SCHEDULING AND LOCKING IN MULTIPROCESSOR REAL-TIME OPERATING SYSTEMS

Björn B. Brandenburg

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2011

Approved by:

James H. Anderson

Sanjoy K. Baruah

Hermann Härtig

Jan F. Prins

F. Donelson Smith

Paul E. McKenney

ABSTRACT

BJÖRN B. BRANDENBURG: Scheduling and Locking in Multiprocessor Real-Time Operating Systems
(Under the direction of James H. Anderson)

With the widespread adoption of multicore architectures, multiprocessors are now a standard deployment platform for (soft) real-time applications. This dissertation addresses two questions fundamental to the design of multicore-ready real-time operating systems: (1) Which scheduling policies offer the greatest flexibility in satisfying temporal constraints; and (2) which locking algorithms should be used to avoid unpredictable delays?

With regard to Question 1, LITMUS$^{\mathrm{RT}}$, a real-time extension of the Linux kernel, is presented and its design is discussed in detail. Notably, LITMUS$^{\mathrm{RT}}$ implements link-based scheduling, a novel approach to controlling blocking due to non-preemptive sections. Each implemented scheduler (22 configurations in total) is evaluated under consideration of overheads on a 24-core Intel Xeon platform. The experiments show that partitioned earliest-deadline first (EDF) scheduling is generally preferable in a hard real-time setting, whereas global and clustered EDF scheduling are effective in a soft real-time setting.

With regard to Question 2, real-time locking protocols are required to ensure that the maximum delay due to priority inversion can be bounded *a priori*. Several spinlock- and semaphore-based multiprocessor real-time locking protocols for mutual exclusion (mutex), reader-writer (RW) exclusion, and k-exclusion are proposed and analyzed. A new category of RW locks suited to worst-case analysis, termed phase-fair locks, is proposed and three efficient phase-fair spinlock implementations are provided (one with few atomic operations, one with low space requirements, and one with constant RMR complexity).

Maximum priority-inversion blocking is proposed as a natural complexity measure for semaphore protocols. It is shown that there are two classes of schedulability analysis, namely suspension-oblivious and suspension-aware analysis, that yield two different lower bounds on blocking. Five

asymptotically optimal locking protocols are designed and analyzed: a family of mutex, RW, and k-exclusion protocols for global, partitioned, and clustered scheduling that are asymptotically optimal in the suspension-oblivious case, and a mutex protocol for partitioned scheduling that is asymptotically optimal in the suspension-aware case. A LITMUS$^{\mathrm{RT}}$-based empirical evaluation is presented that shows these protocols to be practical.

To my parents, Harald and Petra.

ACKNOWLEDGEMENTS

I am indebted to many people who first helped me to get into graduate school, and then helped me to get out again. First of all, I would like to thank my advisor, Jim Anderson, for his unwavering support, and my committee, Sanjoy Baruah, Hermann Härtig, Jan Prins, Don Smith, and Paul McKenney, for their guidance and advice. I would also like to express my thanks and appreciation to all my co-authors, Aaron Block, Hennadiy Leontyev, John Calandrino, Uma Devi, Mac Mollison, Chris Kenna, Andrea Bastoni, Jonathan Hermann, and Alex Mills, and the students in the real-time systems group that I sadly did not have a chance to write a paper with, Cong Liu, Glenn Elliott, Guruprasad Aphale, Jeremy Erickson, Haohan Li, Bipasa Chattopadhyay, and Chih-Hao Sun, for their help and feedback. Many thanks also to the friendly CS department staff, and in particular Jodie Turnbull, John Sopko, Murray Anderegg, and Mike Stone, who helped me in many large and small ways over the years.

I am grateful to the German-American Fulbright Program and the Graduate School's Dissertation Completion Fellowship for financial support during my first and last year in graduate school, respectively. Without Fulbright's support, I probably would not have attended an American graduate school. Special thanks are also due to Jan Prins, for quickly un-rejecting my application to UNC after it ended up on the wrong pile due to a "processing mishap."

Life in Sitterson Hall and Brooks Building would not have been much fun without my friends in the CS department: Aaron Block, Sasa Junuzovic, Jay Aikat, Sean Curtis, Stephen Olivier, Keith Lee, Jamie Snape, Srinivas Krishnan, Anish Chandak, Jeff Terrell, Stephen Guy—thank y'all for a great time in North Carolina!

Outside school, I am greatly indebted to Jasper McChesney. I was very lucky to meet you by chance in the roommate lottery and have greatly enjoyed your company—thank you very much for the many interesting discussions, hiking trips, and wonderful times of advanced nerdery. In a similar vein, I would also like to thank Dot Hall for her compassion, energy, and the occasional crazy idea. Too bad that we overlapped in Chapel Hill for only a short time.

Very special thanks to Aaron and Nicki Block for their unrelenting support and mentorship. Whenever I needed help or guidance, I knew I could count on you. You taught me a lot about American culture; without you my stay in the U.S. would have been only half as interesting and fun. Thank you very much for your generosity and friendship. One day we will finish that bike ride around Berlin!

Another colleague and friend that deserves special mention is Andrea Bastoni, who was a visiting scholar with UNC's real-time group from October 2009 until October 2010. Thanks a lot for all the fun we had inside and outside of the department, for the papers we wrote, and for your contributions to LITMUS$^{\mathrm{RT}}$. You came at exactly the right time; unfortunately you had to leave again after only a year. Thanks for staying involved with LITMUS$^{\mathrm{RT}}$ anyway.

Foremost, I thank my parents Harald and Petra for their unwavering support, understanding, and encouragement, both during graduate school and the years before. A defining event in my life was the high school exchange that brought me to Mayfield High School in Las Cruces, NM during junior year. In Las Cruces, I developed ambition and tenacity, without which this dissertation would not have been possible. For this I would like to thank my former host parents Barbara and George Stafford, who out of the kindness of their hearts opened their door to an unknown teenager.

Finally, I am deeply thankful to Nora, for all the love and support, for her patience, for following me halfway around the world to North Carolina, and for her trust that this was indeed the right way. I could not have finished this without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACPI	Advanced Configuration and Power Interface
ACPI PM	ACPI Power Management
APIC	Advanced Programmable Interrupt Controller
API	Application Programming Interface
ASN	Address Space Number
C-EDF	Clustered Earliest-Deadline First
C-PD^2	Clustered PD^2
CFS	Completely Fair Scheduler
COTS	Components Of The Shelf
CPMD	Cache-related Preemption and Migration Delay
CSI	Cycle-Stealing Interrupt
DI	Device Interrupt
DM	Deadline-Monotonic
DPCP	Distributed Priority-Ceiling Protocol
DSP	Digital Signal Processor
ECU	Engine Control Unit
EDF	Earliest-Deadline First
EOI	End Of Interrupt
ERfair	Early-Release fair
ESP	Electronic Stability Program
FDSO	File-Descriptor-attached Shared Object
FIFO	First-In First-Out
FMLP$^+$	Flexible Multiprocessor Locking Protocol (improved)
FMLP	Flexible Multiprocessor Locking Protocol
FPU	Floating Point Unit
FP	Fixed-Priority
G-EDF	Global Earliest-Deadline First
G-FP	Global Fixed-Priority

GPOS	General-Purpose Operating System
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HPET	High-Precision Event Timer
HRT	Hard Real-Time
HW	Hardware
I/O	Input/Output
IPC	Inter-Process Communication
IPI	Inter-Processor Interrupt
IQR	Inter-Quartile Range
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
JLDP	Job-Level Dynamic-Priority
JLFP	Job-Level Fixed-Priority
LDID	Logical Destination ID
LKM	Loadable Kernel Module
MMU	Memory Management Unit
MPCP-VS	Multiprocessor Priority-Ceiling Protocol with Virtual Spinning
MPCP	Multiprocessor Priority-Ceiling Protocol
MSRP	Multiprocessor Stack Resource Policy
NCP	Non-preemptive Critical section Protocol
NMI	Non-Maskable Interrupt
NP-FMLP$^+$	Non-preemptive Flexible Multiprocessor Locking Protocol (improved)
NUMA	Non-Uniform Memory Access
OMLP	$O(m)$ Locking Protocol
OS	Operating System
P-EDF	Partitioned Earliest-Deadline First
P-FP	Partitioned Fixed-Priority

PCB	Process Control Block
PCP	Priority-Ceiling Protocol
pi-blocking	Priority Inversion blocking
PIC	Programmable Interrupt Controller
PIP	Priority Inheritance Protocol
PIT	Programmable Interval Timer
PS	Fluid Processor Sharing
RMR	Remote Memory Reference
RM	Rate-Monotonic
RPC	Remote Procedure Call
RTOS	Real-Time Operating System
RW	Reader-Writer
s-aware	Suspension-aware
s-blocking	Spin blocking
s-oblivious	Suspension-oblivious
SMI	System Management Interrupt
SMM	System Management Mode
SMP	Symmetric Multiprocessor
SRP	Stack Resource Policy
SRT	Soft Real-Time
TI	Timer Interrupt
TLB	Translation Look-aside Buffer
TPR	Task Priority Register
TSC	Timestamp Counter
UMA	Uniform Memory Access
WCET	Worst-Case Execution Time
WSS	Working Set Size

CHAPTER 1

INTRODUCTION

With the recent advent of multicore chips, multiprocessors are now commonly encountered in servers, personal computers, and embedded systems (Sodan *et al.*, 2010). As a result, real-time applications—that is, applications that must satisfy temporal constraints in order to be deemed correct—are increasingly being deployed on multiprocessors (Baker, 2010). One reason for the proliferation of multicore platforms in real-time systems is that such platforms have come to constitute a significant share of the cost-efficient *components-of-the-shelf* (COTS) market (Child, 2010). Another important factor is their considerable processing capacity, which makes them an attractive choice for hosting compute-intensive tasks such as high-definition video stream processing, object recognition and tracking, and computer vision applications in general.

For example, Villalpando *et al.* (2010) evaluated Tilera's 64-core TILE64 processor (Bell *et al.*, 2008) for use in the real-time hazard detection and avoidance system of the Altair Lunar Lander (NASA, 2008). They found "that for [computer vision and image analysis] the TILE64 architecture provides excellent performance," and further recommended the platform for other on-board tasks, including spacecraft and instrument control (Villalpando *et al.*, 2010). Notably, a radiation-hardened 49-core version of the TILE64 processor, named Maestro, has been developed with the explicit purpose of enabling space-born, embedded multicore real-time systems (Malone, 2009; Crago, 2009).

Many real-time applications can be decomposed into a collection of recurrent *tasks*. The goal of this dissertation is to determine how such tasks should be supported at the operating-system level. In particular, the research presented herein is focused on two questions fundamental to the design of multiprocessor real-time operating systems (RTOSs):

Q1 Which scheduling policies offer the greatest flexibility in satisfying temporal constraints?

Q2 Which synchronization techniques should be used to enable tasks to access shared resources (such as message buffers and I/O devices) without incurring unpredictable delays?

To motivate our research, we begin by illustrating the need for predictable and efficient multicore RTOSs with two examples.

1.1 The Cost of Over-Provisioning

Temporal constraints naturally arise whenever computers interact with the "real world." In particular, this is the case when computers control or observe physical processes, or when human users perceive a device's reaction as slow or delayed. Real-time constraints are commonly classified as either *hard* or *soft*, with the interpretation that hard real-time (HRT) constraints must always be met, whereas violations are tolerable to a limited extent in the soft real-time (SRT) case.

An example of the former is the *electronic stability control* (or *program*, ESP) deployed in modern cars (Liebemann *et al.*, 2004). The purpose of ESP is to prevent roll-over accidents by correcting driver errors such as exaggerated steering during panic reactions. This is achieved by applying corrective forces to individual wheels when a dangerous level of lateral acceleration is detected (Liebemann *et al.*, 2004). HRT constraints are fundamental to ESP because a delay in the sensing of the current acceleration or enactment of appropriate corrections can have catastrophic consequences. Anti-lock brakes and traction control are subject to similar constraints.

In contrast, an unresponsive *graphical user interface* (GUI) usually does not have catastrophic consequences, but results in unsatisfied customers nonetheless. A recent consumer review of HP's TouchSmart interface criticized that "the [TouchSmart interface] is simply too sluggish for everyday use" (Stern, 2010). Barnes & Noble's Nook e-book reader was disparaged similarly (Topolsky, 2010). The underlying failure in both cases is that the user interface did not respond in time—while not safety-critical, noticeable graphics or audio glitches can spell financial disaster for manufacturers of consumer electronics. Similar arguments apply to popular multimedia features such as video playback and interactive games.

To impose minimum levels of predictability in safety-critical applications, governments have instituted certification authorities such as the U.S. Federal Aviation Authority (FAA) and the National Highway Traffic Safety Administration (NHTSA). A key requirement is isolation—the failure of one

subsystem should not affect the correct operation of other components. The easiest and historically most-commonly used way to ensure isolation is to employ a dedicated processor for each functionality. However, this approach has led to an increasingly unmanageable proliferation of such systems, to the effect that some modern cars contain in excess of one hundred processors (Hergenhan and Heiser, 2008). Unsurprisingly, there are strong market forces and design imperatives that require engineers to curb this spread of embedded processors: not only does the overall system complexity (and thus cost) grow with each added component, but every embedded system requires wiring and cooling, adds weight, requires space, drains power, and must be purchased, transported, stored, tested, documented, serviced, eventually replaced, and finally recycled. Thus, instead of embedding one hundred networked, slow uniprocessors throughout a car, it would be desirable to use only ten (or fewer) shared, but ten-times as powerful, multicore processors that are highly utilized.

Over-provisioning GUIs and other soft real-time applications with faster-than-necessary processors is similarly costly, and especially so in the highly competitive mobile sector (*e.g.*, smart phones and tablets), which is subject to stringent energy, cooling, and cost constraints. There are thus strong incentives to "do more with less": an increasing number of real-time tasks need to share limited physical resources on multiprocessors without compromising predictability or efficiency.

1.2 The Divergence of Theory and Practice

A solid RTOS foundation is thus required to fully exploit the promise of multicore technology for real-time applications. Unfortunately, the existing foundation appears increasingly insufficient, as most RTOSs in common use today still rely on approaches rooted in decades-old uniprocessor design concepts.[1] Motivated by these developments, research on multiprocessor real-time systems has surged in recent years and many algorithmic advances have been made. Unfortunately, these results have found only little adoption in practice.

We briefly digress to introduce needed background before discussing the current state of affairs.

1.2.1 Background

Each of the concepts mentioned in the remainder of this chapter is discussed in detail in Chapter 2.

[1] See Section 2.5 for a discussion of relevant RTOSs.

One particularly well-studied abstraction of recurrent real-time activity is the *sporadic task model*. When such a task is triggered by an (external) event such as an interrupt, it releases a sequential *job* to process the triggering event. Each job has a *deadline* by which it should complete. In this section, we restrict our discussion to *implicit deadlines*, where a job's deadline also marks the earliest-possible release time of the next job (of the same task). A task's *utilization* is the fraction of one processor's capacity that must be reserved for it. Similarly, the *total utilization* of a task set is simply the sum of all utilizations of tasks in the set.

From an RTOS point of view, the sporadic task model is a good compromise between practicality and expressiveness. It is easy to implement, well-analyzed, and can represent both cyclic tasks such as ESP and video decoding as well as event-driven tasks such as GUI applications. To avoid confusion, we use the term "task" to refer to a sporadic task on the model level and let "process" denote the OS-level concept of a sequential thread of control. A sporadic task is typically implemented as a process that executes an infinite loop, where each loop iteration corresponds to one job.

Temporal correctness. A task set is *schedulable* if it can be shown that each task meets its timing constraints. A procedure that establishes whether a task set is schedulable under a given scheduler is a *schedulability test*. In an HRT context, a sporadic task set is schedulable if each job meets its deadline (Liu and Layland, 1973). In contrast, some deadline misses are tolerable in SRT applications: a task set is SRT schedulable if *tardiness* is bounded, *i.e.*, if the magnitude of deadline misses does not exceed a (reasonably small) constant (Devi, 2006).[2] A task set is *feasible* if there exists *some* scheduling policy under which it is schedulable. For a task set to be feasible (either HRT or SRT) on m processors, its total utilization must not exceed m—unbounded deadline misses cannot be avoided if processors are overloaded. In the context of our discussion here, a scheduler can be considered *optimal* if any task set with total utilization at most m is schedulable (with regard to either HRT or SRT constraints, respectively; see Section 2.2.3 for a precise definition of scheduler optimality).

Capacity loss. To meet the design goal of maximum flexibility (recall Question Q1), an ideal RTOS should support *any* feasible task set. Given the above limit on total utilization (and our focus on implicit deadlines), this means that any task set with total utilization not exceeding the total processor capacity m should be schedulable. In practice, it is not possible to allocate all processor capacity to

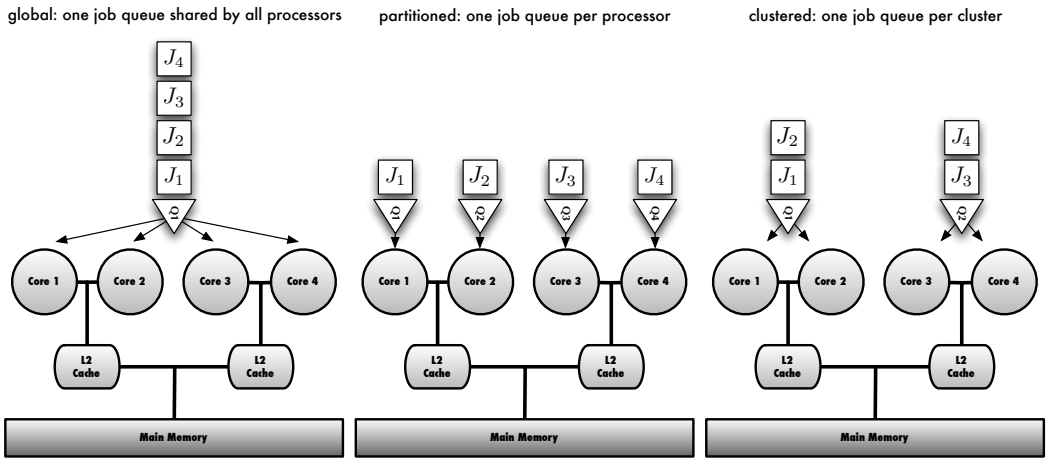
[2] Other notions of SRT constraints exist; see Section 2.2.2.

real-time tasks, *i.e.*, task systems with total utilization less than m may not be schedulable. Such *capacity loss* has two primary causes: the choice of scheduling policy and runtime overheads. With regard to the former, if the RTOS employs a non-optimal scheduling policy, then a feasible task set may not be schedulable due to *algorithmic* capacity loss. The latter, *overhead-related* capacity loss, is due to processor time that is consumed by hardware inefficiencies (*e.g.*, cache misses) and system management activities (*e.g.*, computing a scheduling decision). Such runtime overheads are unavoidable to some degree, but can differ significantly among schedulers and implementations. Time lost to system overheads must be accounted for when testing whether a task set is schedulable on a given platform, which reduces the amount of processor capacity effictively available to real-time tasks.

To minimize overall capacity loss, an RTOS should ideally use a scheduler that is both optimal and that incurs minimal overheads.

Uniprocessor real-time scheduling. The scheduling of sporadic tasks on uniprocessors is well understood. The two most-relevant scheduling policies are *fixed-priority* (FP) and *earliest-deadline first* (EDF) scheduling. Under FP scheduling, each task is statically assigned a unique priority prior to execution. At runtime, competing jobs are then scheduled in order of decreasing task priority. In contrast, under EDF scheduling, jobs are scheduled in order of increasing deadlines (*i.e.*, in order of urgency) and no manual priority assignment is required. In a classic result, Liu and Layland (1973) showed that, for HRT constraints, EDF is optimal, whereas FP is subject to algorithmic capacity loss (Liu and Layland, 1973). Since a task set that is HRT schedulable also has bounded tardiness, EDF is optimal in the SRT case as well.

Nonetheless, FP scheduling is much more common in practice. One likely reason is the relative ease of implementing FP efficiently (using a bitmask where each set bit indicates the presence of pending jobs at a corresponding priority level). While it is also possible to implement EDF efficiently (Short, 2010), the required techniques are conceptually more difficult and are not part of OS developer folklore (in contrast to FP implementation techniques, which are widely known). Buttazzo (2005) argued that the preference for FP is further reinforced by widespread misconceptions and myths concerning supposed advantages in practice (such as failure modes under transient overloads), which he conclusively debunked (Buttazzo, 2005). From a capacity-loss perspective, the pervasive



(a) Global scheduling. (b) Partitioned scheduling. (c) Clustered scheduling.

Figure 1.1: Illustration of multiprocessor scheduling approaches for four cores that share L2 caches in pairs of two. Note that L2 cache affinity is not lost when migrations occur under clustered scheduling. (See Section 2.1.2 for details.)

use of FP as the uniprocessor real-time scheduler of choice is thus not justified. Rather, it could be considered a historical accident—alas, one that has since been codified in the POSIX real-time standard (IEEE, 1993, 2003, 2008b), which mandates FP scheduling. (The POSIX standard does not prohibit additional schedulers, but few RTOSs supplement FP with other real-time schedulers such as EDF.)

1.2.2 Multiprocessor Real-Time Scheduling

There are two fundamental classes of multiprocessor schedulers: *global* and *partitioned*. Under global scheduling (illustrated in inset (a) of Figure 1.1), all processors serve a single ready queue and jobs may migrate among processors. In contrast, under *partitioned* scheduling (illustrated in inset (b) of Figure 1.1), tasks are statically assigned to processors during an offline phase and each processor is scheduled individually using a uniprocessor policy such as EDF or FP.

There is a clear analytical answer to Question Q1: some global schedulers are *provably superior* to any partitioned approach. This is because partitioned scheduling requires tasks to be statically assigned to processors such that no processor is overloaded, which is equivalent to solving a bin-packing problem. As even an optimal packing may leave some bins incompletely filled (*i.e.*, some processors partially idle), there exist task sets with total utilization at most $\frac{m+1}{2} + \epsilon$ that cannot be

partitioned onto m processors (Oh and Baker, 1998; Andersson *et al.*, 2001; Andersson and Jonsson, 2003), where ϵ is an arbitrary small positive number. In other words, nearly up to half of the available processor capacity may be lost to the algorithmic limitations of partitioned scheduling.

In contrast, optimal global schedulers exist. For example, consider *global EDF* (G-EDF), where the uniprocessor EDF policy is globally applied to a shared ready queue. While it has long been known that G-EDF can be subject to severe algorithmic capacity loss in an HRT context (Dhall and Liu, 1978), Devi and Anderson (2005) recently showed that G-EDF does in fact ensure bounded tardiness for any sporadic task set with utilization at most m, *i.e.*, G-EDF is optimal in an SRT context (Devi, 2006; Devi and Anderson, 2008).

For HRT constraints, the global algorithm PD^2 is optimal (Srinivasan and Anderson, 2006). PD^2 is a *proportionate fair* (Baruah *et al.*, 1996), or *pfair*, scheduler, which is a class of schedulers that ensure that a task's processor allocation is always proportional to its utilization (within stringent bounds, see Chapter 2). PD^2 derives its properties from dividing jobs into (many) subtasks of fixed size, each of which is assigned an individual deadline. Subtasks are scheduled on an EDF basis with additional rules to break deadline ties.[3] With an appropriate subtask granularity, PD^2 ensures that all deadlines will be met for any task set with utilization at most m (assuming implicit deadlines).

Therefore, *in theory*, global scheduling is clearly preferable to partitioned scheduling.

However, *in practice*, global schedulers are commonly eschewed. Many OS developers consider global scheduling to be impractical due to the associated runtime overheads and implementation complexity. For example, PD^2's subtask-based scheduling causes jobs to incur frequent preemptions and migrations. While the scheduling literature generally considers such costs to be negligible, the associated loss of cache affinity[4] can significantly increase a job's processor demand in practice by introducing additional cache misses. Preemptions occur less frequently under G-EDF. However, processors must still access a shared ready queue, which OS developers traditionally avoid because frequent accesses to global data structures often entail high lock contention and cache-coherency overheads. Consequently, virtually all multiprocessor-capable RTOSs employ partitioned scheduling, where most data structures and scheduling decisions are processor-local and thus cache-friendly.

[3]The name PD^2 is due to its use of two tie-breaking rules and since it supersedes an earlier algorithm named PD by Baruah *et al.* (1995). Algorithm PD derives its name from the use of "pseudo-deadlines" (Baruah *et al.*, 1995).

[4]Cache affinity and the impact of cache misses are reviewed in Section 2.1.2.

Of course, it would be unrealistic to expect global algorithms to scale to tens or hundreds of processors. However, this does not imply that the other extreme, partitioned scheduling, is the best choice. Instead, *clustered scheduling* (Calandrino *et al*., 2007; Baker and Baruah, 2007) could be a practical, better-performing compromise. Illustrated in inset (c) of Figure 1.1, clustered scheduling is a hybrid of both global and partitioned scheduling that groups processors sharing low-level caches into disjoint *clusters*. As under partitioning, tasks are statically assigned to clusters during an offline phase. Within each cluster, jobs are then scheduled "globally" from a shared per-cluster ready queue. The intuition behind clustered scheduling is to reconcile the advantages of both partitioned and global scheduling: the impact of bin-packing issues is reduced as there are fewer and larger bins (compared to partitioned scheduling), some level of cache affinity is maintained during job migrations, and lock contention is reduced as queues are shared among fewer processors (compared to global scheduling).

Nonetheless, most multiprocessor-capable RTOSs currently rely on *partitioned fixed-priority* (P-FP) scheduling as the primary supported real-time scheduling policy. This choice reflects great conservatism: even among partitioned schedulers, P-FP is inferior to *partitioned earliest-deadline first* (P-EDF) scheduling since EDF is optimal on a uniprocessor and FP is not (Liu and Layland, 1973). Yet P-FP has remained the *traditional* RTOS scheduler due to its uniprocessor legacy and the previously mentioned POSIX-compliance reasons.

Compliance with the POSIX real-time standard (IEEE, 1993, 2003, 2008b), however, should not prevent needed innovation and adaptation. The standard is in fact silent on multiprocessor issues,[5] and understandably so, as multiprocessor real-time systems were still an "academic curiosity" when it was first ratified. It further does not preclude RTOSs from supporting other, possibly better-performing scheduling policies in addition to the mandated FP. With the advent of the multicore age, and given that RTOSs are now being redesigned and restructured to adapt, it is presently an opportune time to revisit the choice of a "default" RTOS scheduler.

[5]The preamble to the original POSIX real-time extension makes this explicit: "It is beyond the scope of these interfaces to support networking or multiprocessor functionality" (IEEE, 1993, p. 2). The original document has since been superseded by the "real-time and embedded application profiles" (IEEE, 2003), which define use-case-specific subsets of the (much larger) POSIX "base profile" (IEEE, 2008b). The former does mention multiprocessors, but only in the context of the absence of memory barriers and spinlocks (IEEE, 2003, p. 49). The latter does not discuss multiprocessor scheduling, but does mandate FP scheduling with at least 32 distinct priorities for OSs that support scheduling at all (IEEE, 2008b, pp. 501–505).

1.2.3 Real-Time Locking

Most published schedulability tests make the simplifying assumption that tasks (and hence jobs) are *independent*, that is, it is assumed that a job is never delayed by actions of other jobs. In practice, this assumption does not hold when jobs share resources that are protected by locks. For example, if a job must transmit a message, but the required network device has already been locked and is in use, then the job is *blocked*—it cannot progress until the shared resource becomes available.

Such blocking can endanger temporal correctness because it can give rise to *priority inversions*, which, intuitively, occur when a high-priority job is forced to wait for a lower-priority job. Uncontrolled priority inversions jeopardize the temporal correctness of real-time tasks because they can result in unpredictable delays. However, if the maximum delay due to resource sharing is known *a priori*, then it can be accounted for when testing whether a given task set is schedulable. An RTOS must thus incorporate a *real-time locking protocol* that allows the maximum length of priority inversion to be analytically bounded. Naturally, it is desirable to employ locking protocols that minimize the occurrence of priority inversions since frequent or long priority inversions severely limit an RTOS's suitability for ensuring stringent real-time constraints. A practical locking protocol should further be easily and efficiently implementable, just as with schedulers.

Unfortunately, few RTOSs implement any algorithm beyond basic uniprocessor priority inheritance and ceiling protocols, which are insufficient under partitioned scheduling (see Section 2.4.4.2). In effect, current RTOSs fundamentally fail to provide the means necessary to *predictably* use locks on multiprocessors (special cases aside). Given that extensions of uniprocessor protocols that are (at least theoretically) appropriate for P-FP scheduling have been available for more than 20 years (Rajkumar *et al.*, 1988; Rajkumar, 1990, 1991), this is a dire state of affairs.

Given the relative immaturity of multiprocessor real-time locking research (compared to advances in real-time scheduling), the lack of proper locking protocol support is hardly surprising. In fact, prior to the work presented herein, locking optimality questions had received little, if any, attention. As a result, goals such as "*minimizing* the occurrence of priority inversion" were only intuitively understood and lacked analytical precision. Instead, *provably optimal* multiprocessor locking protocols that are also simple and efficient to implement are needed.

1.3 Thesis Statement

To *fully* exploit the potential of modern multicore platforms, RTOSs will have to advance beyond the traditional, but increasingly strained uniprocessor approach of FP scheduling with priority inheritance. With regard to Question Q1, several promising multiprocessor scheduling approaches exist. With regard to Question Q2, while the multiprocessor locking protocol design space is still relatively unexplored, and questions of optimality have not received much attention in prior work, proper locking protocols for P-FP scheduling exist. Yet neither such protocols nor promising schedulers have been evaluated, much less adopted, in mainstream RTOSs to date.

A primary cause for this growing disconnect is the open question of practicality. Runtime overheads are typically considered to be negligible in algorithm-centric research, but such overheads are rarely negligible in practice. In fact, many overhead sources can exhibit counterintuitive trends that are difficult to anticipate. As a result, there is a strong focus on overhead measurements in the RTOS community, with a particular focus on *interrupt latency*, which is a metric that reflects the RTOS's responsiveness in enacting scheduling decisions. Unfortunately, this causes a disregard for algorithmic properties, as latency by itself fails to capture schedulability analysis tradeoffs that arise when multiple tasks subject to varying timing constraints compete for processor time.

Either extreme—completely disregarding overheads or making them the sole focus—yields only incomplete and misleading answers. Are algorithmically optimal, high-overhead global schedulers still superior to non-optimal, low-overhead partitioned schedulers if overheads and algorithmic properties are both fully accounted for? If not, is clustered scheduling a viable compromise?

Motivated by these considerations, the main thesis supported by this dissertation is the following.

> *A "multicore-ready" RTOS should employ scheduling and synchronization algorithms that minimize the loss of processor capacity allocable to real-time tasks. When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,* ***(i)*** *partitioned scheduling is preferable to global and clustered approaches in the HRT case,* ***(ii)*** *P-EDF is superior to P-FP, and* ***(iii)*** *clustered scheduling can be effective in reducing the impact of bin-packing limitations in the SRT case. Further,* ***(iv)*** *multiprocessor locking protocols exist that are both efficiently implementable and asymptotically optimal with regard to the maximum duration of priority inversion.*

1.4 Contributions

In the following, we briefly summarize the contributions presented in the subsequent chapters.

1.4.1 LITMUS$^{\mathrm{RT}}$

Central to our work is the proposition that a meaningful comparison of multiprocessor scheduling and locking algorithms must be implementation-based. An actual RTOS is thus required. To this extent, we co-developed the **L**inux **T**estbed for **Mu**ltiprocessor **S**cheduling in **R**eal-**T**ime Systems (LITMUS$^{\mathrm{RT}}$).[6]

As the name implies, LITMUS$^{\mathrm{RT}}$ is an extension of Linux (Torvalds, 1997; Torvalds and contributors, 2010), which is a widely used open-source UNIX-like kernel. Conceptually, LITMUS$^{\mathrm{RT}}$ consists of three parts: **(i)** it extends Linux's scheduling infrastructure with an implementation of the sporadic task model, **(ii)** it provides a plugin interface that allows the active scheduling policy and locking protocols to be changed at runtime, and **(iii)** it provides additional system calls for real-time tasks.

The Linux scheduling framework is structured as a hierarchy of *scheduling classes*. Each scheduling class encapsulates the policy used for a particular process type. For example, regular user-space processes are scheduled using CFS, a timesharing policy inspired by proportional-share fair scheduling (Tijdeman, 1980; Stoica *et al.*, 1996). Similarly, scheduling classes for low-priority background work and high-priority "real-time" processes exist. (In compliance with the POSIX standard, Linux's real-time scheduling class uses FP scheduling with support for 100 distinct priority levels.) Whenever Linux makes a scheduling decision, it polls each scheduling class in the hierarchy in top-to-bottom order until a pending task is found. LITMUS$^{\mathrm{RT}}$ introduces a new scheduling class at the top of the hierarchy such that non-LITMUS$^{\mathrm{RT}}$ tasks are only scheduled when no real-time workload is present.

In contrast to the regular Linux scheduling classes, the LITMUS$^{\mathrm{RT}}$ scheduling class does not implement any particular policy. Instead, it defers all scheduling decisions to the active real-time

[6] LITMUS$^{\mathrm{RT}}$ is a group effort. However, the author is the principal designer and developer of the core LITMUS$^{\mathrm{RT}}$ distribution and has contributed most of its significant features in the process of preparing this dissertation. Various extensions of the LITMUS$^{\mathrm{RT}}$ core have been developed and described by others (Block, 2008; Calandrino, 2009; Leontyev, 2010; Bastoni, 2011); these are not the subject of this dissertation.

scheduler plugin. This simplifies experimental scheduler development and maintenance: instead of interfacing with the full Linux kernel (and its considerable complexities that tend to change from version to version), LITMUS$^{\mathrm{RT}}$ plugins interface only with the LITMUS$^{\mathrm{RT}}$ scheduling class via the plugin interface, which remains (mostly) stable between releases. Notably, LITMUS$^{\mathrm{RT}}$ supports global scheduling policies, which are not well supported by standard Linux.

LITMUS$^{\mathrm{RT}}$ further augments the Linux system call interface with additional real-time-specific system calls. In particular, LITMUS$^{\mathrm{RT}}$ adds calls for real-time task and job control (*e.g.*, to configure task parameters, to wait for the next job release, to obtain the job sequence number, *etc.*) and for invoking real-time locking protocols (*e.g.*, to acquire resource handles, to lock resources, *etc.*).

An early prototype of LITMUS$^{\mathrm{RT}}$ existed prior to the commencement of the work described in this dissertation (Calandrino *et al.*, 2006); however, this early version lacked several critical features. In particular, it did not allow processes to suspend for any reason (such as loading shared libraries, acquiring a semaphore, or using any kind of I/O device), to the effect that the system failed with a "kernel panic" if a real-time task attempted to suspend. Consequently, only artificial, completely processor-bound workloads could be supported. In the current version of LITMUS$^{\mathrm{RT}}$ (which this dissertation is based on), these limitations have been removed and virtually no code from the beginnings of LITMUS$^{\mathrm{RT}}$ remains. The design and implementation of LITMUS$^{\mathrm{RT}}$ is discussed in detail in Chapter 3.

1.4.2 Comprehensive Overhead Accounting

As argued above, it is critical to account for runtime overheads when establishing whether a task set is schedulable. This requires procedures for incorporating delays due to overheads into published schedulability tests that assume ideal, overhead-free execution. In particular, such procedures must be *safe*, *i.e.*, they may not underestimate delays due to overheads at runtime.

Such procedures have long been known for uniprocessor systems and are discussed in detail in (Liu, 2000). Some of these also apply to multiprocessor scheduling and have been used in this context before (Devi, 2006; Calandrino *et al.*, 2006). However, to the best of our knowledge, there is no comprehensive summary of multiprocessor overhead accounting techniques available today.

In Chapter 3, we present a complete description of the techniques that we use to account for multiprocessor scheduling overheads, and how these techniques apply to LITMUS$^{\mathrm{RT}}$ in particular.

We further describe a new *preemption-centric* approach to account for asynchronous interrupts in Section 3.4.5. Under preemption-centric interrupt accounting, each task's execution budget is inflated by a bound on the worst-case delay due to interrupts.

Overheads affect not only scheduling, but also locking protocols. In Chapter 7, we describe locking-related overhead sources (such as system calls, suspensions, and context switches) in detail and derive appropriate accounting methods.

1.4.3 Overhead-Aware Evaluation Methodology

The central contribution with regard to Question Q1 is a methodology for evaluating scheduling choices *under full consideration of overheads* on a given platform. Conceptually, this methodology consists of five steps split across an OS-level phase and a subsequent analytical phase. First, each scheduling algorithm that is to be evaluated is implemented in LITMUS$^{\mathrm{RT}}$. Next, hundreds of benchmark task sets are run to exercise each scheduler over the course of several hours. During these experiments, millions of raw overhead measurements are collected using low-overhead cycle counters; measuring cache-related overheads is particularly challenging (see Section 4.4). In a post-processing step, we extract models of worst- and average-case overheads from the collected traces. Each source of delay (interrupts, scheduling decisions, context switches, *etc.*) is upper-bounded as a function of task-set size using piece-wise linear interpolation. This concludes the OS-level phase of the process.

In the analytical phase, we incorporate the overhead models into existing schedulability tests for each scheduler. This enables testing whether a given task set is schedulable *in the presence of overheads*, thereby rectifying the theoreticians' assumption that overheads are negligible. Finally, we apply each overhead-aware schedulability test to millions of randomly generated sporadic task sets, which is computationally intensive and requires the use of UNC's research cluster. For each scheduler, the resulting fraction of schedulable task sets is an empirical performance measure that reflects both implementation efficiency and algorithmic properties. This allows to answer Question Q1 for a given platform. A predecessor of this approach was described in (Calandrino *et al.*, 2006). However, our methodology as described in Section 4.1 has been considerably evolved and refined.

Since the methodology sketched so far yields a very large number of graphs (several thousand in the case study discussed below), we further describe a method for aggregating these individual graphs into composite graphs that express interesting trends more succinctly (Section 4.1.7).

1.4.4 Case Study: Multiprocessor Real-time Scheduling on 24 Cores

Since schedulers must be compared under consideration of overheads, a proper evaluation is necessarily platform-specific (although it may be possible to extrapolate general trends in some cases). To demonstrate the applicability of our overhead-aware evaluation methodology, and to support our claim that both overheads and algorithmic properties significantly affect scheduler performance, we present a case study that compares 22 scheduler variants (including partitioned, clustered, and global schedulers) on a large (for today's standard) Intel x86-64 multicore platform with a total of 24 cores.

Our results, discussed at length in Sections 4.3– 4.5, show that scheduling policy implementations differ significantly in overheads (up to an order of magnitude in some cases), and that such differences have a major impact: overheads are decidedly not negligible. At the same time, our results also show that low runtime overheads do not imply that a particular scheduler is always preferable: increased overheads are an acceptable price to pay for some, but not all, algorithmic improvements.

Our results further show that there is no single "best" scheduler in practice. Rather, scheduler performance is highly workload-dependent (task parameters, number of tasks, *etc.*). In general, P-EDF is preferable in the HRT case due to its combination of low overheads and uniprocessor optimality. In particular, P-EDF performs as well or better than P-FP for most tested workloads. G-EDF and C-EDF perform well for many SRT workloads and outperform partitioned schedulers in the case of workloads that are difficult to partition (*i.e.*, if there is a large proportion of high-utilization tasks). P-EDF and C-EDF are thus good candidates for RTOS inclusion.

1.4.5 Spin-Based Locking Protocols

Having developed and demonstrated a methodology for choosing a scheduler, we next consider lock-based synchronization protocols for multiprocessor systems in Chapters 5–7. As previously mentioned, research on multiprocessor real-time locking protocols is less mature than scheduling-oriented research. Our contributions thus not only pertain to the selection of such a protocol, but also to questions concerning their design and algorithmic properties.

When a job requires a shared resource that is currently locked by another job, it must wait before it can proceed in its execution. On a multiprocessor, there are two fundamental ways to implement waiting. In suspension-based protocols, a waiting job relinquishes its assigned processor until the resource becomes available, which gives other jobs the chance to execute. In contrast, in spin-based protocols, a waiting job executes a simple delay loop (*i.e.*, it busy-waits or "spins") until it gains access to the desired resource.

While busy-waiting results in a waste of processor cycles when a resource is contended, it has the advantage that spin-based protocols are easier to analyze than suspension-based protocols and that they can be efficiently implemented. The contribution of Chapter 5 is the design and analysis of three spin-based locking protocols, one realizing *mutual exclusion* (mutex), where at most one job may hold a resource at any time, and the other two realizing *reader-writer exclusion* (RW), where only writers require exclusive access.

Mutex protocol. Our mutex protocol relies on non-preemptive execution (jobs spinning or holding a resource may not be preempted) and FIFO queue locks (Anderson, 1990; Mellor-Crummey and Scott, 1991a) to ensure progress. On a platform with m processors, this combination yields an upper bound of $O(m)$ blocking critical sections, since at most one job per processor can precede a job in the wait queue. Spin-based mutex protocols using FIFO queue locks have been previously studied in the context of pfair schedulers (Holman, 2004; Holman and Anderson, 2002a, 2006), P-EDF (Gai *et al*., 2003), and G-EDF (Devi *et al*., 2006). Our contribution is twofold. We present an improved *holistic* analysis framework that reduces pessimism in the derivation of bounds on the worst-case delay by considering all critical sections of a job simultaneously; secondly, we present a mechanism to enact preemptions "lazily" that limits the negative impact of non-preemptive execution under event-driven global schedulers such as G-EDF. The latter is required to prevent some jobs from being repeatedly preempted and thus delayed (see Section 3.3.3 for details). Prior work did not consider the possibility of such repeated preemptions.

RW protocols. Spin-based RW locks have not been considered in prior work on multiprocessor real-time systems. In throughput-oriented computing, RW locks are attractive because they increase average concurrency (compared to mutex locks) if reads occur more frequently than writes. In a real-time context, RW locks should also aid in lowering the maximum length of priority inversions

for readers, *i.e.*, the higher degree of concurrency must be reflected in *a priori* worst-case analysis and not just in observed average-case delays.

Three types of RW locks with predictable ordering of readers and writers have been studied in prior (non-real-time) work: *writer* and *reader preference locks* (Courtois *et al.*, 1971; Mellor-Crummey and Scott, 1991b), and *task-fair RW locks* (Mellor-Crummey and Scott, 1991b). Under a reader (respectively, writer) preference lock, write (respectively, read) requests are only satisfied if there are no unsatisfied read (respectively, write) requests. Under a task-fair RW lock, requests (either read or write) are satisfied strictly in FIFO order.

Unfortunately, neither preference nor task-fair RW locks are well-suited to real-time systems: preference locks allow starvation, which entails unpredictable and excessive delays, and task-fair RW locks can cause mutex-like delays when readers and writers issue requests in an interleaved manner, and thus offer little analytical advantage over regular mutex locks. In Chapter 5, we introduce and analyze *phase-fair* RW locks as an alternative, under which *reader phases* and *writer phases* alternate (unless there are only requests of one kind). At the beginning of a reader phase, all incomplete read requests are satisfied, whereas one write request is satisfied at the beginning of a writer phase. This results in $O(1)$ acquisition delay for read requests without starving write requests. Additionally, an analysis of task-fair RW locks is presented as well.

1.4.6 Efficient Phase-Fair Spinlock Implementations

An efficient implementation is essential to making spin-based locking protocols competitive. That is, a low-overhead implementation is required to compensate for the spin-related waste of processor cycles. If there would be no efficient implementation of phase-fair locks, then their practical value would be severely limited. In Section 5.3, we present three phase-fair RW locks that can be implemented on common multiprocessor instruction set architectures (ISAs) such as Intel's x86 platform.

The first lock requires 16 bytes per lock and is optimized to require a minimum number of atomic instructions, which carry a high execution cost on common multiprocessor platforms. The second lock is more compact and requires only four bytes per lock, which makes it more appropriate for memory-constrained embedded systems. However, this is achieved at the expense of additional atomic instructions. The third lock type is designed to minimize the number of *remote memory*

references, which corresponds to the number of cache invalidations that are required in a cache-coherent shared-memory system (caches are reviewed in Section 2.1.2). Since such invalidations can be a major source of overhead in multiprocessor systems, these locks may be particularly appropriate for large multicore platforms with complex memory architectures. The performance of each lock in terms of acquisition and release overhead is compared in Section 7.2.

1.4.7 A Notion of Optimality for Suspension-Based Locking Protocols

This dissertation presents several contributions in the area of suspension-based multiprocessor locking protocols. Despite the fact that the first such protocol was proposed more than 20 years ago, the fundamental limits of mutual exclusion in multiprocessor real-time systems have not been explored in prior work. In Chapter 6, we propose *maximum priority-inversion blocking* as a natural complexity measure for real-time locking protocols. For this measure, we prove that there are two common classes of schedulability analysis that yield two *different* lower bounds on blocking because they differ in the analysis of suspensions. In the following discussion, recall that we let m denote the number of processors, and let n denote the number of tasks; we assume here that $n \geq m$.

The first kind, *suspension-oblivious* schedulability analysis, does not allow for suspension times to be explicitly accounted for. This lack of expressivity in the task model necessitates such times to be modeled as computation instead. Consequently, suspension-oblivious analysis over-estimates the processor demand of resource-sharing tasks and thereby yields pessimistic but sound analysis results. Since there exist task sets in which a resource is contended by jobs on each processor at the same time, the maximum duration of priority inversion under *any* mutual exclusion protocol is bounded from below in general by $\Omega(m)$, which we show in Section 6.1.3. We further show that, when combined with priority inheritance, neither priority queues nor FIFO queues are sufficient by themselves to ensure an $O(m)$ upper bound (Section 6.2.5).

In contrast, *suspension-aware* schedulability analysis considers suspensions explicitly and thus uses less-pessimistic estimates of processor demand. Perhaps surprisingly, this improvement in schedulability analysis comes at the cost of an increased lower bound for mutual exclusion protocols: as we formally show in Section 6.1.4, there exist task sets for which, under suspension-aware schedulability analysis, priority inversions of length $\Omega(n)$ are unavoidable under *any* mutual exclusion protocol, where n is the number of tasks. We also show that priority inheritance can cause

priority inversions of $\Omega(\phi)$ total duration, where ϕ can be chosen to be arbitrarily large (Section 6.3.2), and further demonstrate that prioritizing resource requests by job deadlines or static priorities gives rise to an $\Omega(mn)$ lower bound because priority queues allow starvation (Section 6.3.3). That is, neither priority inheritance nor priority queues can be used to obtain an optimal protocol.

The difference in lower bounds under suspension-oblivious and suspension-aware analysis arises because the nature of what constitutes a "priority inversion" is changed by the assumption underlying suspension-oblivious analysis. Intuitively, the analytical "trick" is to "reuse" some of the pessimism inherent in treating suspensions as execution time to derive less pessimistic bounds on priority inversion length. Suspension-oblivious schedulability is relevant because suspension-aware analysis has not yet been developed for many global and clustered schedulers.

1.4.8 Optimal Suspension-Based Multiprocessor Locking Protocols

The next contribution of this dissertation is the design and analysis of suspension-based real-time locking protocols, namely the $O(m)$ *locking protocol* (OMLP) family and the *FIFO mutex locking protocol* (FMLP$^+$), which are discussed in detail in Section 6.2 and Section 6.3, respectively.

The OMLP family includes mutex, RW, and k-exclusion[7] protocols for global, clustered, and partitioned job-level fixed-priority schedulers. The defining characteristic of the OMLP family is that each variant is asymptotically optimal with regard to maximum priority inversion length under suspension-oblivious schedulability analysis. To the best of our knowledge, these are the first optimal protocols for suspension-oblivious analysis. Two novel design techniques were developed to achieve this property: the OMLP mutex protocol for global scheduling is the first protocol to employ a two-level hybrid queue that incorporates both a priority queue and a bounded-length FIFO queue, and the OMLP variants for clustered scheduling employ *priority donation*, which is a new mechanism that ensures that resource-holding jobs are scheduled even if higher-priority jobs are released.

By design, the OMLP protocols are not suited to suspension-aware analysis. Instead, we present the FMLP$^+$, a mutex protocol for partitioned scheduling that is asymptotically optimal with regard to maximum priority inversion length under suspension-aware analysis. The FMLP$^+$ is a close derivative of the *flexible multiprocessor locking protocol* (FMLP), an earlier multiprocessor real-time

[7] A k-exclusion protocol allows up to k concurrent critical sections. Such protocols can be used to arbitrate access to resources of which there are multiple, identical replicas such as multiple graphics processing units (GPUs).

locking protocol for global and partitioned scheduling that we developed in joint work with Block *et al*. (2007). In Section 6.3.2, we show that the global FMLP, which relies on priority inheritance, ensures $O(n)$ maximum priority inversion length in certain situations, but that priority inheritance cannot yield an optimal protocol in general. To the best of our knowledge, the FMLP is the first suspension-based multiprocessor real-time locking protocol that relies exclusively on FIFO ordering. FIFO ordering conflicts with the "traditional" uniprocessor protocols, which typically rely on priority queues in real-time systems. However, as mentioned above in Section 1.4.7, priority queues are challenging in a multiprocessor context because they allow starvation. The FMLP$^+$ is the first practical suspension-based multiprocessor locking protocol with an asymptotically optimal bound on the maximum priority inversion length.

1.4.9 Case Study: The Impact of Overheads on Locking Protocols

Having proposed several spin- and suspension-based multiprocessor real-time locking protocols, the final contribution of this dissertation is two case studies that apply the proposed overhead-aware evaluation methodology (Section 1.4.3 above) to compare the proposed locking protocols with each other and with alternatives from prior work. As a precursor to the actual study, we discuss the nature of locking-related overheads and how to account for them in overhead-unaware blocking and schedulability analysis by means of inflating task parameters.

The first of the two studies compares task-fair mutex, task-fair RW, and phase-fair RW spinlocks under G-EDF, C-EDF, and P-EDF. The main result is that use of RW spinlocks can yield significant analytical advantages over mutex spinlocks if the write ratio is low, and that phase-fair RW locks offer advantages over task-fair RW locks if multiple writers access a resource. Further, our results show that spinlocks affect each of the schedulers similarly, that is, the general recommendations summarized in Section 1.4.4 remain valid even if tasks are not independent.

The second study compares task-fair mutex spinlocks under G-EDF, C-EDF, and P-EDF scheduling, the OMLP mutex protocol under G-EDF, C-EDF, and P-EDF scheduling, and the FMLP$^+$ and two mutex protocols from prior work under P-FP scheduling. The two key results are that spinlocks outperform all tested semaphore protocols if critical sections are short (in both the HRT and SRT cases), and that semaphore protocols for suspension-oblivious analysis perform better than semaphore protocols for suspension-aware analysis in some, but not all, cases. The

former observation demonstrates that spinlocks are the best choice for locking in well-designed real-time systems with short critical sections; the latter result substantiates our claim that suspension-oblivious analysis is a viable approach, but also shows that there is a need to develop improved suspension-aware schedulability analysis.

Generally speaking, while there are many nuances and individual tradeoffs, the common theme of these studies is that each highlights the importance of considering both overheads and analytic properties when selecting a locking algorithm for use in an RTOS.

1.5 Organization

The remainder of this dissertation is organized as follows. We review relevant prior work and the hardware and algorithmic foundations of our work in Chapter 2. The design and implementation of our RTOS testbed, LITMUS$^{\mathrm{RT}}$, is documented in Chapter 3, together with a comprehensive discussion of the main runtime overheads that affect real-time tasks and how to account for them. Chapter 4 presents our methodology for incorporating overheads into scheduler comparisons, discusses how overhead magnitudes can be empirically estimated, and reports on a case study that demonstrates that our methodology yields insights that are unobtainable via only theoretic or only overhead-based studies.

Chapters 5–7 present our locking-related contributions. In Chapter 5, we derive and analyze several spin-based mutex and RW protocols (including task-fair, phase-fair, and reader and writer preference variants), and present three phase-fair RW locks. Chapter 5 also introduces the analysis framework that forms the foundation for the analysis of all the locking protocols proposed in this dissertation. Chapter 6 presents our work on suspension-based locking protocols, including definitions of suspension-aware and -oblivious analysis, our notion of blocking optimality, the OMLP, the FMLP^{+}, and the corresponding blocking analysis and proofs of optimality. We compare and contrast our protocols, under consideration of overheads, with each other and with three previously proposed protocols in Chapter 7. Finally, Chapter 8 summarizes our results, raises open questions, and discusses future work.

CHAPTER 2

BACKGROUND AND PRIOR WORK

This dissertation builds upon a large body of prior work. In this chapter, we briefly review the required hardware background and discuss relevant real-time scheduling algorithms, locking protocols, and RTOS implementation techniques that form the foundation of our work.

2.1 Hardware Foundations

We begin with a discussion of relevant hardware fundamentals. Given the breadth of the topic, a comprehensive review of current processor technology and hardware architecture is beyond the scope of this dissertation. Instead, we focus on the parts of a computing platform that have the biggest impact on multiprocessor RTOS design and efficiency, namely multicore processors, caches, interrupts, and clocks and timers. As a concrete example, we consider the hardware platform that underlies the case studies presented in Chapters 4 and 7, which is a 24-core Intel Xeon L7455 system (Intel Corporation, 2008b).

2.1.1 Multiprocessors and Multicore Platforms

The term "multiprocessor" encompasses a wide range of system architectures. This dissertation applies to shared-memory, uniform memory access, identical multiprocessors—that is, systems in which task migrations are conceptually viable and for which the choice of scheduler is not predetermined. In the following, we briefly discuss the meaning and significance of the attributes "shared-memory," "uniform memory access," and "identical."

A *multiprocessor* consists of multiple, independently controlled processing units that communicate via a processor *interconnect* (Tanenbaum, 2005). There are two fundamental classes of multiprocessors that differ in the nature of the interconnect. In a *shared-memory* multiprocessor,

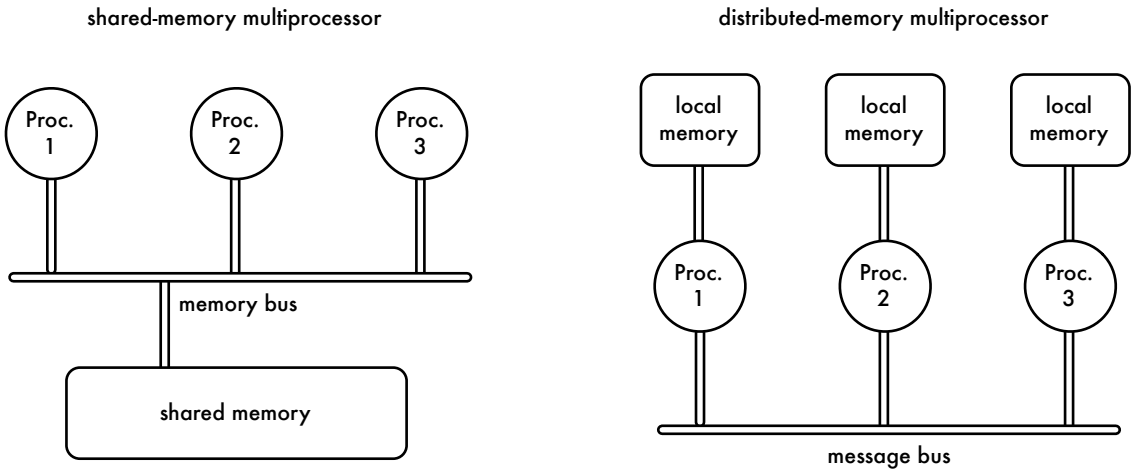


Figure 2.1: Illustration of shared- and distributed-memory multiprocessors. In a shared-memory system, a memory bus enables access to a shared central memory, whereas processors communicate via a message bus that does not allow remote memory access in a distributed-memory system. This dissertation pertains to shared-memory multiprocessors.

there is a central memory that is accessible to all processors and processors are connected to each other and the central memory by means of a shared *memory bus*. In contrast, in a *distributed-memory* system (or *multicomputer*), there are multiple *local* memories that are accessible to only a subset of the processors (historically one). Processors are still connected to each other in a distributed-memory system, but only via a *message bus* that does not allow direct access to non-local memory. Figure 2.1 illustrates the difference on a conceptual level. In practice, interconnect implementations are usually much more refined than a simple "bus" since they are crucial to a multiprocessor's effective speed and capacity; Baer (2010) provides a detailed introduction to multiprocessor interconnects and Bjerregaard and Mahadevan (2006) survey high-performance, on-chip switched networks.

From a scheduling point of view, the main difference between shared- and distributed-memory architectures is how *process migration* is implemented, *i.e.*, how a process may commence execution on one processor and, transparently to the process, continue execution on another processor. When a process is migrated in a shared-memory system, only its hardware state (such as register contents) must be transferred since its data (including its OS state) is accessible from all processors. In contrast, the data of a migrating process must be *copied* from one local memory to another in a distributed-memory system. Copying process state is challenging to implement at the OS level and can create considerable load on the communication bus (see Milojičić *et al*. (2000) for an overview

of process migration techniques and challenges). Consequently, *frequent* process migrations are impractical in distributed-memory systems, which precludes clustered and global scheduling policies.

Shared-memory systems further differ with regard to memory access times. Ideally, a processor should be able to access each memory location with the same maximum latency. Such systems are called *uniform memory access* (UMA) architectures. However, completely centralized memory systems can quickly become a bottleneck in practice because only one or few processors can access the central memory at a time. Hence, the available memory may be split across several modules, which may result in a *non-uniform memory access* (NUMA) architecture where some memory modules are closer to a particular processor than others. To achieve maximum efficiency in NUMA systems, a task should be scheduled on a processor "close" to its data, which precludes global scheduling.

This dissertation applies mainly to UMA multiprocessors; the impact of accommodating NUMA constraints is briefly discussed in Chapter 8.

Processor symmetry. In the scheduling literature, shared-memory multiprocessors are further classified based on the capabilities of their constituent processors, *e.g.*, see (Brucker, 2007; Funk, 2004). In *identical* multiprocessors, there are no differences among processors, that is, a task's execution is not affected by the processor's identity. Identical multiprocessors are also commonly referred to as *symmetric multiprocessors* (SMPs). In *uniform heterogeneous* multiprocessors, processors differ in speed but have otherwise equal capabilities. In this case, the execution requirement of tasks assigned to a slower processor is scaled up proportionally to its speed. A widespread example for uniform heterogeneous multiprocessors is systems that support per-processor frequency scaling as a power-saving measure. Finally, in *unrelated heterogeneous* multiprocessors, each processor may have special capabilities (such as application-specific co-processors) such that execution time requirements are both processor- and task-specific. The three classes of multiprocessors are illustrated in Figure 2.2.

We restrict our focus to identical multiprocessors in the main part of this dissertation and briefly consider how our results apply to the heterogeneous cases in Chapter 8.

Multicore vs. multithreading vs. multiprocessor. As mentioned in the introduction, interest in multiprocessor real-time scheduling is driven in large part by the widespread emergence of multicore

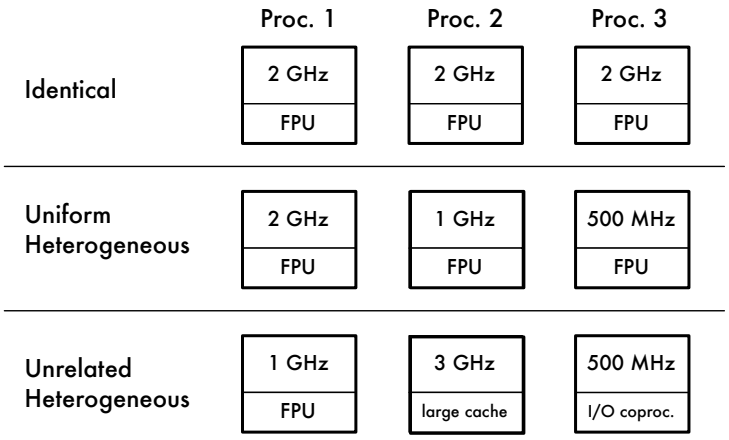


Figure 2.2: Illustration of identical, uniform heterogeneous, and unrelated heterogeneous multiprocessors. No differences exist among processors in identical multiprocessors, *e.g.*, here, each processor has a speed of 2 GHz and a floating point unit (FPU). Processors differ in speed but not capabilities in uniform heterogeneous multiprocessors, *e.g.*, each processor has a FPU but is clocked at different speeds. Processors differ in capabilities (and possibly also speed) in unrelated heterogeneous processors, *e.g.*, only one processor has an FPU whereas the others are targeted at fast integer arithmetic and slow I/O processing. The focus of this dissertation is identical multiprocessors.

platforms. In a *multicore* design, multiple (mostly) independent processing *cores* are manufactured on a single integrated circuit *chip* to exploit increases in transistor density (Olukotun *et al.*, 1996; Sodan *et al.*, 2010). Such systems are sometimes referred to as a *multiprocessor on a chip* (Tanenbaum, 2005). From a scheduling point of view, "multicore" is thus simply a particular way of implementing multiprocessors.

Another technique similarly used to exploit improved transistor densities for increased parallelism is *hardware multithreading*, where some of a core's functional units (such as the instruction pipeline) are replicated such that the core appears as multiple "processors" to the operating system. Multithreading allows multiple task contexts to appear to be scheduled "concurrently," when in fact the core alternates rapidly among hardware threads. This can improve overall throughput by increasing a core's utilization because a core may quickly switch hardware threads whenever the currently executing thread stalls (*e.g.*, due to a cache miss—see Section 2.1.2 below). However, from a real-time perspective, hardware multithreading can be problematic because it can introduce hard-to-predict execution-time variations (Barre *et al.*, 2008; Jain *et al.*, 2002).

The terms "chip," "CPU", "core," and "processor" are used interchangeably in many documents. To avoid confusion, we adopt the following nomenclature in this dissertation. A computer contains

one or more processing *chips*. Each chip consists of one or more *cores* that perform the actual computations. Each core makes one or more *hardware threads* available to the operating system. For scheduling purposes, every hardware thread/context that can be independently scheduled by the operating system is a *processor*.

Xeon L7455. As mentioned above, our experimental platform is a 24-core 64-bit UMA machine, which consists of four physical Intel Xeon L7455 chips (Intel Corporation, 2008b). Each chip contains six cores running at 2.13 GHz. Intel systems support a variety of frequency scaling and power-saving techniques. Due to our focus on identical multiprocessors, we do not use any of these techniques and keep each core clocked at full speed at all times. This generation of Intel's Xeon family does not support "hyperthreading," Intel's implementation of hardware multithreading. Each chip contains two levels of shared caches, which we discuss next.

2.1.2 Processor Caches

Modern processors employ a hierarchy of fast *cache memories* that contain recently accessed instructions and data to alleviate high off-chip memory latencies. Additionally, processors with memory management units (MMU) also have a *translation look-aside buffer* (TLB).

A MMU is used to translate virtual memory addresses into physical memory addresses and is the foundation on which address space separation is implemented in modern OSs. Performing such a translation is relatively slow. The TLB is used to store previously resolved virtual-to-physical address mappings, thereby ensuring that the MMU does not have to perform a translation on every memory reference. There is usually one local TLB per processor.

Caches are typically organized in layers (or levels), where the fastest (and usually smallest) caches are denoted *level-1* (L1) caches, with deeper caches (L2, L3, *etc.*) being successively larger and slower. A cache contains either instructions or data, and may contain both if it is *unified*. In multiprocessors, *shared* caches serve multiple processors, in contrast to *private* caches, which serve only one. Shared caches have become more prevalent with multicore chips. A typical design is shown in Figure 2.3, where each processor has a private L1 cache and groups of two processors each share an L2 cache.

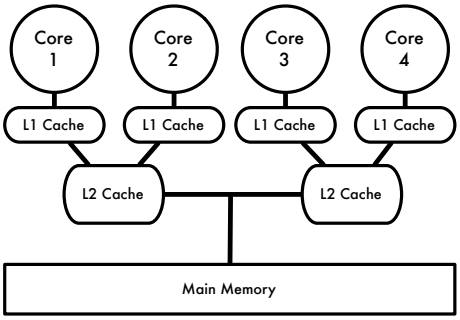


Figure 2.3: Example two-level cache hierarchy with shared L2 caches.

Caches operate on blocks of consecutive addresses called *cache lines* with common sizes ranging from eight to 128 bytes. In *direct mapped* caches, each cache line may only reside in one specific location in the cache. In *fully associative* caches, each cache line may reside at any location in the cache. In practice, most caches are *set associative*, wherein each line may reside at a fixed number of locations. In an x*-way* associative cache, each cache line may be mapped to x distinct cache locations.

In a multiprocessor, caches might become inconsistent if one processor updates a memory location that is currently cached by other processors. We restrict our focus to *cache-coherent* multiprocessors. Such processors employ a *cache-consistency protocol* to transparently evict outdated cache entries from the caches of other processors. Instead of evicting outdated cache entries, a cache-consistency protocol could also propagate the new value; however, this technique is not employed by the platform considered in this dissertation. Hennessy and Patterson (2006) provide a detailed introduction to cache-consistency protocols, and Baer (2010) discusses common implementation techniques.

Cache use. The efficacy of a cache hierarchy depends on the memory requirements and access patterns of the scheduled tasks and the underlying RTOS. Fundamentally, caches work because programs exhibit *temporal* and *spatial locality*—at a given time, a typical program will only access a small subset of its memory. A classic notion of access locality is the *working set* of a task. First proposed by Denning (1968) in the context of virtual memory, he defined the working set as the set of pages that must be present within a given interval to assure efficient execution, *i.e.*, to avoid page faults. Agarwal *et al*. (1989) applied this definition to cache lines. Under their definition, the working set is the set of cache lines that will be referenced (within the analysis time frame). For long-running

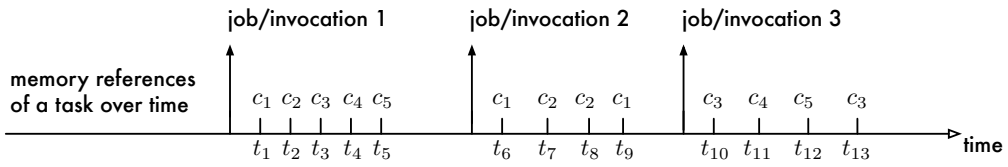


Figure 2.4: Memory references (cache lines $c_1, \ldots, c_5$) of three invocations of a task.

processes, the working set usually changes slowly over time when the program executes in a steady state, and may change abruptly if the program transitions to a different "execution phase." Working sets are thus generally studied with respect to some time interval (or instruction sequence).

Recall that processes implement tasks, and that each distinct invocation of a task is modeled as a job release in the sporadic task model (discussed in detail in Section 2.2 below). Since this dissertation is concerned with sporadic workloads, we interpret "working set" with respect to a single job: the set of all cache lines accessed by a job is its working set. This is similar to Thiebaut and Stone's notion of *cache footprint* (Thiebaut and Stone, 1987), which they define as the "active portion" of a task that is present in a cache. However, strictly speaking, a cache line that is part of the cache footprint is not necessarily also part of the working set (at a given point in execution). This is because a cache line that has previously been brought into the cache could potentially not be accessed again.

Example 2.1. Figure 2.4 depicts the memory references of three invocations (or jobs) of an example task. The working set and cache footprint at each point in time during the task's execution is listed in Table 2.1. The first job sequentially accesses $c_1, \ldots, c_5$ so that each cache line is accessed only once. The working set of the job—cache lines that it requires in the future—thus shrinks with each memory reference. In contrast, its cache footprint—cache lines that have been brought into the cache on the job's behalf—increases with each memory reference. Before the first memory reference at time t_1, the working set encompasses all five cache lines, but the job has no cache footprint yet. After the last memory reference at time t_5, the working set is empty (the first invocation is complete), but the cache footprint is maximal. The first job demonstrates a worst-case scenario from a cache-efficiency point of view: it has a large cache footprint, but no reuse of cache contents occurs since each cache line is accessed only once, *i.e.*, the working set and cache footprint are disjoint.

Job	Time t	Working set (Agarwal *et al.*, 1989) (from time t until next invocation)	Cache footprint (Thiebaut and Stone, 1987) (prior to memory reference at time t)
1	t_1	$\{c_1, c_2, c_3, c_4, c_5\}$	$\emptyset$
	t_2	$\{c_2, c_3, c_4, c_5\}$	$\{c_1\}$
	t_3	$\{c_3, c_4, c_5\}$	$\{c_1, c_2\}$
	t_4	$\{c_4, c_5\}$	$\{c_1, c_2, c_3\}$
	t_5	$\{c_5\}$	$\{c_1, c_2, c_3, c_4\}$
2	t_6	$\{c_1, c_2\}$	$\emptyset$
	t_7	$\{c_1, c_2\}$	$\{c_1\}$
	t_8	$\{c_1, c_2\}$	$\{c_1, c_2\}$
	t_9	$\{c_1\}$	$\{c_1, c_2\}$
3	t_{10}	$\{c_3, c_4, c_5\}$	$\emptyset$
	t_{11}	$\{c_3, c_4, c_5\}$	$\{c_3\}$
	t_{12}	$\{c_3, c_5\}$	$\{c_3, c_4\}$
	t_{13}	$\{c_3\}$	$\{c_3, c_4, c_5\}$

Table 2.1: The working set and cache footprint of the task from Figure 2.4. At time t, the working set is the set of cache lines that will be accessed on or after time t; the cache footprint is the set of cache lines that have been accessed prior to time t. Both definitions are applied with regard to individual jobs.

The second invocation illustrates cache-line reuse. Here, the working set includes only two cache lines (c_1 and c_2), which are both accessed twice. After each cache line has been accessed once at time t_8, the working set equals the cache footprint, an ideal situation with regard to cache efficiency. The third invocation is a mixture of the prior two scenarios such that a subset of the cache footprint is part of the working set. Cache line c_3 is reused at time t_{13} after being brought into the cache at time t_{10}, but the other two cache lines of the cache footprint at time t_{13} are not useful. In general, the cache footprint closely corresponds to the working set (after an initial warm-up phase) if a job exhibits high spatial locality. $\Diamond$

The working set and cache footprint determine the impact of preemptions. The cache footprint of a job that is preempted is likely to be evicted while it is not scheduled. If its working set at the time of the preemption closely matched its cache footprint, then it is penalized by additional cache misses when it resumes execution. That is, immediately after a preemption, a job does not benefit from its spatial and temporal locality since the cache state was disturbed. In contrast, a job with a disjoint working set and cache footprint is not impacted by a preemption at all (*e.g.*, the first job

in Figure 2.4). However, such cases of inherently low cache efficiency are rare in well-engineered systems.

In the context of this dissertation, the distinction between working set and cache footprint is less important since we focus on *worst-case* per-job memory use, and because the time of preemption is unknown in general. In the worst case, preempting jobs create maximal cache footprints and all of a preempted job's working set is evicted. We therefore make the simplifying assumption that the working set encompasses all cache lines accessed by a job over the course of its execution. A job's maximum cache footprint is thus its working set (assuming it fits into the cache by itself). A characteristic measure of a job's working set is its size, denoted *working set size* (WSS).

Cache misses. If a job references a cache line that cannot be found in a level-X cache, then it suffers a *level-X cache miss*. There are four primary causes for cache misses. *Compulsory misses* are triggered the first time a cache line is referenced, *i.e.*, if the referenced cache line was not yet part of the cache footprint. *Capacity misses* result if the WSS of the job exceeds the size of the cache, that is, if needed cache lines were evicted to make room for other data. Further, in direct mapped and set associative caches, *conflict misses* arise if cache lines were evicted to accommodate mapping constraints of other cache lines. Finally, *coherency misses* occur when another processor evicted required cache lines to ensure data consistency.

Ideally, a job should incur few cache misses besides compulsory misses. However, since caches are finite, this is not always the case. Jobs that incur frequent level-X capacity and conflict misses even when executing in isolation are said to be *thrashing* the level-X cache. *Cache affinity* describes the effect that a job's overall miss rate tends to decrease with increasing execution time (unless it is thrashing)—after an initial burst of compulsory misses, most of the working set has been brought into the cache and the rate of compulsory misses decreases. A job's memory references are *cache-warm* after cache affinity has been established; conversely, *cache-cold* references imply a lack of cache affinity.

In a multiprocessor system, a job that does not thrash by itself may still incur frequent cache misses due to the activity of jobs on other processors. In particular, a shared cache must exceed the combined cache footprint of all jobs accessing it, otherwise, frequent capacity and conflict misses may arise due to *cache interference*. Cache consistency can also become a major source of overhead

if processors frequently read and write memory locations that reside in the same cache line. *Cache line bouncing* describes the effect when two or more processors repeatedly evict the same cache line(s) from each other's caches. Cache line bouncing can also be the result of *false sharing*, wherein processors access different, but neighboring memory locations that map to the same cache line.

TLB misses occur either the first time a memory page is accessed (together with a compulsory cache miss), when the TLB entry was displaced (this corresponds to a capacity miss), or when the TLB was *flushed*. TLB flushes are required whenever the address space of a task is modified (*e.g.*, when it maps or unmaps physical addresses and memory-mapped devices). The TLB may also be flushed when the address space is changed on a context switch.

Cache-miss avoidance. Cache misses slow down a task's execution since the processor stalls while it must wait for instructions and data to be fetched from main memory. On a modern, fast processor, servicing a cache miss can take more than 100 processor cycles (Baer, 2010). Avoiding cache misses as much as possible is thus crucial to achieving high efficiency. In real-time systems, cache misses should further be avoided since they are difficult to predict. We briefly discuss some relevant techniques for improving cache hit rates.

In practice, cache interference and cache line bouncing due to false sharing are a significant concern for OS developers. A common solution to reduce cache line bouncing is to include padding bytes in data structures such that their size becomes a multiple of the cache line size. False sharing is impossible for such data structures if they are consistently allocated at cache line boundaries. The only way to avoid cache line bouncing due to "true sharing," *i.e.*, due to data structures that are accessed by multiple processors, is to minimize the frequency of such accesses.

Since the extent of cache interference depends on the set of jobs executing on the processors that share a cache, cache interference can also be reduced using cache-aware scheduling approaches (Calandrino, 2009; Guan *et al.*, 2009). Such approaches are complimentary to this dissertation, in the sense that our overhead-aware evaluation methodology (Chapter 4) can be used to compare scheduling algorithms that aim to reduce cache interference.

As the name implies, compulsory misses are impossible to avoid in general. However, it is possible to exploit spatial locality to avoid some compulsory misses by *predicting* future memory references. Processors that perform *cache prefetching* monitor the sequence of memory references

and proactively transfer cache lines that are likely to be accessed in the near future into a cache. Cache prefetching can be effective from a throughput point of view. However, in real-time systems, prefetching can be detrimental since it might displace part of a job's working set with unrelated, possibly useless data when the prefetching logic mis-predicts future references.

TLB misses can also cause significant slowdowns. To avoid TLB flushes on context switches, some processors support *tagged TLBs*, wherein each TLB entry is marked with an *address space number* (ASN). ASNs are unique identifiers that correspond to address spaces managed by the OS. On a context switch, instead of flushing the TLB, the OS only updates a "current ASN" register to a value corresponding to the scheduled process. TLB entries belonging to the current process can then be differentiated by the MMU from stale entries based on the differing ASN tag. The benefit of a tagged TLB is that it reduces the average cost of context switches: if a process resumes execution shortly after it was preempted, then most of its TLB entries may still be present. However, tagged TLBs are still subject to capacity limitations, and ASNs must be invalidated when address spaces are modified (which is equivalent to a process-specific TLB flush). Consequently, tagged TLBs have only little impact on the worst-case cost of a context switch.

Cache partitioning. Recall that preemptions cause the preempted job to incur additional cache misses when it resumes execution if the cache footprint of the preempting job caused parts of the working set of the preempted job to be evicted. It is difficult to predict such cache misses and those that arise due to cache interference since they depend on the identity of the interfering or preempting task (of which there can be many), and on the point in time at which the preemption or interference takes place. In effect, such misses cannot be accurately anticipated by analyzing individual tasks in isolation. To limit the impact of such cache effects, several hardware- and software-based isolation techniques have been proposed.

At the hardware level, Kirk (1989) proposed *cache partitioning* to reserve parts of a cache for specific processes, thereby limiting cache interference to the non-exclusive parts. As a variant of full partitioning, some architectures allow cache contents to be *locked* such that they will not be evicted. An alternative to caches are *scratchpad memories* (also called *programmable caches*), which function similarly to a cache in that they speed up memory references (Banakar *et al.*, 2002). The benefit of scratchpads is that their contents are completely software-controlled and thus not subject

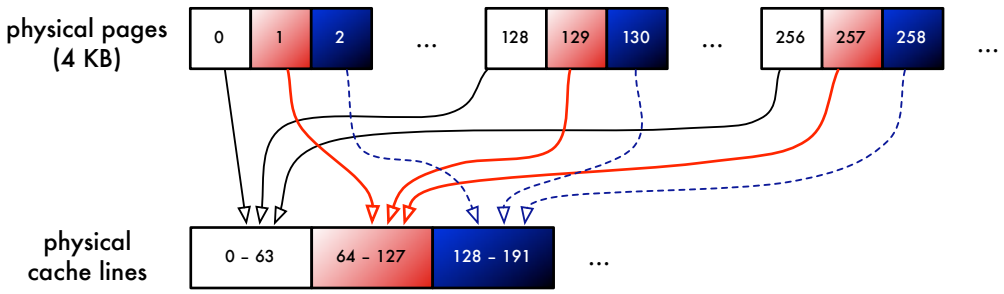


Figure 2.5: Illustration of page coloring. Physical pages are of the same page color if their contents map to the same physical cache lines. In this example, there are 128 distinct page colors (see Example 2.2).

to unexpected interference. These hardware-based isolation techniques have been employed in a number of uniprocessor designs targeted at embedded systems; however, they are typically not available in current mass-market multicore designs.

Page coloring. Cache partitioning can also be realized at the software level based on *page coloring*. The "color" of a memory page describes to which set of cache locations its addresses will be mapped (assuming that caches are direct mapped or set associative, which they are in practice).

Example 2.2. Suppose a system with 1 GB of memory has a 512 KB direct-mapped L2 cache with a cache-line size of 64 bytes. This implies there are 512 KB / 64 bytes = 8,192 physical cache lines in the L2 cache. Assuming a page size of 4 KB, there are 4 KB / 64 bytes = 64 cache lines per page and a total of 1 GB / 4 KB = 262,144 physical pages. Let p denote a physical page number, and let c denote a physical cache-line index (both zero-based, *i.e.*, $0 \leq p \leq 262{,}143$ and $0 \leq c \leq 8{,}191$).

As illustrated in Figure 2.5, in a direct-mapped cache, the contents of the first page ($p = 0$) map to the first 64 physical cache lines $c \in \{0, \ldots, 63\}$. Similarly, the contents of the second page ($p = 1$) are mapped onto the cache lines $c \in \{64, \ldots, 127\}$. Accesses to data stored in the first two physical pages can thus not interfere with each other. However, since the cache is much smaller than the main memory, the direct mapping wraps around after 8,192 / 64 = 128 pages. That is, the contents of page $p = 128$ are also mapped to $c \in \{0, \ldots, 63\}$, and thus can conflict with the contents of page $p = 0$. In general, the contents of a page p are mapped to the cache lines $c \in \{(p \bmod 128) \cdot 64, \ldots, (p \bmod 128) \cdot 64 + 63\}$. This implies that two pages p_1 and p_2 conflict if and only if $p_1 \bmod 128 = p_2 \bmod 128$. Hence, in this example, there are 128 disjoint sets of pages—named colors—such that pages within each set conflict with each other, but not with any

pages from other sets. If the cache in this example were a 2-way set-associative cache (*i.e.*, if each cache line could be mapped to two distinct locations in the cache), then there would be only 64 distinct colors. ◊

Page coloring can be used to implement cache partitioning by dedicating a page color to each real-time task, *i.e.*, by letting each real-time task allocate only pages of a reserved, task-specific color. This can be enforced either statically at the compiler level (Müller, 1995) or dynamically at the OS level in the virtual memory subsystem (Liedtke *et al.*, 1997). However, either approach limits the number of real-time tasks that may run concurrently to the number of distinct page colors. Further, each real-time task uses only a subset of the cache, which can lead to thrashing. This can be avoided by allocating multiple colors to a cache-intensive task; however, this further constrains the total number of real-time tasks and may lead to a considerable waste of memory. In a system with a hierarchy of caches, there is a tradeoff between the number of available colors and the degree of task isolation. When using L1-based colors, real-time tasks can be isolated at all cache levels,[1] but there are only few colors available due to the typically small size of L1 caches. When using L2- or L3-based colors, the number of available colors is much larger, but interference is possible at higher cache levels. Nonetheless, page coloring has the unique advantage that it can provide a high degree of isolation on commodity hardware.

Xeon L7455. Figure 2.6 depicts the cache hierarchy of one six-core chip of the experimental platform used in Chapters 4 and 7. Each core has eight-way set associative private L1 data and L1 instruction caches of size 32 KB each. Two cores each share a unified, twelve-way set associative L2 cache of size 3072 KB. Finally, all cores on the chip share a 12288 KB unified L3 cache, which is also twelve-way set associative. The cache line size is 64 bytes. Cache interference is possible via the L2 and L3 caches. Jobs with a private address space and a WSS of at most 32 KB are not subject to cache interference once their working set has been brought into L1 cache.

The system contains four such six-core chips. As configured, the system does not use cache prefetching. The system also does not contain scratchpad memory or hardware-based cache partitioning facilities, and does not support tagged TLBs. On the software level, Linux unfortunately does not

[1] Strictly speaking, this may not be the case if the L1 cache and lower caches differ in line size.

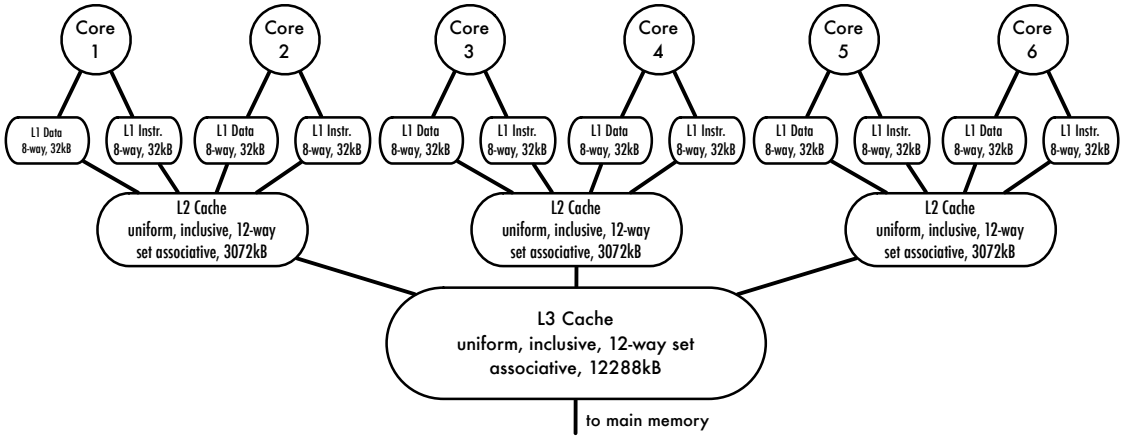


Figure 2.6: Illustration of the cache hierarchy of one chip of the experimental platform used in Chapters 4 and 7. The system contains four chips, for a total of 24 cores.

support strict separation of memory allocations based on page coloring, and such support is also not available in common Linux compilers.

2.1.3 Interrupts

Interrupts notify processors of asynchronous events and may occur between (almost) any two instructions. If an interrupt is detected, the processor temporarily pauses the execution of the currently scheduled task and executes a designated *interrupt service routine* (ISR) instead. This can cause the interrupted task to incur undesirable delays that must be accounted for when determining whether a task is schedulable, as discussed in detail in Chapter 3.

Most interrupts are *maskable*, *i.e.*, the processor can be instructed by the OS to delay the invocation of ISRs until interrupts are unmasked again. However, *non-maskable interrupts* (NMIs), which can be used for "watch dog" functionality to detect system hangs, cannot be suppressed by the OS. In multiprocessor systems, some interrupts may further be local to a specific processor, whereas others may be serviced by multiple or all processors.

Interrupt categories. Interrupts can be broadly categorized into four classes: *device interrupts* (DIs), *timer interrupts* (TIs), *cycle-stealing interrupts* (CSIs), and *inter-processor interrupts* (IPIs). We briefly discuss the purpose of each next.

DIs are triggered by hardware devices when a timely reaction by the OS is required or to avoid costly "polling" (see below). In real-time systems, DIs may cause jobs to be released, *e.g.*, a sensor

may trigger a DI to indicate the availability of newly-acquired data, in response to which a job is released to process said data.

TIs are used by the OS to initiate some action in the future. For example, TIs are used to support high-resolution delays ("sleeping") in Linux. In LITMUS$^{\mathrm{RT}}$, they are also used for periodic job releases and to enforce execution-time budgets. In networking protocols, TIs are commonly employed to trigger timeouts and packet re-transmissions.

CSIs are an artifact of the way that modern hardware architectures are commonly implemented and differ from the other categories in that they are neither controlled nor handled by the OS. CSIs are used to "steal" processing time for use by some component that is—from the point of view of the OS—hardware, but that is implemented as a combination of hardware and software (so called "firmware") and that lacks its own processor. CSIs are intended to be transparent from a logical correctness point of view, but of course do affect temporal correctness. They are usually non-maskable and the OS is generally unaware if and when CSIs occur. A well-known example for the use of CSIs is the system management mode (SMM) in Intel's x86 architecture (Intel Corporation, 2008a,c): when a system management interrupt (SMI) occurs, the system switches into the SMM to execute ISRs stored in firmware. For example, on some chip sets the SMM is entered to control the speed of fans for cooling purposes. CSIs can also occur in architectures in which raw hardware access is mediated by a hypervisor. For example, this is the case in Sony's PlayStation 3 (Kurzak *et al*., 2008) and Sun's sun4v architecture (Saulsbury, 2008). In such systems, the hypervisor may become active at any time to handle interrupts or perform services for devices "invisible" to the OS. CSIs are especially problematic if the code that is being executed is unknown—for example, a CSI could flush instruction and data caches and thereby unexpectedly increase task execution costs.

In contrast to DIs, TIs, and CSIs, the final category considered, IPIs, are specific to multiprocessor systems. IPIs are used to synchronize state changes across processors and are generated by the OS. For example, the modification of an address space on one processor can require software-initiated TLB flushes on multiple processors (Intel Corporation, 2008c,d). IPIs are also commonly used to cause a remote processor to reschedule.

Xeon L7455. Figure 2.7 shows how interrupts are handled in our x86 platform used for the case studies presented in Chapters 4 and 7. Modern x86 systems use the *advanced programmable interrupt*

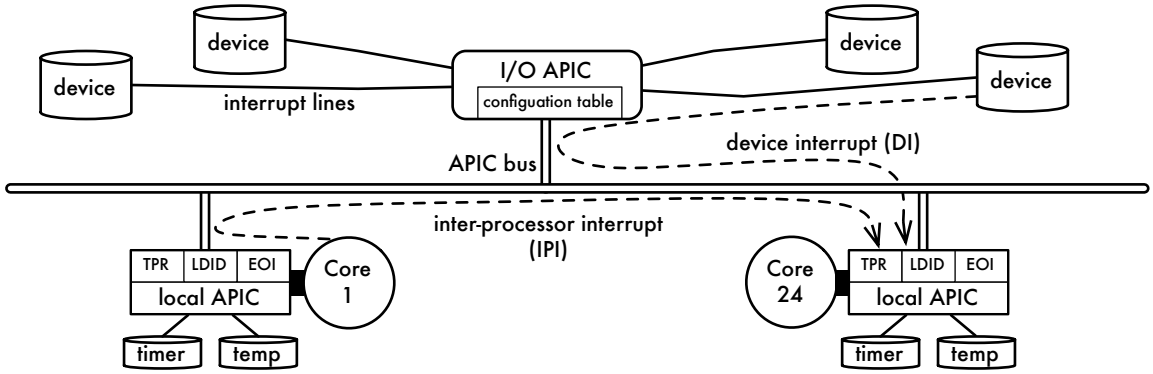


Figure 2.7: Illustration of the I/O APIC and local APICs in the multiprocessor platform underlying the case studies discussed in Chapters 4 and 7. Devices are connected to the I/O APIC with physical interrupt lines. The I/O APIC polls the incoming interrupt lines and dispatches an interrupt to a local APIC when a signal is detected. The recipient is determined based on the interrupt-line-specific target LDID and the current priorities in the TPR registers. Local APICs are also used to send IPIs among processors and to handle per-processor devices (such as the depicted timer and temperature sensors).

controller (APIC) to manage interrupts (Intel Corporation, 1996, 2008c). The name derives from the legacy *programmable interrupt controller* (PIC), which was a single chip that was used in early x86-based designs. In contrast, the APIC is actually a collection of chips that collaborate to deliver interrupts.

Devices interface with the *I/O APIC* that handles interrupt distribution by sending DIs to one of the cores (more on DI distribution below). Each core has a *local APIC* that performs the actual interrupt delivery when it receives a DI from the I/O APIC. Local APICs are also responsible for sending IPIs to each other, and also directly interface with processor-local interrupt sources (such as a timer, performance counters, and thermal sensors). Conceptually, the local APICs and the I/O APIC are connected by a dedicated *APIC bus* (Intel Corporation, 1996); however, current systems actually use the main system bus to transport APIC messages. A processor can temporarily disable the delivery of interrupts with the cli instruction. This does not, however, stop the local APIC from accepting interrupts, which will be delivered once local interrupts are re-enabled by the OS with the sti instruction.

There are 15 distinct interrupt priorities. A local APIC accepts only higher-priority interrupts while an ISR is executing. If a higher-priority interrupt is received while executing an ISR for a lower-priority interrupt, then the higher-priority interrupt is delivered right away, interrupting the

lower-priority ISR (unless interrupt delivery has been disabled—see above). The OS must signal the end of an ISR to the local APIC by writing to a special *end-of-interrupt* (EOI) register. Each local APIC has a *task priority register* (TPR) and will only accept interrupts that exceed the priority of the value in the TPR. This can be used by the OS to mark the local APIC as ineligible to process low-priority interrupts.

The I/O APIC polls 24 physical interrupt lines (each of which is connected to one or more devices) for interrupt signals. For each interrupt line, the OS can configure how a detected interrupt will be relayed to one or more local APICs. In particular, each interrupt can be sent to either a specific local APIC (and hence processor) or to a *logical destination ID* (LDID). Each local APIC belongs to exactly one LDID and multiple local APICs may share the same LDID; this allows interrupts to be distributed among several processors on a line-by-line basis. If there are multiple possible recipients for a logical destination ID, a TPR-aware arbitration protocol is used to dispatch the interrupt to the lowest-priority recipient processor, *i.e.*, to the processor with the lowest priority value stored in its TPR register (the OS should avoid priority ties). The I/O APIC also allows each interrupt line to be individually masked, and to specify the interrupt priority for each line.

Taken together, these options provide the OS with considerable flexibility to determine which processor(s) will handle interrupts.

2.1.4 Timers and Clocks

Precise time management is essential to the correct functioning of an RTOS. Most computing platforms therefore include one or several devices that can serve as a timer and/or clock.

A *clock* is a device that measures the progress of physical time. Physical time is sometimes referred to as "monotonic time," as opposed to "wall-clock time," which is subject to man-made concepts such as time zones, daylight saving time, and leap seconds. Clocks rarely work in units of seconds. Instead, they typically count the occurrence of some physical event that occurs regularly with a known frequency (such as the oscillation of a crystal)—that is, a clock "ticks" at a given frequency and increments a counter on every "clock tick." The OS can read the clock to obtain the number of events (or "ticks") that have occurred since system startup (or since the last time the counter overflowed). This allows the OS to determine the length of an activity (such as a task executing) by taking a timestamp before the activity commences and after it completes.

A *timer* is a device that can be programmed to generate an interrupt at a future point in time. A timer does not necessarily have a notion of current time (*i.e.*, not every timer is also a clock). Instead, knowledge of the remaining time until the interrupt will be generated is sufficient for correct operation. Similar to clocks, frequency-driven timers can be implemented by decrementing a counter every time a physical event occurs until the counter reaches zero, which is when the interrupt fires. Timers generally operate in one of two modes. *One-shot timers* generate a single interrupt and then remain inactive until reprogrammed. In contrast, *periodic timers* automatically reset and generate interrupts in a regular fashion until explicitly stopped.

Periodic timers can be, and traditionally have been, used to implement clocks: whenever the periodic timer interrupt (called the *system tick*) occurs, the value of a "current time" variable is incremented. In Linux parlance, this notion of current time is referred to as "jiffies." A physical clock device is thus not necessarily required. However, such emulation of a clock comes with a tradeoff in precision since the time appears to "stand still" between timer interrupts. In general, the precision of a clock depends on its resolution and accuracy.

The *resolution* of a clock is the smallest difference that can be observed between consecutive readings, *i.e.*, the granularity of time measured by the clock. For example, suppose a periodic timer with a period of one second is used to implement the system clock. The resulting resolution is one second, *i.e.*, it is impossible to correctly tell apart intervals that differ by less than one second.

The *accuracy* of a clock determines its error, *i.e.*, how close the reported time is to the actual physical time. For example, suppose the OS would query the atomic clock of the National Institute of Standards and Technology (NIST) over the Internet. NIST guarantees the resolution to not exceed 200 picoseconds, *i.e.*, the NIST's clock will report distinct readings to queries that are separated by at least 200 picoseconds. However, the accuracy as perceived by the OS would only be in the range of tenths to hundreds of milliseconds due to unpredictable latencies on the Internet, *i.e.*, by the time that the OS has obtained a clock reading the result is no longer accurate.[2] The accuracy of a practical clock, *i.e.*, one that does not artificially delay observations until the next "tick," is limited by its resolution.

[2]Multi-round clock synchronization protocols for real-time systems exist that can greatly reduce the impact of uncertain transmission times (Kopetz and Ochsenreiter, 1987; IEEE, 2008a), but such protocols do not apply here since we are concerned with a single observation.

Clock drift describes the effect that a clock's accuracy may slowly decrease over time. For example, slight changes in tick frequency may cause a clock to drift. Dealing with clock drift in practice can be a non-trivial engineering challenge, and especially so for OSs such as Linux that must work on a wide range of commodity platforms with components and clocks of varying quality and correctness. However, such techniques are beyond the scope of this dissertation; we make the simplifying assumption that clock drift is negligible across short time intervals. Platforms for which this does not hold are of questionable utility in the context of real-time systems.

Resolution and accuracy similarly apply to timers, where the resolution determines the granularity of points in time that can be programmed, and accuracy is a measure of how closely the interrupt occurs to the requested point in time.

Another criterion is the overhead involved in reading a clock or programming a timer. For example, the *programmable interval timer* (PIT), which has traditionally been used in x86-based systems, is theoretically capable of operating as a one-shot timer, but is only used as a periodic timer in practice due to the high cost of re-programming the timer. The underlying reason is that writing to the PIT's memory-mapped registers can cause a processor to stall for many cycles while it waits for the write transaction that was issued to the memory bus to complete. Ideally, each processor should have low-overhead access to a timer and clock with high resolution and high accuracy.

Xeon L7455. In our experimental platform, each processor has access to three clocks and two timers (as illustrated in Figure 2.8 and summarized in Table 2.2).

The system has a *high-precision event timer* (HPET) that is connected to the I/O APIC. The HPET is both a clock and a timer. It contains a circuit that oscillates with a frequency of approximately 14.3 MHz. On each oscillation, a 64 bit counter, which can be read by the OS, is incremented by one. The HPET also implements three independent one-shot timers with three *comparator registers*. The OS can program a value in each register; an interrupt is generated when the incrementing counter equals the comparator value. As discussed above, the I/O APIC can be configured to route the HPET interrupt to a specific set of processors. The frequency of the oscillator limits the HPET to a resolution of about 69.84 ns. (The HPET standard requires a frequency of at least 10 MHz, which corresponds to a minimum resolution of 100 ns.) The HPET is an off-chip device, *i.e.*, it is not part

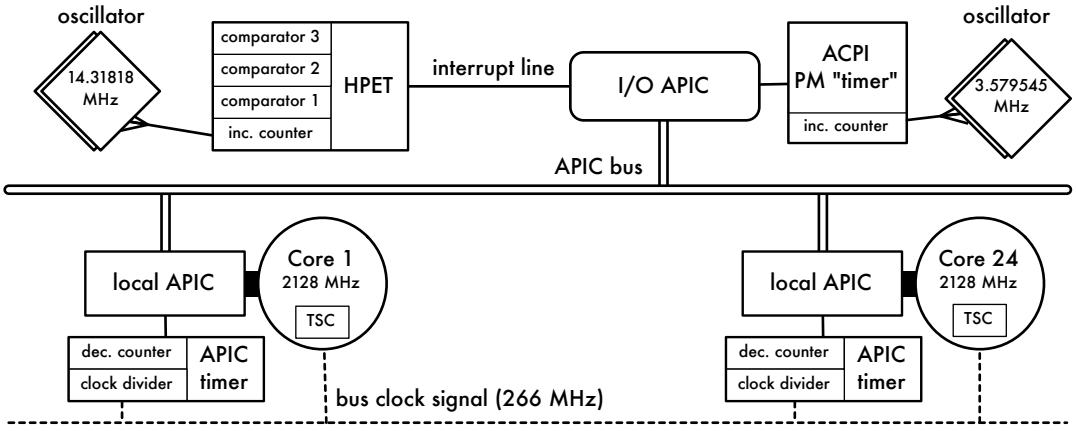


Figure 2.8: Illustration of the timers and clocks available in the multiprocessor platform underlying the case studies discussed in Chapters 4 and 7. Each processor has access to three clocks (the HPET, the ACPI PM timer, and the per-processor TSC) and two timers (the HPET and the per-processor local APIC timer). The local APIC timer and the TSC are preferable to the other alternatives due to their superior resolution and low-overhead, on-chip access.

of the processor die. The accuracy of the HPET is hence impacted by the time it takes the I/O APIC to deliver an interrupt to a local APIC.

Each core's local APIC also contains a one-shot timer that is driven by the main bus clock signal. The local APIC timer consists of a counter (initialized by the OS) that decrements by one every 2^x bus clock cycles, where $x \in \{0, \ldots, 7\}$. The OS configures x by means of a clock divider register. When the counter reaches zero, a local interrupt is generated. Since the local APIC is an on-chip device that is tightly integrated into the processor core, it incurs only little programming overhead. In our system, the system bus clock frequency is 266 MHz, which implies a minimum resolution of about 3.76 ns (for $x = 0$). It is also a fairly accurate timer since interrupt delivery is local. The local APIC has no clock functionality.

Besides the HPET, the system has two additional clocks without timer functionality. The *Advanced Configuration and Power Interface* (ACPI) standard (Hewlett-Packard *et al.*, 2010) requires each system to have an ACPI *power management timer* (PM timer), which—despite its name—is a clock without timer functionality. Similarly to the HPET, it is an off-chip device and consists of an oscillator with a fixed frequency and a 32-bit counter that is incremented on every oscillation. The ACPI PM timer generates an interrupt whenever the counter overflows. The ACPI PM oscillator's frequency of about 3.58 MHz implies a minimum resolution of approximately 279.37 ns.

Device	Type	Location	Frequency (MHz)	Resolution (ns)
HPET	one-shot timer	off chip	14.32	69.84
Local APIC	one-shot timer	on chip	266.00	3.76
HPET	clock	off chip	14.32	69.84
ACPI PM timer	clock	off chip	3.58	279.37
TSC	clock	on chip	2128.00	0.47

Table 2.2: Summary of the timers and clocks available in the multiprocessor platform underlying the case studies discussed in Chapters 4 and 7. Linux uses on-chip devices when available.

The highest-resolution clock in the system is the *timestamp counter* (TSC), which is a per-processor 64 bit register. The TSC is incremented with each processor cycle, which, at 2128 MHz (eight times the bus clock), implies a minimum resolution of only about 0.47 ns. Since the TSC is a register that is part of the processor, TSC accesses incur only negligible overhead. However, the TSC is subject to some accuracy limitations when conducting micro benchmarks (*i.e.*, when measuring the execution time of short code segments) since instruction reordering could move the dependency-less TSC read ahead of other operations. This can be avoided by issuing serializing instructions prior to the TSC read (Paoloni, 2010). The TSC can also be affected by processor frequency changes and TSC values may not be comparable across processors (*e.g.*, the processors may have been initialized at different times).

The available clocks and timers are summarized in Table 2.2. Accesses to on-chip device registers generally incur lower overheads than accesses to memory-mapped registers of off-chip devices because off-chip accesses are subject to memory bus arbitration. Similarly, interrupts from off-chip devices must be dispatched by the I/O APIC, whereas on-chip devices are typically directly connected to a local APIC.

Linux requires just one clock and one timer to function properly. It selects one device of each class during bootup and leaves the others unused. On our system, Linux chooses the local APIC timers and the TSC due to their high resolution and low overheads.

2.2 Real-Time Task Model and Constraints

The defining characteristic of real-time systems is that task execution is subject to temporal constraints. Additionally, many real-time tasks are recurrent, that is, they do not terminate during normal operation

of the system. A recurrent task model that allows *a priori* validation of temporal constraints should thus be used in the design and implementation of such real-time systems. LITMUS$^{\mathrm{RT}}$ is based on the classic sporadic task model (Mok, 1983), because it is analytically sound, well-studied, and yet simple and flexible enough to be implemented as an interface in an RTOS.

Under the *sporadic task model* (Mok, 1983), a real-time workload consists of a set of n sequential tasks $\tau = \{T_1, \ldots, T_n\}$. Each task T_i is repeatedly invoked by asynchronous, external events such as device interrupts or expiring timers. When T_i is invoked, it *releases* a *job* to process the triggering event. The j^{th} job of task T_i is denoted $J_{i,j}$, where $j \geq 1$. We omit the job index in cases where the identity of a job is irrelevant and use J_i to denote an arbitrary job of T_i. In the following, we provide a precise definition of the sporadic task model because it is used as a foundation for formal analysis throughout this dissertation. Our notation for sporadic tasks is summarized in Table 2.3 and illustrated in Figure 2.9.

Tasks. Each task T_i is characterized by three integral[3] parameters (e_i, p_i, d_i): its per-job *maximum execution requirement* e_i, where $e_i > 0$; its *minimum job inter-arrival separation* p_i, where $p_i \geq e_i$; and its *relative deadline* d_i, where $d_i \geq e_i$. The parameters have the following interpretation (which is formalized below): T_i releases jobs at least p_i time units apart, each job executes for at most e_i time units, and each job should finish no more than d_i time units after its release. The per-job execution requirement is naturally machine-specific, that is, e_i depends on the computational speed of the underlying hardware platform. In contrast, p_i and d_i are platform-independent since they express environmental constraints. For historical reasons, p_i is also referred to as a task's *period*.

Jobs. A job $J_{i,j}$ becomes available for execution at its *release* or *arrival time* $a_{i,j}$, where $a_{i,j} \geq 0$. Job releases are rate-limited by the task period such that $a_{i,j+1} \geq a_{i,j} + p_i$. Each $J_{i,j}$ requires at most e_i units of processor time to process the event that caused the task to be invoked, and *completes* or *finishes* thereafter at time $f_{i,j}$ (obviously, $f_{i,j} \geq a_{i,j}$). $J_{i,j}$ is said to be *pending* from its release until it completes. Recall that tasks are sequential: if $J_{i,j+1}$ is released while $J_{i,j}$ is still pending (*i.e.*, if $f_{i,j} > a_{i,j+1}$), then $J_{i,j+1}$ can only be scheduled after $J_{i,j}$ completes. $J_{i,j}$'s *response time* $r_{i,j}$ describes how long $J_{i,j}$ remained pending; formally, $r_{i,j} = f_{i,j} - a_{i,j}$. Across all jobs, T_i's *maximum response time* r_i is defined as $r_i = \max_j\{r_{i,j}\}$. Since tasks release a potentially infinite

[3] In LITMUS$^{\mathrm{RT}}$, task parameters are expressed in nanoseconds.

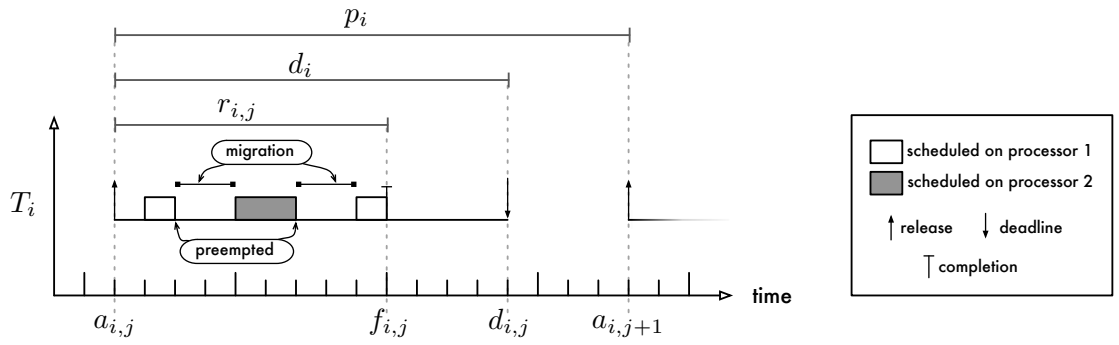


Figure 2.9: Illustration of a constrained-deadline sporadic task. The job $J_{i,j}$ is preempted twice by higher-priority jobs (which are not shown in this example). As a result, $J_{i,j}$ incurs two migrations: it first migrates from processor 1 to processor 2, and then again back to processor 1. See Table 2.3 for a summary of our notation.

number of jobs, r_i may not be well-defined for all tasks; for example, if the system is overloaded, $r_{i,j}$ could be monotonically increasing with each $J_{i,j}$.

Mok (1983) introduced the sporadic task model as a generalization of the earlier *periodic task model* (Liu and Layland, 1973). A *periodic task* releases jobs regularly on every period multiple, *i.e.*, $a_{i,j} = a_{i,1} + (j-1) \cdot p_i$ for every $J_{i,j}$ where $j > 1$. The sporadic task model encompasses periodic task behavior, but also allows tasks to experience *inter-arrival delays*, that is, the interpretation of the period parameter is relaxed to represent a lower bound on job separation (instead of an exact spacing between jobs). This allows tasks to become inactive in the absence of triggering events.

Deadlines. The relative deadline parameter d_i determines the range of acceptable response times. A job $J_{i,j}$ should complete by its *absolute deadline* $d_{i,j} = a_{i,j} + d_i$, and is *tardy* if it completes later (*i.e.*, if $f_{i,j} > d_{i,j}$). A job's *tardiness* is the extent of the deadline miss; formally, $J_{i,j}$'s tardiness is given by $\max(0, f_{i,j} - d_{i,j})$. A tardy job does not affect the release times of subsequent jobs of the same task, but may delay their execution since tasks are sequential.

Task systems are categorized by the range of allowed relative deadlines. The most-restrictive case is *implicit deadlines*: a set of tasks τ has implicit deadlines if $d_i = p_i$ for each $T_i \in \tau$. A more general case is *constrained deadlines*, where $d_i \leq p_i$ for each $T_i \in \tau$. Otherwise, if a task set has neither implicit nor constrained deadlines, then it is said to have *arbitrary deadlines*. The type of deadline constraint has a large impact on schedulability analysis (see Section 2.2.3 below). However,

Notation	Interpretation	Constraint / Definition
τ	A task set.	$\tau = \{T_1, \ldots, T_n\}$
T_i	The i^{th} sporadic task.	$1 \leq i \leq n$
$J_{i,j}$	The j^{th} job of task T_i.	$j \geq 1$
J_i	An arbitrary job of T_i.	
e_i	T_i's per-job execution requirement.	$e_i > 0$
p_i	T_i's period; minimum separation between jobs.	$p_i \geq e_i$
d_i	T_i's relative deadline.	$d_i \geq e_i$
u_i	T_i's utilization.	$u_i = e_i/p_i$
δ_i	T_i's density.	$\delta_i = e_i/\min(d_i, p_i)$
$a_{i,j}$	$J_{i,j}$'s release time.	$a_{i,j} \geq a_{i,j-1} + p_i$
$d_{i,j}$	$J_{i,j}$'s absolute deadline.	$d_{i,j} = a_{i,j} + d_i$
$f_{i,j}$	$J_{i,j}$'s completion time.	$f_{i,j} \geq a_{i,j}$
$r_{i,j}$	$J_{i,j}$'s response time.	$r_{i,j} = f_{i,j} - a_{i,j}$
r_i	T_i's maximum response time.	$r_i = \max_j\{r_{i,j}\}$

Table 2.3: Summary of the sporadic task model's constraints and our notation.

implicit, constrained, and arbitrary deadlines are largely equivalent from an implementation point of view. In this dissertation, we assume implicit deadlines unless noted otherwise.

Processor demand. The execution requirement e_i limits *how much* processor time a single job of T_i will need in the worst case, and the relative deadline d_i determines *how soon* it will be needed. When considering longer intervals during which a task may release multiple jobs, it is useful to normalize a task's processor demand with respect to its period and deadline.

Task T_i's *utilization* $u_i = \frac{e_i}{p_i}$ is the fraction of one processor that T_i requires. T_i's utilization u_i is significant because it corresponds to the maximum rate of execution required for T_i to "keep up" with incoming events, *i.e.*, T_i may require up to e_i time units out of p_i time units. If T_i is allocated less than u_i processor capacity over long intervals, then tardiness may grow unboundedly, that is, jobs of T_i may miss their deadlines by ever increasing amounts due to overload. In the context of fair scheduling algorithms (see Section 2.3.3.3 below), a task's utilization is also called its *weight*.

Constrained-deadline tasks are subject to an additional rate constraint. If $d_i < p_i$, then a job $J_{i,j}$'s rate of execution has to exceed u_i during $[a_{i,j}, d_{i,j}]$ if $J_{i,j}$ is to meet its deadline. The increased "urgency" of a constrained-deadline task T_i is reflected by its *density* δ_i, which is task T_i's execution requirement normalized by its relative deadline. However, if $d_i \geq p_i$, then T_i's rate of execution is

less constrained by its relative deadline than its period. Therefore, density is formally defined as $\delta_i = \frac{e_i}{\min(d_i, p_i)}$, which ensures that $\delta_i \geq u_i$ in all cases.

Max, min, sum, and top. The concepts of utilization and density are also commonly applied to whole task sets. The *total utilization* of a task set τ is defined as

$$u_{\text{sum}}(\tau) \triangleq \sum_{T_i \in \tau} u_i.$$

As a notational convenience, we further let

$$u_{\max}(\tau) \triangleq \max_{T_i \in \tau}\{u_i\} \qquad \text{and} \qquad u_{\min}(\tau) \triangleq \min_{T_i \in \tau}\{u_i\}$$

denote the maximum and minimum utilizations of tasks in τ, respectively. Additionally, we define $u_{\text{top}}(\tau, k)$ to denote the sum of the $\min(k, n)$ largest utilizations of tasks in τ. The maximum and total utilizations are two special cases of $u_{\text{top}}(\tau, k)$ where $k = 1$ and $k = n$, *i.e.*, $u_{\text{top}}(\tau, 1) = u_{\max}(\tau)$ and $u_{\text{top}}(\tau, n) = u_{\text{sum}}(\tau)$.

We analogously define $\delta_{\max}(\tau)$, $\delta_{\min}(\tau)$, $\delta_{\text{sum}}(\tau)$, and $\delta_{\text{top}}(\tau, k)$ with respect to density, and $e_{\max}(\tau)$, $e_{\min}(\tau)$, $e_{\text{sum}}(\tau)$, and $e_{\text{top}}(\tau, k)$ with respect to the execution requirement of tasks in τ.

2.2.1 Temporal Correctness

As mentioned previously, the hallmark of a real-time system is that it must satisfy a temporal specification to be deemed correct. In the sporadic task model, timeliness requirements are expressed as deadline constraints—each job *should* complete by its absolute deadline. In this dissertation, we consider two notions of deadline-based temporal correctness. In a *hard real-time* (HRT), deadlines are strict and may not be violated. In contrast, some deadline misses are permissible in a *soft real-time* (SRT) system. The word "some" in the preceding sentence allows for considerable leeway, and SRT correctness has indeed been defined in many ways in prior work. In contrast, the idea of HRT correctness is less ambiguous and there exists a universally accepted definition (in the context of the sporadic task model). We consider HRT correctness first.

HRT correctness demands that all jobs complete by their absolute deadlines, which is equivalent to requiring that each task T_i's maximum response time be bounded by its relative deadline, *i.e.*,

$r_i \leq d_i$. A job's response time necessarily depends on the underlying scheduling algorithm, its actual processor requirement, and may also depend on the concrete release times and processor use of higher-priority jobs that prevent it from being scheduled. HRT correctness is thus defined with respect to a given scheduling algorithm and only for *legal invocation sequences*, that is, it applies only if all job releases conform to each task's p_i and e_i parameters.

Definition 2.1. A task set τ is *HRT schedulable* under a scheduling algorithm $\mathcal{A}$ if and only if, for each legal invocation sequence, $r_i \leq d_i$ for each $T_i \in \tau$ (when scheduled by $\mathcal{A}$ on a given platform).

While it would be convenient to use this definition for all real-time systems, ensuring that *all* deadlines will be met can result in inefficient resource use (as will be discussed in subsequent sections). It is thus desirable to use a weaker definition for systems that can tolerate deadline misses to "some" extent. In the context of multiprocessor systems, the *bounded-tardiness* interpretation of SRT correctness has received considerable recent attention (Srinivasan and Anderson, 2003; Devi and Anderson, 2005; Devi, 2006; Block, 2008; Calandrino, 2009; Leontyev, 2010). First applied to multiprocessor real-time systems by Srinivasan and Anderson (2003), the bounded-tardiness definition allows any job to be tardy provided that the maximum tardiness is bounded by a constant (with respect to job identity). That is, jobs may miss deadlines, but only by a known extent that does not increase over time.

Definition 2.2. A task set τ is *SRT schedulable* under a scheduling algorithm $\mathcal{A}$ if and only if there exists a constant B such that $r_i \leq d_i + B$ for each $T_i \in \tau$ for each legal invocation sequence (when scheduled by $\mathcal{A}$ on a given platform).

Note that SRT schedulability generalizes HRT schedulability: if τ is HRT schedulable under $\mathcal{A}$, then $B = 0$. For brevity, we use "schedulable" without the 'HRT" or "SRT" prefix when the appropriate interpretation of deadlines is clear from context or irrelevant.

We chose the bounded-tardiness interpretation of SRT correctness because it offers a compelling set of analytical and practical advantages. For one, bounded tardiness is analytically sound and supported by a well-studied theoretical framework for multiprocessors (Devi, 2006; Leontyev, 2010). In particular, the offered SRT guarantees are quantifiable *a priori* (*i.e.*, the tardiness bound B). This allows tardiness to be anticipated and compensated for during the design phase (*e.g.*, by determining appropriate buffer sizes or by checking that maximum tardiness falls within acceptable "engineering

margins"). From a practical point of view, bounded tardiness is convenient as an SRT model because it only affects the analysis, but not necessarily the implementation. The RTOS system call interface remains the same as for HRT tasks since no additional parameters are required (in contrast to some of the approaches discussed below). Further, the same scheduling algorithms that are used to support HRT workloads can be used to support SRT workloads, that is, no special-purpose SRT schedulers are required (see Section 2.3 below). This significantly simplifies the engineering aspects involved in building a multiprocessor RTOS such as LITMUS$^{\mathrm{RT}}$. In essence, the system itself remains unchanged and only the nature of the guarantees provided by it differs between HRT and SRT analysis.

2.2.2 Alternate Soft Real-Time Definitions

The term "soft real-time" tends to be used as a catch-all label that is applied to any approach that does not strictly qualify as "hard." Not surprisingly, the literature pertaining to "soft" real-time systems in one form or another is extensive. The following discussion is not intended as a comprehensive review. Rather, the goal is to explain the benefit of the adopted bounded-tardiness definition of SRT correctness, and to contrast it to the most closely related SRT approaches.

A metric commonly used in practice and systems-oriented work is the *deadline miss ratio*. Under this metric, the extent of an individual miss, *i.e.*, a job's tardiness, is irrelevant. Instead, this metric simply reflects the number of deadlines missed by a task (or a set of tasks) in relation to the total number of deadlines (*i.e.*, jobs) during some interval. For example, in experimental work on supporting SRT workloads in the presence of hardware multithreading, Jain *et al*. (2002) considered a deadline miss ratio of up to five percent as acceptable. The foremost advantage of the deadline miss ratio is that it can be easily measured during experiments and system testing. Further, it provides a simple-to-interpret indicator of "scheduling quality," as lower miss ratios are intuitively preferable to higher miss ratios. A major disadvantage is the lack of an analytic framework for predicting *worst-case* deadline miss ratios. As such, the deadline miss ratio metric is ill-suited for *a priori* analysis and for providing any kind of predictable worst-case performance guarantees. However, deadline miss ratios have proven useful as a "sensor" for system performance in feedback-control-based SRT scheduling approaches (Lu *et al*., 2000, 2002) and are also suitable for making stochastic guarantees (see, *e.g.*, (Manolache *et al*., 2004)).

Optional jobs. Allowing a non-zero deadline miss ratio implies that it is acceptable for a certain ratio of jobs to fail to complete on time (or at all). Several formalizations of this notion of "softness" have been proposed in work on uniprocessor real-time systems. Koren and Shasha (1995) introduced the *skip model* as a means to handle overload situations. Under the skip model, each task T_i has an additional *skip parameter* s_i, where $s_i \geq 2$. The skip parameter is the minimum distance between jobs that fail to finish by their deadline. Any job $J_{i,j}$ may be skipped altogether or aborted mid-execution provided that the previous $s_i - 1$ jobs completed on time, that is, if $J_{i,l}$ is the last job of T_i to have been skipped, then $J_{i,j}$ may be skipped only if $l + s_i \leq j$. This rule limits the long-term deadline miss ratio of T_i to $\frac{1}{s_i}$.

Hamdaoui and Ramanathan (1995) proposed the conceptually similar, but more general (m, k)-*firm model*, which requires that in *any* window of k consecutive jobs (of a task T_i) at least m jobs meet their deadline. The (m, k)-model is suitable to make *a priori* guarantees (Goossens, 2008; Li *et al*., 2004; Quan and Hu, 2000; Ramanathan, 1999; Hamdaoui and Ramanathan, 1997), albeit only when using schedulers that were especially designed for the (m, k)-model. In fact, finding the optimal set of jobs to skip is an NP-hard problem (Quan and Hu, 2000). Intuitively, the (m, k)-firm model limits a task's deadline miss ratio to $\frac{k-m}{k}$ in the long term; Calandrino (2009) explores the exact correspondence in more detail.

Very similar to the (m, k)-firm model is the *window-constrained model* (West and Poellabauer, 2000; West and Schwan, 1999), which associates two parameters (x, y) with each task, with the interpretation that at most x jobs out of a window of y jobs may fail to meet their deadline, just like in the (m, k)-firm model. However, in contrast, windows are non-overlapping in the window-constrained model, that is, the x-constraint applies to consecutive windows, whereas the m-constraint in the (m, k)-model applies to *any* window of k consecutive jobs. For example, suppose that $x = m = 9$ and $x = k = 10$ for a task T_i. Then it would be permissible for both $J_{i,10}$ and $J_{i,11}$ to be skipped in the window-constrained model (the first window includes $J_{i,1}$–$J_{i,10}$, the second window includes $J_{i,11}$–$J_{i,20}$), but not in the (m, k)-model (since for example the k consecutive jobs $J_{i,2}$–$J_{i,11}$ include both jobs). In the long run, the window-constrained model results in a worst-case deadline miss ratio of $\frac{x-y}{x}$, but it allows deadline misses to "cluster" at window boundaries. Interestingly, from a complexity point of view, the less-strict nature of the window-constrained model does not yield a simpler scheduling problem (Mok and Wang, 2001).

Constraints in the models mentioned so far apply on a per-job basis. In contrast, the *imprecise computation model* (Liu *et al.*, 1991; Aydin *et al.*, 2001) splits each job into a mandatory and an optional part. The mandatory part of each job must complete by its respective deadline, whereas the optional part may complete later, only partially, or not at all. The underlying idea is that some applications can degrade gracefully in the face of system overload by skipping parts of a computation that merely serves to refine an already obtained result (*e.g.*, additional filters in a video encoder). A scheduler for imprecise tasks will ensure that all mandatory parts complete and then execute as many optional parts as possible.

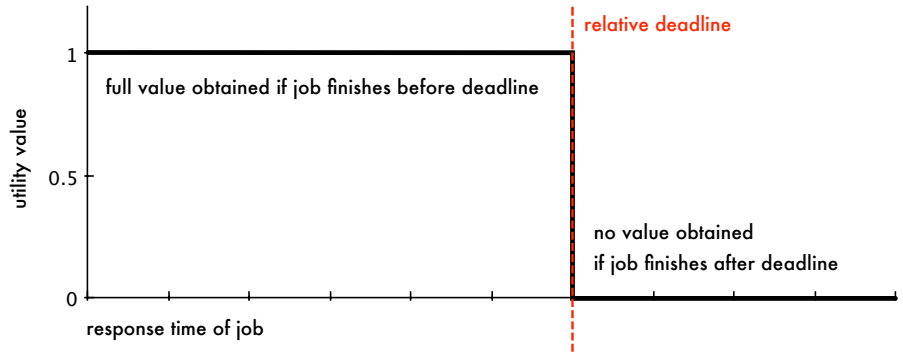
The common theme in the above approaches is that they formalize the notion that some jobs, or parts of each job, are "expendable" in an SRT system. The key benefit of using these models (instead of just relying on measured deadline miss ratios) is that they enable the system designer to make performance guarantees—they are analytically sound. However, in the context of this dissertation, they also share a number of key limitations. First, each mentioned model was only analyzed in the uniprocessor context. Second, they each require additional parameters (besides the core sporadic task parameters) to be known at runtime, which complicates the kernel-space/user-space interface and hence their realization in LITMUS$^{\mathrm{RT}}$. Further, model-specific scheduling approaches are required, *i.e.*, the models do not apply in a meaningful way to all multiprocessor real-time schedulers. Finally, and most importantly, it is very difficult to implement job skipping, or even outright job abortion, in a general way in an RTOS such as LITMUS$^{\mathrm{RT}}$. This is because application-specific "cleanup" code may be required to ensure that application state remains consistent. Aborting jobs in mid-execution is particularly tricky in the presence of lock-based resource sharing, which we consider in Chapters 5–7. In fact, to the best of our knowledge, no real-time locking protocol that supports the forcible revocation of resources (in an analytically sound way) has been published to date.

Bernat *et al.* (2001) further generalized the (m, k)-model by adding the ability to affix multiple window constraints to a single task and by enabling additional types of constraints (such as requiring that m *consecutive* jobs do not miss their deadlines). They also allow jobs to complete past their deadlines (as an alternative to being either skipped or aborted), thereby avoiding the need for "cleanup" support in the RTOS. This allows them to use a generic, POSIX-compliant fixed-priority scheduler that is unaware of (m, k)-constraints (see Section 2.3.1.1 below); however, their method for assigning priorities only applies to periodic (and not sporadic) tasks on a uniprocessor.

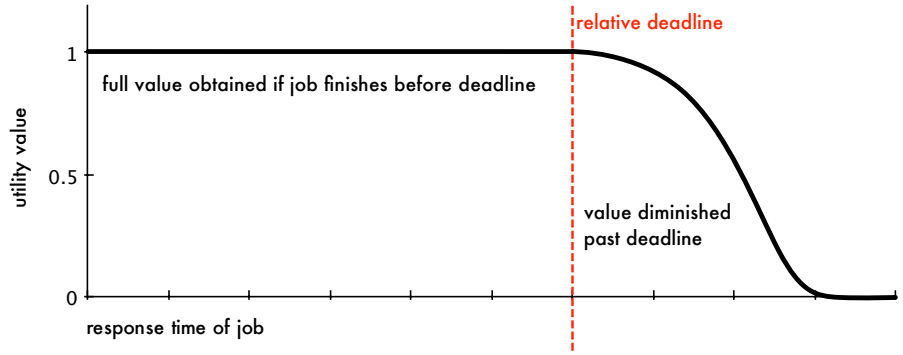
Job utility. Common to the bounded-tardiness SRT definition and the discussed job-skipping-based approaches is that they treat all deadline misses alike. That is, neither the identity of the late job nor the magnitude of its tardiness factor into the undesirability of a deadline miss. In practice, some applications are more sensitive to tardiness than others, and there may in fact exist a cutoff point (other than the deadline) at which the computed result becomes useless. To express such considerations, Jensen *et al*. (1985) developed a scheduling framework based on *time utility functions* that map a job's response time to a utility value (where 0 implies "useless"). Their model is very general and can express both HRT and SRT "valuations" in the sense that the result of an HRT job is useless after its deadline, whereas the result of an SRT job may gradually decline in value after its deadline. Two example value functions are illustrated in Figure 2.10. In this model, the goal of the scheduler is to maximize the accrued utility. To this end, the scheduler may choose to delay or even skip low-remaining-value jobs in overload situations. Additional constraints may be imposed on the scheduler to guide its decisions. For example, it may be required that each HRT job must yield *some* positive utility to ensure that they are never skipped.

While utility accrual scheduling is a very versatile concept, it is not necessarily possible to predict the resulting deadline miss ratio or maximum tardiness. From a practical standpoint, it further suffers from the same drawbacks as the previously mentioned job-skipping-based approaches (such as the need for a special-purpose scheduler and job-skipping support). Although this renders utility accrual scheduling ill-suited for our purposes, it is nonetheless an intriguing approach. We refer the interested reader to Ravindran *et al*. (2005), who survey the major developments in the field of utility accrual scheduling since Jensen *et al*.'s original work.

Job servers. A very different notion of SRT constraints has been studied in the context of *aperiodic job scheduling*. In contrast to sporadic tasks, aperiodic jobs do not necessarily follow a regular arrival pattern, but may arrive at any time without a bound on inter-arrival separation. This creates a hazard for regular sporadic tasks. Aperiodic jobs must thus be isolated such that the temporal correctness of real-time tasks is not endangered while simultaneously minimizing the response times of the aperiodic jobs. The classic approach to integrating aperiodic jobs is to use a server-based scheme, where a sporadic-task-like process acts as a server that processes pending aperiodic jobs whenever the process is scheduled. Much prior work on uniprocessors has investigated how to provision such



(a) HRT utility value function.



(b) SRT utility value function.

Figure 2.10: Illustration of HRT and SRT utility functions. In this example, there is no value in completing a late HRT job, whereas the utility of a late SRT job declines with increasing tardiness.

server tasks to minimize their impact on real-time tasks and to also minimize aperiodic response times (*e.g.*, see Lehoczky *et al.*, 1987; Strosnider *et al.*, 1995; Spuri and Buttazzo, 1994, 1996; Oikawa and Rajkumar, 1999; Abeni and Buttazzo, 1998, 2004; Lin and Brandt, 2005) and such techniques have also been developed for multiprocessors (Srinivasan *et al.*, 2002; Baruah *et al.*, 2002; Baruah and Lipari, 2004a,b; Pellizzoni and Caccamo, 2008; Brandenburg and Anderson, 2007b). The temporal guarantees offered by such server schemes are "soft" because aperiodic jobs have predictable response times as long as the server does not become (severely) backlogged, *i.e.*, if the server is not constantly overloaded. This interpretation of "soft real-time" is orthogonal to the SRT definition adopted in this dissertation: bounded tardiness is required for sporadic real-time tasks, whereas server schemes apply to aperiodic background activity, which is not strictly part of the real-time workload. Conceptually, both notions of "softness" can be combined in a straightforward

manner such that the response times of aperiodic jobs increase by the server's tardiness. One example of aperiodic job servers that may incur bounded tardiness can be found in (Brandenburg and Anderson, 2007b).

On a final note, probabilistic task models and guarantees offer yet another, extensively researched notion of "soft real-time." Devi (2006) reviews several such approaches and how they relate to the bounded-tardiness notion of SRT correctness. Probabilistic guarantees are beyond the scope of this dissertation. However, we note that stochastic reasoning can also be applied to obtain probabilistic tardiness bounds (Mills and Anderson, 2010, 2011).

To summarize, there is a broad consensus that *some* deadline misses are tolerable in a "soft" real-time system, but there is no standard definition of SRT correctness. In the end, the choice of SRT model is a compromise among the expressiveness of the model, its practicality with regard to both implementing RTOS support and obtaining needed parameters, and the ability to make strong analytical guarantees. We chose bounded tardiness as the SRT criterion because it is, in our view, a particularly attractive compromise among the three concerns: it applies without restrictions to the sporadic task model and is well-founded in theory, *a priori* guarantees are an integral part of its definition, and it is comparatively easy to support from an RTOS point of view. Its greatest limitation is that the existence of a tardiness bound for a task does not automatically yield a bound on its deadline miss ratio.

2.2.3 Schedulability Analysis Concepts

In preparation for the review of scheduling policies, we next introduce a few key concepts related to temporal correctness, namely feasibility, optimality, schedulability tests, sustainability, and a set of common assumptions.

Recall that a task set is schedulable if its temporal constraints will always be satisfied (either in an HRT or SRT sense). Feasibility of task sets and optimality of scheduling algorithms are both defined in terms of schedulability. Intuitively, a task set is feasible if it is not impossible to schedule it and a scheduler is optimal if it can successfully schedule all feasible task sets. To avoid repetition, the following definitions should both be interpreted with respect to an implementation on a given platform.

Definition 2.3. A task set τ is HRT (respectively, SRT) *feasible* if and only if there exists a scheduling algorithm $\mathcal{A}$ such that τ is HRT (respectively, SRT) schedulable under $\mathcal{A}$.

Definition 2.4. A scheduling algorithm $\mathcal{A}$ is *optimal* in an HRT (respectively, SRT) sense if and only if every task set τ that is HRT (respectively, SRT) feasible is HRT (respectively, SRT) schedulable under $\mathcal{A}$.

Feasibility and optimality are commonly considered with respect to some restricted class of schedulers or platforms, *e.g.*, "optimal among uniprocessor schedulers" or "feasible on a two-processor platform."

A straightforward feasibility test is given by the total utilization. Recall that the utilization u_i is the fraction of a processor that must be allocated to T_i in order to match its average rate of execution if jobs arrive periodically and execute for e_i time units each. This implies that the total utilization cannot exceed the total number of processors, as otherwise the total processor demand could outstrip the available supply.

Lemma 2.1. *A set of sporadic tasks τ is feasible on m processors only if $u_{\mathrm{sum}}(\tau) \leq m$.*

In the context of job sets (*i.e.*, non-recurrent tasks), a feasibility condition that implies Lemma 2.1 was proven by Horn (1974). Leung and Merrill (1980) stated $u_{\mathrm{sum}}(\tau) \leq m$ as a feasibility condition for periodic task sets. A formal proof of Lemma 2.1 for the case $m = 1$ was given by Baruah *et al.* (1990). This proof can easily be generalized to $m > 1$ and is omitted here.

The necessary condition imposed by Lemma 2.1 applies equally to HRT and SRT feasibility. HRT and SRT constraints, however, differ with regard to sufficient conditions. In the HRT case, Lemma 2.1 can in fact be strengthened to an equivalence, but only for implicit-deadline task sets.

Lemma 2.2. *A set of implicit-deadline sporadic tasks τ is HRT feasible on m processors if and only if $u_{\mathrm{sum}}(\tau) \leq m$.*

Lemma 2.2 follows from the existence of HRT optimal schedulers under which any implicit-deadline task set with $u_{\mathrm{sum}}(\tau) \leq m$ is HRT schedulable (see Sections 2.3.1.2 and 2.3.3.3 below).

Unfortunately, the general case of $d_i \neq p_i$ is much more difficult. In fact, optimal online scheduling of arbitrary-deadline sporadic task sets is impossible on a multiprocessor (Fisher *et al.*, 2010). Further, a sufficient utilization-bound-based HRT feasibility test also does not exist, as there

are task sets with arbitrarily small utilizations that are not HRT feasible. For example, $m+1$ tasks with parameters $e_i = \epsilon$, $d_i = \epsilon$ and $p_i = 1 + \epsilon$ are clearly infeasible (in the HRT sense) on m processors, yet have arbitrarily small utilization for $\epsilon \to 0$.

Using the total density instead of total utilization in Lemma 2.2 also does not yield a necessary HRT feasibility test, as there exist task sets τ with $\delta_{\text{sum}}(\tau) > m$ that are in fact HRT feasible on m processors. For example, consider $m+1$ tasks $\tau = \{T_1, \ldots, T_{m+1}\}$ with parameters $e_i = 1$ and $p_i = m+1$ for each T_i, $d_i = 1$ for $1 \leq i \leq m$, and $d_{m+1} = m+1$. Their total density is $\delta_{\text{sum}}(\tau) = m + \frac{1}{m+1}$, yet τ is HRT feasible on m processors.

Given our focus on implicit-deadline task systems, Lemma 2.2 is adequate for our purposes. See (Fisher, 2007; Fisher *et al.*, 2010) for recent in-depth discussions of HRT feasibility of arbitrary-deadline sporadic task sets (and other, more general task models).

SRT feasibility is much simpler due to the less-strict nature of SRT correctness. In fact, Lemma 2.1 can be strengthened to an equivalence even for arbitrary-deadline task systems.

Lemma 2.3. *A set of sporadic tasks τ is SRT feasible on m processors if and only if $u_{\text{sum}}(\tau) \leq m$.*

Lemma 2.3 directly follows from the existence of SRT optimal schedulers (Devi and Anderson, 2005; Devi, 2006; Leontyev, 2010) under which any implicit-deadline task set with $u_{\text{sum}}(\tau) \leq m$ is SRT schedulable (see Section 2.3.3.1 below).

Schedulability and sustainability. The purpose of a *schedulability test* for a scheduling algorithm $\mathcal{A}$ is to determine *a priori*, that is, during the design phase, whether a task set will be schedulable under $\mathcal{A}$, either in a HRT or SRT sense, when implemented on a given platform. Such a test must be *safe* (*i.e.*, it may not wrongly claim task sets schedulable), but it may be pessimistic (*i.e.*, it may return a negative result for a task set that is in fact schedulable). In the latter case, such a test is also called a *sufficient*, but not *necessary* schedulability test. An *exact* schedulablity test is both necessary and sufficient, that is, it never returns a wrong answer. Exact schedulability tests are highly desirable, but difficult to obtain.

Related to schedulability is the concept of *sustainability* (Baruah and Burns, 2006; Burns and Baruah, 2008; Baker and Baruah, 2009b). Intuitively, sustainability requires that a task set that passed a schedulability test should not suddenly fail, *i.e.*, become unschedulable, when the scheduling problem becomes "easier." That is, a scheduling algorithm $\mathcal{A}$ is *sustainable* if a task set τ that was

deemed schedulable under $\mathcal{A}$ remains schedulable even if τ is modified, *potentially during runtime*, in any of the following ways:

- a period p_i is enlarged;
- an execution requirement e_i is reduced; or
- a relative deadline d_i is increased.

In other words, τ must remain schedulable under $\mathcal{A}$ when the utilization or density of any task is (temporarily) reduced. This is very important in practice because task parameters are typically determined such that e_i is an upper bound, whereas p_i and d_i are lower bounds. From an engineering point of view, all schedulers used in practical systems should thus be sustainable. The concept of sustainability can similarly be applied to schedulability tests (Baruah and Burns, 2006; Burns and Baruah, 2008; Baker and Baruah, 2009b). We make heavy use of sustainability with respect to changes in the execution requirement e_i in Chapters 3, 5, 6, and 7.

The definition of sustainability allows a task set τ that passed a sufficient schedulability test $\mathcal{T}$ to *fail* $\mathcal{T}$ after having been modified in one of the ways listed above (unless $\mathcal{T}$ is an exact test). That is, under a sustainable scheduler, τ must remain schedulable if turned into an "easier" task set τ', but a schedulability test that deemed τ schedulable is not required to identify τ' as schedulable, too. A stricter notion of sustainability is *self-sustainability* (Baker and Baruah, 2009b). A schedulability test $\mathcal{T}$ is self-sustainable if every task set τ that passed $\mathcal{T}$ still passes $\mathcal{T}$ after a reduction in density or utilization of any task in τ. If a schedulability test is not self-sustainable, then a system designer may be faced with the counterintuitive situation that a once-schedulable system design cannot be shown to be correct after it was simplified—in other words, a non-self-sustainable schedulability test can cause a failure in *validation*, but cannot lead to a *runtime* failure. While self-sustainable schedulability tests are desirable, non-self-sustainable tests are less problematic than unsustainable scheduling algorithms because they do not result in incorrect systems (but they may result in increased pessimism). There are a number of useful schedulability tests for which it is not known (in the published literature) whether they are sustainable (see Section 2.3.3.1 below).

Common assumptions. We review a number of relevant scheduling algorithms and associated schedulability tests in the subsequent sections. Most schedulability analysis results make a number of

standard simplifying assumptions. To avoid repeating these for each schedulability test, we assume the following task behavior unless noted otherwise.

1. All tasks are independent: tasks do not share any resources besides the processor(s).
2. Jobs do not self-suspend: pending jobs are always ready to execute when allocated a processor by the scheduler.
3. Jobs are always preemptive: scheduling decisions can be enacted at any time.
4. The task invocation sequence is legal: all job releases of each T_i are separated by at least p_i time units and no job of T_i requires more than e_i time units of processor time to complete.
5. All runtime overheads (such as those caused by job migrations, scheduling decisions, context switches, *etc.*) are negligible.

Each of these assumptions is to some degree problematic in practice. In fact, as discussed in detail in Chapter 1, a driving motivation underlying the research presented herein is to remove Assumptions 1 and 5 by integrating predictable resource-sharing protocols and by accounting for runtime overheads prior to schedulability analysis. Nonetheless, we shall first describe the scheduling algorithms and their schedulability tests assuming idealized task behavior, and describe how to remove these assumptions in subsequent sections and chapters.

2.3 Real-Time Scheduling

In this section, we discuss scheduling from an algorithmic point of view; implementation issues and overheads are considered in detail in Chapter 3. The following review is intended both to provide the needed background for the presentation of our overhead-aware scheduler evaluation methodology in Chapters 3 and 4, and also to document—in one place—the algorithms that are implemented in LITMUS$^{\mathrm{RT}}$ and their supporting analysis. We therefore focus on examples and the intuition underlying the discussed algorithms and omit proofs, which can be found in the cited works.

When the number of pending jobs exceeds the number of processors m, a *scheduler* is required to assign (at most) m of the pending jobs to processors.[4] There are several categories of schedulers.

[4]See (Fisher, 2007) for a formalization of the concepts of "schedule" and "scheduling algorithm."

Static schedulers, also known as *table-driven* schedulers, enact a task-set-specific allocation policy that is pre-computed during the design of the system. Such schedulers are easy to implement and straightforward to validate as only the pre-computed table needs to be checked. This makes static schedulers an ideal choice for safety-critical systems. However, they are generally too inflexible for sporadic workloads (with unpredictable inter-arrival delays). In contrast, *dynamic schedulers* evaluate scheduling rules to make scheduling decisions online based on the current system state such as the set of pending jobs and their parameters, that is, they do not rely primarily on pre-computed information. In this dissertation, we only consider dynamic schedulers.

We further restrict our attention to the class of *priority-driven* schedulers. Conceptually, a priority-driven scheduler instantiated for m processors works as follows: each pending job is assigned a priority, pending jobs are queued in a priority queue ordered by job priority, and the m pending jobs with the highest priorities (if that many exist) are scheduled. If there are fewer than m jobs pending, then some of the processors execute (non-real-time) background tasks or idle. While actual implementations may diverge substantially from this simple description, the conceptual framework of priority-driven scheduling describes a large class of real-time schedulers. This class can further be sub-divided according to when scheduling decisions are made and how job priorities may change over time

Scheduling events. A scheduler is said to *reschedule* when it alters the current job-to-processor mapping. A priority-driven scheduler may have to reschedule either when the set of pending jobs changes, *i.e.*, when a job is released or completes, or when the priority of a job changes. With regard to the time when such a scheduling event is processed, priority-driven schedulers are either *event-driven* or *quantum-driven*. Event-driven schedulers react to job releases, job completions, and priority changes without delay and reschedule immediately when required. In contrast, quantum-driven schedulers only reschedule at times that are multiples of a scheduling quantum Q, that is, the scheduler is invoked only once every Q time units and scheduling events may be processed with a delay of up to Q time units. For now, we only consider event-driven schedulers and revisit quantum-driven scheduling in Section 2.3.3.3 below.

Priorities. A scheduling policy assigns each pending job a priority. We let $\Upsilon(J_i, t)$ denote the priority assigned to J_i at time t. For notational convenience, we require that priorities are unique and

that there exists a total order such that $\mathsf{Y}(J_x, t) < \mathsf{Y}(J_y, t)$ if and only if J_x has a higher priority than J_y at time t. Based on the prioritization function Y, we can differentiate between three fundamental classes of priority-based schedulers. In a *fixed-priority* (FP) scheduler, all jobs of a task share a common, constant priority, *i.e.*, fixed priorities are assigned to tasks and jobs are scheduled according to their tasks' priorities. Formally, $\mathsf{Y}(J_{i,x}, t_x) = \mathsf{Y}(J_{i,y}, t_y)$ for any two jobs $J_{i,x}$ and $J_{i,y}$ and any two times $t_x \in [a_{i,x}, f_{i,x}]$ and $t_y \in [a_{a,y}, f_{i,y}]$. Priority changes are allowed in a *job-level fixed-priority* (JLFP) scheduler, but only at job boundaries, *i.e.*, fixed priorities are assigned to jobs and not tasks. Formally, $\mathsf{Y}(J_{i,x}, t_1) = \mathsf{Y}(J_{i,x}, t_2)$ for any two times $t_1, t_2 \in [a_{i,x}, f_{i,x}]$. The class of JLFP schedulers includes FP schedulers. The final and most general class is *job-level dynamic-priority* (JLDP) schedulers. Under a JLDP scheduler, a job's priority may change at any time.

A fundamental constraint in the scheduling of sporadic tasks is that a job may be assigned to at most one processor at any time: tasks are sequential. This seemingly benign restriction on parallelism has the profound impact that multiprocessor real-time scheduling is substantially more difficult than uniprocessor scheduling (Dhall and Liu, 1978). We review the uniprocessor case next and consider multiprocessor scheduling in Sections 2.3.2–2.3.4 below.

2.3.1 Uniprocessor Real-Time Scheduling

The seminal work on uniprocessor real-time scheduling is due to Liu and Layland (1973). They introduced optimal FP and JLFP policies for the scheduling of periodic tasks and developed corresponding schedulability tests, which Mok (1983) transferred to the sporadic task model.

2.3.1.1 Fixed-Priority Scheduling

In work on FP scheduling, it is common to index tasks in order of decreasing priority, which we adopt as well. The priority function for FP scheduling can then simply be defined as $\mathsf{Y}(J_i, t) = i$. The question is then how to order the tasks in a task set. Liu and Layland (1973) introduced the *rate-monotonic* (RM) priority assignment. Under RM scheduling, tasks are indexed in order of increasing period: if $p_i < p_j$, then $i < j$. Any ties in period are broken arbitrarily.

Example 2.3. Figure 2.11 depicts an example RM schedule of the task set presented in Table 2.4. At time 0, all four tasks release jobs, which are scheduled in order of decreasing task priority. At time 4,

	e_i	p_i	u_i		
T_1	1	4	$\frac{1}{4}$	$= \frac{45}{180}$	$= 0.25$
T_2	1	5	$\frac{1}{5}$	$= \frac{36}{180}$	$= 0.20$
T_3	3	9	$\frac{1}{3}$	$= \frac{60}{180}$	≈ 0.33
T_4	3	18	$\frac{1}{6}$	$= \frac{30}{180}$	≈ 0.17
$u_{\text{sum}}(\tau)$				$\frac{171}{180}$	≈ 0.95

Table 2.4: Example task set.

$J_{3,1}$ is preempted by $J_{2,1}$ and not scheduled again until time 6, when $J_{2,2}$ completes. $J_{4,1}$ is not scheduled until time 7, when all previously released higher-priority jobs have completed. However, it is immediately preempted again at time 8 when $J_{1,3}$ is released. T_3 exhibits an intra-sporadic arrival delay of one time unit at time 9, when its second job is not released until time 10, which gives $J_{4,1}$ a chance to be scheduled during $[9, 10)$. Note that $J_{4,1}$ completes exactly on time, that is, $f_{4,1} = d_{4,1}$.

$\Diamond$

In the context of implicit-deadline periodic tasks, Liu and Layland (1973) proved that RM scheduling is in fact optimal with respect to FP schedulers (*i.e.*, any task set that is schedulable under some FP scheduler is also schedulable under RM scheduling) and derived the following schedulability test (which is given here adapted to sporadic tasks).

Theorem 2.1 (Liu and Layland, 1973)**.** *On a uniprocessor, a set of n implicit-deadline sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ is schedulable under RM scheduling if $u_{\text{sum}}(\tau) \leq n(2^{1/n} - 1)$.*

The bound $n(2^{1/n} - 1)$ converges to $\ln 2 \approx 0.69$ for $n \to \infty$. Liu and Layland showed that this is the highest-achievable utilization bound for RM and hence also FP scheduling—there exist

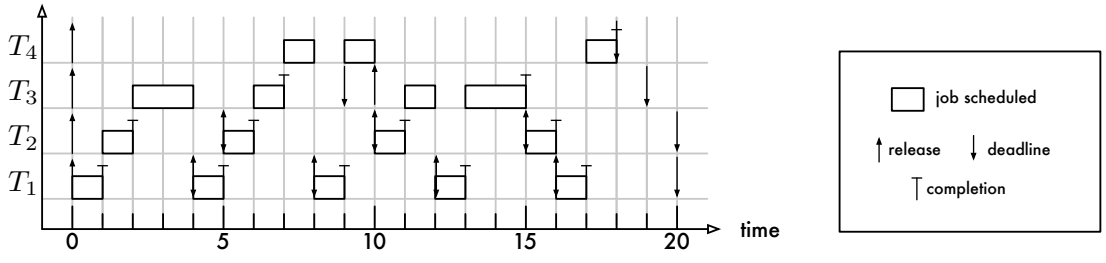


Figure 2.11: Example schedule of four tasks under uniprocessor FP scheduling with RM priorities. The parameters of the tasks are given in Table 2.4. This schedule is discussed in Example 2.3.

implicit-deadline task sets with total utilization just slightly exceeding the bound that are not HRT schedulable under any FP scheduler. Given that any implicit-deadline task set with utilization at most one is feasible on a uniprocessor (Lemma 2.2), this shows that FP scheduling is not HRT optimal on a uniprocessor. Rather, up to 30 percent of a processor's capacity may have to remain unused to ensure HRT schedulability under FP scheduling.

The RM priority assignment is not optimal for non-implicit deadlines. Leung and Whitehead (1982) showed that the *deadline-monotonic* DM priority assignment, where $i < j$ if $d_i < d_j$, is instead optimal for constrained-deadline task sets. Since $p_i = d_i$ for implicit-deadline tasks, DM scheduling generalizes RM scheduling. A simple schedulability test for DM scheduling can be obtained by substituting $\delta_{\text{sum}}(\tau)$ for $u_{\text{sum}}(\tau)$ in Theorem 2.1.

However, note that Theorem 2.1 is not an equivalence: there exist higher-utilization task sets that are in fact schedulable under RM scheduling. For example, two implicit-deadline tasks with parameters $e_i = 1$ and $p_i = 2$ are schedulable under RM scheduling and have a total utilization $1 > 2(2^{1/2} - 1) \approx 0.83$. Similarly, the task set from Example 2.3 has a total utilization of $u_{\text{sum}}(\tau) = \frac{171}{180} \approx 0.95$ (see Table 2.4), which exceeds the bound $4(2^{1/4} - 1) \approx 0.76$, yet appears to be HRT schedulable in the example schedule shown in Figure 2.11.

An exact schedulability test for constrained-deadline tasks under FP scheduling (*i.e.*, under any priority assignment) was later developed by Joseph and Pandya (1986). Their approach is to compute an upper bound R_i on the maximum response time r_i for each task explicitly. Given such a bound for each task T_i, HRT schedulability can be established by checking that $r_i \leq R_i \leq d_i$. The bound R_i can be found using the following recursion.

Theorem 2.2 (Joseph and Pandya, 1986)**.** *Let $\tau = \{T_1, \ldots, T_n\}$ denote a set of constrained-deadline sporadic tasks indexed in order of decreasing priority. On a uniprocessor, under FP scheduling, the response r_i of task $T_i \in \tau$ is bounded by the smallest R_i, where $R_i \geq e_i$, that satisfies the following equation:*

$$R_i = e_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{p_h} \right\rceil \cdot e_h.$$

Each R_i can be iteratively computed by using e_i as an initial value for R_i and by repeatedly re-evaluating the right-hand side until it and the left-hand side converge. Convergence is guaranteed as long as $u_{\text{sum}}(\tau) \leq 1$ (Joseph and Pandya, 1986).

Example 2.4. Consider the task set τ from Example 2.3 with parameters as given in Table 2.4. Applying Theorem 2.2 to each $T_i \in \tau$ yields the following response-time bounds.

$$
\begin{aligned}
R_1 &= e_1 & &= 1 \\
R_2 &= e_2 + \left\lceil \frac{R_2}{p_1} \right\rceil \cdot e_1 & &= 1 + \left\lceil \frac{2}{4} \right\rceil \cdot 1 = 2 \\
R_3 &= e_3 + \left\lceil \frac{R_3}{p_1} \right\rceil \cdot e_1 + \left\lceil \frac{R_3}{p_2} \right\rceil \cdot e_2 & &= 3 + \left\lceil \frac{7}{4} \right\rceil \cdot 1 + \left\lceil \frac{7}{5} \right\rceil \cdot 1 = 7 \\
R_4 &= e_4 + \left\lceil \frac{R_4}{p_1} \right\rceil \cdot e_1 + \left\lceil \frac{R_4}{p_2} \right\rceil \cdot e_2 + \left\lceil \frac{R_4}{p_3} \right\rceil \cdot e_3 & &= 3 + \left\lceil \frac{18}{4} \right\rceil \cdot 1 + \left\lceil \frac{18}{5} \right\rceil \cdot 1 + \left\lceil \frac{18}{9} \right\rceil \cdot 3 = 18
\end{aligned}
$$

Note that each task's response-time bound matches the actual response time of its first job in Figure 2.11. This is because a *critical instant*, which is a point in time when a newly-released job's response time is maximized, occurs when all tasks release a job at the same time (Liu and Layland, 1973). Since $R_i \leq d_i = p_i$ for each T_i, the task set is indeed HRT schedulable under RM scheduling. $\Diamond$

Using Theorem 2.2 in this manner to establish HRT schedulability is called *response time analysis*. Baruah and Burns (2006) showed that response time analysis is sustainable and thus well-suited to determining schedulability in practical systems. Similar tests that also estimate worst-case response times but may converge more quickly were later developed by Lehoczky *et al*. (1989) and Audsley *et al*. (1991). Joseph and Pandya's response time bound, as stated in Theorem 2.2, applies to constrained-deadline sporadic tasks only. A response time test for arbitrary-deadline tasks exists as well, but requires considering longer intervals since multiple jobs of a task may be pending at the same time (Lehoczky *et al*., 1991).

Note that, for a given invocation sequence, the actual maximum response time r_i may be well less than the worst-case response time bound R_i, but the bound R_i is tight in the sense that there exist legal invocation sequences such that $r_i = R_i$ (*e.g.*, as in Figure 2.11). For this reason, the response time test is an exact HRT schedulability test for constrained-deadline sporadic tasks: if $R_i > d_i$ for some T_i, then there exists a legal invocation sequence such that some J_i misses its deadline.

Recall that such deadline misses are allowed in an SRT context provided that tardiness is bounded, *i.e.*, a task set τ is SRT schedulable if there exists a constant B such that $R_i \leq d_i + B$ for each $T_i \in \tau$.

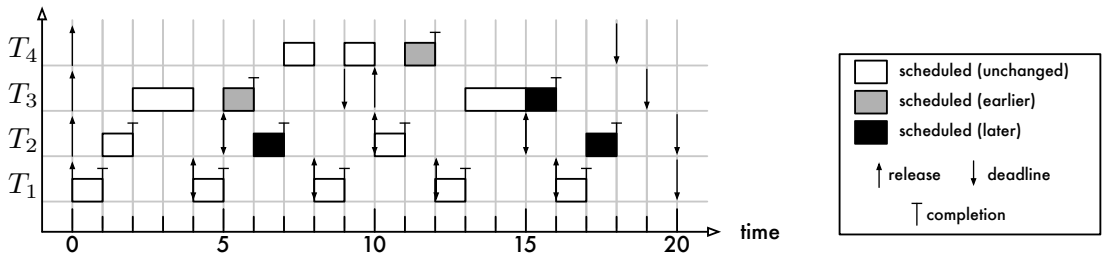


Figure 2.12: Example EDF schedule of the task set given in Table 2.4. Grey and black processor allocations indicate differences from the FP schedule shown in Figure 2.11. A grey allocation indicates that a job received processor service earlier than in the FP case; a black allocation means that the job was processed at a later point in time under EDF scheduling. This schedule is discussed in Example 2.5.

Joseph and Pandya (1986) considered this case as well:[5] as long as a finite R_i for task T_i exists, which it does if $u_{\text{sum}}(\tau) \leq 1$ (Joseph and Pandya, 1986), a valid tardiness bound for T_i is given by $R_i - d_i$.[6] This implies that FP scheduling is in fact SRT optimal with respect to uniprocessor schedulers since any task set with utilization exceeding one is infeasible on a uniprocessor.

2.3.1.2 Job-Level Fixed-Priority Scheduling

The most important JLFP real-time scheduler is the *earliest-deadline-first* (EDF) policy, which prioritizes each job by its absolute deadline. Absolute deadlines are not necessarily unique since multiple jobs may have their absolute deadlines at the same point in time. We therefore use the task index as a consistent deadline tie-break; formally, a job $J_{i,j}$'s priority is denoted as $\mathsf{Y}(J_{i,j}, t) = (d_{i,j}, i)$, with the interpretation $(d_{i,j}, i) < (d_{x,y}, x) \Leftrightarrow d_{i,j} < d_{x,y} \vee (d_{i,j} = d_{x,y} \wedge i < x)$.

Example 2.5. The example EDF schedule depicted in Figure 2.12 shows the same task set previously considered in Example 2.3. Most of the scheduling decisions taken by EDF are identical to those made by FP. However, at time 5, $\mathsf{Y}(J_{3,1}, 5) = (9, 3) < (10, 2) = \mathsf{Y}(J_{2,2}, 5)$, so that $J_{3,1}$ and $J_{2,2}$'s allocations are switched (compared to the FP schedule shown in Figure 2.11). Notably, $J_{4,1}$'s

[5] Joseph and Pandya's work predates the adoption of bounded tardiness as a notion of SRT correctness (Srinivasan and Anderson, 2003; Devi and Anderson, 2005). Joseph and Pandya (1986) actually derived a bound on the maximum input buffer size required to compensate for missed deadlines, which requires bounding maximum tardiness.

[6] Joseph and Pandya proved the existence of a finite R_i if $u_{\text{sum}}(\tau) \leq 1$; however, the formula for R_i given in (Joseph and Pandya, 1986) is unfortunately not necessarily sufficient for the SRT case. A tardy job may still be pending when its successor is released, which causes analytical complications similar to allowing arbitrary deadlines; the generalized approach presented in (Lehoczky *et al.*, 1991) should be used instead.

response time is much lower under EDF ($r_{4,1} = 12$ instead of 18 in the FP case), which requires $J_{3,2}$ and $J_{2,4}$ to be delayed. $J_{2,4}$ and $J_{1,5}$ have the same absolute deadline $d_{2,4} = d_{1,5} = 20$, so the tie in priority at time 16 is broken in favor of $J_{1,5}$ due to its lower task index. $\Diamond$

EDF derives its significance from being optimal on a uniprocessor (Liu, 1969a; Liu and Layland, 1973; Labetoulle, 1974; Dertouzos, 1974), that is, on a uniprocessor, any feasible arbitrary-deadline task set is HRT schedulable under EDF.[7] In the case of implicit-deadline sporadic tasks, this results in a simple utilization bound.

Theorem 2.3 (Liu, 1969a; Liu and Layland, 1973)**.** *On a uniprocessor, a set of implicit-deadline sporadic tasks is HRT schedulable under EDF if and only if* $u_{\text{sum}}(\tau) \leq 1$.

Since SRT schedulability generalizes HRT schedulability, EDF is also optimal in the SRT sense. EDF is optimal in the case of arbitrary deadlines as well; however, testing for HRT schedulability becomes more involved. A simple, but inexact HRT schedulability condition can be obtained by substituting total density for the total utilization in Theorem 2.3, which yields the following density bound.

Theorem 2.4. *On a uniprocessor, a set of arbitrary-deadline sporadic tasks is HRT schedulable under EDF if* $\delta_{\text{sum}}(\tau) \leq 1$.

While simple, this test can be very pessimistic for constrained-deadline task sets. Exact tests have been developed for this case as well (Baruah *et al.*, 1990; Albers and Slomka, 2005);[8] however, these tests are of pseudo-polynomial time complexity and can thus be very expensive to compute. As a compromise, inexact polynomial-time tests, which yield (slightly) more pessimistic results at the benefit of greatly reduced computational effort, have been developed as well (Devi, 2003; Masrur *et al.*, 2008, 2010). Baruah and Burns (2006) showed that any sufficient schedulability test for the EDF scheduling of sporadic tasks on a uniprocessor is sustainable.

Summary. On a uniprocessor, any implicit-deadline task set is feasible as long as the processor is not overutilized. In the HRT case, FP scheduling is subject to capacity loss: up to 30 percent of processor capacity may have to remain unallocated to ensure HRT constraints. On the positive side,

[7] In fact, EDF's optimality extends to arbitrary job sets, *i.e.*, EDF is an optimal uniprocessor scheduler even if job arrivals are not constrained by the sporadic task model (Jackson, 1954; Horn, 1974).

[8] Strictly speaking, Baruah *et al.*'s test requires $u_{\text{sum}}(\tau) < 1$ and thus is *almost* exact.

response-time analysis can identify all schedulable task sets, *i.e.*, it is an exact schedulability test. In contrast to FP scheduling, the JLFP policy EDF is optimal on a uniprocessor and not subject to any capacity loss. Neither FP nor EDF scheduling incur capacity loss in the SRT case, that is, both ensure bounded tardiness for any feasible task set. However, for implicit-deadline task sets, tardiness is always zero under EDF, but not necessarily so under FP scheduling. From a schedulability point of view—that is, if the goal is to guarantee schedulability for as many feasible task sets as possible—there is thus no reason to prefer FP over EDF scheduling. This substantiates the criticism expressed in Chapter 1.

For similar reasons, we do not consider uniprocessor JLDP scheduling. While such policies exist, they offer only little (if any) advantages over EDF scheduling and are thus of limited practical relevance. However, JLDP policies are of great importance in the context of multiprocessor real-time scheduling, as is discussed in Section 2.3.3.

2.3.2 Partitioned Multiprocessor Real-Time Scheduling

There are two fundamental approaches to applying priority-driven scheduling to a shared-memory multiprocessor: either all available processors are scheduled using one scheduler (with, conceptually speaking, a single, shared ready queue), or the set of processors is subdivided into smaller, disjoint sets of processors that are scheduled independently (by separate priority-driven schedulers, each with a local ready queue). The former approach is called *global scheduling*, and the latter approach is called either *clustered* or *partitioned scheduling*, depending on the number of processors per set. Under partitioned scheduling, there is only one processor in each subdivision (called a *partition* in this case), whereas there may be multiple processors per subdivision under clustered scheduling (and each subdivision is called a *cluster*). We first consider partitioned scheduling, followed by global scheduling in Section 2.3.3 below. Finally, clustered scheduling, which can be understood either as a hybrid or as a generalization of global and partitioned scheduling, is discussed in Section 2.3.4.

Partitioned scheduling, first studied in the context of real-time systems by Dhall and Liu (1978), is the multiprocessor real-time scheduling approach most commonly used in practice. Its great appeal stems from the fact that each partition (*i.e.*, each processor) can be scheduled and analyzed using existing uniprocessor techniques. We consider both FP and EDF partitioned schedulers,

namely *partitioned FP* (P-FP) and *partitioned EDF* (P-EDF).[9] However, the reuse of existing uniprocessor scheduling theory comes at a price. To obtain m simpler uniprocessor problems from a multiprocessor platform consisting of m processors, the task set must first be *partitioned*, that is, each task must be statically assigned to a partition such that no processor is overloaded. Solving this task assignment problem requires solving a bin-packing-like problem.

Bin-packing. The bin-packing problem is a classic intractable problem that is NP-hard in the strong sense (Garey and Johnson, 1979). Given a bin capacity V and a set of n items $x_1, \ldots, x_n$ with corresponding sizes $a_1, \ldots, a_n$, the goal is to assign each item to a bin such that the number of bins is minimized, but without exceeding the capacity of any bin. The corresponding decision problem is to determine for a given number B whether there exists a "packing" (*i.e.*, item-to-bin assignment) such that the items fit into B bins.

In the context of partitioned scheduling, the items are tasks and their respective sizes are given by their utilizations. Processors correspond to bins and each bin's effective capacity V is dependent on the scheduling policy in use and the type of deadline constraint. In the case of implicit deadlines and EDF, each processor has a capacity of 1.0. Partitioning a task set is equivalent to solving a bin-packing problem in the sense that an implicit-deadline task set is feasible on m processors under partitioned scheduling if and only if there exists a "packing" of all tasks into m "bins." In other words, the bin-packing decision problem can be reduced to task-set partitioning in polynomial time (by scaling item sizes and bin sizes), which implies that finding an *optimal* task assignment in all cases is intractable (unless $P = NP$). Dhall and Liu (1978) formally proved that the task assignment problem is indeed NP-hard in the strong sense by showing that a solution can be used to solve the 3-partition problem (Garey and Johnson, 1979).

The practical implication is that we can use existing bin-packing heuristics to find valid task assignments. The literature on bin-packing heuristics is extensive and a comprehensive review is beyond the scope of this dissertation. Instead, we briefly summarize the heuristics most relevant to our work and refer the interested reader to (Coffman *et al.*, 1997). In the description of the heuristics below, let S_j denote the set of items assigned to the j^{th} bin and let V_j denote the remaining capacity

[9] In principle, it would also be possible to use different scheduling policies on different processors, *e.g.*, some processors could be scheduled with EDF and some with FP, but this is currently not supported in LITMUS$^{\text{RT}}$ (and we are not aware of any research into such mixed-policy systems). Since our main interest is to compare schedulers under consideration of overheads, we only consider "pure" P-EDF and P-FP.

of the j^{th} bin, *i.e.*, $V_j = V - \sum_{x_i \in S_j} a_i$. Initially, there is only one (empty) bin, but additional bins can be added by the heuristic as required. Bins are indexed in the order in which they are created.

It is generally more difficult to pack large items (in relation to the bin size V) as smaller items are more likely to fit into the remaining capacity of partially used bins. Therefore, it is beneficial to pack items in order of decreasing size, which is easy to accommodate during off-line partitioning. We hence assume that the items have been sorted such that $a_1 \geq a_2 \geq \ldots \geq a_n$ prior to applying one of the following placement heuristics.

First-fit decreasing. To place an item x_i, the *first-fit heuristic* considers each bin in index order and places x_i in the first bin where it fits. That is, item x_i is added to S_j if $V_j \geq a_i$ and $V_l < a_i$ for each $l \in \{1, \ldots, j-1\}$. If no such j exists, then x_i is placed in a new empty bin that is appended to the packing. Asymptotically, *i.e.*, for a large number of bins, the first-fit decreasing heuristic requires at most $\frac{11}{9} \approx 1.22$ times the number of bins used by an optimal solution (Johnson, 1973, 1974). For a small number of bins (less than 4), first-fit decreasing requires no more than 1.5 times the number of bins used by an optimal solution (Simchi-Levi, 1994).

Best-fit decreasing. In contrast to the first-fit heuristic, the *best-fit heuristic* considers all bins and selects the bin that will have minimal remaining capacity after placing item x_i. Let $L = \{l \mid V_l \geq a_i\}$ denote the set of bins that have sufficient remaining capacity. Item x_i is added to S_j, where $j \in L$, such that $V_j \leq V_l$ for each $l \in L$ (prior to placing x_i, with ties broken arbitrarily). If no such j exists, then x_i is placed in a new empty bin. The best-fit heuristic is known to compare well to other online bin-packing heuristics (Kenyon, 1996), usually deviating only little from an optimal solution (with regard to the number of required bins). As it is the case with the first-fit heuristic, the best-fit heuristic requires (asymptotically) at most $\frac{11}{9} \approx 1.22$ times the number of bins used by an optimal solution (Johnson, 1973, 1974), and yields results within a factor of 1.5 of the optimal solution for small numbers of bins (Simchi-Levi, 1994).

Worst-fit decreasing. As the name implies, the *worst-fit heuristic* is the inverse of the best-fit heuristic, that is, item x_i is placed to maximize the remaining capacity of the bin in which it is placed. Define L as above. Item x_i is placed in the j^{th} bin, where $j \in L$, such that $V_j \geq V_l$ for each $l \in L$. Again, x_i is placed by itself into a newly-added bin if no such j exists. The worst-case performance of the worst-fit decreasing heuristic is somewhat worse than that of either the first-fit or best-fit

Item size	Assigned bin using first-fit	best-fit	worst-fit
$0.0 < a_i \leq 0.1$	bin 1	bin 2	bin 3
$0.1 < a_i \leq 0.3$	bin 1	bin 1	bin 3
$0.3 < a_i \leq 0.5$	bin 3	bin 3	bin 3

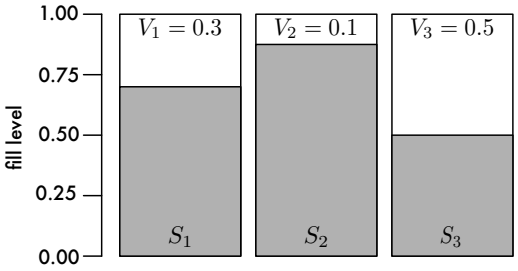


Figure 2.13: Illustration of the first-fit, best-fit, and worst-fit bin-packing heuristics.

decreasing heuristics: asymptotically, the worst-fit decreasing heuristic uses at most $\frac{5}{4} = 1.25$ times the number of bins used by an optimal solution (Johnson, 1973, 1974).

Example 2.6. Figure 2.13 shows three bins with remaining capacity $V_1 = 0.3$, $V_2 = 0.1$, and $V_3 = 0.5$, respectively. The bin in which the next item x_i will be placed depends on both its size a_i and the choice of bin-packing heuristic, as indicated by the table in Figure 2.13. Suppose $a_i = 0.1$. Then the first-fit heuristic would assign x_i to S_1, the best-fit heuristic would assign x_i to S_2, and the worst-fit heuristic would choose S_3. For larger items, there are fewer candidate bins; for $a_i > 0.3$, any (reasonable) heuristic must assign the item to the third bin. ◊

Assigning tasks. The above "textbook" heuristics attempt to minimize the number of bins by assigning items to partially used bins whenever possible and only allocating a new bin when absolutely required. This is a slight mismatch to the problem of partitioning a task set onto a given multiprocessor platform, where the number of processors m is fixed. That is, it is not immediately possible to just add another processor, nor is it of great benefit to leave present processors completely unallocated to real-time tasks.[10] We therefore interpret these heuristics as follows: there are initially m empty bins, and any attempt to allocate additional bins indicates that the heuristic failed to partition the task set. This has little impact on the first-fit and best-fit heuristics, which attempt to fully allocate processors before assigning tasks to idle processors, but causes the worst-fit heuristic to spread out the total utilization more-or-less evenly among all processors.

Spreading out the load among all processors is preferable for several reasons. First, there is little benefit to leaving available processors unused. Second, from an engineering point of view, it makes each processor more resilient to transient overloads. And third, some remaining "idle" capacity is required on each processor to compensate for the inflation of task utilizations caused

[10]Depending on the energy model, this may not be the case in the context of energy-aware systems.

by overhead accounting (see Chapter 3). For these reasons, we use the worst-fit heuristic to assign tasks to processors. While it may appear counterintuitive to use a heuristic with a worse worst-case performance than either first-fit or best-fit, the worst-fit heuristic works well in practice and has been used successfully by us in (Brandenburg *et al.*, 2008; Bastoni *et al.*, 2010b, 2011). This choice is further supported by experiments carried out by López *et al.* (2004a), who considered partitioning heuristics for P-FP scheduling where the Liu and Layland (1973) RM bound is used to check schedulability on each processor. They observed that there is no significant difference among placement heuristics when tasks are placed in order of decreasing utilization (as we do). They also formally showed that all "reasonable" placement heuristics yield the same utilization bound (for the described setup) when tasks are placed in order of decreasing utilization (López *et al.*, 2004a).

Limitations. While finding *optimal* task assignments is intractable in the general case, the above bin-backing heuristics typically find near-optimal solutions. That is, for task sets that can be partitioned onto m processors at all, it is usually possible to find a valid task assignment, and particularly for task sets that do not fully utilize a platform (*i.e.*, if there is ample spare capacity). Unfortunately, there exist task sets with total utilization much less than m that cannot be partitioned onto m processors, even using an optimal assignment algorithm, *i.e.*, one that minimizes the number of required processors.

For example, consider the implicit-deadline task set $\tau^{ABJ} = \{T_1, \ldots, T_{m+1}\}$ due to Andersson *et al.* (2001), where, for each $T_i \in \tau^{ABJ}$, $p_i = 2$ and $e_i = 1 + \epsilon$ for some positive ϵ. The total utilization of τ^{ABJ} is $u_{\text{sum}}(\tau^{ABJ}) = \sum_{i=1}^{m+1} \frac{1+\epsilon}{2} = (m+1) \cdot \frac{1+\epsilon}{2}$. By design, no two tasks of τ^{ABJ} can be assigned to the same processor because their combined utilization would exceed the capacity of a single processor, *i.e.*, $2 \cdot \frac{1+\epsilon}{2} = 1 + \epsilon > 1$ for $\epsilon > 0$. Therefore, τ^{ABJ} requires at least $m + 1$ processors and is thus not feasible on m processors under partitioned scheduling. However, for $\epsilon \to 0$, τ^{ABJ}'s total utilization approaches $\frac{m+1}{2}$. Recall from Section 2.2.3 that implicit-deadline task sets are feasible on m processors if their total utilization does not exceed m. The existence of task sets with utilization $\frac{(m+1)}{2} \cdot (1 + \epsilon)$ that are infeasible under partitioning shows that, asymptotically, *i.e.*, for large m, partitioned scheduling can cause up to nearly 50 percent capacity loss, no matter which scheduling policy is employed on each processor and which partitioning heuristic is being used.

However, this is merely an upper bound, *i.e.*, it leaves open the question whether all implicit-deadline task sets with total utilization at most $\frac{m+1}{2}$ are necessarily schedulable under partitioning. In the case of P-EDF, the optimality of EDF on a uniprocessor implies that any implicit-deadline task set that can be partitioned onto m processors is also HRT schedulable. López *et al*. (2004b) showed that the desired utilization bound can in fact be realized with any of the bin-packing heuristics discussed in this section (among other heuristics).

Theorem 2.5 (López *et al*., 2004b)**.** *An implicit-deadline task set τ is HRT schedulable under P-EDF scheduling on m processors if $u_{\text{sum}}(\tau) \leq \frac{m+1}{2}$ when using either the first-fit, best-fit, or worst-fit decreasing heuristic.*

In fact, López *et al*. proved a more-general result, but the special case stated in Theorem 2.5 is sufficient for our purposes. Note that even the use of an optimal bin-packing algorithm could not yield a higher utilization bound due to the existence of hard-to-pack task sets such as τ^{ABJ} above (López *et al*., 2004b). In this sense, the stated utilization bound is the best possible. However, it is merely a sufficient, but not an exact, schedulability test. It is thus generally preferable to test whether a given task set is schedulable (either HRT or SRT) by simply partitioning the task set (if possible) and by applying an exact test to each partition. This approach also works in the case of non-implicit deadlines.

Since EDF is optimal on a uniprocessor and FP scheduling is not, one might reasonably expect a significantly worse utilization bound under P-FP scheduling. Surprisingly, this is not the case. Oh and Baker (1998) were the first to consider a utilization bound for P-FP scheduling. They showed that no priority assignment can ensure a utilization bound higher than $(m+1) \cdot \left(1 + 2^{\frac{1}{m+1}}\right)^{-1}$, which converges to $\frac{m+1}{2}$ for large m. Asymptotically, the upper bound on a utilization bound under P-FP scheduling matches the general upper bound on any partitioned scheduler and is not worse than the bound for P-EDF. For example, for $m = 24$, the P-FP bound corresponds to $\approx 0.516m$, whereas the P-EDF bound yields $\frac{m+1}{2} \approx 0.521m$, a minuscule difference. Oh and Baker (1998) further showed that any implicit-deadline task set τ is HRT schedulable under P-FP scheduling when used with the RM priority assignment if $u_{\text{sum}}(\tau) \leq (\sqrt{2}-1) \cdot m \approx 0.414m$, hence leaving a utilization difference of $0.086m$ compared to the theoretical upper limit. This gap was later closed by Andersson and Jonsson (2003), who proved the following result.

Theorem 2.6 (Andersson and Jonsson, 2003)**.** *An implicit-deadline task set τ is HRT schedulable under P-FP scheduling if $u_{\mathrm{sum}}(\tau) \leq \frac{m}{2}$.*

A particular, specific-purpose task assignment algorithm must be used to realize the stated bound (Andersson and Jonsson, 2003). However, as it is the case with P-EDF, it is more practical to just partition a given task set (if possible) and apply response-time analysis to each partition due to the inexact nature of Theorem 2.6. The key observation here is that P-EDF and P-FP scheduling have a utilization bound of approximately $\frac{m}{2}$ for large m. It is therefore not readily apparent which is preferable from a capacity-loss point of view. We explore this question in the case study presented in Chapter 4.

2.3.3 Global Multiprocessor Real-Time Scheduling

Under priority-driven global scheduling, the m highest-priority pending jobs are scheduled at any time (if that many exist). Preemptions thus only affect the lowest-priority scheduled job, that is, when a new job J_i is released at time t, it preempts the m^{th} highest-priority scheduled job J_m if $\mathsf{Y}(J_i, t) < \mathsf{Y}(J_m, t)$. The appeal of global scheduling is that it can overcome the algorithmic capacity loss inherent in partitioned scheduling. Conceptually, all processors service a single, shared ready queue under global priority-driven scheduling (though an actual implementation may use more advanced data structures with some processor-local state). This eliminates the need to solve a task assignment problem, which is the source of capacity loss under partitioned scheduling. As a result, there exist optimal global schedulers for implicit-deadline tasks, with regard to both SRT and HRT constraints. Unfortunately, no optimal schedulers exist in the case of constrained- and arbitrary-deadline tasks (Fisher *et al.*, 2010). In this section, we review two global priority-driven schedulers in detail, namely *global EDF* (G-EDF) and the proportionate fair algorithm PD^2. Both are optimal for implicit-deadline tasks: G-EDF only with regard to SRT constraints, and PD^2 also with regard to HRT constraints. Furthermore, we briefly discuss *global fixed-priority* (G-FP) scheduling, which is neither SRT nor HRT optimal.

Global scheduling of real-time tasks was first considered by Dhall and Liu in the context of the periodic task model and HRT constraints (Liu, 1969b; Dhall and Liu, 1978). Notably, they showed that neither RM nor EDF retains its respective optimality property when the number of processors

m exceeds one (discussed below). Specifically, they demonstrated that there exist task sets of total utilization approaching one that are unschedulable under either approach (Dhall and Liu, 1978). In other words, in an HRT context, both G-FP with the RM priority assignment and G-EDF can be subject to algorithmic capacity loss approaching $m - 1$—there is no guarantee that these schedulers can derive any analytical benefit from multiple processors at all. Given these early negative results (and the lack of widespread availability of shared-memory multiprocessor platforms), interest in the global scheduling was quite limited in the first two decades of research into real-time systems (Davis and Burns, 2011b). However, this has changed in recent years.

Renewed interest in the algorithmic properties of global scheduling was sparked by Baruah *et al*. (1996), who designed and analyzed the first optimal (with regard to periodic tasks and implicit deadlines) multiprocessor scheduler that is implementable, *i.e.*, that does not rely on fractional or infinitesimal processor allocations. With the widespread emergence of shared-memory and shared-cache multicore platforms in this decade, global real-time scheduling gained practical relevance as well. For the sake of consistency, we review global scheduling policies relevant to our work in order of fixed to dynamic priorities, even though this order does not reflect the historic development.

2.3.3.1 Global Fixed-Priority Scheduling

Recall that FP scheduling policies only differ in the way that priorities are assigned. As mentioned above, Dhall and Liu (1978) demonstrated that the RM policy is not an optimal priority assignment for G-FP scheduling. To show this so-called *Dhall effect*, they constructed the following task set. Given m processors, consider a set of $n > m$ implicit-deadline tasks $\tau = \{T_1, \ldots, T_{m+1}\}$, where $e_i = 2 \cdot \epsilon$ and $p_i = 1$ for $1 \leq i \leq m$, and $e_{m+1} = 1$ and $p_{m+1} = 1 + \epsilon$. Under an RM priority assignment, τ is not HRT schedulable even though $u_{\text{sum}}(\tau) \to 1$ as $\epsilon \to 0$. An illustration of the Dhall effect for $m = 2$ and $n = 3$ is depicted in Figure 2.14. As J_3 has the latest deadline in this example, it would miss its deadline under G-EDF scheduling as well, which demonstrates that the Dhall effect limits both global RM and G-EDF scheduling. Note, however, that τ would be schedulable if the priorities of T_1 and T_3 were switched, that is, the Dhall effect applies specifically to global RM scheduling and not G-FP scheduling in general.

Devi (2006) further showed that the RM priority assignment does not necessarily ensure bounded tardiness under G-FP scheduling. She constructed a three-task implicit-deadline example for $m = 2$,

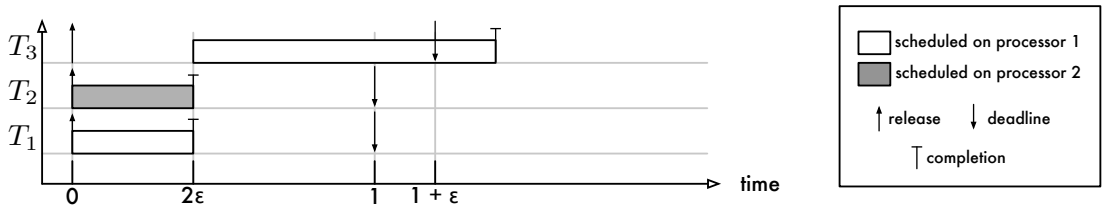


Figure 2.14: Illustration of the Dhall effect for $m = 2$. $J_{3,1}$ misses its deadline at time $d_{3,1} = 1 + \epsilon$ because it is not scheduled until time 2ϵ, which leaves only $1 - \epsilon$ time units until its deadline to complete $e_3 = 1$ time units of work.

where $e_1 = e_2 = 2$, $p_1 = p_2 = 3$, and $e_3 = 4$ and $p_3 = 6$ (Devi, 2006). A resulting schedule with unbounded tardiness is illustrated in Figure 2.15. Since jobs of T_3 are repeatedly preempted when jobs of T_1 and T_2 arrive simultaneously, the response time of successive jobs of T_3 grows unboundedly. Again, note that giving T_3 higher priority would alleviate the problem. Devi's example highlights that tasks are vulnerable to starvation under G-FP scheduling and that priorities must be carefully chosen. This is in contrast to the uniprocessor case, where any priority assignment results in bounded tardiness as long as the processor is not overutilized (Joseph and Pandya, 1986).

In fact, there exist task sets that are unschedulable under any priority assignment. Recall Andersson *et al.*'s task set τ^{ABJ} consisting of $m + 1$ implicit-deadline tasks with parameters $e_i = 1 + \epsilon$ and $p_i = 2$, which is the pathological example that was used to show that there exist "unpartitionable" task sets (see page 68). On m processors, τ^{ABJ} is not schedulable under any FP scheduler: irrespective of the priority assignment, if all $m + 1$ tasks release a job at time $t = 0$, then $J_{m+1,1}$ (the job of the lowest-priority task) will not be scheduled until time $1 + \epsilon$, at which point there is insufficient time left to complete its $1 + \epsilon$ units of work by time 2. Further, if all tasks release jobs periodically, T_{m+1} will have unbounded tardiness since it will receive only $1 - \epsilon$ time units of processor service every 2 time units, which fails to match its utilization. This shows that no global FP scheduler can possess a utilization bound exceeding $\frac{m+1}{2}$, just as is the case with partitioned scheduling.

No provably optimal priority assignment or exact feasibility test is known for the G-FP scheduling of sporadic tasks (Davis and Burns, 2011b). For a given priority assignment, an upper bound on response times can be derived using multiprocessor response-time tests (see (Davis and Burns, 2011a,b) for a recent survey). A schedulable priority assignment may be found with $O(n^2)$ response-

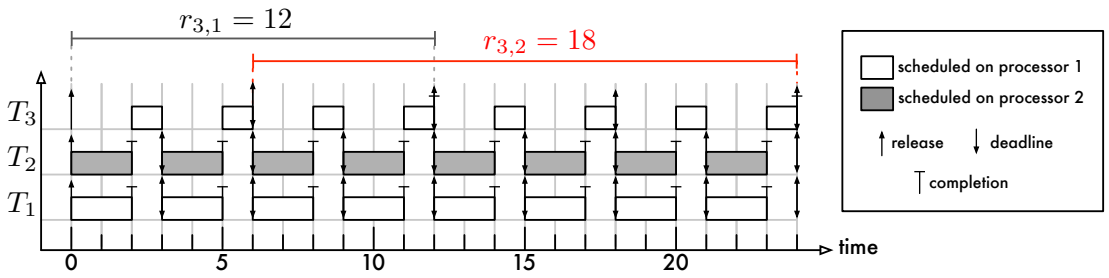


Figure 2.15: Devi's example of unbounded tardiness under G-FP scheduling with RM priorities (Devi, 2006). There are three tasks with parameters $e_1 = e_2 = 2$, $p_1 = p_2 = 3$, and $e_3 = 4$ and $p_3 = 6$. If all tasks release jobs periodically starting at time 0, all jobs of T_3 miss their deadlines and exhibit unbounded tardiness. This is because $u_3 = \frac{2}{3}$, but T_3 receives only an allocation of $\frac{1}{3}$ on average.

time test invocations[11] by assigning priorities in order from lowest to highest (Davis and Burns, 2011a). However, this approach can fail to find a schedulable priority assignment even if one exists since all known global response time bounds are pessimistic in the general case (worst-case response times may be overestimated). In the HRT case, priority assignment rules that yield higher utilization bounds than one (as RM does) have also been devised (*e.g.*, see Andersson, 2008). In the SRT case, besides Devi's original observation, G-FP scheduling has not received much attention. In principle, multiprocessor response time analysis for arbitrary-deadline tasks could be used to derive tardiness bounds, but to the best of our knowledge, this has not been studied in detail to date.

We do not consider G-FP scheduling in the remainder of this dissertation. As we shall demonstrate in Chapter 4, global scheduling is most compelling in the context of SRT constraints. In an SRT context, there appears to be little reason to favor G-FP over G-EDF, as G-FP does not ensure bounded tardiness in all cases (Devi, 2006), but G-EDF does (Devi and Anderson, 2005; Devi, 2006). We therefore focus on G-EDF instead, which we describe next.

2.3.3.2 Global Job-Level Fixed-Priority Scheduling

Other than in the uniprocessor case, global JLFP scheduling is not "better" at satisfying HRT constraints than global FP scheduling when confronted with "difficult" task sets. As mentioned above (page 71), the Dhall effect (Dhall and Liu, 1978) applies equally to both G-FP scheduling with RM priorities and G-EDF. Even a global JLFP scheduler that is optimal with respect to this class of

[11] Not all response-time tests are compatible with this approach; see (Davis and Burns, 2011a) for details.

Task	e_i	p_i	u_i		
T_1	3	10	$\frac{3}{10}$	$= \frac{819}{2730}$	$= 0.30$
T_2	2	7	$\frac{2}{7}$	$= \frac{780}{2730}$	≈ 0.29
T_3	1	5	$\frac{1}{5}$	$= \frac{546}{2730}$	$= 0.20$
T_4	3	9	$\frac{1}{3}$	$= \frac{910}{2730}$	≈ 0.33
T_5	5	13	$\frac{5}{13}$	$= \frac{1050}{2730}$	≈ 0.38
$u_{\text{sum}}(\tau)$				$\frac{4105}{2730}$	≈ 1.50
$u_{\text{max}}(\tau)$				$\frac{1050}{2730}$	≈ 0.38

Table 2.5: Example task set.

schedulers would fail to schedule Andersson *et al.*'s pathological example task set τ^{ABJ} consisting of $m + 1$ implicit-deadline tasks with parameters $e_i = 1 + \epsilon$ and $p_i = 2$ (Andersson *et al.*, 2001): as before, if all tasks release jobs at time 0, then the lowest-priority job would not start execution until time 1, at which point a deadline miss is unavoidable, assuming that job executes for a full $1 + \epsilon$ time units. Therefore, the utilization bound of any global JLFP scheduler is bounded by $\frac{m+1}{2}$ in the HRT case, which is no better than the worst-case utilization bounds of either partitioned or G-FP scheduling. However, this bound applies to partitioned and G-FP scheduling even in the SRT case, whereas there exists a family of global JLFP schedulers that are SRT optimal (Leontyev, 2010), that is, on m processors, they ensure bounded tardiness for any sporadic task set τ with $u_{\text{sum}}(\tau) \leq m$.

Similar to G-FP scheduling, neither an optimal global JLFP scheduler[12] nor an *exact* HRT feasiblity test for global JLFP scheduling is currently known. As EDF is in the uniprocessor case, G-EDF is the best-studied global JLFP scheduler, and it is also the one that we focus on in this dissertation. An example G-EDF schedule of the task set given in Table 2.5 is shown in Figure 2.16. We will use this task set as an example to demonstrate G-EDF schedulability analysis in the following discussion. We begin with HRT analysis.

Density test. Since G-EDF is subject to the Dhall effect, reasonable G-EDF schedulability tests must necessarily be based on task-set-specific parameters besides total utilization to identify task sets that are HRT schedulable. Key to the Dhall effect is the presence of high-utilization tasks (Phillips *et al.*, 1997, 2002). If the maximum utilization $u_{\text{max}}(\tau)$ is significantly less than one, then a higher utilization bound exists. The first to derive such a schedulability test were Goossens *et al.* (2003),

[12]That is, optimal with respect to its class.

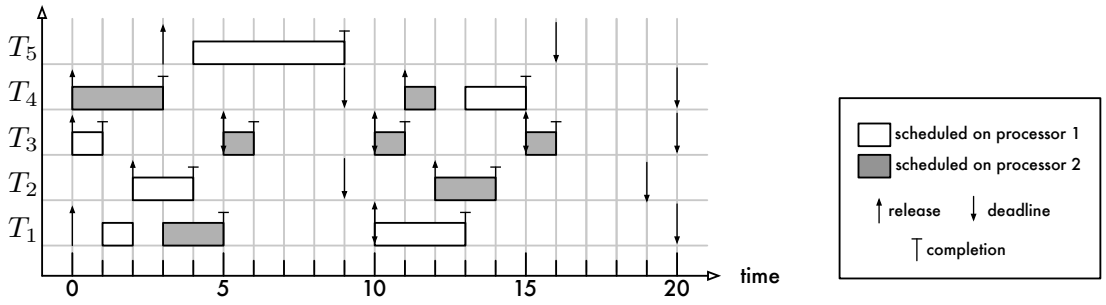


Figure 2.16: Example G-EDF schedule of the task set listed in Table 2.5. Two migrations occur in this example. $J_{1,1}$ is preempted at time 2 on processor 1 by $J_{2,1}$ and is scheduled again at time 3 on processor 2, after the higher-priority $J_{4,1}$ completed. The reverse occurs later in the schedule: at time 12 on processor 2, $J_{4,2}$ is preempted due to the release of $J_{2,2}$ and continues to execute after migrating to processor 1 at time 13. Note that $J_{4,2}$ and $J_{1,2}$ have the same absolute deadline $d_{4,2} = d_{1,2} = 20$, but that $J_{4,2}$ was preempted instead of $J_{1,2}$ because deadline ties are broken in favor of lower-indexed tasks.

who showed that a set of implicit-deadline periodic tasks τ is HRT schedulability under G-EDF if $u_{\text{sum}}(\tau) \leq m-(m-1)\cdot u_{\max}(\tau)$. Bertogna *et al.* (2005) and Baker and Baruah (2007) subsequently showed that this bound also applies to arbitrary-deadline sporadic task sets if density is substituted for utilization. The resulting schedulability criterion is now commonly referred to as the *density test*.

Theorem 2.7 (Goossens *et al.*, 2003; Bertogna *et al.*, 2005; Baker and Baruah, 2007)**.** *An arbitrary-deadline sporadic task set τ is HRT schedulable under G-EDF on m processors if*

$$\delta_{\text{sum}}(\tau) \leq m - (m-1) \cdot \delta_{\max}(\tau).$$

Example 2.7. The total utilization (and hence density) of the implicit-deadline task set given in Table 2.5 is $u_{\text{sum}}(\tau) = \frac{4105}{2730} \approx 1.5$. The maximum utilization is $u_{\max}(\tau) = \frac{1050}{2730}$. On a two-processor platform ($m = 2$), the density test yields the inequality

$$\frac{4105}{2730} \leq 2 - (2-1) \cdot \frac{1050}{2730} = \frac{5460}{2730} - \frac{1050}{2730} = \frac{4410}{2730} \approx 1.62,$$

which is true. The example task set is thus HRT schedulable under G-EDF on two processors. $\Diamond$

Baker's test. Shortly thereafter, Baker (2003) developed a novel proof strategy that has since enabled several key results, including the G-EDF HRT schedulability tests discussed in the remainder of

this section. From a high-level point of view, Baker's *problem window approach* (illustrated in Figure 2.17) consists of the following four steps. Let $J_{k,x}$ denote the first job to miss a deadline.

1. Identify a *problem window* immediately prior to $J_{k,x}$'s deadline miss. For example, a straightforward choice of problem window is the interval $[a_{k,x}, d_{k,x})$.

2. Analyze the problem window to derive a *necessary* condition for a deadline miss to occur. For example, $J_{k,x}$ will only miss its deadline if all m processors execute higher-priority jobs for more than $d_k - e_k$ time units during $[a_{k,x}, d_{k,x})$ (otherwise, $J_{k,x}$ would finish in time). In other words, more than $m \cdot (d_k - e_k)$ time units of higher-priority work were carried out during the problem window prior to $J_{k,x}$'s deadline miss.

3. Bound the maximum amount of higher-priority work processed during the problem window. There are two principle sources of higher-priority work: higher-priority jobs that are released during the problem window, and *carry-in* jobs that were already pending at the beginning of the problem window. Bounding the work due to jobs released during the problem window is relatively straightforward because sporadic tasks are rate-limited. Bounding carry-in work is possible since, by assumption, $J_{k,x}$ is the first job to miss its deadline; however, bounds on carry-in work are more difficult to obtain.

4. Relate both bounds to the necessary condition for a deadline miss from Step 2 to obtain a necessary "unschedulability test." Then invert the necessary "unschedulability test" to obtain a sufficient (but not necessary) HRT schedulability test.

Baker's technique has been hailed as a "seminal result" (Davis and Burns, 2011b) that has since served as the basis for several published schedulability results—see Davis and Burns and Bertogna and Baruah's recent surveys for a comprehensive overview (Davis and Burns, 2011b; Bertogna and Baruah, 2011). Besides the overall proof strategy, Baker's key insight was to extend the problem window to a point in time earlier than $a_{k,x}$. The benefit is that a longer interval results in reduced overestimation of higher-priority work, *i.e.*, in reduced pessimism. In the case of constrained and arbitrary deadlines, Baker's test can identify schedulable task sets that are missed by the density test. However, the converse is also true, *i.e.*, the tests are incomparable (Bertogna *et al.*, 2005). In the case

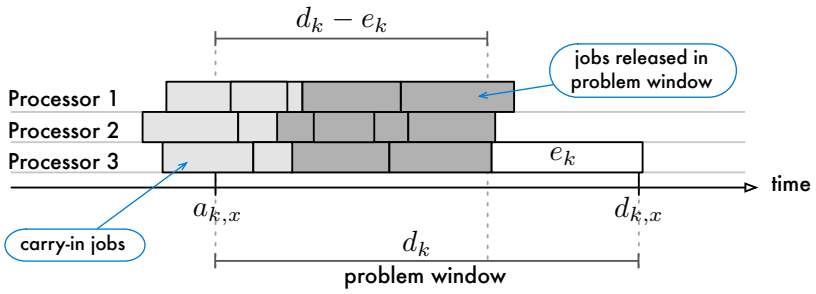


Figure 2.17: Illustration of Baker's problem window analysis (Baker, 2003) for $m = 3$ processors. In order for $J_{k,x}$ to miss its deadline at time $d_{k,x}$, all m processors must be busy executing higher-priority jobs for more than $(d_k - e_k)$ time units during the problem window. Interfering higher-priority jobs were either carried into or released during the problem window. There may exist up to n carry-in jobs.

of implicit deadlines, Baker's test is equivalent to the density test (Bertogna and Baruah, 2011). For this reason, Baker's actual schedulability test is not relevant to our main study (see Chapter 4).

Multiprocessor response-time analysis. Bertogna, Cirinei, and Lipari adopted Baker's approach and developed a series of subsequently less pessimsitic schedulability tests for constrained-deadline sporadic tasks (Bertogna *et al.*, 2005, 2009; Bertogna and Cirinei, 2007). We discuss Bertogna and Cirinei's *multiprocessor response-time analysis* (Bertogna and Cirinei, 2007) since it dominates the other two tests (Bertogna and Baruah, 2011). Like uniprocessor response-time analysis (see Section 2.3.1.1 above), Bertogna and Cirinei's test is an iterative process that is repeated until response-time bounds converge. In the uniprocessor case, the response-time bound of one task does not depend on the response-time bounds of other tasks (recall Example 2.4). That is, the individual response-time bounds are computed on a task-by-task basis and not reconsidered when the response times of other tasks become available. In the multiprocessor case, an outer loop is added that uses updated response-time bounds of *all tasks* to obtain better estimates in subsequent iterations of the outer loop. This is illustrated in Listing 2.1, where Lines 3–7 constitute the outer loop and Lines 5–6 the inner loop. In Listing 2.1 and the following discussion, let $R_i^{(l)}$ denote the response-time bound for T_i determined during the l^{th} iteration of the outer loop, and let $s_i^{(l)} \triangleq \max(0, d_i - R_i^{(l)})$ denote a lower bound on T_i's *slack* based on its l^{th} response-time estimate. Initially, assume $R_i^{(0)} = d_i$, which implies $s_i^{(0)} = 0$.

```
1  compute−response−times(τ, m):
2      set l ← 0
3      do
4          set l ← l + 1
5          foreach T_k ∈ τ:
6              set R_k^(l) using Theorem 2.8 using s_i^(l−1) for each T_i
7      while ∃ T_i ∈ τ such that s_i^(l−1) < s_i^(l)
8      foreach T_i ∈ τ:
9          set R_i ← R_i^(l)
```

Listing 2.1: Multiprocessor response-time analysis pseudo-code.

Let T_k denote the task for which the maximum response time is being bounded (Line 6 in Listing 2.1). As before, let $J_{k,x}$ denote the first job to miss a deadline. Bertogna and Cirinei analyzed the problem window $[a_{k,x}, d_{k,x})$ prior to $J_{k,x}$'s deadline miss. Two bounds on delay due to higher-priority jobs of each task T_i (where $i \neq k$) are crucial to their analysis. The first bound is concerned with how much processor time jobs of T_i can consume in the worst case during L time units (assuming a legal invocation sequence), which they called T_i's *workload* $W_i(L)$. A second notion of delay is the *interference* by jobs of T_i while $J_{k,x}$ is pending, which is denoted as I_i (where $T_i \neq T_k$). Workload and interference differ in that interference only reflects the jobs of T_i that have a higher priority than $J_{k,x}$, whereas the workload includes all of T_i's jobs in the analyzed interval. Both interference and workload yield upper bounds on the delay due to T_i experienced by $J_{k,x}$; to reduce pessimism, the minimum of the two is used in the computation of the response-time bound. Bertogna and Cirinei (2007) showed that a valid bound on T_i's workload $W_i(L)$ is given by

$$W_i(L) = \left\lceil \frac{L + d_i - e_i - s_i^{(l-1)}}{p_i} \right\rceil \cdot e_i + \min(e_i,\ L + d_i - e_i - s_i^{(l-1)})$$

during the l^{th} iteration of the outer loop. Note that $W_i(L)$ is independent of T_k and thus reflects neither $J_{k,x}$'s priority nor the scheduling algorithm in use. In contrast, the interference I_i reflects both. Bertogna and Cirinei showed that, under G-EDF, an upper bound on T_i's interference[13] is given by

$$I_i = \left\lfloor \frac{d_k}{p_i} \right\rfloor \cdot e_i + \min(e_i,\ d_k \bmod (p_i - s_i^{(l-1)})).$$

[13]This definition of I_i diverges slightly from the one given in (Bertogna and Cirinei, 2007). We use the simpler (and more recent) definition from (Bertogna and Baruah, 2011).

Based on the workload and interference bounds, Bertogna and Cirinei derived the following response-time bound.

Theorem 2.8 (Bertogna and Cirinei, 2007)**.** *Let τ denote a set of constrained-deadline sporadic tasks scheduled by G-EDF on m processors. Assuming no job misses its deadline, the maximum response time of task $T_k \in \tau$ is bounded by the smallest $R_k^{(l)}$ that satisfies*

$$R_k^{(l)} = e_k + \left\lfloor \frac{1}{m} \sum_{T_i \in \tau}^{T_i \neq T_k} \min\left(W_i(R_k^{(l)}), \quad I_i, \quad R_k^{(l)} - e_k + 1\right) \right\rfloor,$$

where $R_k^{(l)}$ is determined using a fixed-point iteration with the starting assumption $R_k^{(l)} = e_k$.

If the fixed-point iteration fails to converge for some T_k, *i.e.*, if $R_k^{(l)}$ exceeds d_k, then set $R_k^{(l)} = d_k + 1$. Due to the use of the outer loop, a non-converging response-time bound does not necessarily cause the task set to be claimed unschedulable. During subsequent iterations of the outer loop, improved slack estimates may be available for *other* tasks, which may then allow a response-time bound for T_k to be determined. Theorem 2.8, when used as illustrated in Listing 2.1, yields a safe upper bound on each task's maximum response time *if the task set is HRT schedulable*, *i.e.*, if $R_i \leq d_i$ for each T_i in Line 9 of Listing 2.1. The response-time bounds determined by Theorem 2.8 are not necessarily valid if jobs can be tardy (or if tasks have arbitrary deadlines); extended versions of multiprocessor response-time analysis should be used in this case (Leontyev and Anderson, 2008; Liu and Anderson, 2011).

In many cases, multiprocessor response-time analysis is less pessimistic than either the density test or Baker's test (Bertogna and Baruah, 2011). However, strictly speaking, multiprocessor response-time analysis is incomparable with both tests, that is, there exist task sets that are wrongly deemed unschedulable by Bertogna and Cirinei's analysis, but that pass the density test or Baker's test.

Example 2.8. Consider the task set from Example 2.7 as given in Table 2.5. During the first iteration of the outer loop ($l = 1$), applying Theorem 2.8 to each $T_k \in \{T_1, \ldots, T_5\}$ requires 16 fixed-point iterations and yields the following response-time bounds: $R_1^{(1)} = 10$, $R_2^{(1)} = 8$, $R_3^{(1)} = 6$, $R_4^{(1)} = 10$, and $R_5^{(14)} = 10$, which implies that $s_k^{(1)} = 0$ for each T_k. Since none of the computed response-time bounds improved over the initial assumption of $R_k^0 = d_k$, the outer loop terminates after the first

iteration. Since the resulting $R_k = R_k^{(1)}$ exceeds d_k for each task except T_1, the task set is wrongly claimed unschedulable. $\Diamond$

Baruah's test. The underlying reason for this pessimistic result is that Bertogna and Cirinei's response-time analysis overestimates the amount of carry-in work contributed by each task (recall that carry-in work is the time needed to service higher-priority jobs that were released prior to the beginning of the problem window). In fact, in Theorem 2.8, each task is considered to contribute carry-in work. Baruah (2007) showed that if the problem window is chosen carefully, then at most $m - 1$ tasks contribute carry-in demand. He proposed a schedulability test (based on Baker's proof technique) for constrained-deadline task sets that exploits this observation. Since his schedulability test also yields good results for implicit-deadline task sets, we briefly review it next.

Let $J_{k,x}$ denote the first job to miss its deadline at time $d_{k,x}$. A major source of uncertainty is that every task may have a job pending at time $a_{k,x}$, *i.e.*, every task may contribute carry-in demand. Baruah extended the problem window by considering the last point in time prior to $a_{k,x}$ at which at least one processor was idle or executing a job with lower priority (*i.e.*, a later deadline) than $J_{k,x}$ (or time zero if no such point in time exists prior to $a_{k,x}$). Let t_0 denote this point in time, and let A_k denote the length of the interval $[t_0, a_{k,x})$, *i.e.*, $a_{k,x} = t_0 + A_k$. This is illustrated in Figure 2.18.

The carefully chosen nature of t_0 allows two observations. First, at most $m - 1$ higher-priority jobs are pending at time t_0, and, second, all processors are busy executing higher-priority jobs during $[t_0, r_{k,x})$. Further, for $J_{k,x}$ to miss its deadline, processors must be executing higher-priority jobs for more than $(d_k - e_k) \cdot m$ time units during $[a_{k,x}, d_{k,x})$, and thus for more than $(A_k + d_k - e_k) \cdot m$ time units during $[t_0, d_{k,x})$. This observation corresponds to Step 2 of Baker's proof strategy.

The next step is to bound the amount of higher-priority work carried out during $[t_0, d_{k,x})$. As before, let I_i denote the interference due to T_i prior to $J_{k,x}$'s deadline miss. The upper bound on this interference depends on whether T_i has a carry-in job. Let I_i' denote a bound on interference due to T_i if T_i does *not* have a carry-in job, and let I_i'' denote a bound on interference due to T_i if it does carry in work into the analysis interval. Baruah derived the following bounds:

$$I_i' = \begin{cases} \min(W_i',\ L) & \text{if } i \neq k \\ \min(W_i' - e_k,\ A_k) & \text{if } i = k \end{cases} \qquad I_i'' = \begin{cases} \min(W_i'',\ L) & \text{if } i \neq k \\ \min(W_i'' - e_k,\ A_k) & \text{if } i = k \end{cases}$$

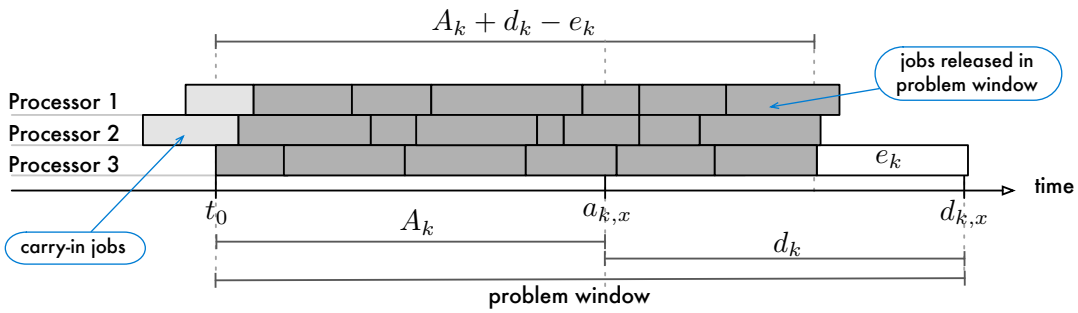


Figure 2.18: Illustration of Baruah's extended problem window (Baruah, 2007) for $m = 3$ processors. Baruah extended the problem window by A_k time units to the first point in time t_0 at which at least one processor is idle or processing a job of lower priority, where $t_0 \leq a_{k,x}$. In order for $J_{k,x}$ to miss its deadline at time $d_{k,x}$, all m processors must be busy executing higher-priority jobs for more than $(A_k + d_k - e_k)$ time units during the problem window. Due to the choice of t_0, there exist at most $m - 1$ carry-in jobs at time t_0.

where

$$L = A_k + d_k - (e_k - 1),$$
$$W_i' = \max\left(0, \quad \left(\left\lfloor \frac{A_k + d_k - d_i}{p_i} \right\rfloor + 1\right) \cdot e_i\right), \text{ and}$$
$$W_i'' = \left\lfloor \frac{A_k + d_k}{p_i} \right\rfloor \cdot e_i + \min(e_i, (A_k + d_k) \bmod p_i).$$

Unfortunately, the identity of the tasks that do have carry-in jobs is unknown in the general case, but due to the definition of t_0 there are at most $m - 1$ such tasks. Let CB denote a bound on the maximum amount of work that could have been carried in such that

$$\sum_{T_i \in \tau} I_i \leq CB + \sum_{T_i \in \tau} I_i'. \tag{2.1}$$

Let $I_i^{\text{diff}} \triangleq I_i'' - I_i'$ denote the difference between the two interference bounds for each task T_i. A safe upper bound CB is then given by the sum of the $m - 1$ largest values of I_i^{diff}.

Recall that a necessary condition for $J_{k,x}$ to miss a deadline is that the total interference $\sum_{T_i \in \tau} I_i$ exceeds $(A_k + d_k - e_k) \cdot m$ during $[t_0, d_{k,x})$. By inverting this necessary condition (Baker's Step 4) and by substituting the total interference using Equation (2.1), Baruah obtained the following sufficient HRT schedulability test.

Theorem 2.9 (Baruah, 2007). *A constrained-deadline task set τ is HRT schedulable under G-EDF on m processors if for all $T_k \in \tau$ and $A_k \geq 0$*

$$CB + \sum_{T_i \in \tau} I'_i \leq (A_k + d_k - e_k) \cdot m.$$

To obtain a finite testing set of A_k values, Baruah (2007) further showed that it is sufficient to test only values of A_k that satisfy

$$A_k \leq \frac{e_{\text{top}}(\tau, m-1) - d_k \cdot (m - u_{\text{sum}}(\tau)) + \sum_{T_i \in \tau}(p_i - d_i) \cdot u_i + m \cdot e_k}{m - u_{\text{sum}}(\tau)}. \tag{2.2}$$

An additional important reduction in the (average) computation cost of the test is due to his observation that it is sufficient to test only values for A_k that correspond to the points in time at which the value of W'_i changes for some T_i; formally, it is sufficient to test only those values of A_k that satisfy

$$A_k = d_i - d_k + j \cdot p_i$$

for some $T_i \in \tau$ and some $j \in \{0, 1, 2, \ldots\}$. Nonetheless, Baruah's test is of pseudo-polynomial time complexity, as the bound on the maximum A_k given in Equation (2.2) grows exponentially as the idle capacity, $m - u_{\text{sum}}(\tau)$, approaches zero. In practice, if there is only little idle capacity, the resulting runtime can become prohibitively large for task sets with several hundred tasks. In the case of the task set from Example 2.7, in total 39 values of A_k must be tested (across all tasks) before the task is correctly claimed to be schedulable on $m = 2$ processors. In general, Baruah's test is incomparable to both the density test and multiprocessor response-time analysis (Bertogna and Baruah, 2011). Notably, Theorem 2.9 is the only G-EDF schedulability test that generalizes Baruah *et al.*'s almost-exact uniprocessor EDF schedulability test (Baruah *et al.*, 1990) in the case of $m = 1$ (Baruah, 2007).

Sustainability. Our work presented in Chapters 3, 5, 6, and 7 requires sustainability with regard to the execution-cost parameter. Baker and Baruah (2009b) studied G-EDF sustainability and found that the G-EDF algorithm is sustainable with respect to execution costs and periods. With respect to G-EDF schedulability tests, the density test is obviously self-sustainable: a decrease in density

or utilization cannot cause a task set to fail the density test if it previously satisfied the bound. To the best of our knowledge, it has not been studied whether Bertogna and Cirinei's multiprocessor response-time analysis and Baruah's test are self-sustainable (Baker and Baruah, 2009b). As G-EDF is itself sustainable with respect to execution costs, this is of no concern to our experiments.

This concludes our review of the tests that we use to check HRT schedulability under G-EDF in this dissertation: the density test, Bertogna and Cirinei' multiprocessor response-time analysis, and Baruah's test. We note that several other HRT schedulability tests for G-EDF have been proposed in recent years. Besides earlier tests that are dominated by the aforementioned tests (Bertogna *et al*., 2005, 2009; Baker, 2003), there exist several tests that only offer advantages in the case of non-implicit deadlines. A number of these are based on the concept of *maximum load*, which is another characterization of a task set's processor demand (in addition to utilization and density). Maximum load is a more-accurate notion of processor demand for constrained-deadline task sets; however, it reduces to total utilization in the case of implicit-deadline task sets. In fact, the least-pessimistic load-based test (Baker and Baruah, 2009a) reduces to the density test for implicit deadline task sets (Bertogna and Baruah, 2011). Load-based tests are thus of limited relevance to the work presented herein. In other recent work, Baruah *et al*. (2010) proposed another pseudo-polynomial HRT schedulability test for G-EDF, which is on average much more computationally expensive than Baruah's test (Baruah, 2007) described above. The primary benefit of the new test is that it yields a provably good resource augmentation bound.[14] However, in a recent empirical comparison (Bertogna and Baruah, 2011), most task sets claimed schedulable by the test presented in (Baruah *et al*., 2010) were also claimed schedulable by one or more of the three tests that we employ. We thus do not use the (Baruah *et al*., 2010) test because our experimental setup (see Chapter 4) does not depend on a test's provable resource augmentation bound (and is not hindered by a lack thereof).

For a comprehensive overview of the state-of-the-art in schedulability tests for G-EDF, we refer the interested reader to (Davis and Burns, 2011b; Bertogna and Baruah, 2011; Baruah *et al*., 2010).

Bounded tardiness. G-EDF was the first JLFP scheduler for which it was shown that *any* sporadic task set with total utilization at most m has bounded tardiness on m processors (Devi and Anderson, 2005, 2008; Devi, 2006). Since then, this property has been shown to hold for a large class of

[14] A resource augmentation quantifies the pessimism of an algorithm or schedulability test—see (Davis and Burns, 2011b) for an introduction and survey.

schedulers, namely *window-constrained schedulers* (Leontyev and Anderson, 2010; Leontyev, 2010), under which each job is prioritized by a point in time that may vary at runtime but must remain within a constant-sized interval anchored at the job's release time (this includes JLDP schedulers). G-EDF is a prominent JLFP member of this class.

Showing that a scheduler ensures bounded tardiness differs from deriving an HRT schedulability condition for constrained-deadline tasks because tasks can potentially carry in multiple jobs worth of higher-priority work into an analysis interval. In other words, tardiness can have a cascading effect when tardy higher-priority jobs induce tardiness in later-arriving lower-priority jobs. Under G-EDF, higher-priority jobs may belong to any task; the tardiness bound of any one task thus depends on the tardiness bound of all other tasks.

Devi and Anderson (2005) solved this cyclic dependency using a novel proof strategy that relates G-EDF to a fluid *processor-sharing* (PS) schedule. In a *fluid schedule*, a processor can be allocated to more than one task at a time such that each task receives a fractional capacity from one or more processors. Such a schedule is clearly impossible on actual hardware, but it is interesting to consider because it trivially yields an HRT optimal multiprocessor scheduler for implicit-deadline task sets. In a PS schedule, this is achieved by allocating each pending job J_i exactly u_i processor capacity (at each point in time), which implies that each job completes after at most $\frac{e_i}{u_i} = p_i = d_i$ time units. Tasks with $d_i \geq p_i$ thus have zero tardiness in a PS schedule; jobs of constrained-deadline tasks are at most $p_i - d_i$ time units tardy. If it is possible to bound the extent to which a given algorithm deviates from a PS schedule, then it is thus also possible to bound maximum tardiness. This "extent of deviation" is called *lag* and denotes how much a task's cumulative processor allocation (up to a given time) diverges from the allocation that it would have received in an ideal PS schedule.

Definition 2.5. Let $\mathcal{A}$ denote a scheduling algorithm. With respect to a fixed legal invocation sequence of a sporadic task set τ, we denote the *total processor time* allocated to jobs of a task $T_i \in \tau$ during the interval $[0, t)$ in the $\mathcal{A}$-schedule for τ as $service(T_i, t, \mathcal{A})$.

Definition 2.6. With respect to a fixed legal invocation sequence of a sporadic task set τ, the *lag* of task $T_i \in \tau$ under a scheduling algorithm $\mathcal{A}$ at time t is given by

$$lag(T_i, t, \mathcal{A}) \triangleq service(T_i, t, \mathsf{PS}) - service(T_i, t, \mathcal{A}).$$

Since implicit-deadline tasks are never tardy in a PS schedule, a job that is tardy under G-EDF must necessarily have positive lag at the time of its deadline miss. Devi and Anderson (2005) used this fact to bound the maximum tardiness under G-EDF by bounding maximum lag. Specifically, their proof strategy consists of the following three steps. Let $J_{k,x}$ denote a job that misses its deadline at time $d_{k,x}$.

1. Assuming that $J_{k,x}$ is the first job to exceed the tardiness bound B given in Theorem 2.10 below, they compute an upper bound UB on the total lag of all tasks at time $d_{k,x}$.

2. Next, they compute a lower bound LB on the total lag that is necessary for $J_{k,x}$ to exceed the stated tardiness bound.

3. Finally, they showed that $UB < LB$, thereby contradicting the assumption that the tardiness bound B was exceeded by $J_{k,x}$.

Using this approach, Devi and Anderson showed that a task's maximum tardiness under G-EDF is limited by the following bound.

Theorem 2.10 (Devi and Anderson, 2005, 2008; Devi, 2006)**.** *Let τ denote an implicit-deadline sporadic task set scheduled under G-EDF on m processors. If $u_{\text{sum}}(\tau) \leq m$, then the maximum tardiness of each $T_i \in \tau$ is bounded by $e_i + B$, where*

$$B = \frac{e_{\text{top}}(\tau, \lceil u_{\text{sum}}(\tau) \rceil - 1) - e_{\text{min}}(\tau)}{m - u_{\text{top}}(\tau, \lceil u_{\text{sum}}(\tau) \rceil - 2)}.$$

Using the same proof strategy, Leontyev and Anderson first showed that a similar bound exists for global FIFO scheduling (where each job is prioritized by its release time) and then generalized the result to the class of all global window-constraint schedulers (Leontyev and Anderson, 2007, 2010; Leontyev, 2010), which includes both G-EDF and global FIFO scheduling. In recent work, Erickson *et al.* presented a polynomial-time iterative method to compute finer-grained, per-task bounds B_i to replace the coarser-grained bound B above (Erickson *et al.*, 2010a), and derived a tardiness bound for arbitrary-deadline task sets (Erickson *et al.*, 2010b). G-EDF is thus SRT optimal: any sporadic task set that does not over-utilize the system has bounded tardiness under G-EDF.

Task	e_i	p_i	u_i		
T_1	6	10	$\frac{3}{5}$	$= \frac{108}{180}$	$= 0.60$
T_2	2	9	$\frac{2}{9}$	$= \frac{40}{180}$	≈ 0.22
T_3	1	5	$\frac{1}{5}$	$= \frac{36}{180}$	$= 0.20$
T_4	3	9	$\frac{1}{3}$	$= \frac{60}{180}$	≈ 0.33
T_5	7	12	$\frac{7}{12}$	$= \frac{105}{180}$	≈ 0.58
$u_{\text{sum}}(\tau)$				$\frac{349}{180}$	≈ 1.93

Table 2.6: Example task set with two high-utilization tasks (T_1, T_5).

Summary. We consider G-EDF as the representative JLFP scheduler in this dissertation because it **(i)** uses a simple, easy-to-implement priority rule, **(ii)** is well-supported by existing HRT schedulability analysis, and **(iii)** is SRT optimal, as discussed above. Note that (iii) does not imply that G-EDF ensures *minimal* tardiness. In fact, for implicit-deadline sporadic task sets, minimal achievable tardiness is zero since there exist global JLDP schedulers that are HRT optimal,[15] which we discuss next.

2.3.3.3 Global Job-Level Dynamic-Priority Scheduling

To motivate why dynamic job priorities are required for HRT optimality, consider a modified version of the task set from Example 2.7 where the utilizations of T_1 and T_5 have been increased (and the utilization of T_2 has been decreased to compensate). The resulting task set is listed in Table 2.6. A possible G-EDF schedule of the task set is shown in Figure 2.19. Here, each task releases jobs in a pattern so that all tasks have a job deadline at time 12 (T_3 releases two jobs). As a result, the order of the pending jobs (except for $J_{3,1}$) is determined by the tie-breaking rule.[16] Consequently, the job of the lowest-indexed task T_5 is only scheduled when there are fewer than $m = 2$ jobs of higher-indexed tasks pending. Since this is only the case before time 2 and after time 8, $J_{5,1}$ has only six time units before its deadline at time $d_{5,1} = 12$ to complete $e_5 = 7$ time units worth of work. A deadline miss results; the task set is thus not HRT schedulable under G-EDF. The root cause for this failure is that G-EDF's priority rule only reflects *urgency* and not *parallelism*, or rather the

[15] Very recent work indicates that G-EDF does not ensure minimal tardiness even within the class of JLFP schedulers (Erickson and Anderson, 2011).

[16] We note that the deadline miss seen in the figure does not depend on the choice of tie-breaking rule. A similar example can be constructed for any tie-breaking rule. For example, $J_{5,1}$'s release time could be moved to time 1.

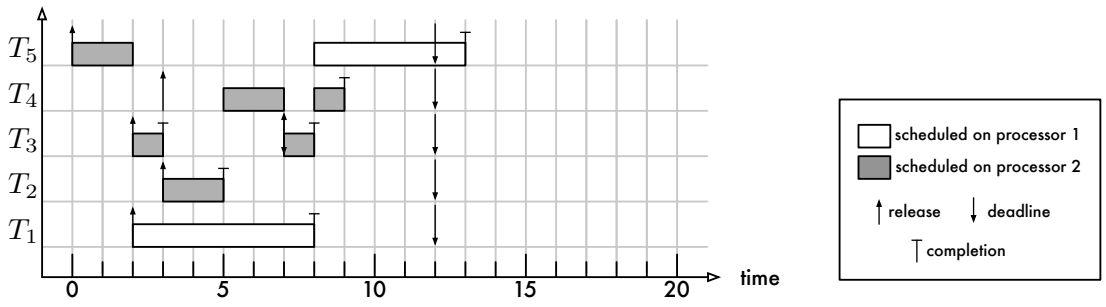


Figure 2.19: Example G-EDF schedule of the task set listed in Table 2.6. If jobs are released as shown, $J_{5,1}$ misses its deadline at time $d_{5,1} = 12$ because an excessive amount of its execution requirement has been deferred in favor of higher-priority jobs.

lack thereof due to the sporadic task model's sequentiality constraint. This is evident in the interval $[9, 12)$ in Figure 2.19, during which processor 2 is idle. Used together, both processors could have completed six time units of work, but since $J_{5,1}$ can only execute on one processor at a time, it could only use half of the processing capacity available during this interval. If the scheduler had delayed $J_{4,1}$'s first allocation by only one time unit (*i.e.*, if $J_{4,1}$'s first allocation would have been scheduled during $[6, 7)$ instead of during $[5, 6)$), then all jobs would have met their deadlines. This, however, would have required dynamically lowering $J_{4,1}$'s priority. The key to designing an HRT optimal multiprocessor scheduler is thus to be cognizant of each job's sequentiality constraint (in addition to its deadline).

Quantum-based schedulers. Before discussing an optimal JLDP scheduler, a brief digression concerning the scheduling model is required. So far, we have only discussed event-driven scheduling, where a scheduler reacts to scheduling events (such as job releases and completions) instantaneously (in the absence of overheads). As previously remarked on page 57, an alternate approach is *quantum-driven scheduling*, where the scheduler is only invoked at integer multiples of a scheduling quantum Q. The time between integer multiples of Q is called a *slot* or *quantum*; the points in time separating quanta are called *quantum boundaries*. For example, if $Q = 10$, then rescheduling would only take place at times 0,10,20,30,... and the first slot corresponds to the interval $[0, 10)$.

A benefit of quantum-driven scheduling is that it is somewhat easier to implement since the scheduler is only invoked at fixed, pre-determined points in time (*i.e.*, each quantum boundary). Further, in the context of JLDP schedulers, quantum-driven scheduling offers the advantage of

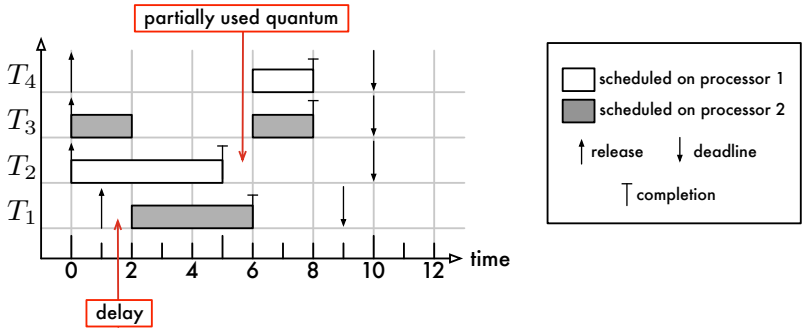


Figure 2.20: Illustration of a delayed preemption and an only partially used quantum under quantum-driven scheduling for $Q = 2$. The scheduler is only invoked at quantum boundaries, which occur at integer multiples of two. When $J_{1,1}$ is released at time 1, it *should* preempt the lower-priority job $J_{3,1}$ on processor 2. However, the scheduler is not invoked until time 2; $J_{1,1}$ suffers an additional delay as a consequence. When $J_{2,1}$ completes at time 5, processor 1 is idled until time 6 when the scheduler is finally invoked to reschedule; as result, the quantum $[4, 6)$ is only partially used on processor 1 despite the fact that pending jobs exist at the time. In theory, such effects become negligible as $Q \to 0$.

clearly defined points at which priority changes are relevant as any changes in job priority are only enacted at quantum boundaries; it is thus sufficient if a job's priority is well-defined only at these points. A negative side effect of quantum-based scheduling is that individual quanta may only be partially used (*i.e.*, if a job completes early) and that newly-released jobs may not be scheduled until up to Q time units after their release, even if they have one of the m highest priorities at the time of their release. Both effects are illustrated in Figure 2.20. Nonetheless, at least in theory, capacity lost to such delays can be considered negligible since the quantum size Q can be chosen to be sufficiently small to avoid any negative effects. In practice, the minimum viable quantum size is constrained by the resolution of hardware timers and efficiency considerations; this is discussed in Chapter 3. When dealing with a quantum-based scheduler (and, in particular, in the remainder of this section), we require that all task parameters are integer multiples of the scheduling quantum, *i.e.*, for each e_i, p_i, and d_i, there exist $e'_i, p'_i, d'_i \in \mathbb{N}$ such that $e_i = e'_i Q$, $p_i = p'_i Q$, and $d_i = d'_i Q$. Since each e_i, p_i, and d_i are integer as well (see Section 2.2), this assumption is trivially satisfied for a sufficiently small Q. For simplicity, we assume $Q = 1$ in the following discussion (*e.g.*, e_i denotes the number of quanta require by jobs of T_i).

Pfairness. Based on the quantum-based scheduling model, Baruah *et al.* (1996) proposed *proportionate fair* (or *pfair*) scheduling and designed the first (non-fluid) global multiprocessor real-time

scheduler, named PF, that is HRT optimal for implicit-deadline periodic tasks. Pfair scheduling was later extended to more general task models, including a generalization of the sporadic task model (Srinivasan and Anderson, 2002, 2006). In the following, we review pfair scheduling as it applies to sporadic tasks; see (Srinivasan, 2003; Holman, 2004; Devi, 2006) for an in-depth treatment of the subject.

An implicit-deadline task T_i does not place any constraints on job execution besides that a job J_i must sequentially receive up to e_i quanta of processor time within a *scheduling window* of length p_i. Specifically, it does not constrain when the processor time may be allocated, *e.g.*, a job J_i may be scheduled for e_i consecutive time units immediately after its release or just before its deadline. Baruah *et al*. (1996) showed that HRT optimality can in fact be achieved by imposing *additional* constraints on the allocation of a job's processor service. Recall from Definition 2.6 that a task's lag describes the extent to which its processor allocation up to a point in time t deviates from a fluid PS schedule (in which each pending job J_i is allocated exactly u_i processor capacity at any point in time). Baruah *et al*. introduced the following constraint on job execution.

Definition 2.7. Let τ denote a feasible implicit-deadline task set. A scheduling algorithm $\mathcal{A}$ is *pfair* if and only if for all tasks $T_i \in \tau$ and times t

$$-1 < lag(T_i, \mathcal{A}, t) < 1. \tag{2.3}$$

In other words, in a pfair schedule, a task's cumulative allocation deviates by strictly less than one quantum from a fluid PS schedule at any point in time. For implicit-deadline tasks, this immediately implies HRT correctness, as can be seen with a simple argument. Suppose a job $J_{i,j}$ is the first to miss its deadline (at time $d_{i,j}$) under a pfair scheduler $\mathcal{A}$. In a quantum-based scheduler, this implies that $J_{i,j}$ has received at most $e_i - 1$ quanta during $[a_{i,j}, d_{i,j})$. Because the implicit deadline implies that $J_{i,j-1}$ (if $j > 1$) must have completed by $a_{i,j}$ if $J_{i,j}$ is the first job to miss a deadline, it follows that both PS and $\mathcal{A}$ must have scheduled T_i for equal amounts of time prior to $a_{i,j}$, *i.e.*, $service(T_i, a_{i,j}, \mathcal{A}) = service(T_i, a_{i,j}, \mathsf{PS})$. In a fluid PS schedule, $J_{i,j}$ would have received

$(d_{i,j} - a_{i,j}) \cdot u_i = p_i \cdot u_i = e_i$ time units of processor time during $[a_{i,j}, d_{i,j})$. This implies that

$$\begin{aligned} lag(T_i, d_{i,j}, \mathcal{A}) &= service(T_i, d_{i,j}, \mathsf{PS}) - service(T_i, d_{i,j}, \mathcal{A}) \\ &\geq (service(T_i, a_{i,j}, \mathsf{PS}) + e_i) - (service(T_i, a_{i,j}, \mathcal{A}) + e_i - 1) \\ &= service(T_i, a_{i,j}, \mathsf{PS}) - service(T_i, a_{i,j}, \mathcal{A}) + 1 \\ &= 1, \end{aligned}$$

which contradicts Equation (2.3). Any pfair scheduler is thus HRT optimal with respect to implicit deadlines.

Under a quantum-based scheduler, a job J_i can be thought of as consisting of (up to) e_i quantum-sized allocations. Anderson and Srinivasan termed these allocations *subtasks* (Anderson and Srinivasan, 2004). We denote the k^{th} subtask of a job $J_{i,j}$ as $J_{i,j,k}$. In addition to implying HRT correctness, the pfair constraint also imposes an interval for each subtask $J_{i,j,k}$, called its *pfair window*, during which $J_{i,j,k}$ must be scheduled. The start and end of a subtask's pfair window is given by its *pseudo-release time* $r_{i,j,k}$ and *pseudo-deadline* $d_{i,j,k}$, respectively. These times are defined as

$$r_{i,j,k} = a_{i,j} + \left\lfloor \frac{k-1}{u_i} \right\rfloor \qquad \text{and} \qquad d_{i,j,k} = a_{i,j} + \left\lceil \frac{k}{u_i} \right\rceil.$$

In other words, $J_{i,j}$ must have received at least k time units of processor time by $d_{i,j,k}$, but not earlier than $r_{i,j,k} + 1$. This constraint is implied by Equation (2.3): suppose $J_{i,j}$ has received only $k-1$ quanta of processor service by $d_{i,j,k}$ under a pfair scheduler $\mathcal{A}$. In a fluid PS schedule, $J_{i,j}$ would have received exactly $(d_{i,j,k} - a_{i,j}) \cdot u_i = \left\lceil \frac{k}{u_i} \right\rceil \cdot u_i \geq k$ time units by then, which implies that $lag(T_i, d_{i,j,k}, \mathcal{A}) \geq 1$ at time $d_{i,j,k}$. An analogous argument yields the definition of $r_{i,j,k}$.

Example 2.9. Table 2.7 lists the (relative) pseudo-release times and pseudo-deadlines of the high-utilization task set given in Table 2.6 (as well as two values, the "successor bit" and "group deadline," which are discussed below); the parameters are also illustrated in Figure 2.21. For example, consider the pfair windows of subtasks of $J_{5,1}$ in comparison to the G-EDF schedule depicted in Figure 2.19. Under a pfair scheduler, $J_{5,1,3}$ must be scheduled some time during $[3, 6)$, whereas $J_{5,1}$ received its third quantum of processor service only during $[8, 9)$ in Figure 2.19. Under a pfair scheduler, the

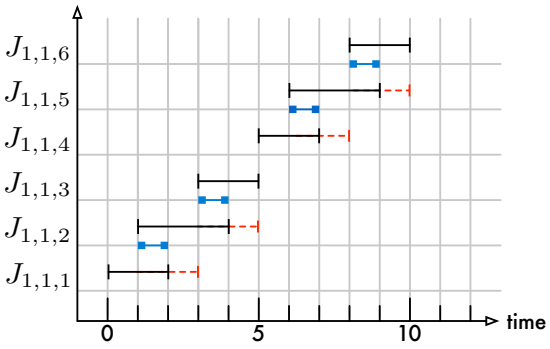
Task T_i	util. u_i	subtask k	pseudo-release $r_{i,j,k} - a_{i,j}$	pseudo-deadline $d_{i,j,k} - a_{i,j}$	successor bit $bbit(J_{i,j,k})$	group deadline $gdl(J_{i,j,k}) - a_{i,j}$
T_1	$\frac{3}{5}$	1	0	2	1	3
		2	1	4	1	5
		3	3	5	0	5
		4	5	7	1	8
		5	6	9	1	10
		6	8	10	0	10
T_2	$\frac{2}{9}$	1	0	5	1	0
		2	4	9	0	0
T_3	$\frac{1}{5}$	1	0	5	0	0
T_4	$\frac{1}{3}$	1	0	3	0	0
		2	3	6	0	0
		3	6	9	0	0
T_5	$\frac{7}{12}$	1	0	2	1	3
		2	1	4	1	5
		3	3	6	1	8
		4	5	7	1	8
		5	6	9	1	10
		6	8	11	1	12
		7	10	12	0	12

Table 2.7: Subtask parameters of the task set given in Table 2.6. Note that pseudo-release, pseudo-deadline, and group deadline times are given relative to the job's release time $a_{i,j}$. See Figure 2.21 for a visual representation of these parameters.

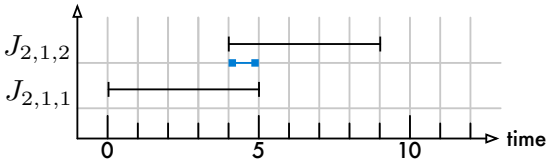
earlier pseudo-deadlines would have ensured that $J_{5,1}$'s execution is not deferred until it is "too late."

$\Diamond$

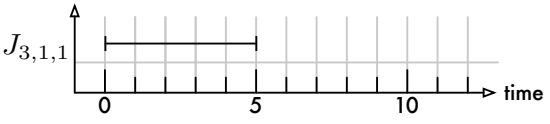
Algorithm PD^2. To this point, we have discussed only the desirable properties of pfair schedules; however, we have not specified how to obtain such a schedule. While it is relatively straightforward to compute a pfair schedule offline, it is a considerable challenge to design a succinct set of rules to order pending (*i.e.*, pseudo-released) subtasks at runtime such that a pfair schedule results. While tempting, simply scheduling all subtasks by pseudo-deadline does in fact not yield an HRT optimal scheduler, though it does ensure bounded tardiness (Srinivasan and Anderson, 2003, 2005). This is because pseudo-deadlines still do not adequately encode the "sequentiality" of a task; they merely reflect its weight (*i.e.*, high-utilization tasks have short pfair windows). Baruah *et al*. designed two pfair algorithms, the original **p**roportionate-**f**air algorithm PF (Baruah *et al*., 1996) and the later PD algorithm based on **p**seudo-**d**eadlines (Baruah *et al*., 1995). PD used a relatively complex priority



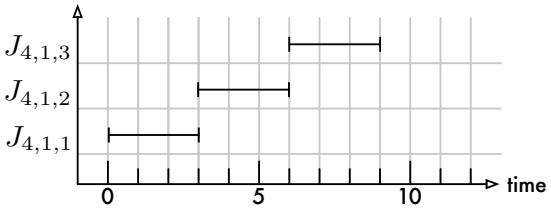
(a) $J_{1,1}$'s subtask parameters for $a_{1,1} = 0$.



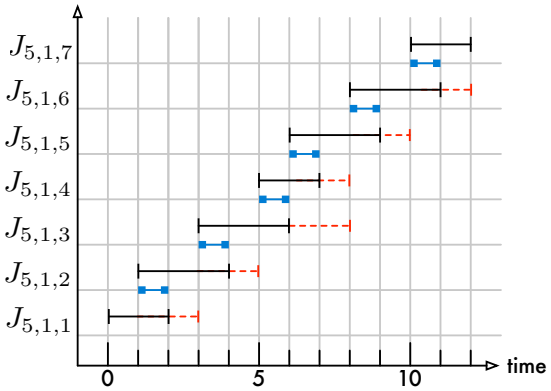
(b) $J_{2,1}$'s subtask parameters for $a_{2,1} = 0$.



(c) $J_{3,1}$'s subtask parameters for $a_{3,1} = 0$.



(d) $J_{4,1}$'s subtask parameters for $a_{4,1} = 0$.



(e) $J_{5,1}$'s subtask parameters for $a_{5,1} = 0$.

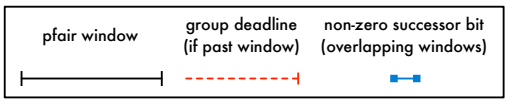


Figure 2.21: Illustration of the subtasks of the first job of each task assuming it is released at time 0. The corresponding task parameters are listed in Table 2.6; the subtask parameters are listed in Table 2.7.

rule and distinguishes between a total of seven subtask classes (such as "urgent," "heavy," "light," *etc.*). Srinivasan and Anderson later found a much simpler variant called PD^2 that only requires two pseudo-deadline tie-breaking rules (Srinivasan and Anderson, 2002, 2006; Anderson and Srinivasan, 2004). Of the three pfair algorithms, PD^2 is the most efficient since it requires the fewest number of tie-breaking rules, both of which can be evaluated in constant time (or easily pre-computed). We therefore only consider PD^2 in this dissertation.

As its predecessor, PD^2 schedules pending subtasks in order of non-increasing pseudo-deadlines. However, the key to its optimality is how it chooses among subtasks with equal pseudo-deadlines. Intuitively, PD^2 follows a simple rule: choose subtasks such that future scheduling decisions are the least constrained possible. For example, consider the subtasks of $J_{1,1}$, assuming $J_{1,1}$ is released at time $a_{1,1} = 0$ as shown in Figure 2.22. Further, suppose $J_{1,1,1}$ is not scheduling during $[0, 1)$, but

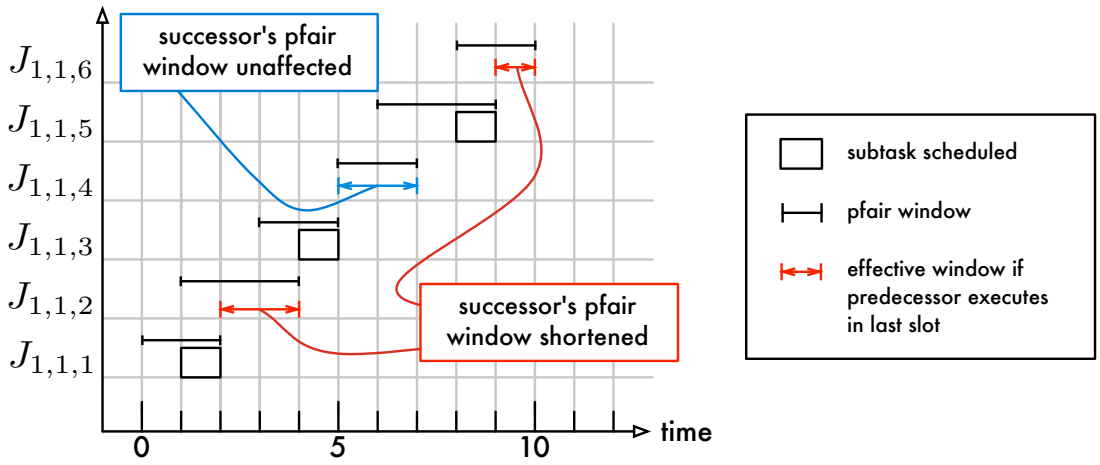


Figure 2.22: Illustration of constrained pfair windows. This figure shows the pfair windows of $J_{1,1}$'s subtasks assuming that $a_{1,1} = 0$ (as in Figure 2.21). If $J_{1,1,1}$ and $J_{1,1,6}$ are scheduled in their latest-possible slot, then the effective scheduling window of their respective successor subtasks, $J_{1,1,2}$ and $J_{1,1,6}$, are additionally constrained by the sequentiality requirement of the sporadic task model. In contrast, $J_{1,1,3}$'s pfair window does not overlap with the pfair window of its successor, so it can be scheduled in its latest-possible slot without impeding $J_{1,1,4}$. Overlapping pfair windows are indicated by the successor bit in Figure 2.21.

only during the later slot $[1, 2)$. Due to the sequentiality of sporadic tasks, this constrains $J_{1,1,2}$ to a reduced effective window of $[2, 4)$ instead of its regular pfair window of $[1, 4)$. Similarly, if $J_{1,1,5}$ is scheduled only during $[8, 9)$, the latest-possible legal allocation time, then $J_{1,1,6}$ *must* be scheduled during $[9, 10)$. In contrast, $J_{1,1,3}$ can be safely delayed until its last slot without affecting the pfair window of its successor subtask. These examples show the impact of overlapping pfair windows: delaying one of the earlier subtasks of a task with overlapping pfair windows can have a cascading effect that forces later allocations. PD^2 avoids this by favoring subtasks that have a pfair window that overlaps with their successor's pfair window. Formally, a subtask's *successor bit*, also referred to as the *boundary bit* by Devi (2006), is defined as

$$bbit(J_{i,j,k}) \triangleq \left\lceil \frac{k}{u_i} \right\rceil - \left\lfloor \frac{k}{u_i} \right\rfloor .$$

The successor bit of a subtask $J_{i,j,k}$ is one if $J_{i,j,k}$'s pfair window overlaps with the pfair window of $J_{i,j,k+1}$ (if $k < e_i$), and zero otherwise. Under PD^2, if two subtasks have equal pseudo-deadlines but differ in their successor bits, then PD^2 assigns a higher priority to the one with a non-zero successor

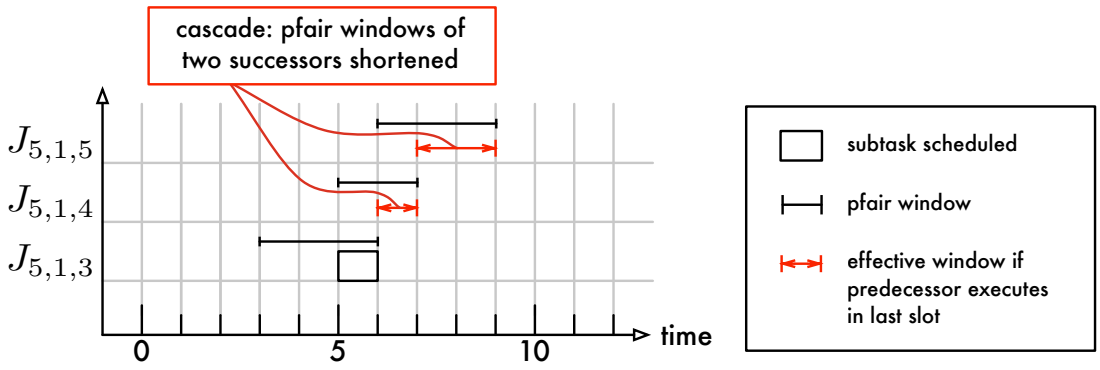


Figure 2.23: Illustration of multiple constrained pfair windows. This figure shows the pfair windows of the subtasks $J_{5,1,3}$, $J_{5,1,4}$, and $J_{5,1,5}$ assuming that $a_{5,1} = 0$ (as in Figure 2.21). If $J_{5,1,3}$ is scheduled in the latest-possible slot, then the effective scheduling window of both its immediate successor $J_{5,1,4}$ and the later successor $J_{5,1,5}$ are constrained. In other words, delaying $J_{5,1,3}$ until the latest-possible slot triggers a cascade of additional constraints that extends until time 8, which is $J_{5,1,5}$'s group deadline.

bit. (If both subtasks have a zero successor bit, then the tie in pseudo-deadlines can be broken arbitrarily.)

This leaves the case of choosing among two (or more) subtasks with equal pseudo-deadlines and non-zero successor bits. Here, PD^2 again follows the intuition to constrain future allocations as little as possible. While all subtasks with a non-zero successor bit constrain their successor subtasks when delayed to their last legal slot, they differ with regard to the "length" of the resulting cascade. For example, consider the example shown in Figure 2.23. If $J_{5,1,3}$ from Figure 2.21 is scheduled in its latest legal slot $[5, 6)$, then this forces $J_{5,1,4}$ to be scheduling during $[6, 7)$, which in turns reduces the length $J_{5,1,5}$'s effective scheduling window. In other words, if $J_{5,1,3}$ is delayed until the latest-possible moment, then *two* subsequent subtasks have implicitly been constrained as well. In contrast, the pfair windows of $J_{1,1,1}$ and $J_{1,1,2}$ overlap, but $J_{1,1,3}$ retains some flexibility even if $J_{1,1,1}$ is not scheduled until $[1, 2)$. PD^2 breaks ties in favor of subtasks that would cause a "long cascade" when postponed. As it turns out, long cascades are only possible for tasks that have overlapping pfair windows of length two or three, which is only the case for tasks with $u_i > \frac{1}{2}$. The notion of "cascade

length" is formalized by the *group deadline* of a subtask, which is defined as

$$gdl(J_{i,j,k}) \triangleq a_{i,j} + \begin{cases} \left\lceil \frac{\left\lceil \frac{k}{u_i} \right\rceil - k}{1 - u_i} \right\rceil & \text{if } 1 > u_i > \frac{1}{2}, \\ 0 & \text{if } u_i \leq \frac{1}{2}. \end{cases} \tag{2.4}$$

PD^2 breaks ties in favor of subtasks with *later* group deadlines (*i.e.*, longer cascades). Note that Equation (2.4) is not defined for the case $u_i = 1$. This is because tasks with a weight of one require a dedicated processor and hence are effectively not part of the scheduling problem, that is, a task with $u_i = 1$ can be allocated a processor exclusively without loss of optimality. We refer the interested reader to Srinivasan (2003) and Devi (2006) for an in-depth discussion and derivation of Equation (2.4).

Early-releasing. Under the pfair constraint (Equation (2.3)), subtasks have a pseudo-release time $a_{i,j,k}$ prior to which they cannot be scheduled (by a pfair scheduler). However, in an actual RTOS, subtasks are merely an accounting abstraction that is used to determine the current priority of a job (which itself is typically an abstraction backed by a process). Enforcing pseudo-relase times, while certainly possible, creates additional overheads because "waiting" subtasks would have to be merged into the ready queue when they become eligible to be scheduled. Further, if the system is (temporarily) lightly loaded, enforcement of pseudo-release times can result in non-work-conserving schedules, that is, processors may be idled even though there exist processes that could be dispatched. Luckily, Anderson and Srinivasan showed that PD^2 remains HRT optimal for implicit-deadline sporadic tasks even if pseudo-release times are not enforced (Anderson and Srinivasan, 2000, 2004). The resulting pfair variant is called *early-release pfair*, or *ERfair* in short. With early-releasing, a subtask is eligible to be scheduled as soon as its predecessor subtask (if any) has been completed, but the definitions of pseudo-deadline, successor bit, and group deadline remain unchanged. The pfair constraint is relaxed to allow arbitrary negative lag (negative lag implies that a task is ahead of its "fair" allocation); formally, ERfairness requires of a scheduling algorithm $\mathcal{A}$ only that, for all tasks T_i

and times t,

$$lag(T_i, t, \mathcal{A}) < 1. \tag{2.5}$$

In practical terms, early-releasing is desirable because it turns PD^2 into a work-conserving scheduler. Since we are not aware of any reason to favor a non-work-conserving scheduler in an actual RTOS, we only consider PD^2 with early-releasing in the remainder of this dissertation.

PD^2 priority. With PD^2's work-conserving behavior restored, it fits within the framework of priority-driven scheduling. Based on PD^2's two tie-breaking rules, we can define a prioritization function for PD^2 as follows. Let $J_{i,j}$ denote a job pending at time t, and let $J_{i,j,k}$ denote $J_{i,j}$'s pending subtask at time t, *i.e.*, $J_{i,j}$ has been scheduled for $k-1$ quanta prior to time t. Under PD^2, $J_{i,j}$'s priority is given by a four-element tuple

$$\mathsf{Y}(J_{i,j}, t) = (d_{i,j,k},\ bbit(J_{i,j,k}),\ gdl(J_{i,j,k}),\ i)\,,$$

with the interpretation that $\mathsf{Y}(J_{i,j}, t) < \mathsf{Y}(J_{x,y}, t)$ if and only if

$$\begin{aligned}
&(d_{i,j,k} < d_{x,y,z}) && \{\text{earlier pseudo-deadline}\}\\
\vee\ &(d_{i,j,k} = d_{x,y,z} && \{\text{tie-break 1}\}\\
&\wedge\ bbit(J_{i,j,k}) > bbit(J_{x,y,z}))\\
\vee\ &(d_{i,j,k} = d_{x,y,z} \wedge bbit(J_{i,j,k}) = bbit(J_{x,y,z}) = 1 && \{\text{tie-break 2}\}\\
&\wedge\ gdl(J_{i,j,k}) > gdl(J_{x,y,z}))\\
\vee\ &(d_{i,j,k} = d_{x,y,z} \wedge bbit(J_{i,j,k}) = bbit(J_{x,y,z}) = 1 && \{\text{tie-break 3}\}\\
&\wedge\ gdl(J_{i,j,k}) = gdl(J_{x,y,z}) \wedge i < x)\\
\vee\ &(d_{i,j,k} = d_{x,y,z} \wedge bbit(J_{i,j,k}) = bbit(J_{x,y,z}) = 0 && \{\text{tie-break 4}\}\\
&\wedge\ i < x),
\end{aligned}$$

where $J_{x,y,z}$ denotes $J_{x,y}$'s pending subtask at time t. The first two tie-breaks are essential to PD^2's correctness; any remaining priority ties may be broken "arbitrarily" from a correctness point of view.

Here, the third and fourth tie-break ensure that any remaining ties are consistently broken in favor of lower-indexed tasks.

Example 2.10. Figure 2.24 depicts a PD^2 schedule of the task set given in Table 2.6. The pfair window and group deadline for each subtask are shown in Figure 2.24 above each job; all relevant subtask parameters for each task can be found in Table 2.7 and Figure 2.21. Recall from Figure 2.19 that $J_{5,1}$ misses its deadline when scheduled under G-EDF. Under PD^2, this is avoided because the subtasks of $J_{5,1,1}$ have a comparatively high priority since T_5's high utilization results in short pfair windows.

At time 0, $J_{5,1}$ is released. The first two subtasks of $J_{5,1}$ are scheduled automatically since there is no contention. However, $J_{5,1}$ remains scheduled even at time 2 when $J_{1,1}$ and $J_{3,1}$ are released. This is because $J_{5,1,3}$ (the currently pending subtask of $J_{5,1}$) has an earlier pseudo-deadline at time 6, whereas $J_{3,1}$'s only subtask is not due until time $d_{3,1} = d_{3,1,1} = 7$. In contrast, under G-EDF in Figure 2.19, $J_{3,1}$ preempts $J_{5,1}$ on arrival.

At time 3, $J_{4,1}$ and $J_{2,1}$ are released. $J_{4,1,1}$ has an earlier pseudo-deadline (at time 6) than $J_{5,1,4}$ (at time 7); $J_{5,1}$ is thus preempted by $J_{4,1}$. $J_{1,1}$ remains scheduled because its current subtask, $J_{1,1,2}$, has an equally early pseudo-deadline at time 6.

At time 4, the tie-breaking rules take effect for the first time in this example. This is because there are three subtasks ($J_{1,1,3}$, $J_{3,1,1}$, and $J_{5,1,4}$) with equal pseudo-deadline at time 7. Of these three subtasks, only $J_{5,1,4}$ has a non-zero successor bit; it thus has the highest priority and is scheduled. Since $bbit(J_{1,1,3}) = bbit(J_{3,1,1}) = 0$, the tie is resolved according to tie-break 4, which favors the lower-indexed task's subtask, which is $J_{1,1,3}$ in this case.

At time 5, no tie-breaks are required as there are two subtasks that have earlier pseudo-deadlines than the rest of the pending subtasks; $J_{2,1,1}$ and $J_{3,1,1}$ are thus scheduled. However, a tie-break is again required at time 6 as $J_{1,1,4}$, $J_{4,1,2}$, and $J_{5,1,5}$ each have a pseudo-deadline at time 9. Of these, only $J_{1,1,4}$ and $J_{5,1,5}$ have a non-zero successor bit, so $J_{4,1,2}$ is deferred. In contrast, $J_{5,1}$ is the *lowest-priority* pending job at time 6 in the corresponding G-EDF schedule (Figure 2.19).

All later pseudo-deadline ties in Figure 2.24 are resolved in favor of lower-indexed tasks. Under PD^2, all jobs meet their deadlines. In fact, when comparing Figure 2.24 to Figure 2.19, one can observe that the schedule length (*i.e.*, the time until all jobs have completed) is shorter by two time

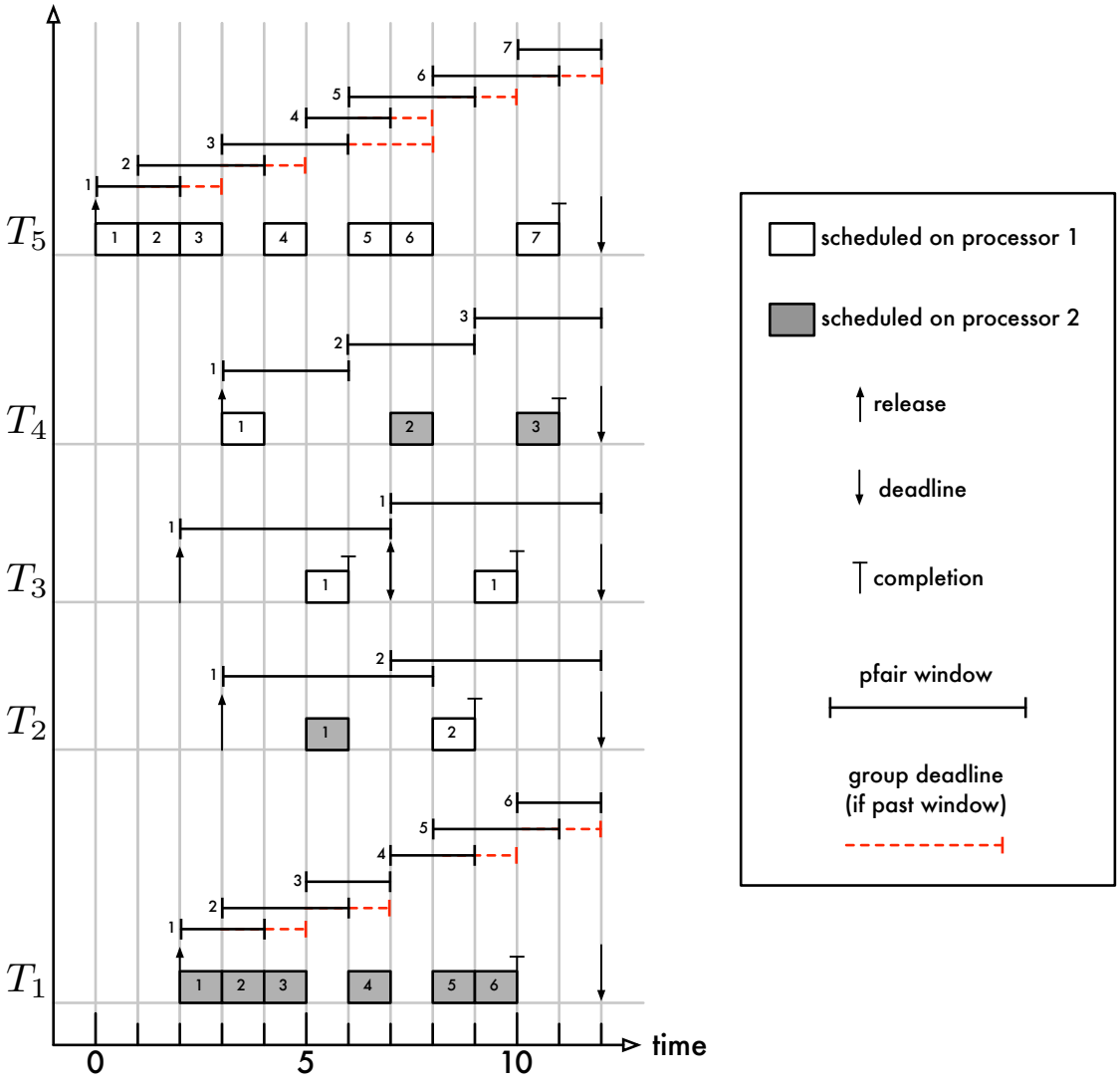


Figure 2.24: Example PD^2 schedule of the task set listed in Table 2.6 for $m = 2$ and $Q = 1$. The digit inside each processor allocation indicates the subtask number k. The corresponding pfair window and group deadline (if applicable) are shown above each job and are similarly marked. Overlapping subtask windows imply a non-zero successor bit. The subtask parameters are listed in Table 2.7 and shown in Figure 2.21. T_3 has two subtasks with $k = 1$ because it releases two jobs at times 2 and 5 in the depicted scenario. Note that $J_{1,1,3}$, $J_{1,1,4}$, and $J_{1,1,6}$ and also $J_{5,1,3}$, $J_{5,1,4}$, and $J_{5,1,6}$ are scheduled in a slot *before* their pfair window due to early-releasing.

units under PD^2 than under G-EDF. This is a witness to PD^2's ability to maximize parallelism, which is key to its ability to correctly schedule any feasible implicit-deadline task set. $\Diamond$

We summarize PD^2's HRT optimality with the following theorem.

Theorem 2.11 (Srinivasan and Anderson, 2002, 2006; Anderson and Srinivasan, 2000, 2004)**.** *An implicit-deadline sporadic task set τ is HRT schedulable under PD^2 with early-releasing on m processors if and only if $u_{\text{sum}}(\tau) \leq m$.*

Unfortunately, it is not obvious whether PD^2's theoretical prowess translates into meaningful real-world performance. As mentioned above, there are physical limits to the minimum achievable quantum size Q. Further, jobs tend to be preempted and migrated frequently under PD^2. For example, in Figure 2.24, both $J_{2,1}$ and $J_{4,1}$ are preempted after each subtask, and both jobs migrate once. This creates additional overheads that must be accounted for. An implementation-based study of PD^2's ability to ensure HRT correctness is thus required. We report on such a study in Chapter 4.

2.3.4 Clustered Multiprocessor Real-Time Scheduling

Under clustered priority-driven scheduling (Calandrino *et al.*, 2007; Baker and Baruah, 2007), the m processors are split into $\lceil \frac{m}{c} \rceil$ disjoint sets (or *clusters*) of c processors each. For notational convenience, we assume that m is an integer multiple of c unless noted otherwise, *i.e.*, there are exactly $\frac{m}{c}$ clusters of size c. As under partitioning, tasks are statically assigned to clusters during an offline partitioning phase. At runtime, an instance of a priority-driven scheduling algorithm is instantiated for each cluster. As a result, jobs do not migrate across cluster boundaries. However, since each cluster may contain multiple processors, jobs are scheduled "globally" within each cluster; consequently, a job may migrate among its constituent processors.

Clustered scheduling is a generalization of both global and partitioned scheduling: if $c = 1$, then clustered scheduling yields pure partitioned scheduling, and if $c = m$, then clustered scheduling is equivalent to global scheduling. From a historical perspective, however, clustered scheduling was introduced as a hybrid approach to combine the advantages of partitioned and global scheduling (Calandrino *et al.*, 2007). Specifically, the appeal of clustered scheduling is that it is believed to have lower implementation overheads than global scheduling and that it yields a simpler bin-packing instance than partitioned scheduling. The reason for the former is that clusters are

typically defined to align with the underlying hardware topology such that all processors in a cluster share a common cache, which may help to lessen migration overheads. With regard to the latter, the task assignment problem is much simpler under clustered scheduling (unless $c = 1$) because there are fewer and larger bins of size c while the size of each item, u_i, remains unchanged (or mostly so, when considering overheads). In other words, in relation to the bins in which they must be placed, the items are smaller under clustered scheduling; this makes it more likely that a bin-packing heuristic will find a (near-)optimal solution. As in the case of partitioned scheduling, we use the worst-fit-decreasing heuristic to distribute tasks more-or-less evenly among all clusters.

We consider two clustered schedulers in this dissertation: *clustered EDF*, denoted C-EDF, and *clustered PD*^{2}, which we denote as C-PD^2. While it is conceivably possible to "mix and match" cluster sizes and schedulers (*e.g.*, a 24-core system could be split into a 12-core PD^2 cluster and a two six-core G-EDF clusters), this is currently not supported by LITMUS$^{\mathrm{RT}}$; we restrict our focus to homogeneous clusters with respect to both cluster size and scheduling policy.

No clustering-specific schedulability tests are required. As is the case with partitioned scheduling, we simply partition a task set and apply one or more appropriate (global) schedulability test(s) to each cluster. The task set is deemed schedulable (either HRT or SRT) if each cluster passes (one of) the applied test(s). In fact, since typically $c > 1$, issues related to parallelism must be handled within each cluster. Therefore, clustered scheduling is essentially "global scheduling with an additional task assignment phase," both from an implementation and from a schedulability-analysis point of view. However, we note that global and clustered scheduling differ greatly when it comes to supporting certain types of locking protocols (which are discussed in the next section). This difference is explored in detail in Chapter 6.

Section summary. This concludes our review of real-time scheduling policies. To summarize, no partitioned scheduler or global JLFP scheduler can be optimal in the HRT sense, but HRT optimal global JLDP schedulers exist. From a theoretical point of view, there is thus little reason not to use global scheduling in general, and PD^2 in particular. However, concerns of higher overheads call the practicality of global scheduling into question. As a compromise, clustered scheduling applies global scheduling to clusters of processors. In this dissertation, we consider two schedulers of each category: two partitioned schedulers, P-FP and P-EDF, two global schedulers, G-EDF and PD^2, and their

clustered counterparts, C-EDF and C-PD^2. In Chapter 3, we discuss how to account for runtime overheads under each of these schedulers and provide a description of how they were implemented in LITMUS$^{\mathrm{RT}}$. We report on a study in which we compared them empirically under consideration of real, measured overheads in Chapter 4.

2.4 Real-Time Locking Protocols

Most published schedulability analysis, and each of the major schedulability tests reviewed in the preceding section, assume that all tasks are *independent*. That is, jobs are assumed to share no resources besides processors. Task independence implies that the m highest-priority pending jobs are always *ready* to be scheduled, *i.e.*, a job progresses towards its completion whenever it is assigned a processor by the scheduler. However, in most real systems, there are *shared resources* that are used by multiple tasks. For example, tasks may have to share limited physical I/O facilities or might share program state such as queues, buffers, and other data structures. Even if all tasks are indeed independent at the application level, there is typically a considerable amount of shared state in an RTOS kernel, including memory maps, device state, task information, and ready queues. Concurrent access to such shared resources must be *synchronized*, lest inconsistencies or conflicts arise. This is commonly achieved with *locks*.[17] As a result, tasks are no longer independent: if a job requires a shared resource that is already in use, then it must *wait* for the resource to become available and cannot progress towards completion, even if it is among the m highest-priority pending jobs.

There are two principal mechanisms to realize waiting on a multiprocessor: a job can either *busy-wait* (or *spin*) in a tight loop, thereby wasting processor time, or it can relinquish its processor and *suspend*. Locks of the former type are called *spinlocks*; the latter kind of locks are called *semaphores*. In theory, semaphores are always preferable because spinning jobs waste processor cycles. In practice, spinlocks benefit from low overheads (compared to the cost of suspending and resuming jobs), so that spinning can in fact be preferable if all critical sections are short. We explore this tradeoff in detail in Chapter 7. On a uniprocessor, suspending is the only option because a

[17] An alternative to locking is the use of specialized *non-blocking* algorithms (*e.g.*, see Anderson and Holman, 2000). However, compared to lock-based synchronization, which allows in-place updates, non-blocking approaches usually require additional memory, incur significant copying or retry overheads, and can only be used for synchronizing shared data structures, but not devices. With regard to synchronization, the focus of this dissertation is locking.

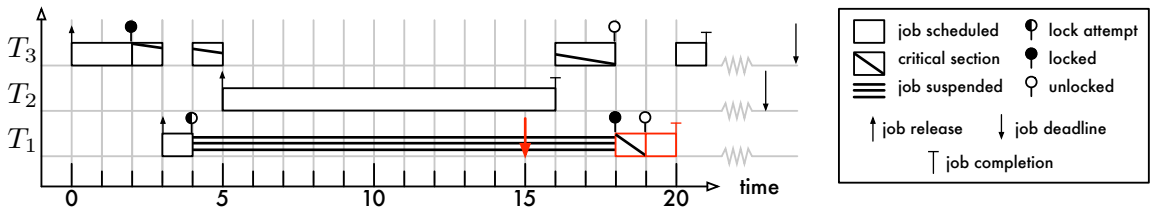


Figure 2.25: Example of an "unbounded" priority inversion. Three tasks are scheduled under FP scheduling. Two of the tasks (T_1, T_3) share one resource that is protected by a semaphore. The job $J_{1,1}$ is suspended during $[4, 17)$ because it tried to acquire the lock when it was already being held by $J_{3,1}$. The release of $J_{2,1}$ at time 5 transitively delays $J_{1,1}$ for $e_2 = 11$ time units. $J_{1,1}$ suffers a priority inversion during this time since it is the highest-priority pending job but not scheduled. As a result, $J_{1,1}$ misses its deadline at time 15.

spinning job prevents other jobs from executing on its processor. No matter whether waiting is implemented by means of spinning or suspending, it increases the response time of both the waiting and, indirectly, other jobs. In a real-time system, these additional delays must be controlled and accounted for.

Example 2.11. Figure 2.25 depicts a classic uniprocessor example that illustrates the dangers of uncontrolled locking. There are three tasks scheduled under FP, two of which (T_1 and T_3) share a non-processor resource that is protected by a semaphore. At time 0, $J_{3,1}$ is released first and locks the semaphore protecting the shared resource at time 2. It begins to execute its critical section, but is preempted at time 3 when the higher-priority job $J_{1,1}$ is released. At time 4, $J_{1,1}$ attempts to lock the shared resource, too. However, since it is unavailable (*i.e.*, already locked), $J_{1,1}$ must suspend and $J_{3,1}$ continues to execute its critical section instead. Then an inopportune preemption occurs: $J_{2,1}$ is released at time 5 and, since $\mathsf{Y}(J_{2,1}, 5) < \mathsf{Y}(J_{3,1}, 5)$, preempts the resource-holding job $J_{3,1}$. Consequently, $J_{3,1}$ cannot finish its critical section until $J_{2,1}$ completes. As a result, $J_{2,1}$ *transitively* delays the highest-priority job $J_{1,1}$ for the entirety of $J_{2,1}$'s execution, so that $J_{1,1}$ misses its deadline at time 15. This is a *priority inversion*: while $J_{1,1}$ is suspended, a lower-priority job is scheduled instead of the highest-priority pending job (formalized below). Since e_2 could be arbitrarily large (assuming an appropriately scaled p_2), $J_{1,1}$ is said to be at risk of an *unbounded priority inversion*. ♢

When using locks, priority inversions are unavoidable to some extent since shared (non-processor) resources are non-preemptable. That is, a lock once granted cannot be forcefully revoked from

a lower-priority job mid-critical-section because this might leave the resource in an inconsistent state. Priority inversions are a major concern because they create delays that are not anticipated by schedulability analysis that assumes independent tasks. To enable the predictable use of locks (and thus shared resources) in a real-time system, two issues need to be resolved: **(i)** the employed schedulability test must take priority inversions into account, and **(ii)** the maximum duration of priority inversion must be bounded.

The purpose of a *real-time locking protocol* is to ensure that (ii) is indeed the case at runtime, and that a suitable bound can be derived *a priori*. This encompasses two requirements. First, a real-time locking protocol must order conflicting requests so that no job incurs an excessive number of priority inversions. Second, it must enforce that resource-holding jobs quickly complete their critical sections when they delay higher-priority jobs, so that each occurrence of priority inversion is of bounded length.

In Chapters 5 and 6, we design and analyze several new multiprocessor locking protocols, and in Chapter 7, we present an implementation-based evaluation of several locking protocols. To establish the required foundation, we review relevant prior uni- and multiprocessor locking protocols in the remainder of this section, after first formalizing our model of resource sharing and defining priority inversion.

2.4.1 Resource Model

In this dissertation, we consider three types of shared resources that differ with respect to their sharing constraint. *Mutual exclusion* of accesses is required for *serially reusable* resources, which may be used by at most one job at any time. *Reader-writer exclusion* (Courtois *et al.*, 1971) is sufficient if a resources's state can be observed without affecting it: only *writes* (*i.e.*, state changes) are exclusive and multiple *reads* may be satisfied simultaneously. Resources of which there are k identical replicas are subject to a *k-exclusion* constraint: each replica is only serially reusable and thus requires mutual exclusion, but up to k requests may be satisfied at the same time by delegating them to different replicas.

Mutex constraints are most common in practice. However, the need for RW synchronization arises naturally in many situations, too. Two common examples are few-producers/many-consumers relationships (*e.g.*, obtaining and distributing sensor data) and rarely changing shared state (*e.g.*,

Notation	Interpretation	Constraint / Definition
ℓ_q	A shared resource.	$1 \leq q \leq n_r$
$\mathcal{R}_{i,q,v}$	The v^{th} resource request by any J_i for resource ℓ_q.	$v \geq 1$
$\mathcal{L}_{i,q,v}$	The request length of $\mathcal{R}_{i,q,v}$.	$\mathcal{L}_{i,q,v} \leq L_{i,q}$
$N_{i,q}$	Maximum number of requests for ℓ_q issued by any J_i.	
$L_{i,q}$	Maximum length of any request for ℓ_q issued by any J_i.	$N_{i,q} = 0 \Rightarrow L_{i,q} = 0$
$N_{i,q}^R$	Maximum number of read requests for ℓ_q issued by any J_i.	
$L_{i,q}^R$	Maximum length of any read request for ℓ_q issued by any J_i.	$N_{i,q}^R = 0 \Rightarrow L_{i,q}^R = 0$
$N_{i,q}^W$	Maximum number of write requests for ℓ_q issued by any J_i.	
$L_{i,q}^W$	Maximum length of any read write for ℓ_q issued by any J_i.	$N_{i,q}^W = 0 \Rightarrow L_{i,q}^W = 0$
k_q	Number of replicas of ℓ_q.	$k_q \geq 1$
b_i	Maximum duration of priority inversion incurred by any J_i.	see Section 2.4.2
s_i	Maximum duration of spinning incurred by any J_i.	see Section 2.4.4.1

Table 2.8: Summary of notation related to resource sharing.

configuration information). Of the three constraints, we expect k-exclusion constraints to be the least common. However, k-exclusion is required whenever there are multiple identical co-processors. For example, a system might contain multiple *graphics processing units* (GPUs) or *digital signal processors* (DSPs). Theoretically, one could further consider replicated resources with RW constraints, but we are not aware of any practical applications where such constraints arise and do not consider this combination of constraints.

Notation. We formalize resource sharing among sporadic tasks as follows (and as summarized in Table 2.8). Besides the m processors, the system contains n_r *shared resources* $\ell_1, \ldots, \ell_{n_r}$. When a job J_i requires a resource ℓ_q, where $q \in \{1, \ldots, n_r\}$, it *issues* a *request* for ℓ_q. We let $\mathcal{R}_{i,q,v}$ denote the v^{th} resource request by task T_i for resource ℓ_q, where $v \geq 1$. In other words, $\mathcal{R}_{i,q,v}$ corresponds to the v^{th} time that any J_i requires ℓ_q since T_i was launched. In the case of RW constraints, we analogously let $\mathcal{R}_{i,q,v}^R$ and $\mathcal{R}_{i,q,v}^W$ denote the v^{th} read and write request for ℓ_q, respectively.

A request $\mathcal{R}_{i,q,v}$ is *satisfied* as soon as J_i *holds* ℓ_q, and *completes* when J_i *releases* ℓ_q. The *request length* is the time that J_i must execute before it releases ℓ_q; we let $\mathcal{L}_{i,q,v}$ denote the request length of $\mathcal{R}_{i,q,v}$. If a request $\mathcal{R}_{i,q,v}$ cannot be satisfied immediately, then J_i incurs *acquisition delay* and cannot proceed with its computation while it *waits* for $\mathcal{R}_{i,q,v}$ to be satisfied. Jobs wait by either spinning or by suspending, depending on the type of lock that protects ℓ_q. The *request span* of $\mathcal{R}_{i,q,v}$

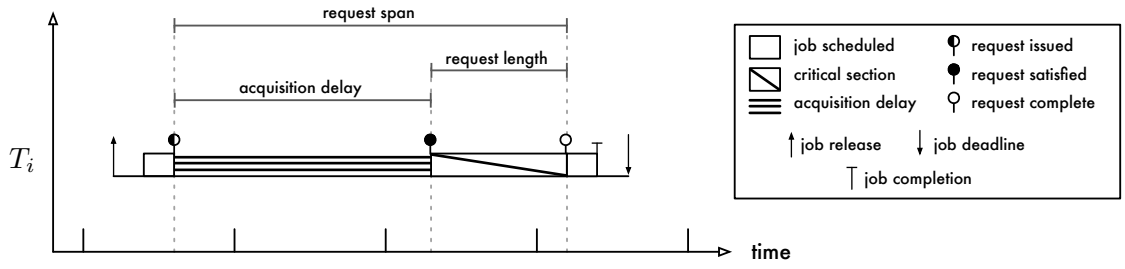


Figure 2.26: Illustration of the phases of a request. A job must wait by either spinning or suspending while it incurs acquisition delay.

starts when $\mathcal{R}_{i,q,v}$ is issued and lasts until it completes, *i.e.*, it includes the request length and any acquisition delay. The different phases of a resource request are illustrated in Figure 2.26.

Requests may be issued at any time during a job's execution, and no particular order of requests is assumed. In particular, we do not assume a minimum separation between consecutive requests. We let $N_{i,q}$ denote the maximum number of times that any J_i requests ℓ_q, and let $L_{i,q}$ denote the maximum length of such a request, where $L_{i,q} = 0$ if $N_{i,q} = 0$. When discussing RW resources, we let $N_{i,q}^R$ and $N_{i,q}^W$ denote the number of read and write requests, respectively, with the interpretation that $N_{i,q}^R + N_{i,q}^W = N_{i,q}$. We let $L_{i,q}^R$ and $L_{i,q}^W$ denote bounds on the maximum read and write request length (analogously to $L_{i,q}$). In the context of shared resources with k-exclusion constraints, we let k_q denote the number of replicas of resource ℓ_q.

While executing a request, J_i is said to be in a *critical section*. We assume that J_i must be scheduled on a processor while using a resource, that is, J_i's request only progresses towards completion when J_i is allocated a processor. This is required for shared data objects, but may be pessimistic for some I/O devices. The latter can be accounted for at the expense of more-verbose notation. We assume that each task's execution requirement e_i already accounts for critical sections of T_i (but not for cycles lost to spinning, if any).

If job J_i issues a request $\mathcal{R}_{i,q,w}$ after a previous request $\mathcal{R}_{i,q,v}$ has been satisfied but before $\mathcal{R}_{i,q,v}$ is complete, then $\mathcal{R}_{i,q,w}$ is an *inner request* that is *nested* within the *outer request* $\mathcal{R}_{i,q,v}$. An *outermost* request is not nested within any other requests. Unless noted otherwise, we assume that resource requests are not nested within each other, *i.e.*, jobs request at most one resource at any time. Resource nesting is considered in the context of uniprocessor locking protocols in Section 2.4.3 below and briefly in Chapter 5. When discussing nested requests, we assume that the request length

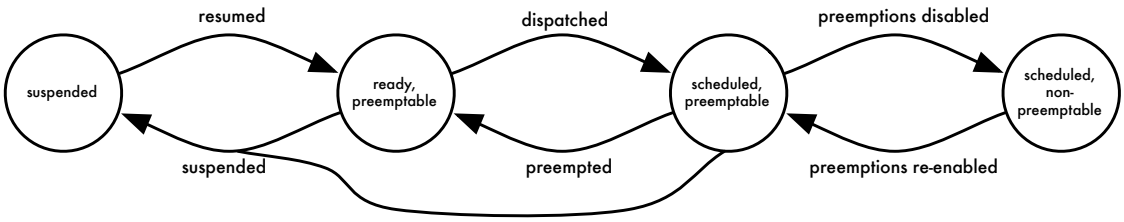


Figure 2.27: Job state transition diagram.

of outer requests includes the request length of inner requests. We further require that tasks do not hold resources across job boundaries.

Job states. To accommodate the locking protocols discussed below, we extend our task model as illustrated in Figure 2.27. A pending job can be in one of the following states: a *ready* job is available for execution, whereas a *suspended* job cannot be scheduled. A job *resumes* when its state changes from suspended to ready. A scheduled job is further either *preemptable* or *non-preemptable*, and cannot be descheduled while it is non-preemptable. In other words, a job that enters a *non-preemptive section* cannot be preempted or migrated by the scheduler until it exits its critical section. We assume that pending jobs are ready unless suspended by a locking protocol. A ready job may be suspended regardless of whether it is scheduled; however, jobs may not suspend while being non-preemptable.

Effective priority. A second extension concerns the prioritization function $\Upsilon(J_i, t)$. As the example shown in Figure 2.25 illustrates, the "regular" priority of a low-priority job may be insufficient to guarantee progress when it blocks a higher-priority job. To avoid such situations, a locking protocol may have to temporarily raise the priority of a resource-holding job. To reflect this, we call $\Upsilon(J_i, t)$ the *base priority* assigned by the scheduling policy, and let $\upsilon(J_i, t)$ denote the *effective priority* of J_i, which is determined by the locking protocol based on the resource usage of J_i at time t. We assume that the two priorities match, *i.e.*, $\upsilon(J_i, t) = \Upsilon(J_i, t)$, if J_i is not using any shared resource at time t. A priority-driven scheduler instantiated for m processors will schedule the m ready jobs with the highest effective priorities at any time, subject to non-preemptivity constraints of earlier-scheduled jobs. In contrast to base priorities, effective priorities are not necessarily unique; any ties are resolved such that the number of preemptions is minimized; that is, if $\upsilon(J_i, t) = \upsilon(J_x, t)$, then J_i and J_x cannot preempt each other.

Partitions and clusters. When bounding locking-related delays under partitioned or clustered scheduling, it is often required to consider the subset of jobs assigned to a particular processor or cluster. We adopt the following notation. Under partitioned scheduling, we let τ_k denote the set of tasks assigned to processor k, and let P_i denote the partition to which task T_i is assigned, *i.e.*, $\forall\, 1 \leq k \leq m :\ \tau_k \subseteq \tau$ and $T_i \in \tau_{P_i}$. Under clustered scheduling, we analogously let τ_k denote the set of jobs assigned to the k^{th} cluster and let P_i denote the cluster to which T_i is assigned. A task T_l is *local* to task T_i if $P_l = P_i$, and *remote* otherwise.

2.4.2 Priority Inversion Blocking

Intuitively, a priority inversion occurs whenever a job that *should* be scheduled (according to its base priority) is *not* scheduled, *i.e.*, either when a lower-priority job is scheduled instead or when the processor is idle. Such inversions are problematic because schedulability analysis that requires task independence (implicitly) assumes that a job always progresses towards completion whenever it has the highest priority. When priorities are inverted, this is no longer the case. As a result, the response time of a job J_i that suffers a priority inversion can exceed the worst-case response-time bound assuming independence (*e.g.*, this happens to $J_{1,1}$ in Figure 2.25). Formally, a priority inversion exists on a uniprocessor at time t if the following criteria are met.

Definition 2.8. Let J_t denote the job scheduled at time t (unless the processor is idle). On a uniprocessor, a job J_i incurs a *priority inversion* at time t if J_i is pending but not scheduled at time t, and either the processor is idle or $\Upsilon(J_i, t) < \Upsilon(J_t, t)$.

Under partitioned scheduling, Definition 2.8 applies on a per-processor basis; priority inversions under global and clustered scheduling are discussed in Chapter 6. Note that the definition does not require J_i to be ready; J_i must merely be pending and may be suspended. At the same time, a suspended job J_i does not necessarily incur a priority inversion: as long as another job with higher priority is scheduled, there is no priority inversion. This occurs when J_i's suspension happens to overlap with an interval where J_i would not have been scheduled anyway due to the processor demand from higher-priority jobs.

This subtle observation has a significant impact on real-time systems: from a real-time correctness point of view, the length of the actual acquisition delay is *irrelevant*. Instead, only the cumulative

length of priority inversion—the increase of a job's response time—is relevant for schedulability analysis. The purpose of a real-time locking protocol is thus to minimize the maximum priority inversion, and not to minimize the maximum acquisition delay. This matches the intuition that high-priority jobs should be granted access to contended resources sooner than lower-priority jobs.

In the real-time literature, acquisition delay that coincides with a priority inversion is traditionally called "blocking," whereas acquisition delay that does not coincide with a priority inversion lacks an established name (since it is irrelevant for analysis purposes). We avoid the term "blocking" in this dissertation because it is overloaded. In a real-time context, many other sources of schedulability-relevant delays are also commonly labeled "blocking," even if they do not coincide with a priority inversion. For example, release jitter and deferred execution[18] are causes of "blocking" without priority inversion (Liu, 2000). In the (non-real-time) synchronization literature, "blocking" is a synonym for acquisition delay. To further confuse matters, in an OS context, "blocking" is often used as a synonym for "suspending," which is not the same as the intended interpretation: in suspension-based protocols, the length of a suspension corresponds to the acquisition delay, but not necessarily to the length of priority inversion. Indeed, "blocking" on a lock is not even required to incur a priority inversion: in each of the locking protocols reviewed below, a job can incur a priority inversion even if it does not request any resources itself. "Blocking" in the real-time sense is thus only tangentially related to "blocking" in either the OS or synchronization sense: each kind of "blocking" can occur independently of the other two.

In this dissertation, we consider the definition specific to real-time resource sharing, which we denote as *priority inversion blocking* (*pi-blocking*) to avoid ambiguity. To reiterate, pi-blocking occurs whenever a job J_i's completion is delayed and this delay cannot be attributed to higher-priority demand—that is, if and only if J_i suffers a priority inversion. We let b_i denote a bound on the total pi-blocking incurred by any J_i. We say that a task T_i is at risk of *unbounded priority inversion* if no finite bound b_i exists or if the bound b_i includes one or more execution requirement parameters e_k. A locking protocol ensures *bounded priority inversions* if each bound b_i can be expressed in terms of a finite number of $L_{k,q}$ parameters. To establish that a task set is schedulable, all priority inversions must be bounded and each b_i must be accounted for in the schedulability test. With these definitions

[18] A job J_i defers a part of its execution when J_i is not yet complete but not available for scheduling (*e.g.*, when waiting for I/O operations to complete).

in place, we are now ready to briefly review the prior locking protocols most relevant to our work. We begin with uniprocessor locking protocols.

2.4.3 Uniprocessor Real-Time Locking Protocols

In contrast to the multiprocessor case reviewed below (Section 2.4.4), uniprocessor locking is well-studied and understood. In particular, protocols that support nested resource requests exist and have been adopted in several commercial RTOSs and, to some degree, even in the POSIX standard. Additionally, both FP response time analysis (Theorem 2.2) and the EDF density test (Theorem 2.4) have been extended to account for priority inversions. In the presence of bounded pi-blocking, HRT schedulability can be established with the following two theorems.

Theorem 2.12 (Audsley *et al.*, 1993)**.** *Let $\tau = \{T_1, \ldots, T_n\}$ denote a set of constrained-deadline sporadic tasks indexed in order of decreasing priority. On a uniprocessor, under FP scheduling, the response time r_i of task $T_i \in \tau$ is bounded by the smallest R_i that satisfies the following equation:*

$$R_i = e_i + b_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{p_h} \right\rceil \cdot e_h.$$

Theorem 2.13 (Baker, 1990, 1991)**.** *An arbitrary-deadline sporadic task set $\tau = \{T_1, \ldots, T_n\}$ is HRT schedulable under EDF on a uniprocessor if, for each $T_i \in \tau$,*

$$\frac{b_i}{\min(d_i, p_i)} + \sum_{T_k \in \tau} \delta_k \leq 1.$$

Both HRT schedulability tests are sustainable with respect to execution requirements and periods (Baruah and Burns, 2006; Burns and Baruah, 2008).

In the discussion of the following four locking protocols, we first assume FP scheduling and that all resources are subject to mutex constraints, which is the model most commonly assumed in prior work. EDF scheduling and RW and k-exclusion constraints are considered thereafter.

Non-preemptive sections. The example shown in Figure 2.25 illustrates that ill-timed preemptions of resource-holding jobs can cause unbounded priority inversions. The simplest approach to avoiding unbounded priority inversions is to disallow preemptions during critical sections completely. That is, when a job J_i's request for a resource ℓ_q is satisfied, J_i becomes non-preemptable until the end of its

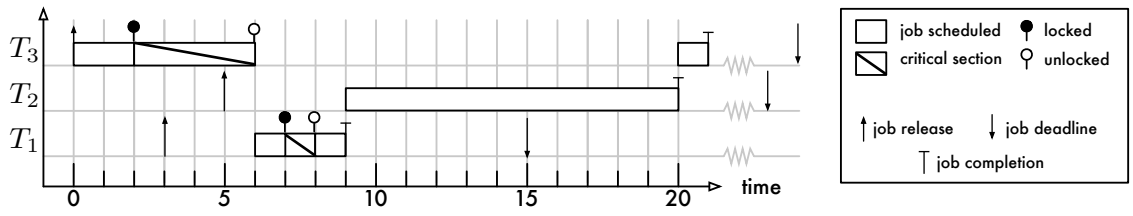


Figure 2.28: NCP schedule of the scenario shown in Figure 2.25. Since $J_{3,1}$ is non-preemptable at time 5, $J_{1,1}$ is not transitively delayed by $J_{2,1}$. $J_{1,1}$ suffers pi-blocking during $[3, 6)$; $J_{2,1}$ suffers pi-blocking during $[5, 6)$. Maximum pi-blocking is limited to $b_1 = 4$, $b_2 = 4$, and $b_3 = 0$.

critical section. This prevents later-arriving, higher-priority jobs from preempting a resource-holding job, thereby ensuring that each request will be completed as soon as possible (once it has been satisfied). We refer to this locking protocol as the *non-preemptive critical section protocol* (NCP).

Example 2.12. An example schedule is shown in Figure 2.28, which depicts the same jobs that were shown in Figure 2.25. Since $J_{3,1}$ executes its critical section non-preemptively during $[2, 6)$, it is not preempted when $J_{1,1}$ and $J_{2,1}$ arrive at times 3 and 5, respectively. As a result, $J_{2,1}$ does not transitively delay $J_{1,1}$. $J_{1,1}$ still suffers pi-blocking during $[3, 6)$, but in contrast to Figure 2.25, it is limited to the length of $J_{3,1}$'s critical section. Note that there are no suspensions and no failed lock attempts (*i.e.*, unsatisfied requests)—under the NCP, once a job starts executing, all its resource requests are satisfied immediately (assuming resource-holding jobs do not suspend for locking-unrelated reasons). ◊

In general, the NCP ensures that any job incurs pi-blocking at most once, and only immediately when it is released. Further, the NCP limits the maximum duration of priority inversion to the length of one outermost request of any lower-priority job. Thus, under FP scheduling, a bound on the maximum duration of priority inversion incurred by any J_i is given by $b_i = \max\{L_{k,q} \mid k > i \wedge 1 \leq q \leq n_r\}$. The NCP is in widespread use in practice, and particularly so in deeply embedded systems that execute all tasks in kernel mode, because non-preemptivity can be very efficiently realized by disabling interrupt delivery, which can be achieved with a single instruction on many platforms (recall Section 2.1.3).

The biggest disadvantage of the NCP is that jobs that do not access any shared resources may still suffer a priority inversion on release, *i.e.*, each b_i reflects the resource usage of other tasks and not T_i. This is illustrated in Figure 2.29. In the example, task T_1 does not access the shared resource

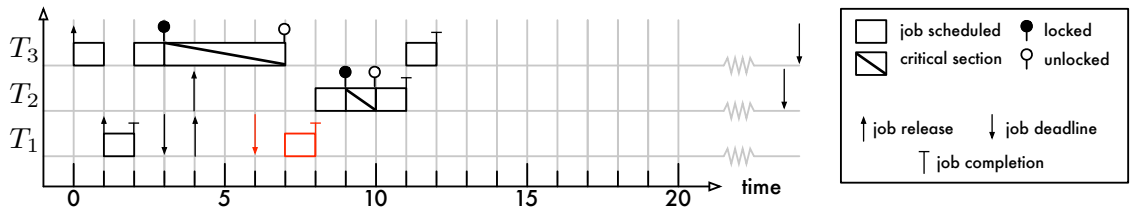


Figure 2.29: NCP schedule that demonstrates that the highest-priority task is subject to pi-blocking even if it is independent. T_2 and T_3 share one resource ℓ_q, but $N_{1,1} = 0$. The first job of T_1, released at time 1, is "lucky" and can preempt $J_{3,1}$. However, the second job $J_{1,2}$ is released while $J_{3,1}$ executes a long critical section. As a result, $J_{1,2}$ is not scheduled until after its deadline.

(*i.e.*, $N_{1,1} = 0$), but tasks T_2 and T_3 do. Further, T_1 has a very tight deadline. Because $J_{3,1}$ executes its request non-preemptively, $J_{1,2}$ cannot preempt $J_{3,1}$ and consequently misses its deadline. This shows that it can be problematic if locking protocols penalize independent tasks.

Priority inheritance. Sha *et al.* (1990) designed an elegant mechanism that avoids delaying higher-priority jobs if they do not request resources. Their *priority inheritance protocol* (PIP) only intervenes when preempting a lower-priority job would delay a higher-priority job (Sha *et al.*, 1990; Rajkumar, 1991). Priority inheritance prevents such preemptions by raising the effective priority of resource-holding jobs to that of the highest-priority waiting job. Let $W(J_i, t)$ denote the set of jobs that are waiting for J_i at time t, *i.e.*, that have issued a request for a resource that is currently being held by J_i. Under the PIP, J_i's effective priority is defined as

$$\mathsf{y}(J_i, t) \triangleq \min\left(\{\mathsf{Y}(J_i, t)\} \bigcup \{\mathsf{y}(J_k, t) \mid J_k \in W(J_i, t)\}\right).$$

Note that priority inheritance is transitive since $\mathsf{y}(J_i, t)$ is defined in terms of each $\mathsf{y}(J_k, t)$. That is, if J_k is waiting for a resource that a job J_j holds, and J_j is waiting for a resource that J_i holds, then $\mathsf{y}(J_i, t) \leq \mathsf{y}(J_j, t) \leq \mathsf{y}(J_k, t) \leq \mathsf{Y}(J_k, t)$. This yields a powerful progress property: whenever a waiting job J_k suffers a priority inversion (*i.e.*, if J_k has the highest base priority), then the job at the end of the "wait-for" dependency chain is guaranteed to be scheduled. This limits the duration of each priority inversion to the length of one outermost critical section. Further, since jobs are preemptable at all times, the highest-priority job is only delayed if it issues requests itself. If there are

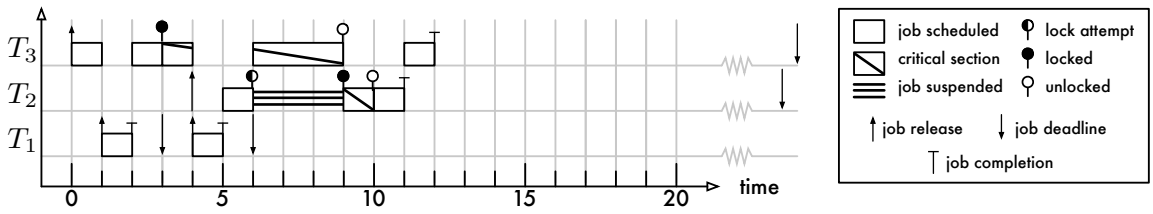


Figure 2.30: PIP schedule of the scenario that results in a deadline miss under the NCP (Figure 2.29). Under the PIP, jobs are always preemptable. Hence, $J_{1,2}$ can preempt $J_{3,1}$ in the middle of a critical section and thus completes on time. Note that $J_{2,1}$ does not incur pi-blocking immediately upon release (since $J_{3,1}$ is preemptable), but is nonetheless delayed when it requires the resource still held by $J_{3,1}$.

multiple jobs waiting for a resource, then their requests are satisfied in order of decreasing effective priority.

Example 2.13. Figure 2.30 shows the example scenario from Figure 2.29 scheduled using the PIP instead of the NCP. Here, $J_{1,2}$, can preempt $J_{3,1}$ during its critical section and thus meet its deadline. A second example shown in Figure 2.31 further demonstrates that the PIP is effective at preventing unbounded priority inversions. Unlike the case in Figure 2.25, $J_{2,1}$ cannot preempt $J_{3,1}$ at time 5 because $W(J_{3,1}, 5) = \{J_{1,1}\}$, and thus $\mathsf{y}(J_{3,1}, 5) = \min\{3, 1\} = 1 < \mathsf{y}(J_{2,1}, 5)$. $\Diamond$

In contrast to the NCP, a job may incur a priority inversion *each time* that it requests a resource (that is has not previously acquired). Further, jobs that do not request any shared resources may still be subject to priority inversion if they are not the highest-priority job. For example, if a low-priority job J_3 holds a resource that a high-priority job J_1 requires, then a middle-priority job J_2 will not be scheduled while J_3 inherits J_1's priority—both J_1 and J_2 are subject to priority inversion while J_3 completes its critical section. In Figure 2.31, this happens during the interval $[5, 7)$.

The PIP yields the following bound on the maximum cumulative priority inversion duration results (assuming FP scheduling). Let A_i denote the set of resources that are shared by both tasks of equal-or-higher and lower priority than T_i, *i.e.*,

$$A_i \triangleq \{\ell_q \mid \exists h, l \text{ s.t. } 1 \leq h \leq i \wedge i < l \leq n \wedge N_{h,q} > 1 \wedge N_{l,q} > 1\}. \tag{2.6}$$

In the worst case, each resource in A_i is held by a lower-priority job when J_i commences execution and requested by a higher-priority job (or J_i itself) while J_i is pending; the maximum duration of

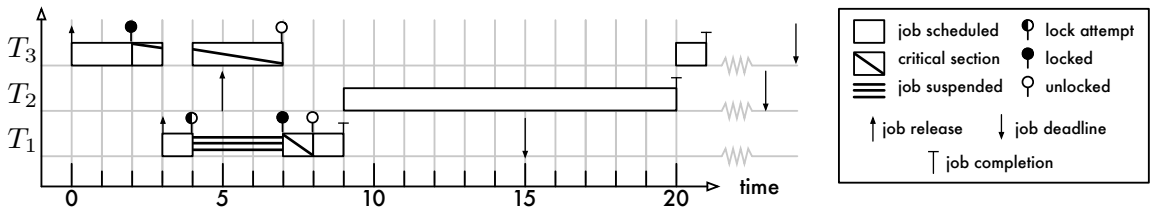


Figure 2.31: PIP schedule of the scenario shown in Figure 2.25. Since $J_{3,1}$ inherits $J_{1,1}$'s priority during $[4, 7)$, $J_{1,1}$ is not transitively delayed by $J_{2,1}$. $J_{1,1}$ incurs pi-blocking during $[4, 7)$. Since $J_{3,1}$ has a lower base priority than $J_{2,1}$, $J_{2,1}$ suffers a priority inversion during $[5, 7)$.

priority inversion is thus $b_i = \sum_{\ell_q \in A_i} \max\{L_{l,q} \mid l > i\}$. The bound stated here has been somewhat simplified; see (Sha *et al.*, 1990) for a bound that is less pessimistic if $|A_i| > (n - i)$. However, even then it shows that tasks can have a greater risk of priority inversion under the PIP than under the NCP. Nonetheless, the PIP is mandated by the POSIX real-time standard and thus widely used in practice.

Another limitation of the PIP is that it is susceptible to *deadlock*, *i.e.*, to jobs waiting indefinitely for each other to release a resource. Deadlock is only an issue when requests may be nested, that is, when jobs may acquire more than one resource at a time.

Example 2.14. A typical deadlock scenario involving two resources ℓ_2 and ℓ_3 is shown in Figure 2.32. Job $J_{3,1}$ acquires ℓ_3 at time 1 and is then preempted by the simultaneous arrivals of $J_{2,1}$ and $J_{1,1}$. $J_{1,1}$ requires neither ℓ_2 nor ℓ_3 and completes without incurring any pi-blocking. $J_{2,1}$, however, first locks ℓ_2 and then requests ℓ_3, which is still being held by $J_{3,1}$. Consequently, $J_{2,1}$ suspends and $J_{3,1}$ continues to execute its critical section until it, too, requests ℓ_2. $J_{3,1}$ and $J_{2,1}$ are now deadlocked. $\Diamond$

Priority ceiling. Sha *et al*. (1990) solved both the deadlock problem and the potential for excessive pi-blocking under the PIP with the *priority-ceiling protocol* (PCP) (Sha *et al*., 1990; Rajkumar, 1991). The intuition underlying the PCP is to delay potentially problematic resource requests until deadlock is impossible. Whether a given request is "potentially problematic" is determined based on the priorities of the tasks sharing a resource according to the following three definitions.

The *priority ceiling* of a resource ℓ_q, denoted $\Pi(\ell_q)$, is the base priority of the highest-base-priority task that accesses ℓ_q; formally, $\Pi(\ell_q) \triangleq \min\{i \mid N_{i,q} > 0\}$ under FP scheduling. Based on

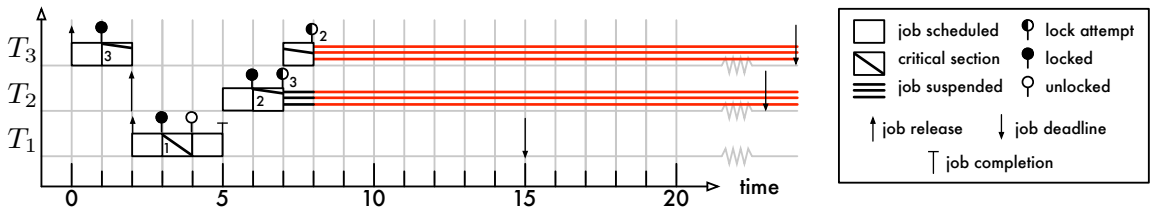


Figure 2.32: Example schedule of three tasks showing that jobs can deadlock under the PIP. There are three resources ℓ_1, ℓ_2, ℓ_3. The numbers within each critical section and next to failed lock attempts indicate the index of the requested resource. $J_{3,1}$ and $J_{2,1}$ issue nested requests for ℓ_2 and ℓ_3, respectively. Deadlock results since neither job can release the resource it holds without first acquiring the other.

the priority ceiling, the *system ceiling* at time t, denoted $\hat{\Pi}(t)$, is the highest priority ceiling of any resource in use at time t, or $n+1$ if no resource is in use; *i.e.*,

$$\hat{\Pi}(t) \triangleq \min\left(\{n+1\} \bigcup \{\Pi(\ell_q) \mid \ell_q \text{ is in use at time } t\}\right).$$

Finally, let $L(t)$ denote the job that raised $\hat{\Pi}(t)$ to its current value. In other words, if $\hat{\Pi}(t) < n+1$, then $L(t)$ holds a resource ℓ_q at time t such that $\Pi(\ell_q) = \hat{\Pi}(t)$.

Based on the two ceilings, the PCP extends the PIP by enforcing an additional constraint prior to satisfying requests: *a request $\mathcal{R}_{i,q,v}$ issued by job J_i is only satisfied at time t if either* **(i)** $\mathsf{Y}(J_i, t) > \hat{\Pi}(t)$ *or* **(ii)** $L(t) = J_i$. Condition (i) implies that deadlock is impossible because it prevents a later-arriving, higher-priority job J_i from acquiring *any* resources while a lower-priority job holds at least one resource that J_i *might* request. As a necessary precondition for J_i to deadlock, there must exist a resource ℓ_q, where $N_{i,q} > 1$, that is already being held by a lower-priority job when J_i locks a second resource ℓ_x at time t. (In Figure 2.32, this is the case for $J_{2,1}$ at time $t = 6$ with $q = 3$ and $x = 2$.) Since $N_{i,q} > 1$, it follows that $\Pi(\ell_q) \leq i$ and thus $\hat{\Pi}(t) \leq i$, which implies that condition (i) is false at time t. Therefore, J_i cannot acquire any resources until all resources that it might request have become available, which precludes deadlock. Condition (ii) is required to prevent jobs from deadlocking with themselves when they issue nested requests. As under the PIP, conflicting requests are satisfied in order of decreasing effective priority, and jobs inherit the priority of blocked higher-priority jobs.

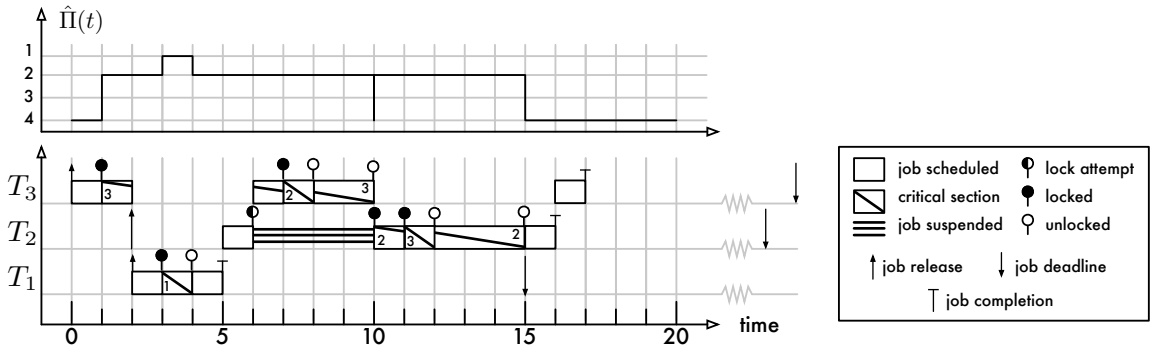


Figure 2.33: PCP schedule of the scenario that results in deadlock under the PIP (Figure 2.32). The step function above the schedule shows the corresponding system ceiling $\hat{\Pi}(t)$ as a function of time. Deadlock is avoided because $J_{2,1}$'s request for ℓ_2 at time 6 remains unsatisfied until time 10, when the system ceiling is momentarily lowered to $n + 1 = 4$. Since $J_{2,1}$ immediately acquires ℓ_2, the system ceiling is instantaneously restored to ℓ_2's priority ceiling $\Pi(\ell_2) = 2$.

Example 2.15. Figure 2.33 demonstrates how condition (i) prevents deadlock. As in Figure 2.32, $J_{3,1}$ and $J_{2,1}$ acquire both ℓ_2 and ℓ_3 in a nested fashion. However, under the PCP, $J_{2,1}$ is prevented from acquiring ℓ_2 at time 6 because $J_{3,1}$ raised the system ceiling to ℓ_3's priority ceiling $\Pi(\ell_3) = \min\{3, 2\} = 2$ at time 1. $J_{2,1}$'s request thus fulfills neither condition (i) nor (ii) until time 10, when the system ceiling is (briefly) lowered to $n + 1 = 4$ when $J_{2,1}$ releases ℓ_3. $J_{3,1}$'s request for ℓ_2 at time 7 is satisfied immediately even though $J_{3,1}$'s effective priority does not exceed the system ceiling because it is the last job to have raised the system ceiling, *i.e.*, $J_{3,1}$'s request satisfies condition (ii) but not condition (i). Note that $J_{1,1}$'s request for ℓ_1 at time 3 is not delayed by the PCP since $\mathsf{y}(J_{1,1}, 3) = 1 < \hat{\Pi}(3) = 2$. ◊

In general, the PCP avoids deadlock and limits the maximum duration of priority inversion to one (outermost) critical section of any lower-priority job. Define A_i as for the PIP in Equation (2.6). Assuming FP scheduling, the maximum (cumulative) duration of priority inversion experienced by any J_i under the PCP is limited to $b_i = \max\{L_{l,q} \mid \ell_q \in A_i \wedge l > i\}$. Note that, like the NCP and unlike the PIP, the pi-blocking bound is an $O(1)$ term, *i.e.*, it does not depend on the number of tasks n or the number of shared resources s. Further, unlike the NCP, the highest-priority job is not subject to priority inversion if it does not access any shared resources. The PCP is thus an asymptotically optimal uniprocessor real-time locking protocol.

Release blocking. Baker (1990, 1991) proposed a second priority-ceiling-based protocol, namely the *stack resource policy* (SRP). The bound on maximum pi-blocking under the SRP is identical to the bound ensured by the PCP, but the time at which the priority inversion (if any) is incurred differs. Recall that the PCP works by delaying "potentially problematic requests." The key insight underlying the SRP is that the resource-sharing problem can be considerably simplified by delaying the time of "problematic preemptions" instead. We shall use a similar approach in Chapter 6.

Under the PCP, a job may execute immediately after its release, but may become blocked once it issues a resource request. In contrast, under the SRP, a job suffers a priority inversion either immediately upon release or not at all. Define each resource ceiling $\Pi(\ell_q)$ and the system ceiling $\hat{\Pi}(t)$ as before in the definition of the PCP. The SRP consists of a single scheduling rule: *a newly-released job $J_{i,j}$ is not eligible to execute until time t, where $t \geq a_{i,j}$, such that $\mathsf{Y}(J_{i,j}, t) < \hat{\Pi}(t)$.* In other words, each job is initially suspended and only resumed once its priority exceeds the system ceiling. Note that the SRP does not require priority inheritance. When J_i later issues a request during its execution, the SRP scheduling rule implies that J_i's request will be satisfied immediately: all resources that J_i *might* request are available after its initial wait. This implies that deadlock is impossible.

Example 2.16. An example SRP schedule is shown in Figure 2.34. As before in Figures 2.32 and 2.33, $J_{2,1}$ is released at time 2. However, under the SRP, it may not commence execution until time 9 because $\hat{\Pi}(t) \leq 2$ for each $t \in [1, 9)$. As under the PCP, the highest-priority job $J_{1,1}$ is not affected by the resource-sharing of lower-priority jobs. $\Diamond$

Locking under EDF. We have assumed FP scheduling in the discussion so far. However, with only small changes, the NCP, PIP, and SRP, and PCP can be applied to EDF as well. A key property of EDF is that lower-priority jobs released prior to job J_i's arrival stem from tasks with strictly longer relative deadlines.

Lemma 2.4. *Under EDF (either uniprocessor, global, clustered, or partitioned), if a lower-priority job $J_{l,k}$ arrived prior to $J_{i,j}$, i.e., if $a_{l,k} < a_{i,j}$ and $\mathsf{Y}(J_{i,j}, t) < \mathsf{Y}(J_{l,k}, t)$, then $d_i < d_l$.*

Proof. This follows from the definition of EDF priorities:

$$\mathsf{Y}(J_{i,j}, t) < \mathsf{Y}(J_{l,k}, t)$$

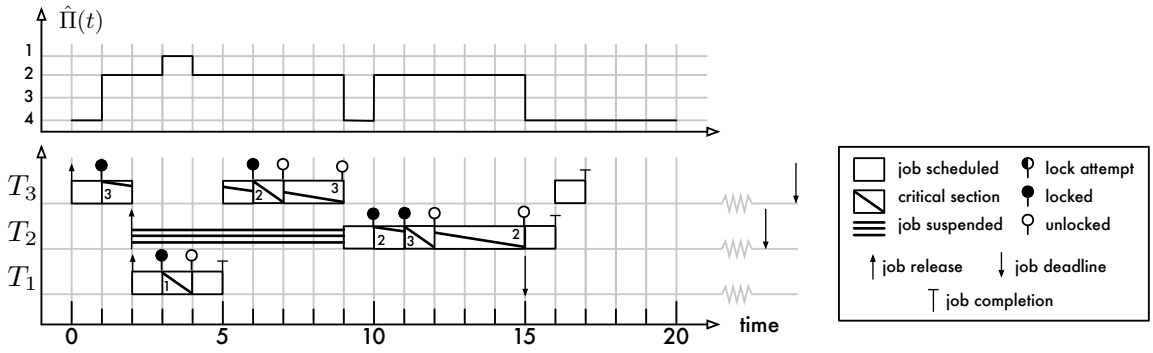


Figure 2.34: SRP schedule of the scenario that results in deadlock under the PIP (Figure 2.32). Similarly to the PCP (Figure 2.33), the SRP avoids deadlock by means of the system ceiling. $J_{2,1}$ is suspended immediately upon release because its priority does not exceed the system ceiling. At time 9, when $J_{3,1}$ releases ℓ_3, the priority ceiling is lowered to four and $J_{2,1}$ becomes eligible to execute.

$$
\begin{aligned}
& \{ \text{ definition of EDF base priorities } \} \\
\Leftrightarrow \quad & d_{i,j} < d_{l,k} \vee (d_{i,j} = d_{l,k} \wedge i < l) \\
& \{ \text{ definition of absolute deadline } \} \\
\Leftrightarrow \quad & a_{i,j} + d_i < a_{l,k} + d_l \vee (a_{i,j} + d_i = a_{l,k} + d_l \wedge i < l) \\
& \{ \text{ since } a_{i,j} > a_{l,k} \} \\
\Rightarrow \quad & d_i < d_l \vee (d_i < d_l \wedge i < l) \\
\Leftrightarrow \quad & d_i < d_l.
\end{aligned}
$$

Since this relies only on the definition of the EDF prioritization function, the lemma applies equally to EDF, P-EDF, C-EDF, and G-EDF. □

Common to the NCP, PIP, PCP, and SRP is that only earlier-arrived, lower-priority jobs can cause J_i to incur a priority inversion (since lower-priority jobs cannot issue outermost requests while J_i is pending). This, together with Lemma 2.4, implies that only critical sections of tasks with strictly larger relative deadlines pi-block J_i under EDF. In contrast, under FP, only critical sections of larger-indexed tasks pi-block J_i. Therefore, the bound on pi-blocking for each of the protocols must be adjusted to correctly reflect to which tasks lower-priority jobs belong.

In the case of the NCP, the protocol itself remains unchanged and the bound is stated in terms of relative deadlines. Under EDF, the NCP ensures a maximum duration of priority inversion of $b_i = \max\{L_{k,q} \mid d_k > d_i \wedge 1 \leq q \leq n_r\}$ for any J_i.

In the case of the PIP, PCP, and SRP, the definition of A_i must be adjusted to reflect relative deadlines instead of task indices. Under EDF, the set of resources that could be accessed by both higher- and lower-priority jobs is defined as $A_i \triangleq \{\ell_q \mid \exists h, l \text{ s.t. } d_h \leq d_i \wedge d_i < d_l \wedge N_{h,q} > 1 \wedge N_{l,q} > 1\}$. Maximum pi-blocking is then given by $b_i = \sum_{\ell_q \in A_i} \max\{L_{l,q} \mid d_l > d_i\}$ under the PIP.

For the PCP and SRP to work correctly, the definition of priority ceiling must be changed to reflect relative deadlines as well. Since ties in absolute deadlines are broken in favor of lower-indexed tasks, the priority ceiling of a resource ℓ_q is defined as $\Pi(\ell_q) \triangleq \min\{(d_i, i) \mid N_{i,q} > 0\}$, with the interpretation that $(d_i, i) < (d_x, x) \Leftrightarrow d_i < d_x \vee (d_i = d_x \wedge i < x)$. Condition (i) of the PCP request rule and the SRP scheduling rule must be changed accordingly to compare the relative deadline of each T_i with the system ceiling. With these changes in place, the maximum duration of priority inversion under either the SRP or the PCP is given by $b_i = \max\{L_{l,q} \mid \ell_q \in A_i \wedge d_l > d_i\}$.

Locking can thus be optimally controlled under both EDF and FP scheduling to cause $O(1)$ per-job priority inversions. That there are only small differences in the above definitions highlights that EDF and FP do not differ much when it comes to real-time locking. In fact, when assigning fixed priorities according to the optimal DM policy, the definitions for FP and EDF scheduling are almost equivalent since $i < j \Leftrightarrow d_i \leq d_j$.

RW and k-exclusion locks. We have reviewed the PCP and SRP as they apply to mutex constraints. However, in a generalized form, both protocols support *multi-unit resources* as well (Baker, 1990, 1991; Rajkumar, 1991). A multi-unit resource is a shared resource of which there are x (integral) units in total and each job may request *multiple* instances for use. Concurrent requests may be satisfied simultaneously as long as the total number of the requested units does not exceed the number of units x. While somewhat rare in practice—their primary use case is stack space management (Baker, 1990, 1991)—multi-unit resources are a versatile abstraction that generalizes both mutex ($x = 1$) and k-exclusion constraints (where $x = k$ and each job requests only one unit at a time). Further, multi-unit resources can be used to emulate RW semantics by setting $x = n$ and letting writers request all n units at once, but readers only single units at a time. The fundamental operation of the SRP and PCP, as well as their properties, remain unchanged in the presence of multi-unit resources; we refer the interested reader to (Baker, 1991; Rajkumar, 1991).

To summarize, shared resources with mutex, RW, and k-exclusion constraints and nested requests are well supported in uniprocessor real-time systems. Optimal protocols exist (SRP and PCP) that limit maximum pi-blocking to the length of one (outermost) request while ensuring deadlock-freedom, and there are schedulability tests that correctly account for priority inversions under both EDF and FP scheduling. In essence, predictable locking is a solved problem on uniprocessors.

2.4.4 Multiprocessor Real-Time Locking Protocols

A multiprocessor real-time locking protocol is required when a shared resource may be requested concurrently by jobs on multiple processors. Shared resources can be classified as either *global* or *local* resources. Under partitioning, a resource is local if it is shared only among jobs residing on the same processor, and global otherwise. Local resources can be optimally managed using the uniprocessor PCP or SRP in each partition; multiprocessor locking protocols are thus only needed for global resources. Under global scheduling, all shared resources are necessarily global since it is not known in advance on which processors jobs will execute. Under clustered scheduling, resources could be considered local if they are only accessed by jobs assigned to the same cluster. If $c > 1$, then such "local" resources are still global with respect to their cluster, *i.e.*, local resources do not necessarily reduce to a simpler uniprocessor problem under clustered scheduling. However, intra-cluster resource sharing is equivalent to resource sharing under global scheduling and thus does not warrant individual attention. We therefore assume in the following that each resource ℓ_q is a global resource—shared across clusters or partitions—and that local resources (if there are any) have been dealt with separately.

Compared to the state-of-the-art in uniprocessor locking protocols, the sharing of global resources is much less understood. In fact, prior to our work, RW and k-exclusion protocols for multiprocessor real-time systems had not been studied (in the published literature). Further, suspension-based mutex protocols had only been developed and analyzed in the context of partitioned scheduling, but not for global scheduling, and locking protocols for clustered scheduling had not been considered at all. In Chapters 5 and 6, we study real-time locking protocols for the sharing of global resources with mutex, RW, and k-exclusion constraints under global, clustered, and partitioned scheduling.

We require that requests for global resources are not nested. That is, jobs are not allowed to request additional resources while holding a global resource and are further barred from requesting

global resources while holding a local resource. Global resource requests can thus not lead to deadlock. This limitation is common to all multiprocessor locking protocols proposed to date. In Chapter 5, we discuss a simple "workaround" applicable to any protocol that allows nested requests by reducing nesting to coarse-grained "group locking." To the best of our knowledge, predictable, deadlock-free support for nested requests (without resorting to coarse-grained coalescing of resources into groups) is an open problem.

Prior work on the sharing of global resources has focused on supporting mutex constraints. Most relevant to our work are three suspension-based mutex protocols for partitioned scheduling, and two spin-based mutex protocols (one for partitioned scheduling and one for global scheduling). We begin by reviewing spin-based protocols because they are easier to analyze.

2.4.4.1 Spinlock Protocols

In a spin-based protocol, a job J_i that incurs acquisition delay busy-waits by executing a delay loop until its request is satisfied. Since J_i remains scheduled while spinning, it does *not* incur pi-blocking at the time. Nonetheless, the spinning delays both J_i and also lower-priority jobs since it increases J_i's execution requirement; and must thus be accounted for. To avoid ambiguity, we refer to the delay that J_i incurs due to its *own* busy-waiting as *spin blocking* (*s-blocking*). We let s_i denote an upper bound on the maximum cumulative duration of s-blocking incurred by any J_i. In other words, J_i incurs acquisition delay—and thus spins—for at most s_i time units.

If jobs spin non-preemptively (which they do in all of the considered spin-based protocols), then a spinning job may cause other, newly-released, higher-priority jobs to incur a priority inversion (this is similar to pi-blocking in the NCP). The bound s_i does *not* account for such delays that affect *other* jobs. For example, if a higher-priority job J_1 cannot preempt J_2 because it is spinning non-preemptively, then J_1 incurs pi-blocking (but not s-blocking), which is accounted for by b_1, whereas J_2 incurs s-blocking (but not pi-blocking), which is accounted for by s_2. This is illustrated in Figure 2.35.

Since busy-waiting consumes processor cycles, the execution requirement must be adjusted to reflect the increased processor demand. We denote T_i's *effective execution requirement* as $e'_i = e_i + s_i$. When applying any schedulability test in the presence of spinlocks, the effective execution requirement e'_i must be substituted for e_i when determining u_i, δ_i, *etc.* For example, the

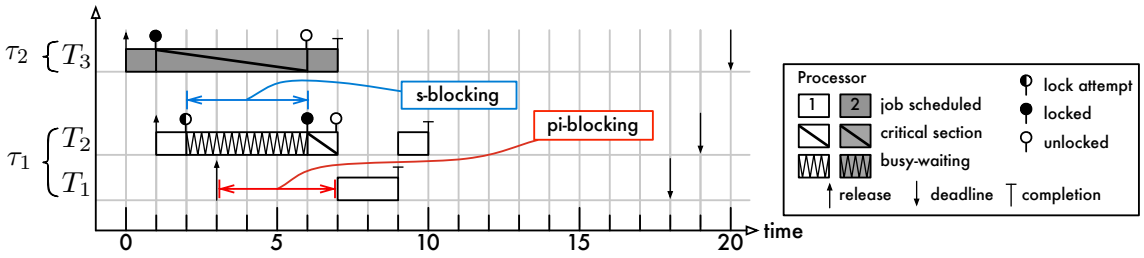


Figure 2.35: Illustration of the difference between pi-blocking and s-blocking. The example shows three tasks under P-FP scheduling on $m = 2$ processors. The tasks $\tau_1 = \{T_1, T_2\}$ are assigned to processor 1; task T_3 is assigned to processor 2. $J_{2,1}$ spins non-preemptively while it waits for $J_{3,1}$ to release the shared resource. $J_{2,1}$ does not incur pi-blocking while it waits because it is scheduled; nonetheless, its response time increases because it is wasting processor cycles. $J_{1,1}$ incurs pi-blocking upon release until time 7, when $J_{2,2}$ becomes preemptable again.

uniprocessor response time bound (Theorem 2.12) is applied as follows:

$$R_i = e'_i + b_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{p_h} \right\rceil \cdot e'_k. \tag{2.7}$$

Since this is a straightforward substitution, we omit restating all relevant schedulability tests and henceforth assume that e'_i is used instead of e_i.

Non-preemptive spinlocks. Gai *et al.* (2003) proposed the *multiprocessor SRP* (MSRP), an extension of Baker's SRP for partitioned scheduling (either P-EDF or P-FP). Somewhat confusingly, the name MSRP derives from the fact that the SRP is used on each processor to arbitrate access to *local* resources; the SRP does not apply to global resources. Priority ceilings are in fact irrelevant for global resources in the MSRP. Instead, the MSRP uses *FIFO spinlocks* to protect global resources. In a FIFO spinlock, waiting jobs form a *spin queue*, and jobs (atomically) append themselves to the end of the spin queue when an acquisition attempt fails.[19] Under the MSRP, to request a global resource ℓ_q, a job J_i becomes non-preemptable and then enqueues itself onto the FIFO spin queue. Once J_i has become the head of the queue, it holds ℓ_q and executes its critical section (and remains non-preemptable). J_i re-enables preemptions when it releases ℓ_q.

Example 2.17. An example MSRP schedule is shown in Figure 2.36. In the depicted scenario, $J_{3,1}$ acquires ℓ_1 at time 1 on processor 1. This causes $J_{4,1}$ to spin non-preemptively on processor 2 during

[19] See (Anderson *et al.*, 2003) for a survey of spinlock implementation techniques.

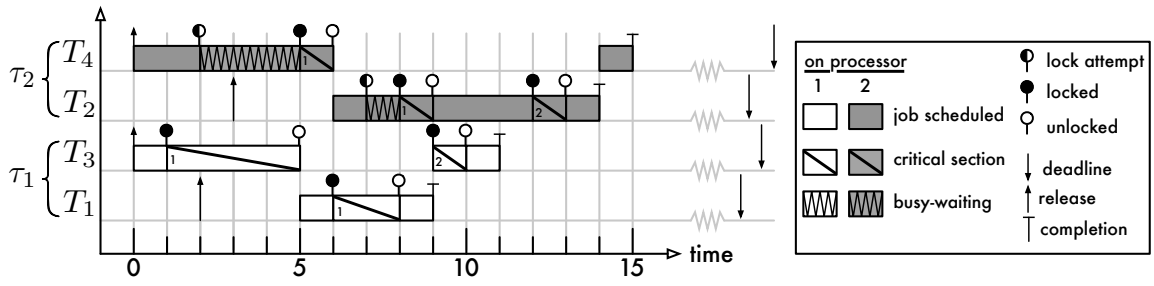


Figure 2.36: Example MSRP schedule. There are four tasks $T_1, \ldots, T_4$ assigned to $m = 2$ processors sharing two resources ℓ_1, ℓ_2 under P-FP scheduling. The two resulting partitions are indicated as τ_1 and τ_2. The digit within each critical section indicates which resource was requested.

$[2, 5)$, which in turn causes $J_{2,1}$ to incur pi-blocking during $[3, 6)$. Similarly, $J_{1,1}$ incurs pi-blocking during $[2, 5)$ on processor 1 because $J_{3,1}$ is executing its critical section and thus is non-preemptable. $J_{3,1}$ and $J_{2,1}$ acquire ℓ_2 later during their execution, but their requests happen to not overlap, and thus neither job incurs s-blocking in this particular case.[20] ◊

Because jobs spin and execute requests non-preemptively, there can be at most m concurrent requests for global resources. Together with the FIFO ordering, this implies that at most $(m-1)$ jobs precede J_i in the spin queue for ℓ_q. This greatly simplifies bounding maximum s-blocking. Recall that τ_k denotes the set of tasks assigned to processor k, and that P_i denotes the processor to which T_i is assigned. A single request to resources ℓ_q causes J_i then to spin for at most

$$spin(T_i, \ell_q) = \sum_{\substack{k=1 \\ k \neq P_i}}^{m} \max\{L_{j,q} \mid T_j \in \tau_k\}. \tag{2.8}$$

In other words, only the longest request issued by any job on each remote processor must be considered when bounding J_i's maximum spin blocking due to a single request. Based on this per-request bound, Gai *et al*. (2003) derived the following bound on the maximum s-blocking incurred by any J_i under the MSRP:

$$s_i = \sum_{q=1}^{n_r} N_{i,q} \cdot spin(T_i, \ell_q).$$

[20] These requests are included to expose certain protocol properties in the later Examples 2.18, 2.19, and 2.20.

We derive a similar but more accurate bound for FIFO spinlocks that is less pessimistic if $N_{i,q} > 1$ in Chapter 5.

As mentioned above, a second concern is a priority inversion that might be caused by a non-preemptively spinning job at the time of J_i's release. This is similar to a priority inversion caused by the NCP or the SRP's scheduling rule. In fact, since both the priority inversion due to the SRP (local resource sharing) and the priority inversion due to spinning (global resource sharing) have to occur at the time of J_i's release, any single J_i cannot incur both types of pi-blocking. Let b_i^{SRP} denote the maximum pi-blocking due to local resource requests, and let b_i^{NP} denote the maximum pi-blocking due to global resource requests. Then each J_i incurs at most $b_i = \max(b_i^{NP}, b_i^{SRP})$ pi-blocking. The local pi-blocking b_i^{SRP} can be determined by uniprocessor analysis (since local and global resources cannot be nested). Under P-FP scheduling, the maximum duration of priority inversion due to a lower-priority job accessing a global resource is bounded by

$$b_i^{NP} = \max\{spin(T_k, l_q) + L_{k,q} \mid T_k \in \tau_{P_i} \wedge N_{k,q} > 0 \wedge k > i\}.$$

As in the uniprocessor case, "$k > i$" should be substituted with "$d_k > d_i$" under P-EDF scheduling. With both b_i and s_i bounded, schedulability under the MSRP can be established by applying either Theorem 2.12 or Theorem 2.13 under consideration of effective execution requirements to each processor.

Devi *et al*. (2006) analyzed global resource sharing with non-preemptive FIFO spinlocks under G-EDF. Their protocol works essentially the same as discussed above in the description of the MSRP: jobs first become non-preemptable, then enqueue themselves in a FIFO spin queue and busy-wait until their request is satisfied, and finally become preemptable again when they release the source. Since jobs are not bound to processors, the maximum duration of spinning cannot be bounded on a per-processor basis as in Equation (2.8) above. However, the non-preemptive spinning still limits the maximum number of concurrent requests to m, and thus to $(m-1)$ preceding jobs. Devi *et al*. (2006) derived the following simple bound:

$$spin(T_i, \ell_q) = (m-1) \cdot \max_{\substack{1 \le k \le n \\ k \neq i}} \{L_{k,q}\}. \tag{2.9}$$

We derive a less-pessimistic bound that takes the identity of blocking jobs and the frequency at which they issue requests into account in Chapter 5.

A key contribution of Devi *et al.*'s work was to show that G-EDF ensures bounded tardiness even in the presence of non-preemptive sections (Devi *et al.*, 2006; Devi, 2006; Devi and Anderson, 2008). However, their analysis assumes that each job suffers a priority inversion at most once, and only at the time of its release, as it is the case on a uniprocessor (and thus also under partitioning). As we show in Chapter 3, a straightforward, "eager" implementation of non-preemptive sections in a global scheduler does *not* ensure that this is indeed the case. Instead, a "linking mechanism" is required to enact preemptions "lazily"; our solution is detailed in Section 3.3.3.

This concludes our review of spin-based locking protocols under event-driven schedulers. In Chapter 5, we generalize the use of non-preemptive FIFO spinlocks to RW constraints and to clustered JLFP scheduling, and derive a flexible blocking analysis framework that takes both response-time bounds and task periods into account to derive less-pessimistic bounds on s-blocking.

The protocols discussed so far do not apply to PD^2. Locking in pfair-scheduled systems is more challenging than under JLFP scheduling since jobs are preempted more frequently. Holman and Anderson studied this problem in detail and proposed several spin-based and suspension-based locking protocols for PD^2 (Holman, 2004; Holman and Anderson, 2006). However, in our overhead-aware evaluation of schedulers presented in Chapter 4, we found PD^2 to not perform well on our test platform even if tasks are independent. We therefore focus on locking under event-driven JLFP schedulers in this dissertation and review suspension-based protocols for such schedulers next.

2.4.4.2 Semaphore Protocols

Rajkumar *et al.* proposed the first real-time locking protocols for multiprocessors (Rajkumar *et al.*, 1988; Rajkumar, 1990, 1991). They proposed two extensions of the PCP for distributed- and shared-memory multiprocessors under P-FP scheduling, namely the *distributed PCP* (Rajkumar *et al.*, 1988; Rajkumar, 1991) and the *multiprocessor PCP* (Rajkumar, 1990, 1991), which are denoted as DPCP and MPCP, respecitvely.[21] Even though we investigate only shared-memory multiprocessors

[21] Originally, the DPCP was referred to as the "multiprocessor PCP" in (Rajkumar *et al.*, 1988) and the MPCP was called the "shared memory synchronization protocol" in (Rajkumar, 1990). Later, the names DPCP and MPCP as they are used today were introduced in (Rajkumar, 1991). Unfortunately, this renaming has caused some confusion; for example, Liu (2000) describes the DPCP but refers to the described protocol as the "MPCP."

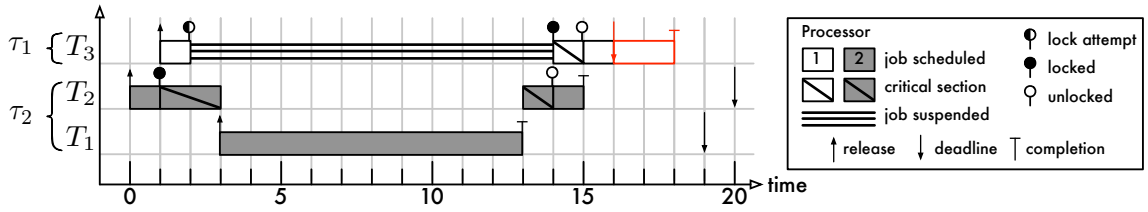


Figure 2.37: Example schedule of three tasks on two processors under P-FP scheduling. Despite using the PIP, $J_{3,1}$ suffers a priority inversion during $[2, 14)$, which includes the entirety of $J_{1,1}$'s execution. Consequently, $J_{1,1}$ misses its deadline at time 16. This shows that priority inheritance is ineffective at preventing unbounded priority inversions when applied across partition (or cluster) boundaries.

in this thesis, we include the DPCP because it is also applicable to shared-memory systems, and because neither design is obviously "better" than the other, *i.e.*, it is not clear which yields lower blocking for a given task set. Similarly to the MSRP, the names are slightly confusing since the PCP rules are not applied across partition boundaries. Indeed, one of the primary features of the PCP—deadlock avoidance—is irrelevant for global resources because nesting is disallowed. Further, priority inheritance, which is a key part of the PCP, is (by itself) insufficient to ensure the progress of resource-holding jobs since it is ineffective across partition (or cluster) boundaries.

The limits of priority inheritance. Priority inheritance is ineffective under partitioned scheduling in the sense that priority inversions are not necessarily bounded under it. Figure 2.37 depicts a schedule that may arise if the PIP is applied across partitions. J_3 misses its deadline because it is delayed for virtually the *entire* duration of J_1's execution *despite* priority inheritance since $\mathsf{y}(J_1, 2) = \mathsf{Y}(J_1, 2) = 1 < \mathsf{y}(J_2, 2) = \min\{2, 3\} = 2$. Note that J_3 is subject to a priority inversion throughout the entire interval since it is pending but its assigned processor is idle. The underlying problem is that priority comparisons across cluster boundaries are essentially meaningless—even though J_3's priority is numerically lower than J_1's priority, J_3 is still the highest-priority job on its assigned processor and should thus be scheduled. This example shows that a priority inversion can be effectively unbounded even if priorities are inherited, *i.e.*, its length cannot be bounded solely in terms of request lengths.

Instead, both multiprocessor PCP variants use a technique that we call *priority boosting* to expedite the completion of requests for global resources (once granted). Under priority boosting, the

priority of a resource-holding job is unconditionally raised above that of other, non-resource-holding jobs, *i.e.*, above the range of "normal" priorities. This ensures that a resource-holding job cannot be preempted by a newly-released job (which cannot hold a resource yet) and thus prevents unbounded priority inversion. In this regard, priority boosting is similar to non-preemptive execution. However, unlike non-preemptive execution, priority-boosted jobs may be preempted by other priority-boosted jobs.

Besides the use of priority boosting, the MPCP and DPCP also share the same queuing discipline. If multiple jobs request the same resource at the same time, the requests are satisfied in order of decreasing base priority (*i.e.*, increasing task index). The major difference between the two protocols is *where* requests are executed. We first review the earlier-proposed DPCP.

Remote resources. The DPCP was first described by Rajkumar *et al*. (1988) and later discussed in greater detail by Rajkumar (1991). Due to its intended use in distributed-memory multiprocessors, the DPCP assumes that each resource is (physically) accessible only from a specific processor. We let Q_q denote the processor that resource ℓ_q is accessible from. In a shared-memory system, Q_q can be determined arbitrarily, but Q_q must be fixed prior to analysis. For example, in Chapter 7, we use the assignment rule $Q_q \triangleq (q \bmod m) + 1$.

The DPCP implements the *remote procedure call* (RPC) model; that is, jobs do not access resources directly, but instead delegate their requests to *local agents*. A local agent A_i^x is located on processor x and carries out requests for resources local to processor x on behalf of T_i. When a job J_i requires a resource ℓ_q and ℓ_q is located on processor x (*i.e.*, $Q_q = x$), then it submits a request to its local agent A_i^x. The local agent becomes *active* when it receives the request. RPCs are synchronous under the DPCP: after issuing a request $\mathcal{R}_{i,q,v}$ to its local agent A_i^x, J_i suspends until it is resumed again when $\mathcal{R}_{i,q,v}$ has been completed by A_i^x. Recall that execution of requests for global resources are priority boosted. Since jobs do not execute requests themselves, their effective priority is left unchanged when they request global resources. Instead, the base priority of a local agent A_i^x is defined as $\mathsf{Y}(A_i^x, t) = i - n$, where the term "$-n$" ensures that the lowest-priority agent has a higher priority than the highest-priority "regular" job.[22] On each processor x, the agents (if any) access the resources that are accessible from processor x as local resources according to the rules of the PCP

[22]The description of the DPCP in (Rajkumar *et al*., 1988; Rajkumar, 1991) allows priorities to be assigned in any way that satisfies certain properties. We follow the simpler description given in (Liu, 2000).

(where the priority ceiling of global resources is defined with respect to agent priorities). If multiple jobs request the same resource at the same time, then the PCP ensures that requests are serviced in order of the issuing jobs' base priorities since they are reflected in the agents' base priorities. On its assigned processor P_i, J_i may serve as its own agent (in which case J_i's effective priority is set accordingly).

Example 2.18. Figure 2.38 shows resource sharing under the DPCP for the scenario previously discussed in Example 2.17. In this example, there are two global resources that reside on processor 1. Thus, four agents $A_1^1, \ldots, A_4^1$ are also assigned to processor 1 in order to act on behalf of $T_1, \ldots, T_4$. (T_1 and T_3 could act as their own agents in a real implementation; we show their agents here for clarity.) Agent A_3^1 is invoked at time 1 when $J_{3,1}$ requests ℓ_1. Shortly thereafter, A_4^1 becomes active at time 2 when $J_{4,1}$ requests ℓ_1. However, since A_3^1 is executing at the time, A_4^1 has insufficient priority to be scheduled. Similarly, A_2^1 becomes active at time 4. Since A_2^1 has higher priority than A_3^1, it can attempt to lock ℓ_1. This, however, fails since A_3^1 still holds ℓ_1 at the time. When A_3^1 releases ℓ_1, A_2^1 gains access next because it is the highest-priority active agent on processor 1. Note that, even though the highest-priority job $J_{1,1}$ is released at time 2, it is not scheduled until time 7 because agents have an effective priority that exceeds the base priority of $J_{1,1}$. A_2^1 becomes active again at time 9 since $J_{2,1}$ requests ℓ_2. However, A_1^1 is accessing ℓ_1 at the time and has a priority that exceeds A_2^1's priority. Therefore, A_2^1 is not scheduled until time 10. $J_{1,1}$ is also resumed at time 10, but not scheduled until time 11 since A_2^1 is now executing the request $J_{2,1}$. In total, $J_{1,1}$ suffers priority inversions for a total of six time units during $[2, 7)$ and $[10, 11)$ before it completes. $\Diamond$

Besides priority inversions due to the sharing of local resources, there are three sources of pi-blocking under the DPCP. (Since neither jobs nor agents spin, there is no s-blocking.)

1. Every time a job J_i requests a resource ℓ_q, the corresponding local agent may be delayed for the duration of one critical section because an agent of some lower-priority job raised the priority ceiling on Q_q. For example, in Figure 2.38, this happens to $J_{2,1}$ at time 4. In total, J_i may thus be pi-blocked by $\sum_q N_{i,q}$ requests of lower-base-priority tasks.

2. Every time some A_i^x is active on behalf of J_i, it may be preempted or delayed by agents of higher-priority tasks (on every remote processor), which in turn causes J_i to be delayed. For

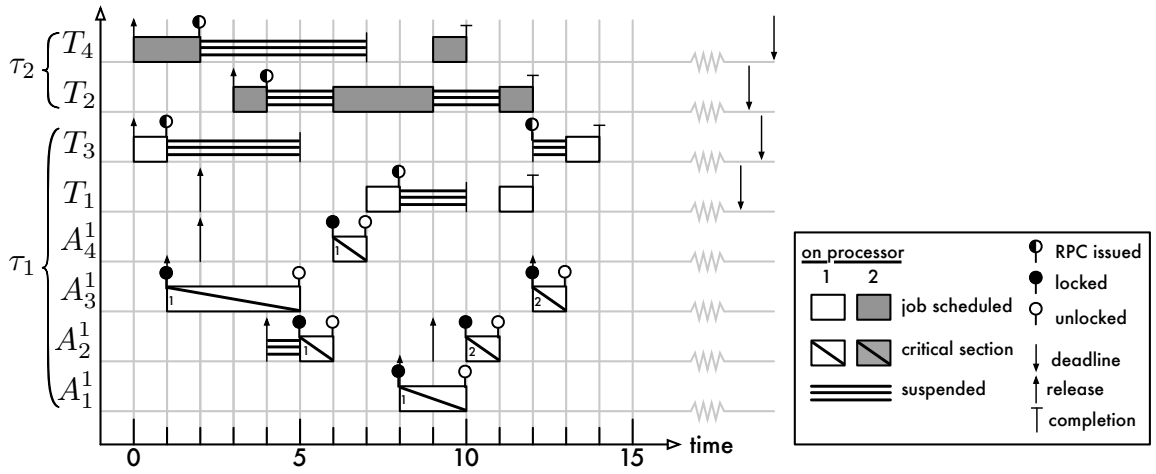


Figure 2.38: Example DPCP schedule for the same scenario as shown in Figure 2.36. There are four tasks $T_1, \ldots, T_4$ assigned to $m = 2$ processors sharing two resources ℓ_1, ℓ_2 under P-FP scheduling. Both resources are local to processor 1; there are thus four agents, $A_1^1, \ldots, A_1^4$, assigned to processor 1. An agent is "released" when it receives an RPC request from a client job. The schedule is discussed in Example 2.18.

example, in Figure 2.38, A_2^1 is not scheduled at time 9 because the higher-priority $A_{1,1}$ is active, which lengthens the priority inversion experienced by $J_{2,1}$.

3. All agents on T_i's processor (*i.e.*, each A_k^x for which $x = P_i$) that act on behalf of tasks other than T_i potentially delay the execution of J_i. This happens repeatedly to both $J_{1,1}$ and $J_{3,1}$ in Figure 2.38. For example, $J_{3,1}$ is pi-blocked during $[6, 7)$ since A_4^1 is active on behalf of $J_{4,1}$.

Deriving an analytical expression that bounds all three sources of delay is somewhat tedious and omitted here in the interest of brevity; the interested reader is referred to (Rajkumar *et al.*, 1988; Rajkumar, 1991; Liu, 2000) instead. From the discussed example, however, it is intuitively apparent that each resulting b_i is typically much larger than in the case of uniprocessor protocols. A potential benefit of the DPCP is that requests that are executed on remote processors (*i.e.*, not on P_i) do not contribute to e_i.

Direct access. The MPCP is a later version of the DPCP for shared-memory multiprocessors (Rajkumar, 1990, 1991). It assumes that each shared resource can be accessed from any processor, which removes the need for RPCs and local agents. Instead, jobs execute their requests themselves on their assigned processors and are priority-boosted while doing so. Specifically, a job J_i's priority while

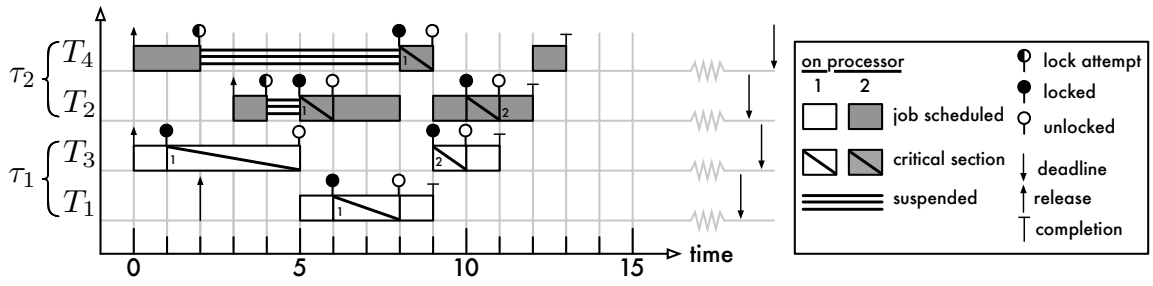


Figure 2.39: Example MPCP schedule for the same scenario as shown in both Figure 2.36 and Figure 2.38. The schedule is discussed in Example 2.19.

holding a resource ℓ_q is boosted to the highest priority of any *remote* task that shares ℓ_q (again offset by "$-n$" as a boosting mechanism). Formally, J_i's effective priority at time t is given by

$$\mathsf{y}(J_i, t) = \begin{cases} \min\{k \mid T_k \in \tau \wedge P_k \neq P_i \wedge N_{k,q} > 1\} - n & \text{if } J_i \text{ holds } \ell_q \text{ at time } t, \\ \mathsf{Y}(J_i, t) = i & \text{otherwise.} \end{cases} \tag{2.10}$$

Should there be a tie in effective priority (because the "highest remote priority" is not necessarily unique), then the earlier-boosted job has precedence to avoid unnecessary preemptions. If there are multiple concurrent unsatisfied requests for a resource ℓ_q, then jobs gain access to ℓ_q in order of decreasing base priority. Note that the MPCP does not use priority inheritance (for global resources); priority boosting is sufficient to guarantee rapid completion of critical sections that delay higher-priority jobs.

Example 2.19. An example MPCP schedule is shown in Figure 2.39. The depicted scenario is the same as previously discussed in both Example 2.17 and Example 2.18. Local agents are no longer required since jobs access global resources directly. $J_{4,1}$ suspends at time 2 since $J_{3,1}$ already holds ℓ_1. Similarly, $J_{2,1}$ suspends at time 4 until it holds ℓ_1 one time unit later. Meanwhile, on processor 1, $J_{1,1}$ is scheduled at time 5 after $J_{2,1}$ returns to its base priority and also requests ℓ_1 at time 6. Since resource requests are satisfied in priority order, $J_{1,1}$'s request has precedence over $J_{4,1}$'s request, which was issued much earlier at time 2. Thus, $J_{4,1}$ must wait until time 8 to access ℓ_1. Note that $J_{4,1}$ preempts $J_{2,1}$ when it resumes at time 8 since it is holding a global resource. $\Diamond$

Similar to the DPCP, there are multiple sources of pi-blocking due to global resource sharing that must be accounted for.

1. Each time that J_i requests some resource ℓ_q, ℓ_q may already be held by a lower-priority job. This can occur at most $\sum_q N_{i,q}$ times. In Figure 2.39, this happens to $J_{2,1}$ at time 4 when $J_{3,1}$ is still holding ℓ_1.

2. While J_i waits to acquire ℓ_q, the resource may be requested by higher-priority jobs. This may happen repeatedly since the MPCP uses a priority queue to order waiting jobs. For example, in Figure 2.39, $J_{4,1}$ is delayed by a later request of $J_{1,1}$ at time 6.

3. While J_i waits on processor P_i for some other job J_x to release ℓ_q on processor P_x, where $P_x \neq P_i$, J_x may be preempted by other jobs on P_x with higher effective priority. Such preemptions will transitively delay J_i. (This does not happen in Figure 2.39.)

4. Prior to J_i's release and each time J_i suspends waiting for a global resource, lower-priority jobs on P_i may execute and request global resources. This may cause them to subsequently become priority-boosted while J_i is scheduled with its effective priority equal to its base priority. In total, this may happen up to $\sum_q N_{i,q} + 1$ times with *each* local lower-priority task. The "$+1$" term is due to the fact that lower-priority jobs may issue requests before J_i is released. For example, in Figure 2.39, $J_{2,1}$ is pi-blocked during $[8, 9)$ because $J_{4,1}$ is priority-boosted due to a request that $J_{4,1}$ issued before $J_{2,1}$ was released.

Unsurprisingly, accounting accurately for each of the four listed factors is a (tedious) challenge. The MPCP was first proposed by Rajkumar (1990, 1991). In recent work, Lakshmanan *et al.* (2009) presented much improved analysis, which we use in Chapter 7, and an MPCP-aware partitioning heuristic. We omit restating Lakshmanan *et al.*'s MPCP-aware response time analysis for the sake of brevity and refer the interested reader to (Lakshmanan *et al.*, 2009).

Deferred execution. Both the MPCP and the DPCP, as well as any other suspension-based multiprocessor locking protocol, are subject to an additional scheduling penalty. Perhaps counterintuitively, there is a fundamental difference between suspensions due to remote jobs, which are also called *self-suspensions*, and suspensions due to local jobs. When jobs self-suspend while waiting for a remotely held resource or when waiting for an agent's reply, jobs *defer* part of their execution to

a later point in time. Deferred execution has a major impact on schedulability analysis because it allows jobs to carry additional work into the analysis window (Rajkumar *et al*., 1988; Audsley *et al*., 1993; Strosnider *et al*., 1995). Therefore, a *suspension-aware* schedulability test must be used to account for additional delays created by remote blocking. In the case of FP scheduling, Audsley *et al*. (1993) developed response-time analysis that accounts for release jitter, which is equivalent to a self-suspension immediately after a job's release. Their analysis was adapted by Ming (1994) to allow for self-suspensions during a job's execution. In the context of suspension-based multiprocessor locking protocols, Audsley *et al*. and Ming's work yields the following response-time bound, which was used in (Lakshmanan *et al*., 2009).

Theorem 2.14 (Audsley *et al*., 1993; Ming, 1994)**.** *Let b_i^r denote the maximum duration J_i that is suspended waiting for* ***remote*** *jobs to release some resource, and let b_i^l denote a bound on the maximum duration of priority inversion due to* ***local*** *jobs. An upper bound on the maximum response time r_i under FP scheduling (on each processor) is given by the smallest R_i that satisfies:*

$$R_i = e_i + b_i^l + b_i^r + \sum_{k=i+1}^{n} \left\lceil \frac{R_i + b_k^r}{p_k} \right\rceil \cdot e_k.$$

Under P-FP, Theorem 2.14 should be applied on a per-processor basis, *i.e.*, R_i only depends on T_k for which $P_k = P_i$. Note that if $b_i^r = 0$ for each T_i, then Theorem 2.14 reduces to Theorem 2.12, which applies to suspension-based uniprocessor locking protocols. Why are suspensions due to local jobs less troublesome than self-suspensions? Unfortunately, a detailed explanation (or proof) is beyond the scope of this dissertation. However, as an intuitive explanation, consider a job $J_{i,j}$ that arrives at time $a_{i,j}$ and that finishes at time $f_{i,j}$. If $J_{i,j}$ blocks only on local jobs, then the processor is never idle during $[a_{i,j}, f_{i,j})$—$J_{i,j}$'s execution window corresponds to a so-called "busy-interval." However, if $J_{i,j}$ blocks on remote jobs, *i.e.*, if $J_{i,j}$ self-suspends, then idle time may exist during $[a_{i,j}, f_{i,j})$. This idle time complicates schedulability analysis considerably. A detailed explanation of the effects of self-suspensions can be found in (Audsley *et al*., 1993; Strosnider *et al*., 1995; Liu, 2000).

One of the main contributions of this dissertation is an investigation of "pi-blocking optimality." As it turns out, how self-suspensions are accounted for has a significant impact on pi-blocking. We revisit the topic of suspension-aware schedulability tests in Chapter 6.

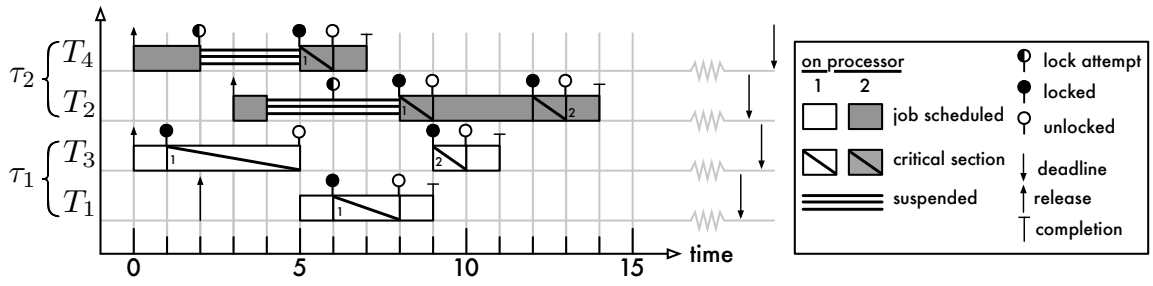


Figure 2.40: Example MPCP-VS schedule for the same scenario as shown in Figure 2.39. "Virtual spinning" occurs on processor 2: $J_{2,1}$ executes during $[3, 4)$ while $J_{4,1}$ waits, but when $J_{2,1}$ requires ℓ_1, it may not issue its request until $J_{3,1}$ releases ℓ_1 at time 6.

Virtual spinning. Besides deriving better blocking analysis for the original MPCP, Lakshmanan *et al*. (2009) also proposed a new variant of the MPCP based on a request rule that they termed "virtual spinning." Under "virtual spinning," jobs do *not* in fact spin while waiting for a global resource to become available. Instead, they suspend just as under the original MPCP and other local (lower-priority) jobs may be scheduled. However, no other local jobs may issue global requests until the suspended job has ceased to spin "virtually." We refer to this MPCP variant as MPCP-VS and reserve the word "spinning" to denote busy-waiting that does not involve a context switch or suspension.

Example 2.20. Figure 2.40 shows how the MPCP-VS rule changes the MPCP schedule from Figure 2.39. Since the jobs on processor 1 do not incur acquisition delay, they are not affected by the MPCP-VS rule. On processor 2, $J_{4,1}$ must wait at time 2 when it requests ℓ_1. It suspends, which gives $J_{2,1}$ a chance to execute when it is released at time 3. If $J_{4,1}$ were actually spinning non-preemptively, then $J_{2,1}$ would not have been scheduled until time 6 (*e.g.*, this is the case in the MSRP schedule shown in Figure 2.36). However, $J_{2,1}$ may not issue requests. When it requires ℓ_1 at time 4, it is forced to suspend *without* issuing a request because $J_{4,1}$'s request is not complete yet. $J_{2,1}$'s request is issued belatedly at time 6 when $T_{4,1}$ releases ℓ_1. $\Diamond$

The MPCP-VS rule has an impact on the analytical blocking bounds derived in (Lakshmanan *et al*., 2009), but from an inspection of the bounds it is not obvious whether the MPCP-VS is more or less pessimistic than the MPCP. We consider both variants in our locking protocol comparison study presented in Chapter 7. Interestingly, the MPCP-VS analysis presented by Lakshmanan *et al*. (2009)

is not subject to scheduling penalties due to deferred execution because it accounts for suspensions as if suspended task were spinning. We refer to this approach as "suspension-oblivious" schedulability analysis (see Chapter 6). The MPCP-VS could also be implemented with actual spinning; however, a priority-ordered spinlock would be required to do so, which takes away some of the simplicity that makes (FIFO) spinlocks appealing in the first place.

Local resources. As noted in the discussion of the MSRP, DPCP, and MPCP, local resources can be handled with a uniprocessor locking protocol under partitioned scheduling. In this dissertation, we are only concerned with global resources because uniprocessor real-time locking is essentially a solved problem, and because virtually any multiprocessor real-time locking protocol for partitioned scheduling can be integrated with either the PCP or SRP (since nesting of local and global resources is disallowed).

Nonetheless, there is one interaction between suspension-based multiprocessor locking protocols and the PCP that should be pointed out. On a uniprocessor, the PCP and SRP both ensure that any job is blocked at most once by lower-priority jobs. When jobs may defer execution, *i.e.*, if they self-suspend while waiting for any kind of external event (such as a remote job releasing a resource or an I/O device returning data), then this is no longer the case. Lower-priority jobs may acquire local resources during those times, which may cause the priority ceiling to be raised. Thus, each time that a job resumes after a self-suspension, it may incur an additional priority inversion.

For example, under the DPCP and the MPCP, J_i may suffer a priority inversion due to local resources up to $1 + \sum_q N_{i,q}$ times.[23] The MPCP-VS prohibits only *global* resource requests while a job is self-suspended;[24] consequently, each J_i may suffer up to $1 + \sum_q N_{i,q}$ priority inversions due to the PCP as well. In contrast, the MSRP is not affected by this since jobs do not self-suspend while waiting for global resources. The sharing of global resources thus makes local resource sharing more pessimistic if jobs suspend while waiting. As all resources are presumed global in the remainder of this dissertation, this is of little concern to our experiments.

[23] This is in addition to $\sum_q N_{i,q} + 1$ priority inversions due to requests for *global* resources by each lower-priority task.

[24] Specifically, the definition of "virtual spinning" states: "[...] other tasks are allowed to execute unless they try to access global critical sections." (Lakshmanan *et al.*, 2009, p. 474).

2.4.5 Further Results in Real-Time Locking

Besides the closely related locking protocols discussed in detail in the preceding sections, several other real-time locking protocols have been proposed in prior work.

Uniprocessors. In early work on real-time locking protocols for uniprocessors, Chen and Lin extended the PCP to multi-unit resources (Chen and Lin, 1990b) and to EDF scheduling (Chen and Lin, 1990a). Jeffay (1992) proposed an alternative to the SRP for EDF-scheduled systems.

Lamastra *et al*. (2001) proposed a *bandwidth inheritance protocol* for uniprocessors that applies the concept of priority inheritance to *open real-time systems*, where tasks may join and leave task sets at runtime, and are expected to (occasionally) overrun their processor allocations. The bandwidth inheritance protocol proposed by Lamastra *et al*. (2001) ensures that a high-priority job is not transitively pi-blocked when a lower-priority, resource-holding job exhausts its processor budget.

In work on the uniprocessor SRP, Baruah (2006) derived an algorithm to find an optimal resource replication strategy to meet a given upper bound on pi-blocking. Fisher *et al*. (2007a) applied the SRP to open real-time systems and derived schedulability tests suitable for online admission control They further derived bounds on the maximum resource-holding times under both FP scheduling (Bertogna *et al*., 2007) and EDF scheduling (Fisher *et al*., 2007b).

Multiprocessors. In early work on multiprocessors, Chen and Tripathi (1994) considered PCP-based locking protocols for P-EDF scheduling. In later work, López *et al*. (2004b) presented a partitioning heuristic that transforms global resources into local resources, thereby avoiding remote pi-blocking completely (albeit at the expense of creating a much more difficult partitioning problem), and derived corresponding worst-case utilization bounds under P-EDF scheduling.

In work on preemptable busy-waiting, Takada and Sakamura studied preemptive spinlocks (Takada and Sakamura, 1994) and spinlocks with "helping," where blocked jobs complete the requests of preempted resource-holding jobs (Takada and Sakamura, 1997). In the same context, Wang *et al*. (1996) proposed spinlocks with priority inheritance. Anderson *et al*. (1998) proposed a much simpler preemptive spinlock with optional helping for quantum-driven schedulers.

In work on pfair-scheduled multiprocessors, Holman and Anderson proposed protocols based on *blocking zones*, where an ill-timed request is automatically delayed until the next quantum,

request skipping, where requests can be preempted and restored, and *request rollback*, where requests can be aborted and retried (Holman, 2004; Holman and Anderson, 2006). They further studied suspension-based locking protocols under PD^2 and showed how to account for acquisition delays under pfair schedulability analysis. We do not consider PD^2 in our evaluation of locking protocols in Chapter 7 since we found PD^2 to not perform well on our test platform even if tasks are independent, as discussed in detail in Chapter 4.

In very recent work, Easwaran and Andersson (2009) studied locking protocols for G-FP scheduling and presented the suspension-based *parallel priority-ceiling protocol* and an analysis of the PIP under G-FP scheduling. Andersson and Easwaran (2010) also designed a multiprocessor locking protocol for which they derived a resource augmentation factor. Macariu and Cretu (2011) proposed an extension of the PIP under G-FP scheduling that limits the extent of pi-blocking due to the execution of resource-holding lower-priority jobs with raised effective priorities. Finally, Faggioli *et al*. (2010) presented a scheduler-agnostic, spin-based locking protocol that applies Lamastra *et al*.'s notion of bandwidth inheritance to multiprocessors.

This concludes our review of the algorithmic foundations of real-time scheduling and locking, on both uniprocessor and multiprocessor systems. Next, we summarize some needed RTOS background before we discuss our implementation of the reviewed schedulers and locking protocols in LITMUS$^{\mathrm{RT}}$ in Chapter 3.

2.5 Real-Time Operating Systems

Unlike the desktop OS market, the RTOS market is split among many major and minor vendors. As a result, scores of RTOSs have been developed and are in use today. For example, Wikipedia's (incomplete) list of RTOSs contains more than one hundred products and projects (Wikipedia, 2011). One reason for this wealth of choices is that the label "RTOS" is liberally applied to many rather minimal OSs intended for use in embedded systems. The small feature set required of such specific-use "micro controller runtime environments" makes it viable for a skilled developer to design and implement an "RTOS" from the ground up with reasonable effort. In fact, an industry survey conducted in 2005 found that 17 percent of the polled companies develop an RTOS internally rather than (or in addition to) licensing an established RTOS (Turley, 2005). This contributes to

the proliferation of "embedded RTOSs" with near-identical features and design. However, the vast majority of RTOSs of this type do not support multiprocessors and are thus of little relevance to this dissertation. Nonetheless, even when only considering established "major" RTOSs that support shared-memory multiprocessors, a large number of competing RTOSs with many similarities remain.

In the following, we provide an overview of current multiprocessor-capable RTOSs to provide a context for our work on LITMUS$^{\mathrm{RT}}$. Given the scope of the RTOS market, we group similar RTOSs into categories and discuss one representative example from each relevant category. Since each RTOS vendor strives to differentiate its products from the offerings of its competitors, any categorization is necessarily going to gloss over (minor) differences in feature set, implementation details, and customer focus. Further, many RTOSs are quite versatile and fit into more than one category, depending on the employed configuration. Nonetheless, we believe that the following discussion provides a reasonably complete overview of the current RTOS landscape. An alternate taxonomy, together with a survey of RTOSs for uniprocessors and distributed-memory multiprocessor real-time systems can be found in (Stankovic and Rajkumar, 2004).

From a high-level view, current RTOSs can be distinguished either as having been purpose-built specifically for use in real-time systems, or as being based on a GPOS that has been retrofitted for real-time use. The majority of commercial offerings fall into the former category, which we review in Section 2.5.1 below. "Real-Time Linux," of which there are many variants, is the most prominent example of the latter category and is considered thereafter in Section 2.5.2. Next, we briefly digress to define two important RTOS-related concepts prior to the actual RTOS survey.

Interrupt latency. A metric that is commonly used when discussing and comparing RTOSs is interrupt latency, sometimes also called scheduling latency. Informally speaking, interrupt latency describes how quickly the RTOS reacts to interrupts. Suppose an event that triggers the release of a job of the highest-priority task occurs at time t_e, but the corresponding job is not scheduled until time t_s, where $t_s \geq t_e$. The difference $t_s - t_e$ is called *interrupt latency*. If a kernel disables interrupt delivery or suppresses calls to the scheduler for extended times, then it risks high interrupt latency.

In practice, "real-time" is sometimes equated with low interrupt latency, *i.e.*, an OS's ability to transfer control to real-time tasks as quickly as possible. In this interpretation, "real-time" is reduced to a suitably low bound on worst-case latency, which is a regrettable over-simplification. Obviously,

latency crucially impacts the kind of guarantees that can be made, but even a system with negligible observed latency is not truly a real-time system if it is constructed from algorithms that do not lend themselves to schedulability analysis; such a system is merely fast.

So what is interrupt latency, and how should it be accounted for? Interrupt latency prevents a job that *should* be scheduled from being scheduled—a priority inversion. From a schedulability analysis point of view, interrupt latency is thus simply a cause of pi-blocking. When determining the worst-case response time of a task, interrupt latency during $[t_e, t_s)$ is equivalent to the job having self-suspended during that time. Special accounting techniques are thus not required to incorporate interrupt latency into schedulability analysis. In Section 3.4.2, we discuss how to account for "release overhead," which includes interrupt latency.

POSIX compliance. In practice, the label "real-time POSIX compliant" is sometimes used as a synonym for "UNIX-like real-time kernel." However, actual compliance to the POSIX real-time standard is much more differentiated. The POSIX standard for "real-time and embedded application support" (IEEE, 2003) defines four *system profiles* that target different use cases:

1. the *Minimal Realtime System Profile*, denoted PSE51;
2. the *Realtime Controller System Profile*, denoted PSE52;
3. the *Dedicated Realtime System Profile*, denoted PSE53; and
4. the *Multi-Purpose Realtime System Profile*, denoted PSE54.

Each profile includes the functionality of lower-indexed profiles.

Profile PSE54 essentially matches the notion "fully UNIX-compatible real-time system." In contrast, profiles PSE51 and PSE52 do not even require multiprocessing: the whole system may be executed in a single address space. These profiles, however, do include multithreading and FP scheduling. They further do not require a real file system, networking, or virtual memory. From a high-level view, the main difference between profiles PSE51 and PSE52 is that profile PSE52 requires more advanced I/O interfaces and a simple file system.

Profile PSE53 matches the feature set of a "UNIX-like OS" much more closely. It requires full multiprocessing with dynamic process creation, FP scheduling, synchronization, and IPC support, a hierarchical file system, networking support, and virtual memory. The main difference between

PSE53 and PSE54 is that PSE54 includes multiuser support (*i.e.*, access rights, *etc.*), whereas PSE53 assumes execution in an embedded systems environment without multiple user identities.

In the following, we follow common practice and denote an RTOS as "POSIX compliant" to indicate that it is (mostly) compatible with profile PSE53 or better.

2.5.1 Purpose-Built Real-Time Operating Systems

The systems discussed in this section represent the "classic" definition of an RTOS: conceived, architected, and implemented specifically for hosting real-time workloads, with GPOS concerns such as throughput or developer comfort being secondary goals (at best). Their uncompromising design makes them typically the best fit for environments with stringent real-time constraints (*i.e.*, very short periods and relative deadlines), and most well-established, proprietary RTOS kernels fall within this category. A characteristic trait of purpose-built RTOSs is low interrupt latency.

The RTOSs in this category can be further classified into four categories based on their main intended use case. The first category targets *deeply embedded* applications where resources are at a premium and mostly-static, single-purpose workloads are prevalent. A typical example for such a use case is an engine control unit (ECU) regulating a car engine's fuel injection, or the baseband chip in a smart phone implementing access to a wireless network. RTOS kernels in this category are also sometimes called "executives" due to their small size.

The second category of RTOSs targets applications where UNIX-like OS abstractions and multitasking are required (or desired for developer productivity), and the system design can afford the additional resources required to support them (*e.g.*, extra memory and faster processors). This category spans a wide range of embedded and non-embedded use cases such as factory automatization, consumer electronics, and telecommunication infrastructure. At the lower end (in terms of resource usage), many RTOSs can be configured into a "slimmed-down" version that is suitable for use in deeply embedded systems, *i.e.*, there is no clear cutoff point between the first two categories.

The third category targets safety-critical applications subject to mandatory certification requirements. In the industry literature, RTOSs in this category are also called *separation kernels*. From an OS-design point of view, RTOSs in the latter category can also be understood as hypervisors that isolate client RTOSs. Separation kernels are commonly used in avionics applications, where

government-mandated standards such as the DO-178B standard for avionics impose strict isolation of safety-critical subsystems from faults in non- or less-safety-critical subsystems.

Finally, academic RTOS projects that are primarily developed to explore new algorithmic concepts or to pioneer novel implementation techniques constitute a fourth, cross-cutting category.

2.5.1.1 Category I: Deeply-Embedded Real-Time Operating Systems

The primary concern in deeply embedded RTOSs, which are in some cases deployed in billions of devices, is to minimize resource usage, and in particular their memory footprint. Consequently, such RTOSs offer only a minimum of abstractions. Support for processes and address-space separation (*i.e.*, MMU support) are commonly absent (or deactivated), so that all tasks and the RTOS kernel itself run in a single address space with unrestricted access to all devices (*i.e.*, in kernel mode). The lack of access mediation avoids overheads and thus lowers resource requirements, but prevents effective fault containment. This category corresponds in spirit to the POSIX profiles PSE51 and PSE52.

Not all RTOSs in this category support scheduling, but those that do typically support multi-threading, FP scheduling, and mutex semaphores with optional priority inheritance (*i.e.*, the PIP discussed in Section 2.4.3). Further, since tasks execute in kernel mode, supporting the NCP is trivially possible by disabling interrupts. Multiprocessor support is rare in this category, but does exist, although often only in the form of message passing. That is, each processor is managed independently as a uniprocessor and communicates only by exchanging messages with remote processors. Scheduling is inherently partitioned in this design, and direct access to global resource is often not supported.

Proprietary RTOSs for deeply embedded systems with advertised multiprocessor support include Quadros Systems' *RTXC/mp* (message passing only), Express Logic's *ThreadX* (FP scheduling with only 32 distinct priorities), Mentor Graphics' *Nucleus RTOS* (advertised as requiring as little as 13 KB of memory, and only 50 KB when multiprocessor support is included), and ENEA's OSE (message passing only). Besides these proprietary, closed-source systems, there are also open-source RTOSs in this category: the TOPPERS consortium provides several multiprocessor-capable reference implementations of the ITRON standard (Monden, 1987; Takada and Sakamura, 1991,

1995), which is in widespread use among Asian car and consumer electronics manufacturers, and the OAR corporation distributes its *RTEMS* system under a liberal open-source license.

RTEMS. We consider RTEMS, the *Real-Time Executive for Multiprocessor Systems*, as the representative RTOS of this category.[25] We chose RTEMS because it is one of the oldest RTOSs to support multiprocessors that is still in use (RTEMS has been in development and use since 1988) and since it is particularly attractive for academic research due to its open-source nature. The RTEMS version discussed in the following is release 4.10, which is the latest stable release at the time of writing. Notably, RTEMS supports over 15 processor architectures, among them Intel's x86 architecture and several radiation-hardened processor designs.

RTEMS does not support MMUs; instead, the "kernel" and the user applications, which are just procedures that are executed in separate threads, are compiled into one binary image. In a multiprocessor environment, such an image is required for each processor. This implies that the assignment of tasks occurs already at design time or, at the latest, at compile time. Task migrations are not supported by the OS. However, task migrations could be emulated by the user by compiling a task's code into the images for multiple processors and then simultaneously "terminating" the task on one processor and starting it on another. However, since RTEMS does not inherently support task migrations, these are technically two (or more) tasks to the OS and the transfer of runtime state from one processor to another must be programmed manually by the developer. Global scheduling is not feasible in such an environment.

Uniprocessor support. All OS resources are statically allocated. This requires the number of tasks and locks to be statically defined at compile time. RTEMS supports P-FP scheduling with 256 distinct priorities and has explicit runtime support for scheduling (*e.g.*, tasks can declare a period and detect deadline misses), but currently does not support EDF. For local resources, RTEMS supports two queuing policies (jobs may wait in FIFO or in priority order) and two priority management policies: the well-known PIP and a second protocol that the RTEMS documentation calls the "priority ceiling protocol," which, however, is not the PCP as defined in Section 2.4.3. Rather, under RTEM's priority ceiling protocol, a job's priority is elevated to the priority ceiling immediately

[25] Curiously, the acronym "RTEMS" stood first for "Real-Time Executive for Missile Systems" and then "Real-Time Executive for Military Systems" in prior versions.

when it acquires the resource instead of relying on priority inheritance as in the PCP. This protocol, sometimes also referred to as the "basic ceiling protocol," is in fact equivalent to the SRP. By raising the job's priority to the priority ceiling, it is ensured that newly-released jobs of equal or lower priority are not scheduled until the resource is released. This has exactly the same effect as raising the system ceiling and considering a newly-released job to be ineligible to execute until its priority exceeds the system ceiling—the SRP's schedulability rule. RTEMS thus supports the SRP, even though not under that name. However, the priority ceiling for each lock must be configured manually, which allows for human error.

RTEMS includes compatibility layers for both the ITRON (Monden, 1987; Takada and Sakamura, 1991, 1995) and POSIX (IEEE, 1993, 2003, 2008b) interface definitions. While not fully compliant, RTEMS closely resembles the POSIX profile PSE51.

Multiprocessor support. As mentioned above, each processor requires an individual binary RTEMS image (*e.g.*, the unique processor index is hard-coded in the image). Further, at runtime, each processor maintains only its own local state. In contrast, in a *single-image multiprocessor OS* such as UNIX, Linux, or Windows, all processors are initialized from a single binary image, processor indices are assigned dynamically, and all processors share the (mostly) global runtime state. RTEMS thus resembles more a distributed system than a traditional shared-memory OS, which enables it to support both distributed- and shared-memory platforms. Processors communicate with each other by exchanging message packets as over a network link. This packet-based communication is medium-agnostic (*e.g.*, it could be implemented over an actual message bus or via shared memory) and must in fact be provided by the application developer—RTEMS itself does not include drivers to interface with processor interconnects.

RTEMS allows semaphores (and other OS objects) to be marked as being a global object. The identifiers of global objects are broadcasted to all processors so that remote tasks can issue requests. Due to the distributed nature of RTEMS, requests for remote resources follow the RPC model: a request packet is sent to the processor to which the resource is local and the requesting job suspends. On each processor, there is *one* local agent, called the *multiprocessing server*, that handles requests from *all* remote tasks. This local agent executes at the highest priority (*i.e.*, at priority zero). When a processor receives a packet, the local agent is resumed by the (developer-provided) processor-

interconnect driver and acquires the local semaphore on behalf of the remote task (subject to either the PIP or the SRP, depending on the configuration of the semaphore). Once it has locked the semaphore, the local agent sends a reply packet to the remote task and suspends again to wait for the next request. Superficially, this protocol resembles the DPCP, but it does in fact deviate in significant ways.

- The critical section itself is not executed by the local agent, which merely carries out lock and unlock requests.
- There is only one local agent and the priority of the currently requesting remote task is not reflected by the local agent's priority.
- If there are multiple requests arriving at the same processor, then there is no guarantee in which order they will be processed. This cannot be "fixed" by the developer-provided processor-interconnect driver since the lock-request packet does not relay the priority of the requesting task (the DPCP assumes priority order).
- The remote task is *not* priority boosted while it executes its critical section, nor is it subject to priority inheritance, nor is the (implicit) system ceiling modified.

The DPCP analysis does thus not apply and unbounded priority inversion is in fact possible due to the lack of priority boosting (recall Figure 2.37).

Since RTEMS effectively supports the SRP, and because tasks execute in kernel mode and thus can disable interrupts, the MSRP can be implemented in a straightforward way by the application developer. A close approximation of the DPCP could similarly be implemented on top of the provided primitives. However, neither protocol is part of the standard RTEMS distribution. In fact, to the best of our knowledge, neither the MPCP nor the DPCP (as defined and analyzed in the real-time literature) are supported by any of the RTOSs in this category.

2.5.1.2 Category II: UNIX-like Real-Time Operating Systems

The next category includes RTOSs that resemble a full-featured UNIX-like GPOS; *i.e.*, these RTOSs match the expectations of the POSIX profiles PSE53 and PSE54. Like a GPOS, RTOSs in this category support many higher-level features such as processes and threads, dynamic process creation,

files and hierarchical file systems, interprocess communication (IPC), networking with a full TCP/IP protocol stack, *etc.*, and virtually all RTOSs in this category implement large parts of the POSIX API (possibly in addition to other interfaces), or at least offer comparable services. Multiprocessor-capable RTOSs in this category typically fully function as a single-image OS and provide transparent task migrations and inter-processor communication via global state in shared memory (instead of packet-based communication links). Prominent examples include WindRiver's *VxWorks*, LynuxWorks' *LynxOS*, and Research in Motion's *QNX Neutrino* system. Due to their flexibility, these RTOSs are used in virtually every role that is not severely resource-constraint (Category I above) or that imposes particular certification requirements (Category III below).

QNX Neutrino. We briefly discuss QNX Neutrino as a representative example, as it has offered full multiprocessor support since 1997 and was one of the first RTOSs in this category to do so. QNX Neutrino is a microkernel-based RTOS that requires an MMU to function properly. Real-time tasks execute in user mode (and not kernel mode) and address space separation is fully enforced by the kernel. By default, tasks do not share any memory; containment of faults due to out-of-bounds memory accesses is thus ensured.

Uniprocessor support. As a POSIX-compliant OS, QNX Neutrino supports FP scheduling with 256 distinct priorities. If priorities are not unique, tasks of the same priority are queued either in FIFO order (SCHED_FIFO in POSIX parlance) or alternate in a round-robin fashion based on a configurable *time slice* parameter (SCHED_RR). As a third option, QNX Neutrino also supports SCHED_SPORADIC, a scheduling variant defined in the POSIX standard (IEEE, 2008b) that is intended to enforce the sporadic task model. However, Stanovich *et al.* (2010) recently pointed out some defects in SCHED_SPORADIC as defined by POSIX that can result in tasks exceeding the utilization of a corresponding sporadic task (as defined in Section 2.2). EDF scheduling is not supported by QNX Neutrino, and there is no explicit support for making a job's deadline known to the kernel.

QNX Neutrino supports a number of task synchronization abstractions (*e.g.*, pipes, signals, mailboxes) including mutex semaphores with two choices for priority management that are mandated by the POSIX standard. The default choice is the PIP. The second option, called PRIO_PROTECT by the POSIX standard, has exactly the same semantics as the "priority ceiling protocol" in RTEMS:

priority of a process is elevated to the ceiling priority when it acquires a semaphore, and restored when the mutex semaphore is unlocked. POSIX-compliant RTOSs such as QNX Neutrino thus support the PIP and the SRP. As it is the case with RTEMS, appropriate priority ceilings must be determined and configured manually by the programmer.

Multiprocessor support. The POSIX standard itself is silent on multiprocessor scheduling issues. In particular, it mandates the availability of FP scheduling in the form of SCHED_FIFO, but does not specify whether this should be implemented in a partitioned or global (or clustered) fashion on a multiprocessor. That is, to be POSIX-compliant, it is sufficient for an RTOS to offer either G-FP or P-FP scheduling. The pragmatic approach taken by QNX Neutrino (and many other RTOSs) has been to adopt *processor affinity masks*. A task's processor affinity mask is an unsigned word W consisting of m bits that indicates on which processor the task may execute. Specifically, a task may execute on processor x, where $x \in \{1, \ldots, m\}$, if and only if the $(x-1)$-th least-significant bit in W is set. For example, a task with affinity mask $W = 2 = 2^1$ may only execute on processor $x = 2$, whereas a task with affinity mask $W = 5 = 2^2 + 2^0$ may execute on processors $x = 3$ and $x = 1$.

FP scheduling with processor affinity masks works as follows. When a real-time process becomes runnable (*i.e.*, ready to execute), all processors on which it may execute (according to its affinity mask) are checked sequentially. If there is a processor x that is idle, then the process resumes execution immediately on processor x. Otherwise, if there exists a processor x on which a lower-priority process is scheduled, then the lower-priority process is preempted. If there are multiple processors to choose from, then the one with the lowest priority is preempted. Finally, if there are no lower-priority processes to preempt, then the newly-runnable process is enqueued in the ready queue until one of the processors becomes available. Note that processes will migrate as needed.

Processor affinity masks can be used to implement clustered scheduling, and hence also global and partitioned scheduling. Let C_i denote the cluster of processors that a task T_i is assigned to, where $C_i \subseteq \{1, \ldots, m\}$. T_i can be confined to its assigned cluster by assigning it the processor affinity mask $W_i = \sum_{x \in C_i} 2^{x-1}$. Depending on the configuration, QNX Neutrino (and the other RTOSs in this category) can thus support both G-FP and P-FP scheduling.

However, while default processor affinity masks allow a task to be scheduled on all m processors, the accompanying documentation emphasizes the benefits of restricting tasks to individual processors

in the interest of increased cache affinity and ease of development. Specifically, the documentation for QNX Neutrino advises that "[partitioned scheduling] eliminates the cache thrashing that can reduce performance [under global scheduling]" (QNX Software Systems, 2011).

No specific multiprocessor locking protocols are supported in QNX Neutrino besides the PIP and PRIO_PROTECT. The PIP is effective under global scheduling (*i.e.*, if no processor affinity mask is constrained) and Easwaran and Andersson (2009) recently presented matching analysis of the PIP under G-FP scheduling. However, as demonstrated in Figure 2.37, the PIP is insufficient across partitions (*i.e.*, if there exist processes with disjoint processor affinity masks that share resources).

Could the PRIO_PROTECT policy be used instead under partitioning? Since the priority ceiling of PRIO_PROTECT locks is configured manually, it could in fact be "tweaked" to implement priority boosting as required by the MPCP. This approach *almost* allows an implementation of the MPCP. Recall from Equation (2.10) on page 129 that, under the MPCP, the effective priority of a resource-holding job depends on the highest priority of any *remote* task sharing the resource. In contrast, for an appropriately chosen priority ceiling, the PRIO_PROTECT policy can be used to priority-boost resource-holding processes to an effective priority that corresponds to the highest priority of *any* task sharing the resource (which could be a local task). The published analysis of the MPCP thus does not directly apply to the PRIO_PROTECT policy, though it could likely be adjusted to take this difference into account. However, we are not aware of any such analysis in the published literature. Since the semantics of the PRIO_PROTECT policy are mandated by the POSIX standard, the other RTOSs in this category are similarly affected by this discrepancy.

None of these RTOSs implements the DPCP, though an implementation could be emulated in user space using POSIX message queues, local mutex semaphores, and appropriately chosen priorities and affinity masks, albeit at the expense of numerous system calls per lock request.

2.5.1.3 Category III: Separation Kernels

The third category of RTOSs are small, low-complexity kernels intended to host applications with stringent safety or security requirements. Such separation kernels are designed to isolate several software components at runtime in a certifiably safe and secure way. In particular, it must be ensured that a temporal or local fault in one component (*e.g.*, an infinite loop, out-of-bounds array access, exhaustion of assigned resources) does not affect the operation of other correct components. This

requirement is referred to as *logical* and *temporal isolation* in the real-time community, and as *space* and *time partitioning* in the RTOS industry.

Since certification efforts benefit from low complexity, separation kernels are typically simpler subsets of Category II kernels with similar but restricted capabilities. Depending on the nature of the "components," a separation kernel is simply either a regular kernel with strict budget enforcement (if components are regular processes) or a hypervisor (if components are virtualized OSs). In practice, most separation kernels support both processes and virtualization, so that this distinction is mainly a question of primary focus. An example of the former kind (regular kernel) is Green Hill's INTEGRITY kernel, which, according to marketing materials (Green Hills Software, 2011), is used in the Airbus A380, Boeing 777, and Boeing 787 aircraft (among many others); an example of the later type (hypervisor) is SYSGO's PikeOS, which is used in Airbus A350 and A400M airplanes (SYSGO AG, 2011). Similarly certified variants of LynxOS, VxWorks, and QNX Neutrino exist as well. Since this dissertation does not consider certification standards *per se*, we do not review this category of RTOSs in detail.

2.5.1.4 Category IV: Research Kernels

Academic research has resulted in many RTOSs that have been used to investigate, and at times pioneer, many of the use cases and types of RTOSs mentioned in Categories I–III. However, until very recently, most academic RTOSs focused on uniprocessors and distributed systems. One of the earliest research kernels to support multiprocessor systems, and arguably one of the most influential academic RTOSs, was the Spring operating system (Stankovic and Ramamritham, 1991). Besides its contributions with regard to kernel design for predictability and end-to-end guarantees in distributed real-time systems, the Spring kernel introduced *dedicated interrupt handling*, which we use and evaluate in Chapters 3 and 4. Under dedicated interrupt handling, one of the available processors is reserved as the *systems processor*; the remaining processors are called *application processors*. Real-time tasks are only scheduled on application processors, that is, the computing capacity of the systems processor is essentially lost to real-time tasks. The systems processor is solely responsible for system management tasks including interrupt handling (besides processor-local devices such as timers), all I/O operations, processing job releases, and (most) scheduling decisions. This approach, also called *interrupt shielding*, prevents real-time tasks from being disturbed by interrupts and

other system management overheads, and thus makes their execution more predictable. In contrast, if real-time tasks execute on processors where interrupts may occur, then the delays due to ISR execution must be accounted for similarly to priority inversions, which can result in severe pessimism (see Section 3.4.4). This results in a tradeoff: is it better to reserve a systems processor and lose $\frac{1}{m}$ of the system capacity, or is it preferable to use all m processors for real-time tasks at the expense of increased analysis pessimism? We explore this choice in detail in Chapter 4.

Besides the Spring system, another early multiprocessor-capable kernel used for both research and to support avionics applications is the *Hawk* kernel from Sandia National Laboratories (Holmes *et al.*, 1987). The Hawk kernel implements a UNIX-like interface and was primarily intended for "hard-real-time for missions of short duration" (Holmes and Harris, 1989). It was built specifically for a now-obsolete embedded multiprocessor and appears to not have been used in (published) research after 1990.

In more recent work, another approach to interrupt handling is implemented in the TU Dresden's *Fiasco* microkernel (Härtig *et al.*, 1998; Härtig and Roitzsch, 2006), which is a member of L4 family of the microkernels (Liedtke, 1995) that is designed to support real-time workloads. While originally targeted at uniprocessor systems, recent Fiasco versions support partitioned multiprocessor scheduling using either P-FP or *waited-fair-queuing* scheduling (Demers *et al.*, 1989) on each processor. Due to its design as a pure microkernel, interrupts are not processed in kernel space (besides the minimal required management). Instead, interrupts are converted to regular IPC messages that are sent to processes that implement the driver functionality for the device that triggered the interrupt. By assigning these drivers a lower priority than real-time tasks, this allows interrupt processing to be safely deferred. In the context of monolithic kernels, this technique is also known as *split interrupt handling*, where the management carried out by the kernel's ISR is called the *top half*, and the deferred part is called the *bottom half*.

In a pure microkernel-based OS (Liedtke, 1995), tasks do not synchronize by means of globally shared semaphores, but rather invoke RPC services offered by server processes that encapsulate shared state. Locking-related pi-blocking is thus not an issue; however, priority inversions can still arise while a high-priority job waits for a server response while the server processes earlier-issued requests of lower-priority tasks, or when the server process exhausts its processor allocation (*i.e.*, its current time slice) before finishing the request. Such priority inversions can be bounded using *time*

slice donation, which is an inheritance protocol that allows servers to execute using the budget of the waiting high-priority task (Steinberg *et al*., 2005, 2010).

Due to their minimalistic nature, microkernels are more appropriate for building predictable real-time systems than monolithic kernels, where it can be difficult to identify all potentially troublesome parts. In particular, since virtually all OS services are encapsulated in schedulable server processes, they can be assigned priorities such that they do not interfere with real-time tasks. Härtig *et al*. (1998) describe such a system design in detail. In more recent work, Ruocco (2008) provides a comprehensive overview of L4 microkernels and how they can be used as a foundation for real-time systems.

2.5.2 Real-Time Extensions of General-Purpose Operating Systems

In the 1970s and 1980s, there was a clear distinction between real-time and general-purpose computing: an OS was either a GPOS, with no expectations of real-time performance, or an OS was an RTOS, with only a small, specific-purpose feature set. This distinction started to blur in the early 1990s as some GPOSs were augmented with real-time features. For example, the *RT-Mach* kernel was derived from the *Mach* GPOS kernel by adding a real-time thread interface, real-time scheduling policies, and real-time locking protocols (Tokuda *et al*., 1990). RT-Mach was at the forefront of a trend that is still strong today: instead of building an RTOS from the ground up, it is now common to use an existing full-featured GPOS as a base that is then "retrofitted" to acquire desired real-time features (*e.g.*, a real-time scheduler, temporal isolation, or lowered interrupt latency).

This approach is desirable for two main reasons. First, it greatly reduces development costs since basic OS functionality such as device drivers, process abstractions, or memory management do not have to be reimplemented. This is a particularly compelling reason in an academic context, where it allows researchers to focus on actual research questions rather than expending much effort on mundane details. The second major reason is that real-time extensions of well-known GPOSs are more convenient for application developers. Instead of having to deal with "esoteric" RTOS interfaces, application developers can leverage their existing knowledge of the familiar GPOS APIs, which increases developer productivity, improves code quality, and reduces development times. When using a popular GPOS, development may further benefit from a vibrant market for third-party libraries and tools, which are less established for RTOSs. In short, building an RTOS on top of an

existing GPOS such as Linux or Windows reduces costs significantly, for both the developers and users of the RTOS.

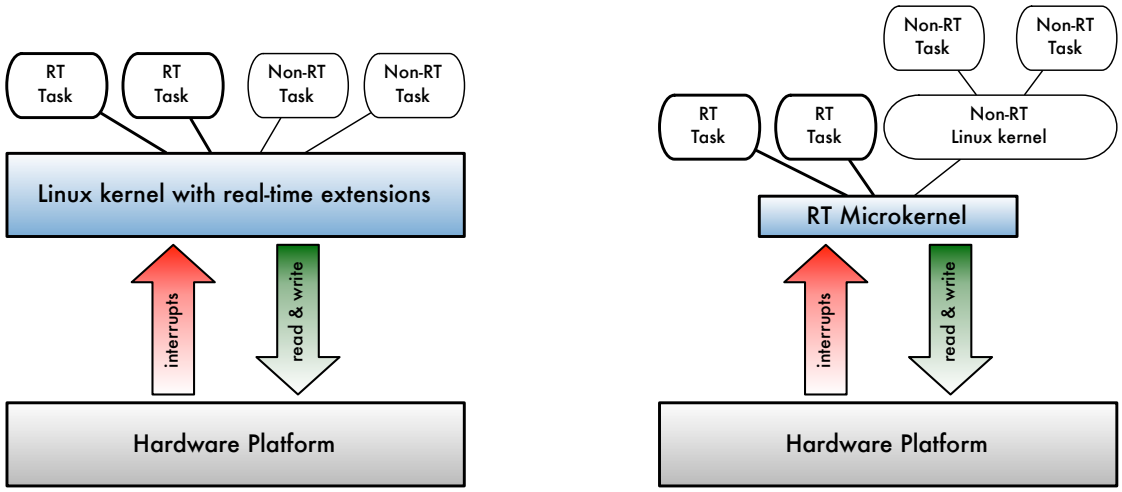
The biggest disadvantage of using a GPOS for real-time computing is that some fundamental design choices made in the interest of throughput can be detrimental in a real-time context, and it may be impossible to fix all of those decisions (short of reimplementing the whole OS). The design and implementation of real-time extensions of GPOSs thus frequently consist of compromises, which can limit the application domains in which they are a viable choice. For example, a large, monolithic, informally specified kernel such as Linux is likely never going to be amenable to formal verification and certification, and thus is not a suitable base for an RTOS targeted at truly safety-critical real-time applications. Similarly, there are limits to the minimal resource requirements of a full-featured GPOS kernel; some deeply embedded applications may thus always require specialized designs.

Nonetheless, from an economic point of view, RTOSs derived from well-known GPOSs are a compelling choice whenever the provided (real-time) capabilities are "good enough"; that is, when the tradeoffs inherent in such a design can be accommodated.

Real-time Linux. Given the rapid spread of Linux over the course of the last two decades and its open-source nature, it is hardly surprising that Linux is frequently chosen as the base GPOS (and may indeed be the most common choice). In fact, a casual web search reveals dozens of "real-time Linux" variants of both a commercial and academic nature.

Real-time Linux variants can generally be categorized into two groups, as illustrated in Figure 2.41. In a *native* design, the Linux kernel is the only kernel present and responsible for meeting real-time requirements, and real-time tasks are regular Linux processes. In contrast, in a *para-virtualized* design, a specialized (hard) real-time-capable microkernel (or hypervisor) is inserted between Linux and the actual hardware. Such implementations follow a classical microkernel design in which Linux takes over the role of an OS server and is scheduled as a background, non-real-time thread by the microkernel. Real-time tasks are specialized threads (i.e., not Linux processes) that are directly dispatched by the microkernel.

LITMUS$^{\mathrm{RT}}$ is a native design. We first review para-virtualized real-time Linux variants, and discuss prior and contemporary native real-time Linux variants in Section 2.5.2.2 below. Having established the necessary context, we then discuss LITMUS$^{\mathrm{RT}}$ in detail in Chapter 3.



(a) Native real-time Linux.

(b) Para-virtualized real-time Linux.

Figure 2.41: In a native real-time Linux design, real-time tasks are implemented as Linux processes. In a true para-virtualized real-time Linux design, real-time tasks are implemented as processes running on top of the purpose-built real-time microkernel and isolated from Linux. Some para-virtualized real-time Linux variants such as RTLinux (Barabanov, 1997; Yodaiken and Barabanov, 1997) execute the Linux kernel, the microkernel, and real-time threads in a single address space.

2.5.2.1 Para-Virtualized Real-Time Linux

Due to Linux's beginnings as a traditional monolithic kernel, all stable (*i.e.*, non-development) versions of Linux prior to Linux 2.6 executed every system call and interrupt as one long non-preemptive section. Requiring all kernel code paths to be non-preemptive greatly simplifies synchronization requirements on a uniprocessor. However, in the context of real-time systems, it leads to excessively long non-preemptive sections: when a high-priority real-time job is released (*i.e.*, an interrupt is triggered), the corresponding real-time process should be scheduled without delay, which may not be possible if the kernel executes non-preemptively on behalf of a lower-priority task. Consequently, a non-preemptive kernel (with long code paths) may cause real-time tasks to incur unacceptable pi-blocking in the worst case. Early real-time Linux variants, and in particular those focused on HRT applications, thus chose to work around the Linux kernel and its inherent limitations using para-virtualization, where Linux is executed as a background task of a real-time kernel or hypervisor. As result, the Linux kernel is not in full control of the hardware, does not disable interrupts, and thus can be preempted at any time.

There are two key advantages to such a *dual kernel* design. First, low interrupt latencies can be guaranteed to real-time tasks regardless of any deficiencies in the Linux kernel, some of which may be unknown. Second, only relatively small changes to the Linux kernel are required, which means that integrating improvements made in newer Linux versions is relatively easy. A good example of a dual kernel design is the *L*^{4}*Linux* system (Hohmuth, 1996; Härtig *et al*., 1998; Lackorzynski, 2004), in which Linux is para-virtualized on top of TU Dresden's Fiasco: initially released in 1996 for Linux 1.3.94 (Hohmuth, 1996), L^{4}Linux is still reliably tracking the latest Linux kernel versions in 2011 (Lackorzynski, 2011).

A disadvantage of para-virtualized real-time Linux variants is that real-time tasks execute directly on top of the microkernel and cannot make use of Linux services (such as device drivers, POSIX IPC, synchronization primitives, filesystems, etc.). This limitation is fundamental since para-virtualization does not improve Linux's real-time capabilities; rather, it enables real-time tasks to safely co-exist with the Linux kernel. One common way to enable communication between real-time threads and Linux processes in such systems is the use of non-blocking queues and buffers (Anderson and Ramamurthy, 1996; Anderson *et al*., 1995, 1997; Holman and Anderson, 2002b); for example, this is done in *RTLinux* (Barabanov, 1997; Yodaiken and Barabanov, 1997). Compared to ordinary Linux-based development, para-virtualized designs pose a greater engineering challenge (for application developers) due to the unusual and more complex system architecture and the restricted runtime environment for real-time tasks.

There are two main classes of para-virtualized real-time Linux variants. L^{4}Linux is an example of a "proper" para-virtualized system where both Linux and real-time tasks are executed in private address spaces and thus isolated from each other. Several commercial RTOSs now offer Linux (para-)virtualization support as well, among them Green River's INTEGRITY, Sysgo's PikeOS, and LynuxWorks' LynxOS. In contrast, real-time tasks execute in kernel mode in RTLinux and are thus not isolated from the Linux kernel. From a software engineering point of view, proper isolation is preferable; however, it creates some additional overheads and is more challenging to implement efficiently. Besides RTLinux, two other well-known real-time Linux variants based on the dual kernel approach that omit isolation are the *Real-Time Application Interface* (RTAI) developed at the Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano (Cloutier *et al*., 2000), which targets

industrial applications, and the *Xenomai* project, which targets similar use cases but also focuses on providing RTOS compatibility layers (so-called *skins*) to support legacy applications (Gerum, 2008).

Para-virtualization fundamentally takes the existing Linux kernel as a given; it is focused on engineering a solution that circumvents Linux's deficiencies rather than solving them. With regard to real-time scheduling and locking, real-time tasks are limited to the services offered by the underlying purpose-built microkernel or hypervisor (*i.e.*, mostly FP scheduling with PIP support, as discussed in the previous section). Para-virtualized Linux versions have therefore little in common with LITMUS$^{\mathrm{RT}}$.

2.5.2.2 Native Real-Time Linux

In a native real-time Linux design, the Linux kernel is the only kernel present and in full control of the hardware platform. Instead of working around Linux's limitations as in a para-virtualized design, a native real-time Linux variant must directly modify the kernel to enhance its real-time capabilities. While para-virtualization may be the only feasible (Linux-based) design for applications with very stringent timing constraints (*e.g.*, engine control software), a native design is generally preferable for the vast majority of applications if timing constraints can be met, *i.e.*, if Linux's limitations such as high interrupt latencies can be addressed.

Given the limitations of early versions of Linux, the first native designs had to introduce substantial infrastructure changes. In contrast, modern Linux (*i.e.*, Linux versions 2.6+) is much better suited for use in real-time systems, so that current native designs mostly focus on scheduling and locking algorithmic changes. In the following, we review the major academic native real-time Linux variants in (roughly) chronological order. These can be classified into two groups: those initially developed in the late 1990s and early 2000s (*i.e.*, before Linux 2.6), and those introduced in recent years.

Embedded HRT systems. Srinivasan *et al*. (1998) introduced *Kansas University Real-Time Linux* (KURT Linux), which pioneered many of the real-time infrastructure changes now found in standard Linux. In particular, it introduced high-resolution (software) timers based on hardware timers operating in one-shot mode (the so-called "UTIME" patch), a design that was later generalized and reimplemented in a POSIX-compliant way and merged into standard Linux under the name *hrtimers*

(Gleixner and Niehaus, 2006). Additionally, KURT Linux also added a number of scheduling features aimed at HRT systems such as table-driven static scheduling, the ability to upload static schedules into the kernel at runtime, and "focused mode," where the kernel would schedule only real-time processes (and no background processes). KURT Linux's high-resolution timers and scheduling support were used to conduct networking simulations requiring microsecond accuracy (House and Niehaus, 2000). The last public release of KURT Linux occurred in 2002 in the form of a patch against 2.4.18, which has since been removed from the web.

Linux started being used in robotics applications in the mid to late 1990s. An early real-time Linux version targeted at this domain is *Advanced Real-Time Linux* (ART Linux), which added the PIP and a system call interface in support of periodic tasks (Ishiwata and Matsui, 1998). ART Linux was developed by Japan's National Institute of Advanced Industrial Science and Technology (AIST) and used in the *Open Humanoid Robotics Platform* (Kanehiro *et al.*, 2004) and, among others, in the humanoid robot HRP-4C (Kaneko *et al.*, 2009). ART Linux is only sparsely documented and not much publicized (*e.g.*, it does not appear to have a current project homepage), and consequently has seen little use elsewhere.

Another early real-time Linux is *Real-Time and Embedded Linux* (RED Linux), which initially was also known as *Real-Time Enhanced* (RTE) Linux (Wang and Lin, 1998). RED Linux was developed at the University of California, Irvine to provide a flexible base for the study of uniprocessor real-time scheduling within a real OS (Wang and Lin, 1998, 1999). (LITMUS$^{\mathrm{RT}}$ serves the same purpose in a multiprocessor context.) With regard to infrastructure changes, RED Linux incorporated Kansas University's UTIME patch and further changed the kernel to reduce the length of non-preemptive sections by adding *preemption points*. That is, the kernel still executes system calls and ISRs non-preemptively most of the time in RED Linux, but checks more frequently whether a preemption is required. RED Linux also introduced significant algorithmic changes. In particular, it added a flexible hierarchical scheduling framework, which included support for EDF and several notions of fair scheduling (Wang *et al.*, 2002; Lin and Wang, 2003). It is not clear whether the code for RED Linux was made publicly available; it appears that the RED Linux project is now dormant and not (anymore) available online.

Resource kernels. When extending a UNIX-like OS, the question arises of how to map sporadic tasks, which are analysis-time entities, to processes, which are runtime entities. In a "classic" RTOS such as QNX Neutrino or even standard Linux, the kernel (and thus the scheduler) is unaware of the task parameters that were used during analysis to establish schedulability of the system. As a result, such a kernel cannot reliably detect when processes deviate from the assumed behavior. This is undesirable from a reliability point of view, and in particular in open systems (*i.e.*, if tasks are added and removed at runtime). The RT-Mach project incorporated the idea of *resource reservations* (Mercer and Rajkumar, 1995), which was later developed into the *resource kernel* design (Rajkumar *et al*., 1998; Oikawa and Rajkumar, 1999; Rajkumar *et al*., 2001). The resource kernel approach was implemented by Oikawa and Rajkumar (1998) in Linux under the name *Linux/RK*.

In a resource kernel, the sporadic task model is interpreted at runtime as an execution time budget that is replenished periodically or sporadically. That is, sporadic "tasks" are instantiated by the kernel as an accounting abstraction, and each "job" is a time slice with a deadline. By performing an admission test (*i.e.*, schedulability test) before granting a reservation, the kernel can ensure at runtime that resources do not become overcommitted. Once a reservation has been granted, a process is attached to it. Instead of scheduling processes directly, the scheduler in a resource kernel first selects a pending "job" (*i.e.*, a budget that has not yet been exhausted) to "execute" and then dispatches the attached process.

Resource reservations have a number of advantages. Since a process is only scheduled when its reservation has remaining budget, it cannot consume more resources than assumed during schedulability analysis. This provides temporal isolation among processes, which makes the system amenable to *a priori* analysis. Further, development is simplified because misbehaving processes (*i.e.*, those that routinely overrun their budget) are straightforward to identify. Another advantage of the resource reservation model is that it can be easily extended to support group scheduling (by attaching more than one process to a processor reservation) and hierarchical scheduling (by having multiple layers of reservations). The idea of resource reservations can further be applied to non-compute resources such as disk and network bandwidth or memory to isolate tasks from interference via those resources as well.

Resource reservations as described so far are called *hard reservations* in Linux/RK because processes are not eligible to execute after exhausting their current budget. Additionally, Linux/RK

supports two relaxed reservation types: processes with *firm reservations* may be scheduled even if they have currently no budget, but only if the processor would idle otherwise, whereas processes with *soft reservations* are scheduled on a round-robin basis after exhausting their budget. The resource reservation model is a natural way to realize the sporadic task model in a UNIX-like OS, and we follow this approach in the implementation of LITMUS$^{\mathrm{RT}}$ as well.

Besides adding resource reservations to Linux's FP scheduler, Linux/RK also added high-resolution timers and shortened the duration of non-preemptive sections in the Linux kernel. The last public release of Linux/RK with support for processor, disk, and network reservations occurred in 2002 on the Linux/RK project homepage in form of a patch against Linux 2.4.18 (CMU Real-Time and Multimedia Systems Lab, 2006). Although a limited "alpha version" based on Linux 2.6.18 with support for processor reservations (but not disk or network reservations) appeared in early 2007 (Lakshmanan, 2007), Linux/RK does not appear to be actively maintained any longer.

QoS scheduling. Two somewhat different native real-time Linux versions are *QLinux* (Sundaram *et al*., 2000) and *Linux-SRT* (Childs and Ingram, 2001). Instead of targeting HRT workloads common in embedded systems, they were aimed at supporting "soft" workloads with *quality-of-service* (QoS) requirements. Like Linux/RK, both variants were based on resource reservations at their core. QLinux emphasizes resource reservations for network and disk bandwidth (in addition to processor time) and implements the *start-time fair queuing policy* (Goyal *et al*., 1996). Linux-SRT supports several types of processor time and disk bandwidth reservations with semantics similar to Linux/RK's hard, firm, and soft reservations. Notably, Linux-SRT provides an additional system call to let servers such as X11 (Linux's graphical user interface) explicitly "bill" clients for resources used on their behalf.

Modern Linux. Common to all of the so-far mentioned native real-time Linux variants is that they are no longer actively maintained,[26] and that they do not explicitly consider multiprocessor issues (besides the multiprocessor support offered by the underlying Linux version). The latter is understandable because multiprocessor platforms for real-time and embedded systems were a rare occurrence at the time. One reason that has likely contributed to the cessation of project

[26]With the possible exception of ART Linux. Given the unavailability of (English) documentation, recent publications, or an up-to-date project homepage, we were unable to determine its current status.

maintenance is that Linux 2.6 gained several improvements (over the course of several versions) that greatly improved its viability as an RTOS, namely high-resolution timers, priority inheritance, mostly preemptable kernel execution, much-shortened non-preemptive sections, and an improved lower-overhead FP scheduler. Since these improvements have become available in standard Linux, the need for alternate infrastructure-modifying patches has been lessened. In fact, mainline Linux is now (virtually) POSIX-compliant and supports FP scheduling (SCHED_FIFO and SCHED_RR) with 100 distinct priorities, processor affinity masks, and the PIP. While not directly supported in the kernel, the PRIO_PROTECT policy is available as a user-space implementation in the *pthreads* library. The SRP is thus available, albeit at the cost of two additional system calls per resource request to raise and lower task priorities.

However, while the Linux kernel and its associated runtime libraries are now compliant with the real-time POSIX specification (similar to a purpose-built Category II kernel), the Linux kernel itself still contains non-preemptive code paths that are long (in the context of real-time systems) and architectural design choices that were made with throughput in mind. For example, interrupts are, by default, not serviced using split interrupt handling; rather, ISRs are typically executed immediately when an interrupt is raised and are not subject to scheduling. Executing ISRs right away benefits network and disk bandwidth, but can also delay real-time tasks. Thus, while API-compatible, current mainline Linux is (understandably) not yet comparable to purpose-built RTOSs such as VxWorks or QNX in terms of predictability and interrupt latency.

PREEMPT_RT patch. Moving Linux further in the direction of being a true RTOS is the goal of the PREEMPT_RT patch,[27] which is the only "semi-official" real-time Linux variant being maintained by several core Linux kernel developers (Molnar, 2004; McKenney, 2005; Gleixner, 2006; Zijlstra, 2008). The PREEMPT_RT patch changes the core Linux infrastructure significantly. To reduce the length of non-preemptive sections, it converts most spinlocks in the kernel to semaphores, and further enables the PIP by default for all semaphores in the kernel (in standard Linux, the PIP is not necessarily active, which can result in priority inversion). Another major change introduced by the PREEMPT_RT patch is to force split interrupt handling for all ISRs (except timers). However, in contrast to a microkernel such as Fiasco, ISR bottom halves are still executed in kernel mode without

[27] The name derives from an option in the configuration of the build system of the Linux kernel.

any isolation (and the ISR top halves are not necessarily as minimal as in Fiasco). Given that error rates in device drivers are notoriously high (Chou *et al.*, 2001), this lack of fault containment poses a considerable reliability hazard.

The PREEMPT_RT patch, originally proposed by Molnar (2004), has been continuously maintained since 2004 and is still under active development, with the latest released version applying to Linux 2.6.33 at the time of writing. Besides serving as a staging ground for real-time features that are (intended to be) incorporated into mainline Linux at a later point, it is also widely used in practice. For example, both Novell and Red Hat corporations have developed commercial Linux distributions that incorporate the PREEMPT_RT patch.

The PREEMPT_RT patch does not add new scheduling algorithms or locking protocols; rather, its main appeal is a considerable reduction in interrupt latency (Arthur *et al.*, 2007). However, strictly speaking, even with the PREEMPT_RT patch, Linux is not a "true" RTOS, and will for the foreseeable future contain a number of compromises between throughput and predictability concerns. Nonetheless, despite these compromises, or maybe in part even because of them, Linux today is "good enough" for many (soft) real-time applications. As noted by Paul McKenney in (Harris, 2005),

> *I believe that Linux is ready to handle applications requiring sub-millisecond process-scheduling and interrupt latencies with 99.99+ percent probabilities of success. No, that does not cover every imaginable real-time application, but it does cover a very large and important subset.*

Many practitioners, it appears, are in agreement, given that industry surveys indicate a steadily growing Linux market share in the embedded sector (Linux Devices, 2007).

Recent developments. One area beyond the scope of the PREEMPT_RT project are (experimental) scheduling algorithm improvements such as resource reservations like those offered in Linux/RK, QLinux, or Linux-SRT. Around the time that the LITMUS$^{\mathrm{RT}}$ effort was started in 2006, two other Linux-based real-time projects were initiated, namely the *Adaptive Quality of Service Architecture* (AQuoSA) developed at the Scuola Superiore Sant'Anna (Palopoli *et al.*, 2009) and *Redline Linux* developed at the University of Massachusetts Amherst (Yang *et al.*, 2008). Both are native real-time Linux variants that are focused mainly on "soft" workloads, albeit in very different ways.

The Redline Linux project's main goal is to ensure a highly responsive graphical user interface even in situations where the system is exposed to severe, potentially malicious overload (such as

"fork bombs," where the number of processes increases exponentially until memory is exhausted). To achieve this, Redline Linux features a modified variant of Linux's now-standard *completely fair scheduler* (CFS),[28] a timesharing policy inspired by proportional-share fair scheduling (Tijdeman, 1980; Stoica *et al*., 1996). However, CFS is not well suited for providing *a priori* guarantees, *i.e.*, actual scheduling is not predictable. Redline Linux thus has little in common with LITMUS$^{\mathrm{RT}}$. The Redline Linux patch has not been updated since Linux version 2.6.22.5, which was released in 2007.

In contrast, AQuoSA is under active development (Cucinotta, 2011). Similar to Linux/RK, QLinux, and Linux-SRT, AQuoSA's chief research goal is to provide soft QoS guarantees to time-sensitive applications (*e.g.*, such as video conferencing) based on reservations. In addition, AQuoSA incorporates adaptive scheduling techniques to adjust reservation parameters dynamically (Cucinotta *et al*., 2004). AQuoSA further implements a *bandwidth inheritance protocol*, which applies the idea of priority inheritance to reservation-based scheduling (Lamastra *et al*., 2001). While AQuoSA was a uniprocessor effort during the first couple of years of the project, it recently gained multiprocessor support (Checconi *et al*., 2009).

The uniprocessor reservation scheduler in AQuoSA is based on EDF. In later work, Faggioli *et al*. (2009a,b) developed SCHED_DEADLINE, a standalone, multiprocessor-capable EDF implementation with the stated purpose of inclusion in mainline Linux, which is a major engineering undertaking. (In contrast, LITMUS$^{\mathrm{RT}}$ is not designed to be merged into Linux.) Together, with Linux's affinity mask support, this patch adds G-EDF, C-EDF, and P-EDF to Linux and thus would be a significant extension of Linux's algorithmic real-time capabilities. While key kernel developers have voiced their support (Corbet, 2010), the SCHED_DEADLINE patch has not been accepted into mainline Linux to date. From a predictability point of view, there are some concerns with Linux's implementation of affinity masks in the context of global and clustered scheduling; these also affect SCHED_DEADLINE and are discussed in more detail in Section 3.2.4.

Finally, in very recent work, Kato *et al*. (2010) presented the *Advanced Interactive Real-Time Scheduler* (AIRS) for Linux. AIRS is based on SCHED_DEADLINE but adds a *semi-partitioned* EDF variant called EDF-WM. In a semi-partitioned algorithm (Anderson *et al*., 2005), most tasks are statically assigned to a processor (as under partitioning) and only a few tasks migrate (as under

[28] In contrast to most other published proportional-share fair schedulers, upper and lower bounds on lag under CFS are unknown; the name "completely fair" is thus not necessarily descriptive of its properties.

global scheduling). Like AQuoSA, QLinux, Linux-SRT, and Linux/RK, AIRS is based on resource reservations and aims to enable high QoS levels for "soft" interactive workloads such as video playback on desktop systems (Kato *et al.*, 2010). Besides processor scheduling, recent versions of AIRS also support memory reservations (Kato *et al.*, 2011). On a related note, Bastoni (2011) recently presented an implementation and evaluation of EDF-WM and several other semi-partitioned schedulers in LITMUS$^{\mathrm{RT}}$ (Bastoni *et al.*, 2011; Bastoni, 2011).

Summary. This concludes our review of prior work and illustrates why we developed LITMUS$^{\mathrm{RT}}$. Global JLFP or JLDP schedulers such as G-EDF or PD^2 are not supported by any of the discussed RTOSs and Linux variants (with the exception of SCHED_DEADLINE, which has only recently become available). Further, global scheduling is only supported by means of affinity masks (if at all), which are ill-suited to implementing policies with frequent migrations such as PD^2 (as discussed in Section 3.3.4). None of the major RTOSs implement multiprocessor locking protocols for partitioned scheduling for which pi-blocking analysis has been derived (in the published literature). Additionally, in 2006, when the development of LITMUS$^{\mathrm{RT}}$ commenced, most prior academic real-time Linux extensions were no longer being maintained. To study multiprocessor scheduling and locking in a real OS, development of a new, extensible testbed was thus required.

In the next chapter, we discuss our native real-time Linux extension LITMUS$^{\mathrm{RT}}$, how it implements the sporadic task model and schedulers described in this chapter, and how to incorporate runtime overheads that arise in LITMUS$^{\mathrm{RT}}$ into the idealized sporadic task model and associated schedulability analysis.

CHAPTER 3

THEORY, PRACTICE, AND OVERHEADS*

In this chapter, we present our work aimed at reconciling practice and theory in multiprocessor real-time systems. In Section 3.1, we revisit the assumptions underlying the real-time theory reviewed in Chapter 2, discuss why some of them are problematic in practice, and argue in favor of a realistic compromise between pure analysis and practical limitations. In Section 3.2, we review the Linux foundation underlying LITMUS$^{\mathrm{RT}}$. Thereafter, in Section 3.3, we present LITMUS$^{\mathrm{RT}}$, which was designed to closely correspond to the sporadic task model, and discuss solutions to implementation problems that we faced when building LITMUS$^{\mathrm{RT}}$. In Sections 3.4 and 3.5 we detail how to account for runtime overheads during schedulability analysis and finally present overhead-aware versions of the schedulability tests reviewed in Chapter 2.

3.1 A Practical Interpretation of the Sporadic Task Model

The sporadic task model is a highly idealized view of a real-time system. When implemented in a real OS on commodity hardware, several practicality issues must be addressed. We briefly discuss a number of issues that we faced in the LITMUS$^{\mathrm{RT}}$ project, and the solutions that we adopted.

---

* Contents of this chapter previously appeared in preliminary form in the following papers:
Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57;
Brandenburg, B., Block, A., Calandrino, J., Devi, U., Leontyev, H., and Anderson, J. (2007). LITMUS$^{\mathrm{RT}}$: A status report. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123;
Brandenburg, B. and Anderson, J. (2008b). An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 185–194;
Brandenburg, B. and Anderson, J. (2009a). On the implementation of global real-time schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 214–224; and
Brandenburg, B., Leontyev, H., and Anderson, J. (2011). An overview of interrupt accounting techniques for multiprocessor real-time systems. *Journal of Systems Architecture: Embedded Software Design*, 57(6):638–654.

Process vs. task. There are a number of ways in which the sporadic task model can be implemented in a process-based OS.[1] One approach (that we did not adopt) is to implement all real-time tasks inside a single process that provides a special runtime environment. The OS kernel is then not concerned with scheduling and locking issues since sporadic tasks are not OS entities. This has the advantage that no kernel modifications are required. However, this approach has the limitations that there is no address space separation between tasks, and that tasks must be trusted not to call library routines and system calls that interfere with the runtime environment (such as the *exit* system call). Such a user-mode runtime environment is thus most appropriate for systems that implement safe languages. For example, real-time Java and some ADA runtime environments follow this approach; Burns and Wellings (2009) provide a comprehensive introduction to this approach of real-time programming. Since tasks are generated by a runtime-environment-aware compiler in these languages, it can be ensured that each task is fully cooperative.

If tasks are to be implemented in an unchecked language such as C, then a pure user-mode approach is problematic for reliability reasons. Further, in the context of our research goal—to assess the viability of multiprocessor real-time schedulers on real hardware—a user mode implementation has the disadvantage that it is not directly in control of the hardware, which may make it difficult to correctly attribute all observed delays. We therefore chose to follow the resource reservation model in LITMUS$^{\mathrm{RT}}$ in which each task is implemented by a separate process (LITMUS$^{\mathrm{RT}}$ also supports real-time threads, *i.e.*, address space separation is not mandatory). In this interpretation of the sporadic task model, the task parameters become a contract between the kernel and each application: once a reservation is granted (*i.e.*, when a task T_i is admitted by the kernel), it is the kernel's responsibility to ensure that each job J_i will be allowed to execute for up to e_i time units prior to its deadline (in the case of HRT reservations) or by its guaranteed tardiness bound (in the case of SRT reservations).

Measurement vs. analysis. When using resource reservations, tasks cannot use the processor for more time units than allowed by their reservation and are thus isolated from each other. An application's temporal correctness then hinges on whether each of its jobs stays within the allocated

[1] Confusingly, processes are commonly called tasks in the context of Linux. We reserve "task" to denote sporadic tasks as defined by the sporadic task model, and refer to the OS abstraction as process.

budget. Determining a safe upper bound on a task's *worst-case execution time* (WCET) is thus essential to building a correct real-time system.

Ideally, the execution requirement parameter e_i should be determined analytically for each task. In theory, this can be achieved by applying control- and data-flow analysis to the task itself and all invoked kernel services and by building a model of the processor's execution pipeline, all possible cache states (for each cache level, and for both instruction and data caches, and the TLB), the memory system itself, and bus interference—for all possible inputs and schedules. Alas, given the complexity of modern commodity architectures, this is an extraordinarily challenging problem in practice.

While WCET analysis for uniprocessors has progressed significantly in recent years (there are now commercial WCET analysis tools available for certain embedded processors[2]), the state-of-the-art in WCET analysis has not yet advanced to the point that current multicore processors and complex architectures such as Intel's x86 architecture are analytically tractable (Wilhelm *et al.*, 2008). This is particularly true for assessing the impact of preemptions, since predicting cache contents and hit rates is notoriously difficult. Even though there has been some initial success in bounding cache-related preemption delays caused by simple, idealized data (Ramaprasad and Mueller, 2010) and instruction caches (Staschulat and Ernst, 2007), analytically determining preemption costs, even on uniprocessors with essentially private caches, is still generally considered to be an open problem (Wilhelm *et al.*, 2008).

On multicore platforms with a complex hierarchy of shared caches, there is currently no choice but to resort to empirical approximation—execution time requirements must be measured during development. This has wide-ranging implications. When using a measurement-based approach, *worst-case* execution costs cannot be determined with certainty, since any amount of finite testing may not reveal the true worst case. Thus, even when a task set is deemed schedulable based on empirically estimated parameters, it may in fact not *always* be correct—if the worst case was underestimated, a job might miss its deadline when its reservation is exhausted.

If schedulability cannot be asserted with certainty, then one may ask: why analyze at all? Why not simply observe the system for some time (say, one hour), and deem it "correct" if no failure is

[2]One example is AbsInt's *aiT* tool suite (AbsInt Angewandte Informatik GmbH, 2011).

observed? Regrettably, this "let's see if it works" approach to correctness is certainly not unheard of in practice. We argue, however, that analysis is beneficial even if some parameters are estimated.

Suppose a system is subject to HRT constraints. When the system is only observed for one hour, then one can only assert that "no failure was observed during the first hour of operation." However, in general, this does *not* preclude a deadline miss at a later time *even if later execution times are less than observed during the first hour*. In contrast, when conducting proper schedulability analysis based on estimated execution costs, then a much stronger guarantee results: "no deadline will ever be missed as long as the execution requirement of each task does not exceed the observed maximum." The latter is a strong, if conditional, guarantee, whereas the former is merely a more or less confidence-inspiring observation. We argue that a conditional guarantee is better than none at all.

In fact, there is no unconditional HRT correctness in the real world, since schedulability analysis (implicitly) assumes the absence of complete hardware failure. As Paul McKenney humorously observed "if you show me a hard real-time system, I will show you the hammer that will cause it to miss its deadlines" (McKenney, 2007). We therefore believe that conducting schedulability analysis based on carefully estimated parameters has merit—and is useful in practice—as long as assumed maxima are determined such that they are unlikely to be exceeded. As it is well-established practice in other engineering disciplines, safety margins can be integrated into estimates to provide a buffer against unlikely events.

Further, in the case of SRT constraints, it may be difficult to detect slowly increasing, unbounded tardiness if the test interval is short (in relation to the tardiness growth rate). Analytical derivation of a tardiness bound based on estimated parameters is thus useful even in the SRT case.

Extent of measurement. One desirable but currently infeasible extreme is pure analysis (and no measurement), whereas the other common but undesirable extreme is mere observation (and no analysis). When conducting experiments, which tradeoff between measurement and analysis should be chosen? For example, is it sufficient to measure only the execution times of jobs and assume that such measurements reflect kernel overheads, or should kernel overheads be measured individually and accounted for analytically?

The fundamental purpose of an RTOS is to provide temporal guarantees. These guarantees should be as strong as possible. In the context of analytically sound real-time systems, this means that

measurements are a stopgap measure only that is to be avoided when possible. We therefore propose the *principle of least measurement*: everything that can be analytically derived with reasonable effort should not be measured.

To illustrate the principle of least measurement, consider measuring the aggregate impact of overheads on a job. If there is simply a single "system overhead" measurement, then different kernel overheads are averaged and their individual worst-case impacts are masked. That is, besides likely not observing the true worst-case for each source of overhead, such coarse-grained sampling additionally risks underestimating the combined effect of overheads since the maximum for each overhead source is likely not observed simultaneously. For example, suppose a context switch requires $2\mu s$ on average, but the measured maximum delay is $15\mu s$. Further, suppose a scheduling decision requires $10\mu s$ on average, but the measured maximum scheduling cost is $50\mu s$. If these two sources of overhead are sampled as one entity, then there is good chance that the observed joint maximum of the two is much less than $65\mu s$, the sum of the two individual maxima. Instead, each source of kernel overhead should be accounted for analytically (as we do in Sections 3.4 and 3.5) and measured individually (until WCET analysis becomes available).

As WCET analysis improves, the need for measurement-based estimation will subside. Ideally, in the (distant) future, it would be desirable for RTOSs to be distributed with built-in WCET analysis support that determines upper bounds on kernel overheads for a given kernel configuration, just like Linux's build system today contains built-in support for static source code analysis that automatically detects certain classes of logical program errors. This is not yet feasible, but given recent advances in bounding migration delays (Hardy and Puaut, 2009) and analyzing interference due to shared caches (Chandra *et al.*, 2005; Yan and Zhang, 2008; Lv *et al.*, 2010), we expect practical multicore WCET analysis (for simple architectures) to be developed eventually.

In the meantime, the principle of least measurement implies two RTOS design guidelines. First, kernel functionality should be structured and instrumented such that delays due to individual code paths (*e.g.*, making a scheduling decision or a context switch) can be attributed correctly and measured individually rather than treated as aggregate "kernel overhead." Second, the kernel should only use algorithms and data structures that are amenable to WCET analysis in principle, even if not (yet) on current hardware platforms. That is, the kernel should not make use of randomized or probabilistic data structures, or algorithms that rely on amortized runtime analysis. Not every

application will require full WCET analysis in practice, but an RTOS should be built in way that it can be analyzed when required.

Large, monolithic GPOS kernels such as Linux may never be amenable to WCET analysis, and, as a research project, LITMUS$^{\mathrm{RT}}$ will not be deployed in safety-critical environments. Nonetheless, we believe that forward-looking real-time research should be rooted in a conceptually sound basis. We therefore adopt the principle of least measurement in the design of LITMUS$^{\mathrm{RT}}$ and as the guiding rule for our overhead-aware evaluation methodology presented in Chapter 4.

Non-conforming tasks. Another question that an implementation of the sporadic task model must address is how to deal with non-conforming tasks. Besides overrunning their budget, jobs of tasks that violate their parameters can be released too early (*i.e.*, less than p_i time units apart) or self-suspend when not expected to do so. For example, a job release may be triggered prematurely by an interrupt due to a faulty sensor, or may suspend briefly due to a page fault. Regarding the latter, while the virtual memory of real-time tasks should be locked into physical memory, the POSIX semantics require pages to be locked into memory only after they have been accessed once. This requires developers to program real-time tasks to pre-fault all required pages; accidental omission is thus possible. Similarly, resolution of symbols in dynamic libraries can cause brief suspensions in Linux.

Strictly speaking, temporal correctness cannot be guaranteed in such cases since the assumptions underlying the schedulability analysis are not met. When interpreting the sporadic task model as a contract, the RTOS is then under no obligation to support tasks that exhibit such behavior. For example, the initial prototype of LITMUS$^{\mathrm{RT}}$ simply did not implement suspension support for real-time tasks, to the effect that a suspending real-time task would cause a kernel panic. This, clearly, is less than ideal behavior during the development phase, and of questionable utility when deployed.

Instead, the RTOS should revert to best-effort behavior in such cases. That is, while temporal correctness cannot be guaranteed any longer in such cases, small infractions likely have only little impact and should not crash the RTOS. Similarly, tasks should not be expected to pass valid parameters to the kernel, nor should they be trusted to execute system calls in the right order (or at all, or only once). While this may seem obvious to OS developers and researchers, it is worth pointing out since not all academic real-time GPOS extensions always live up to this standard.

Non-atomicity of scheduling events. Since the sporadic task model and associated schedulability analysis assumes overhead-free execution, scheduling algorithms are commonly described from the point of view of an "outside observer" that knows the exact state of all processors and assuming sequential, atomic scheduling events. For example, if two jobs are released at the same time, then it is assumed that the scheduler considers both simultaneously when making preemption decisions.

On a multiprocessor, actual scheduling is much more difficult because scheduling events can occur in parallel on different processors (*e.g.*, two jobs may be released at the same time, but by interrupts handled by two different processors) and because scheduling events take time. Further, a processor servicing an interrupt cannot directly initiate a context switch on a remote processor; it can only send an IPI to notify the remote processor that a preemption is required.

Preemption decisions in a global scheduler are particularly challenging. In an ideal, overhead-free model, there exists exactly one scheduled job on each processor (unless a processor is idle). In practice, the notion of when a job is "scheduled" is not clear-cut because processes can be in transition states such as "still scheduled but currently suspending" or "selected by the scheduler, but not yet dispatched and awaiting a context switch." For example, suppose that a job J_i is one of the m highest-priority jobs when released, but a higher-priority job J_k arrives before the context switch to J_i's implementing process is complete. Now further suppose that a third job J_l with priority less than J_k's but higher than J_i's arrives concurrently on another processor. Which job is "scheduled?" Is a preemption required? Should an IPI be sent?

This non-atomicity of scheduling events complicates scheduling decisions considerably because it allows for concurrent scheduling decisions that must take possibly inconsistent system states into account. In the worst case, a scheduling race could result in a priority inversion if a high-priority job is "overlooked." In our experience, relying on the OS's notion of whether a particular *process* is scheduled (*i.e.*, is its stack in use?) for assigning *jobs* to processors is both difficult and error-prone due to unanticipated corner cases and race conditions. This is especially true when jobs (accidentally) self-suspend for short durations (due to blocking on semaphores or page faults, library loading, *etc.*). Instead, in LITMUS$^{\mathrm{RT}}$, we split *job scheduling*, which is at the level of the sporadic task model and is only concerned with assigning abstract jobs to abstract processors, from *process scheduling*, which is concerned with address spaces, stacks, register sets, and other hardware peculiarities.

This simplifies the real-time scheduling logic since any processor's job assignment can be updated on any processor by any event handler (in contrast to performing context switches, which can only be done in one specific code path on the target processor). Process scheduling is then reduced to tracking which process *should* be executing based on the current job-to-processor mapping; any delay in tracking can be accounted for as part of overhead analysis.

In fact, a majority of the time spent developing LITMUS$^{\mathrm{RT}}$ has been devoted to concurrency issues that arise under global scheduling. These issues and our solutions are discussed in detail in Section 3.3 below. Next, we summarize the Linux foundations that underly LITMUS$^{\mathrm{RT}}$. Interestingly, Linux's implementation of global and clustered FP scheduling is also affected by the non-atomicity of scheduling decisions.

3.2 The Linux Scheduling Framework

As a whole, the Linux scheduler is a subsystem of considerable complexity. In Linux 2.6.36, the processor scheduling code spans over ten files approaching 19,000 lines of code and comments in total. Bovet and Cesati (2005) provide a thorough description of process management and scheduling in Linux; a more recent treatment including a discussion of CFS is given by Maurer (2008). Relevant to our work are how the scheduler is invoked, how it is structured into "scheduling classes," how concurrent state updates are synchronized, and how the FP policy and support for processor affinity masks are implemented in Linux.

3.2.1 Invocation of the Scheduler

The Linux scheduler can be invoked in two ways. A process invokes the scheduler *directly* with a call to the schedule() procedure when it suspends (*e.g.*, after invoking the read() system call to perform blocking I/O). From the point of view of the calling process, the call to schedule() returns when the process is resumed.

The scheduler is invoked *indirectly* when the preemption of the currently scheduled process is required. This is achieved by setting the *rescheduling flag* in the *process control block* (PCB) of the

currently scheduled process, which is called struct task_struct in Linux.[3] Prior to returning from an ISR, an exception handler (such as a page fault), or from a system call, the kernel checks to see if the rescheduling flag of the process currently scheduled on the local processor is set, and invokes the scheduler if required.

For example, if an ISR causes a higher-priority process to resume, then the scheduler is not invoked right away. Instead, the ISR sets the rescheduling flag of the currently scheduled process, which causes the required preemption to be enacted at the end of the ISR. This has two advantages: first, the involuntary preemption code path is simplified since it is always invoked from the same context (*i.e.*, at the end of an ISR, exception handler, or system call), and second, rescheduling can be requested even while holding kernel locks that must be released prior to calling the scheduler.

Associated with each process is a *preempt count*. A process is considered to be non-preemptable whenever it has a positive preempt count. The preempt count can only be positive while a process executes in kernel mode; in standard Linux, user space cannot request a process to be non-preemptable. If the rescheduling flag is set while a process has a positive preempt count, then the scheduler is not invoked until the preempt count drops to zero. This allows an ISR that is handled while a process executes non-preemptively in kernel mode to request rescheduling without interfering with the non-preemptive section. While this allows interrupts to be handled immediately when they are raised, the worst-case scheduling latency is not necessarily improved by this approach since a required preemption may still be delayed.

3.2.2 Hierarchical Scheduling Classes and Per-Processor Runqueues

The Linux scheduler is structured along two dimensions. From a scheduling policy point of view, the Linux scheduler is organized as a hierarchy of *scheduling classes*, where processes of lower-priority scheduling classes are only considered if higher-priority scheduling classes are idle. From an implementation point of view, the Linux scheduler is fundamentally organized in a partitioned, per-processor fashion to ensure (mostly) cache-local operation. Associated with each processor is a *runqueue* that contains the state of each scheduling class pertaining to that processor (such as a processor-local ready queue, the current time, and scheduling statistics).

[3] To be precise, the rescheduling flag is called TIF_NEED_RESCHED and located in the struct task_info structure, which is conceptually part of the PCB but allocated separately.

Policies. Each process belongs to exactly one scheduling class at any given time, but may change its scheduling class at runtime by means of the sched_setscheduler() system call. In order of decreasing priority, the three main scheduling classes are the FP real-time scheduler (SCHED_FIFO and SCHED_RR), the general-purpose scheduler CFS (denoted SCHED_OTHER in POSIX parlance), and the idle class used for background work. Each scheduling class implements an interface consisting of 22 methods.[4] Whenever the Linux scheduler is invoked, it traverses the scheduling class hierarchy in order from the real-time class to the idle class by invoking the pick_next_task() method of each class. When a scheduling class returns a non-null PCB, the traversal is aborted and the corresponding process is scheduled. This ensures that real-time processes always take precedence over non-real-time processes.

Locking. Each runqueue is protected by a spinlock. A runqueue's spinlock must be acquired before its state may be modified (*e.g.*, before processes may be enqueued or dequeued). Perhaps surprisingly, there are no per-process locks in Linux to synchronize accesses to PCBs. Instead, runqueue locks are used to serialize process state updates. To this end, each process is assigned to exactly one runqueue at all times, and a processor must first acquire the lock of the assigned runqueue before it may access a PCB. While ready, a process naturally belongs to the runqueue of the processor to which it is currently assigned; when a process suspends, it remains under management of the processor where it was last scheduled until it is resumed (when it may be assigned to the runqueue of another processor as a means of load-balancing).

Due to the processor-local nature of runqueues, a processor must acquire two (or more) locks whenever a consistent modification of the scheduling state of multiple processors is required. For example, this arises whenever a process is migrated: it must be atomically dequeued from the ready queue of the source processor and enqueued in the ready queue of the target processor. Unless coordinated carefully, such "double locking" could quickly result in deadlock. Therefore, Linux requires that runqueue locks are always acquired in order of increasing memory addresses; that is, once a processor holds the lock of a runqueue at address A_1, it may only attempt to lock a runqueue at address A_2 if $A_1 < A_2$. This imposes a total order on runqueue lock requests; deadlock is hence

[4] In an object-oriented language such as C++ or Java, an interface is a set of named virtual methods that a conforming class must implement. In the C language, an interface is implemented as a record of function pointers. Following established object-oriented terminology, we refer to these function pointers as methods as well.

impossible. Consequently, a processor that needs to acquire a second, lower-address runqueue lock must first release the lock that it already holds and then (re-)acquire both locks. As a result, the state of either runqueue may change in between lock acquisitions. Considerable complexity in the Linux scheduler is devoted to detecting and reacting to such concurrent state changes.

Linux further supports hierarchical scheduling of process groups in both the CFS and FP scheduling classes by means of the *cgroups* mechanism. Hierarchical scheduling is beyond the scope of this dissertation and not further considered.

3.2.3 The Timesharing Scheduling Class

The POSIX standard does not specify how non-real-time timesharing should be implemented, which is reflected by the fact that the timesharing policy is simply called SCHED_OTHER. In Linux, CFS is the default scheduling class used for virtually all non-real-time tasks.

The two primary goals of CFS are to maximize throughput and to minimize lag as defined by proportional-share fair scheduling (Tijdeman, 1980; Stoica *et al.*, 1996). To reduce cache-related overheads, which impede throughput, CFS operates on a partitioned, per-processor basis. That is, when selecting the next process to run, CFS only considers processes assigned to the local runqueue. Additionally, there is a load-balancing heuristic that migrates processes to spread out runnable processes among all processors, as permitted by the processor affinity mask of each process. The load-balancing heuristic is triggered periodically and also when processors would otherwise become idle. Notably, the load-balancing heuristic considers the cache topology on multicore chips and prefers processors that are "close" to a task's previous processor, *i.e.*, the load balancer preferentially migrates processes among processors that share a cache.

On each processor, ready processes that are not scheduled are stored in a red-black tree. The red-black tree is keyed by a quantity that (roughly) corresponds to the negated lag of a process at the time of insertion. At any time, the left-most element in the tree is considered to be the highest-priority queued process. By default, a scheduled process may run uninterruptedly for about $5 \cdot (1 + \log m)$ milliseconds, and no less than $2 \cdot (1 + \log m)$ milliseconds,[5] after which its lag is recomputed and its position in the red-black tree updated (recall that m denotes the number of processors). If the

[5] There does not appear to be an analytic justification for these values; rather, they are assumed to yield an acceptable compromise between lag and throughput for many workloads in practice.

lag update causes the currently scheduled process to no longer be the left-most element in the tree, then the process is preempted. However, the precise time between lag updates is subject to many additional rules and configurable settings, and difficult to predict with certainty.

Since load-balancing occurs only infrequently (compared to the number scheduler invocations in total), CFS is essentially a partitioned scheduler with a corresponding high cache affinity for both processes and the CFS implementation itself, which is essential to enabling high throughput. A major weakness of CFS is that it is not formally (or even informally) specified (aside from its implementation in the Linux kernel) and heavily reliant on heuristics. Worse, these heuristics are subject to frequent "tweaking" and may change from kernel release to kernel release. As a result, CFS is ill-suited to formal analysis and does not lend itself to providing *a priori* guarantees. It therefore is of little relevance to analytically sound, predictable real-time systems.

3.2.4 The Real-Time Scheduling Class

Linux's scheduling class for real-time tasks implements FP scheduling with 100 distinct priorities. Both the SCHED_FIFO and SCHED_RR polices mandated by the POSIX standard are mapped to this scheduling class. Since Linux supports processor affinity masks, this scheduling class can be configured to realize either P-FP or G-FP scheduling (or hybrids thereof). To begin, we restrict our focus to the simpler P-FP case, where each real-time process has a processor affinity mask that allows only one processor and no migrations are required.

Ready queue. Each runqueue contains a ready queue for the FP scheduling class that consists of an array of 100 linked lists used to queue ready processes at each priority level. Additionally, each runqueue contains a bitfield containing one bit per priority level to indicate non-empty lists. This allows a ready process of the highest priority to be found quickly in two simple steps: first, the bitfield is scanned for the first non-zero bit, which most architectures support with special-purpose instructions; second, the head of the linked list corresponding to the index of the first non-zero bit is dequeued. If this causes the list to become empty, the corresponding bit is reset.

In the Linux documentation and literature, this implementation is commonly called the "$O(1)$ scheduler" since the number of instructions required to dispatch a process is independent of the number of ready processes. That is, since there are only 100 distinct priorities, a non-empty queue

is found after at most 100 comparisons (assuming a lack of hardware-based bit searching), even if there are thousands of processes pending. Strictly speaking, however, this is not a constant-time algorithm: in the worst case, the number of required instructions is linear in the number of distinct priorities—which happens to be constant in Linux. In other words, claiming that Linux's FP scheduler has $O(1)$ runtime complexity is analogous to stating that a fixed-size array can be sorted in constant time: true in the narrow special case, but misleading in general. For example, Linux's approach of using a simple, sequentially scanned bitfield would not scale to supporting 2^{16} distinct priorities, as offered by some proprietary RTOSs. Nonetheless, with only 100 distinct priorities and hardware-supported bit search, the first set bit can be found with only four instructions on a 32-bit platform and two instructions on a 64-bit platform (in addition to conditional jumps). In practice, Linux's FP scheduling class thus quickly identifies and dispatches the highest-priority ready process with little overhead.

When a process resumes, it is enqueued at the end of the linked list corresponding to its current priority. This ensures that processes of equal priority are scheduled in FIFO order on each processor. Processes of type SCHED_FIFO are allowed to run for as long as required until they suspend, whereas processes of type SCHED_RR are preempted and moved to the end of the linked list after exhausting a time slice of 100 ms (and thus forced to alternate in a round-robin fashion if there are multiple processes of the same priority). As long as process priorities are unique, SCHED_FIFO and SCHED_RR are equivalent to each other and amenable to standard response time analysis (Section 2.3.1.1). However, by default Linux executes processes without temporal isolation (*i.e.*, a high-priority process that overruns its budget can delay lower-priority processes for arbitrarily long times). Recent versions of Linux have gained support for "real-time throttling," which provides a limited means of temporal isolation.

On-demand migrations. In the case of processes that may be scheduled on multiple processors (as is the case under G-FP scheduling), the FP scheduling class must consider multiple processors in its scheduling decisions. That is, it must ensure that a process is scheduled as soon as at least one of the processors permitted by its affinity mask is not executing a higher-priority process. This may require a process to be migrated "on demand," that is, as part of a scheduling decision.

```
1  pull−process−to(RQ_i):
2      foreach x ∈ {1, ..., m} \ {i} such that RQ_x is not empty:
3          lock RQ_x // while holding lock for RQ_i

5          let PC_r ← the highest-priority ready process queued in RQ_x
6                          that includes processor i in its affinity mask (if any)
7          let PC_l ← the highest-priority ready process queued in RQ_i (if any)

9          if (PC_r exists
10             and (PC_l does not exist
11                 or PC_r's priority exceeds PC_l's priority)):
12             migrate PC_r from RQ_x to RQ_i

14         unlock RQ_x
```

Listing 3.1: Linux's target-initiated pull operation.

In contrast to the CFS scheduling class, the FP scheduling class thus does not have a separate load-balancing heuristic. Instead, processes are migrated as needed prior to and after each scheduling decision. A process migration always involves a target and a source processor, either of which may initiate a migration: either a source processor *pushes* waiting processes to other processors, or a target processor *pulls* processes from processors backlogged with higher-priority processes. These operations are expressed as simplified pseudo-code in Listings 3.1 and 3.2. In the pseudo-code and the following discussion, we let RQ_i denote the runqueue of processor i and let PC_r denote a process.

The pull operation is carried out by the FP scheduling class prior to selecting the next process to schedule whenever it is about to schedule a process of lower priority than previously scheduled (*e.g.*, when the previous job of a higher-priority task is completed). A migration is required at this point if there exists a process PC_r that satisfies four conditions:

1. PC_r is currently not scheduled;

2. PC_r is currently assigned to a remote processor's runqueue;

3. the pulling processor is included in PC_r's processor affinity mask; and

4. PC_r's priority exceeds that of any local process.

```
1  push−process−from(PC_l, RQ_i):
2      let attempt ← 1
3      let success ← ⊥
4      while attempt ≤ 3 and ¬success: // retry loop
5          set attempt ← attempt + 1
6          foreach x ∈ {1, ..., m} \ {i} such that x is included in PC_l's affinity mask:
7              // check if PC_l is eligible to be scheduled on processor x
8              let PC_r ← process currently scheduled on processor x
9              if (PC_r does not exist or PC_l's priority exceeds PC_r's priority):
10                 // attempt migration to potential target runqueue
11                 lock RQ_x // while holding lock for RQ_i
12                 if attempt did not race with concurrent state change:
13                     migrate PC_l from RQ_i to RQ_x
14                     set success ← ⊤
15                 unlock RQ_x
16                 break // abort foreach loop after locking one remote runqueue
```

Listing 3.2: Linux's source-initiated push operation.

Linux's pull operation (Listing 3.1) thus sequentially checks every remote processor that has more than one real-time process assigned to it (line 2 in Listing 3.1) to see if there exists a queued (but not scheduled) process that is eligible to be migrated. That is, if the highest-priority queued process satisfies each of the four listed conditions, then it is migrated to the pulling processor (lines 5–12).

The push operation is carried out after a process has been selected to be scheduled, and also when a process is resumed. It checks to see if any of the locally queued, ready processes (*i.e.*, those not selected to be scheduled) could be scheduled immediately on a remote processor. That is, a process PC_l must be pushed to a target processor x if it satisfies the following four conditions:

1. PC_l is currently not scheduled;
2. PC_l is currently not assigned to RQ_x;
3. processor x is included in PC_l's processor affinity mask; and
4. PC_l's priority exceeds that of any process currently assigned to RQ_x .

To reduce overhead, Linux attempts to push each process only once each time that it is preempted or resumed; if the push attempt fails because no processor is currently available, then the process will not be considered again until it has been scheduled or migrated. To check whether a local process PC_l should be migrated, Linux's push operation (Listing 3.2) checks each processor permitted by

PC_l's processor affinity mask (line 6 in Listing 3.2) to see if condition 4 is satisfied (lines 7–9). If it identifies a processor that appears to be a valid migration target (line 9), it locks the corresponding runqueue (line 11), attempts to carry out the migration (lines 12–14), and then terminates the candidate search (lines 15–16). Due to the non-atomicity of scheduling decisions, a push attempt may also fail if it races with a concurrent scheduling event on the target processor (lines 11–15). Linux will try to push the task up to three times before giving up (lines 2–5). After three failed attempts to find a suitable target processor, Linux abandons the process and relies on remote processors to pull it at a later time.

Processes with processor affinity masks that allow scheduling on only a single processor are exempt from pushing and pulling and are thus skipped by the migration code. Linux's push operation also considers a platform's cache topology; such details are omitted in Listing 3.2 for the sake of clarity.

Limitations. The primary advantage of Linux's partitioned runqueue architecture is low average-case overheads: in most cases, processors can make local scheduling decisions without the need for global coordination. However, this inherently partitioned design greatly complicates support for (occasional) pushing and pulling due to the need to (re-)acquire multiple runqueue locks for each migration (*e.g.*, in line 3 in Listing 3.1 and line 11 in Listing 3.2), with the resulting chance of concurrent state changes. Worse, since the current scheduling state is split across multiple data structures, it is not possible for processors to obtain a *consistent snapshot* of the current process-to-processor mapping (without locking all runqueues simultaneously, which would defeat the purpose of having per-processor locks). As a result, there is a chance that processors may either trigger superfluous migrations or may fail to trigger required migrations. We briefly demonstrate each case with an example.

Both pull and push operations may trigger superfluous migrations; here, we consider the pulling of lower-priority processes. In theory—if scheduling decisions are atomic—at most one process needs to be pulled since only it will be scheduled. Other processes may remain where they are since it is irrelevant where a process is queued while it is not scheduled. However, as implemented in Linux, a pull operation may cause up to $m - 1$ migrations to occur if processes are encountered in order of increasing priority.

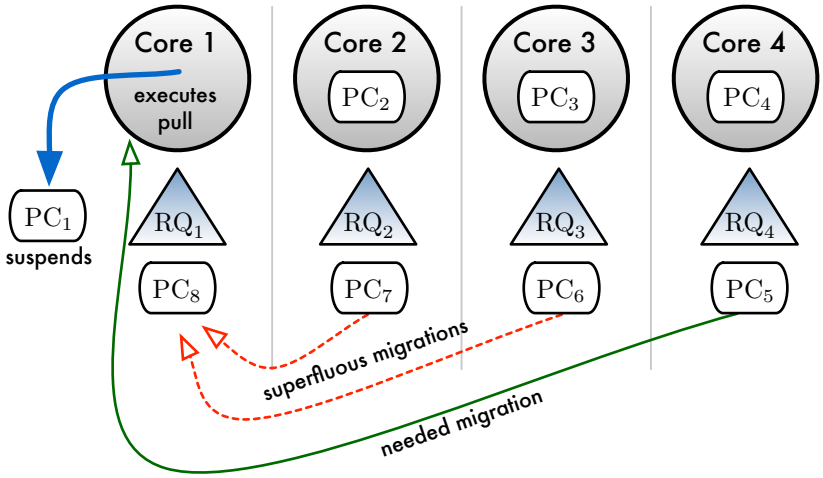


Figure 3.1: Example of unneeded migrations caused by Linux's pull operation. When process PC_1 suspends, process PC_5 must be pulled from runqueue RQ_4 to runqueue RQ_1 so that it can be dispatched next. However, since Linux's pull operation examines remote runqueues in order of increasing indices, two unneeded migrations are triggered as well.

Example 3.1. An example scenario that triggers such unneeded migrations is illustrated in Figure 3.1. There are $m = 4$ processors and eight processes PC_1–PC_8 with corresponding unique priorities 1–8. There are two processes assigned to each processor, such that processes PC_1–PC_4 are scheduled and processes PC_5–PC_8 are queued and waiting, as indicated in Figure 3.1. Now suppose that process PC_1 on processor 1 suspends (*i.e.*, its current job completed). At this point, the next-highest-priority process to be scheduled is process PC_5, which is currently assigned to processor 4. However, since Linux's pull operation checks processors in order of increasing index, it migrates process PC_7 from processor 2 and process PC_6 from processor 3 to processor 1 prior to discovering process PC_5 on processor 4. Depending on when the processes PC_2–PC_4 suspend, the just-migrated processes PC_6 and PC_7 may eventually have to be returned to where they came from. ◊

This shows that processes may be subject to unneeded migrations. In general, it is impossible to anticipate under global scheduling on which processor the next-highest-priority process is queued. An example similar to Example 3.1 can thus be constructed for any processor traversal order. The risk of unneeded migrations is fundamental to a one-pass pull operation and stems from the lack of global knowledge, *i.e.*, the pulling processor does not know which processes are ready on remote processors. Some unneeded migrations could possibly be avoided by a two-pass pull operation that

first checks each processor and then selects the highest-priority encountered process for migration; however, such an approach would still suffer from the lack of a consistent snapshot.

Besides causing unneeded migrations, both push and pull operations may miss required migrations if the scheduling state of remote processors changes concurrently. We consider push migrations as an example. To push a local process to a remote runqueue, the scheduler must identify a processor that currently executes a process of lower priority or that is idle (lines 6–9 in Listing 3.2). Since it is not possible to obtain a consistent snapshot (without locking all runqueues), the set of ready processes assigned to each processor may change repeatedly while the pushing processor attempts to identify the "best" migration target.

After the migration target has been selected, the pushing processor must lock the runqueue of the target processor (line 11 in Listing 3.2). Again, there is a possibility of a concurrent state change: the identity of the highest-priority scheduled process may change between the time when the migration target has been selected (line 6) and the time when the pushing processor has acquired the target's runqueue lock (line 11). This is especially problematic if a higher-priority process resumes during this race window—the prior selection of the migration target has then been invalidated and the attempt to push the process has failed (line 12).

As discussed above, the push operation has a retry loop that re-attempts pushing the process up to three times to cope with failed push attempts due to concurrent updates (lines 2–5).[6] Alas, after a push operating has failed three times due to races with concurrent scheduling events, the pushing processor gives up. This can lead to a needed migration being missed.

Example 3.2. Such a scenario is illustrated in Figure 3.2. Initially, only process PC_5 is ready on processor 1. Suppose process PC_4 resumes on processor 1 (*i.e.*, it was previously scheduled on processor 1 and is then resumed by an ISR that is also executed on processor 1). This causes PC_5 to be preempted and it must thus be pushed to one of the idle processors.

As shown in Figure 3.2(a), processor 2 is picked as the migration target for the first attempt. However, a concurrent scheduling event occurs and process PC_2 is resumed on processor 2 before the migration is enacted. As a result, processor 2 is no longer a valid migration target; therefore the first attempt fails. Processor 1 then retries finding a valid migration target, as shown in Figure 3.2(b).

[6]The retry limit of three is hard-coded in Linux's find_lock_lowest_rq() function.

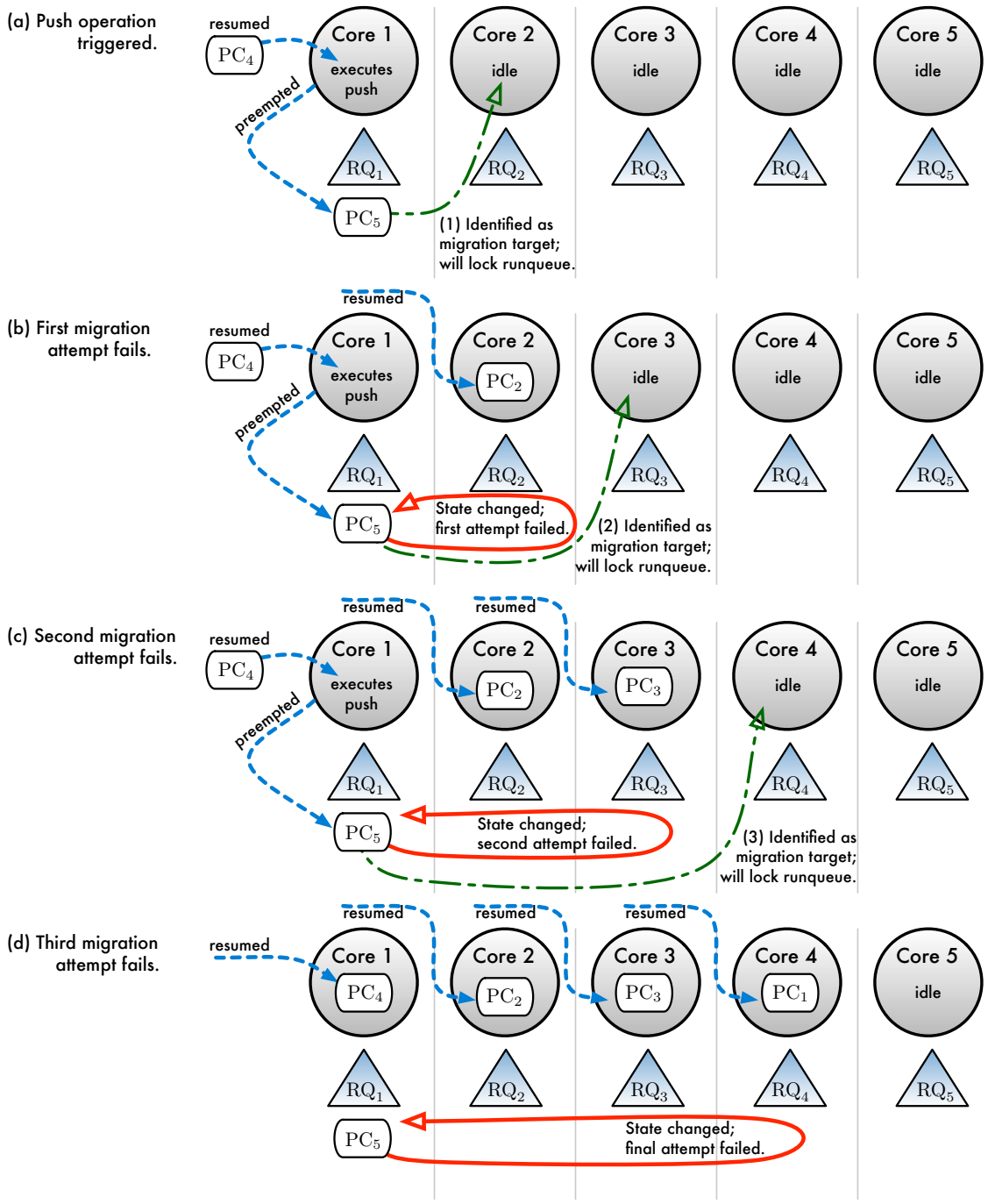


Figure 3.2: Example of a required migration missed by Linux's push operation. As PC_5 is preempted by PC_4, it should be pushed to another processor. The four insets depict four states encountered during the search for a migration target: **(a)** RQ_2 is identified as a suitable target; **(b)** PC_2 is resumed, RQ_2 becomes unsuitable, and RQ_3 is identified as a suitable target; **(c)** PC_3 is resumed, RQ_3 becomes unsuitable, and RQ_4 is identified as a suitable target; and **(d)** PC_1 is resumed, RQ_4 becomes unsuitable, and the push operation fails without enacting a migration.

Processor 3 is picked as the migration target for the second attempt. However, in the scenario depicted in Figure 3.2(c), the migration attempt races with the process PC_3 as it resumes. As a result, the second migration attempt fails as well.

Finally, suppose that processor 4 is picked as the migration target for the third and final attempt. Again, a higher-priority process is resumed after processor 4 was observed as idle and before the migration could be enacted, *i.e.*, before the runqueue RQ_4 was locked by processor 1. The last attempt thus failed, too, and processor 1 gives up trying to push the process since the operation has reached the retry limit. As a result of the failure to push, process PC_5 remains queued on processor 1 and must wait even though there exists an idle processor. ◊

In practice, push failures are likely rare and quickly corrected because other processors will likely initiate pull operations with a compensating effect. Nonetheless, this example results in a non-work-conserving schedule and thus breaks a fundamental assumption of all published schedulability analysis for G-FP scheduling. This example can further be trivially modified to yield an unbounded priority inversion instead of non-work-conserving behavior by adding a lower-priority process PC_6 to processor 5: while PC_6 executes, processor 5 will not invoke the scheduler and thus not pull the higher-priority PC_5 (the presence of PC_6 would not affect the failed migration attempts and scheduling decisions of processors 2–4).

As the two preceding examples demonstrate, Linux's push- and pull-based implementation of global (or clustered) scheduling is vulnerable to "migration glitches." As a result, a process may be needlessly pushed and pulled before it is scheduled, and may suffer additional delays due to non-work-conserving phases or additional priority inversions, which are likely brief but nevertheless incorrect. A comment in Linux's source code (Torvalds and contributors, 2010) notes that unnecessary migrations are a "low-probability event"; we agree that it indeed has likely no noticeable impact on average-case overheads and latencies (for most workloads). Similarly, another comment notes that failure to push a task is acceptable since "the other [processors] will pull from [the local processor] when they are ready." Again, this is likely true on average, but it is not clear from Linux's source code itself or from the accompanying comments that the retry limit of three iterations is sufficient in *all* cases (Example 3.2 suggests it is not; we are not aware of a proof to the contrary).

Taken together, the possibility of such "migration glitches" greatly complicates any analytical worst-case charatcterization of Linux's current migration logic.[7] In fact, to the best of our knowledge, no such analysis has been published, and it is far from obvious how it could be derived. As an aside, we note that the SCHED_DEADLINE project adopted Linux's push- and pull-based approach to enacting migrations, including the fixed retry limit. Their implementation of G-EDF and C-EDF is thus similarly vulnerable to the "migration glitches" illustrated in this section.

As discussed in the next section, in the implementation of LITMUS$^{\mathrm{RT}}$ we avoided distributing scheduler state across processors in global schedulers in favor of globally shared state with coarse-grained locking. While coarse-grained locking is certainly less desirable from a throughput point of view, it has the benefit of being relatively straightforward to analyze, which is crucial to our work.

3.3 The Design and Implementation of LITMUS$^{\mathrm{RT}}$

LITMUS$^{\mathrm{RT}}$ is a native real-time Linux version that is focused on extending the stock Linux kernel with multiprocessor real-time scheduling policies and locking protocols. The first prototype of LITMUS$^{\mathrm{RT}}$ was developed by Calandrino *et al.* (2006) based on Linux 2.6.9, but not publicly released. The first public version of LITMUS$^{\mathrm{RT}}$, named LITMUS$^{\mathrm{RT}}$ 2007.1 and released in May 2007,[8] was based on Linux 2.6.20 and described in detail in (Brandenburg *et al.*, 2007). Since May 2007, there have been eight releases of LITMUS$^{\mathrm{RT}}$ spanning 16 kernel versions. The version of LITMUS$^{\mathrm{RT}}$ discussed in this dissertation is based on LITMUS$^{\mathrm{RT}}$ 2011.1, which in turn is based on Linux 2.6.36. Aside from the project's purpose, the current version of LITMUS$^{\mathrm{RT}}$ has little in common with the first prototype (Calandrino *et al.*, 2006) and virtually none of the initial prototype's source code has been retained in the current codebase.

Linux 2.6 was chosen as the base for LITMUS$^{\mathrm{RT}}$ due to its open-source nature, and due to its excellent multiprocessor and hardware support. This made it possible to focus on the desired algorithmic changes instead of having to develop custom drivers or multiprocessor support. Indeed, Linux's support for a wide range of processor architectures has made it possible to develop and maintain

[7] Given the rapid pace of development in Linux's kernel, these issues will likely be addressed eventually. However, they persist in the latest stable kernel version at the time of writing (Linux 2.6.39).

[8] The version numbering convention is *year-of-release.per-year-sequence-number*. For example, version 2007.2 was the second release of LITMUS$^{\mathrm{RT}}$ in 2007 and version 2008.1 was the first release in 2008.

LITMUS$^{\mathrm{RT}}$ versions with support for Intel x86, Sun SPARC, and ARM-based multiprocessors. The current version of LITMUS$^{\mathrm{RT}}$ supports Intel's 32-bit and 64-bit x86 architectures, as well as the ARMv6 architecture (support for Sun's SPARC architecture was retired in late 2010).

Design goals. LITMUS$^{\mathrm{RT}}$ does not aim for POSIX-compliance, and also is not intended to replace Linux's standard scheduler. Rather, LITMUS$^{\mathrm{RT}}$'s implementation primarily extends Linux by adding new functionality while trying to minimize the changes required to the existing Linux code. This was done for two reasons, one pragmatic and one conceptual. The pragmatic reason is that the more invasive a patch is, the harder maintenance becomes in the face of a rapidly evolving base Linux. For a small project team, it is thus imperative to reduce the amount of "surface friction" when maintaining external patches. The second, more fundamental reason is that POSIX and Linux are constrained by backwards compatibility—in our work, we are not interested in legacy support, but rather seek to investigate multiprocessor real-time systems assuming a "clean slate." That is, we use Linux as a convenient base for our research, but do not explicitly target Linux improvements *per se*.

The focus of LITMUS$^{\mathrm{RT}}$ is algorithmic changes; it does not improve the Linux kernel's real-time capabilities in terms of interrupt latencies and does not force split interrupt handling. LITMUS$^{\mathrm{RT}}$ is thus complimentary in goals to the PREEMPT_RT patch, which targets these areas but does not change or augment the implemented scheduling algorithms (see page 156). However, this also implies that LITMUS$^{\mathrm{RT}}$ is not applicable to workloads with very short relative deadlines, like mainline Linux itself. While not ideal, we consider this to be an acceptable tradeoff given our research goals.

LITMUS$^{\mathrm{RT}}$ was not based on the PREEMPT_RT patch because the PREEMPT_RT patch was less stable in 2006 than it is today (and particularly so on multiprocessors) and, at least until recently, was itself subject to a high rate of change. In the future, it would be beneficial to rebase LITMUS$^{\mathrm{RT}}$ on top of the PREEMPT_RT patch to incorporate its infrastructure improvements and latency reductions. From a conceptual point of view, both real-time patches could easily be merged since they address orthogonal concerns; however, from a practical point of view, it does pose a considerable engineering challenge due to conflicting changes and slight differences in assumptions (*e.g.*, LITMUS$^{\mathrm{RT}}$ would require some changes to properly accommodate split interrupt handling).

In the following, we first provide a high-level overview of the design and architecture of LITMUS$^{\mathrm{RT}}$ and then discuss some of the encountered challenges and proposed solutions in detail.

As in any non-trivial OS project, many implementation-level, software engineering, and tool-related challenges and limitations had to be resolved during the development of LITMUS$^{\mathrm{RT}}$. Many of these were due to concurrency issues inherent in parallel and interrupt-driven programming, the considerable complexity of the Linux kernel, and the resulting increased incidence of programming errors. However, such details are of only limited interest from an academic point of view; the truly interested reader is referred to the LITMUS$^{\mathrm{RT}}$ project homepage and the project's public source code repository.[9] Instead, the following discussion focuses on design choices and algorithmic solutions of a more fundamental nature that are applicable beyond a Linux setting. We first provide a high-level overview of the components that comprise LITMUS$^{\mathrm{RT}}$.

3.3.1 The Architecture of LITMUS$^{\mathrm{RT}}$

LITMUS$^{\mathrm{RT}}$ consists of four main parts: the core infrastructure, a number of scheduler plugins, the user-space interface, and the user-space library and tools. Their respective purpose and dependencies are illustrated in Figure 3.3.

3.3.1.1 The LITMUS$^{\mathrm{RT}}$ Core Infrastructure

The two primary purposes of the LITMUS$^{\mathrm{RT}}$ core infrastructure are **(i)** to provide policy-agnostic "glue code" needed to integrate with the Linux kernel, and **(ii)** to provide a foundation of reusable components (such as a ready queue implementation) and shared functionality (such as task parameter validation and debugging infrastructure) that simplify the development of scheduler plugins.

Regarding (i), the core infrastructure contains the LITMUS$^{\mathrm{RT}}$ scheduling class, thereby hooking into the Linux scheduling framework. The LITMUS$^{\mathrm{RT}}$ scheduling class is added to the Linux scheduling hierarchy as the highest-priority (*i.e.*, first-checked) scheduling class, which allows LITMUS$^{\mathrm{RT}}$ to override Linux's normal scheduling decisions. Besides adding a custom scheduling class, LITMUS$^{\mathrm{RT}}$ also extends the PCB to hold LITMUS$^{\mathrm{RT}}$-specific state and hooks into a few process management functions such as the implementation of the fork() and exit() system calls to initialize and cleanup said state.

[9] See `http://www.cs.unc.edu/~anderson/litmus-rt`.

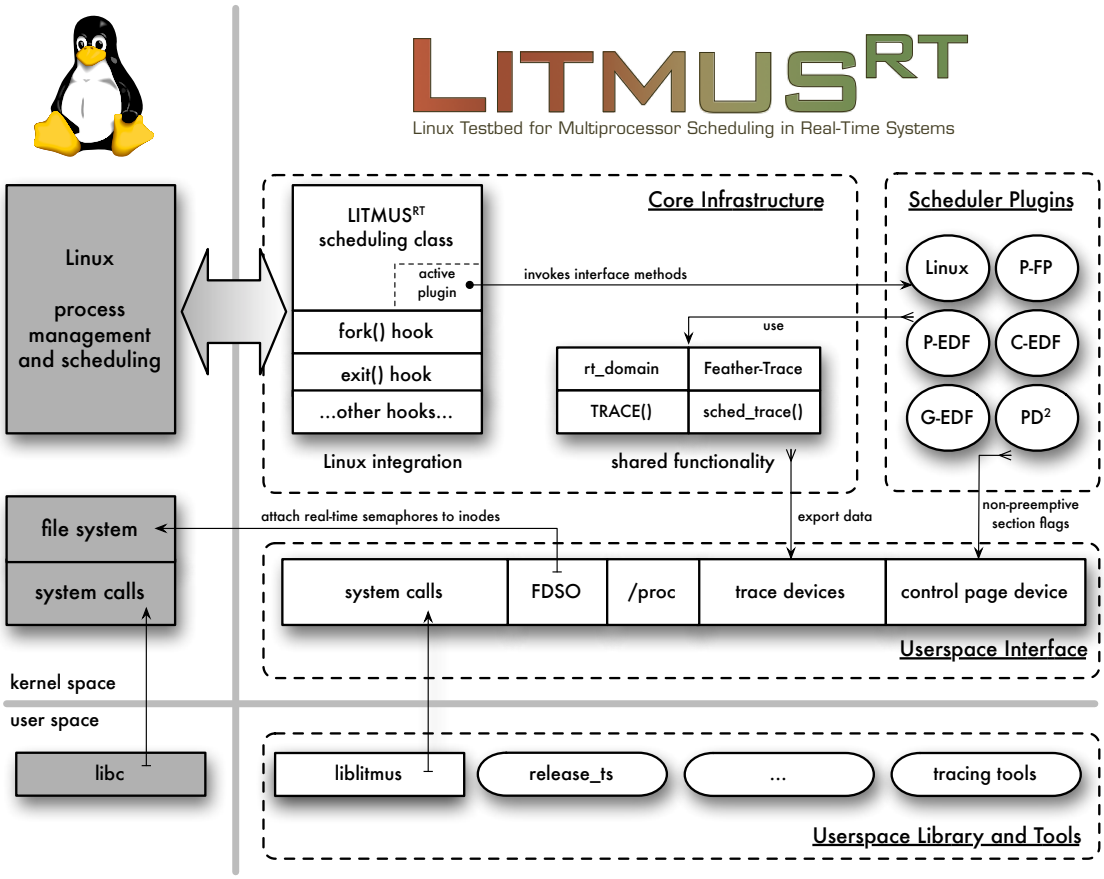


Figure 3.3: Diagram of the structure of LITMUS$^{\mathrm{RT}}$, which is comprised of four main components.

Since LITMUS$^{\mathrm{RT}}$ is designed to be non-invasive, Linux's standard scheduling classes are not modified. Only processes admitted to the LITMUS$^{\mathrm{RT}}$ scheduling class are considered real-time processes in LITMUS$^{\mathrm{RT}}$; "real-time" processes of type SCHED_FIFO or SCHED_RR are considered to be background, best-effort processes. This has the pragmatic benefit that the system behaves like a normal Linux system in the absence of real-time processes, which greatly simplifies system administration and the preparation and evaluation of experiments.

The LITMUS$^{\mathrm{RT}}$ scheduling class itself does not implement any particular policy. Rather, it is a (mostly) thin wrapper that delegates all scheduling decisions to the currently active scheduler plugin (discussed below). This layer of indirection between scheduler plugins and Linux's scheduling hierarchy is required for two reasons. First, Linux does not support adding, removing, or switching scheduling classes at runtime (in contrast, scheduler plugins can be switched at runtime). Second, global and clustered scheduler plugins require specific, particularly tricky migration support that is

factored out into a common "migration path" in the LITMUS$^{\mathrm{RT}}$ scheduling class (see Section 3.3.4 below). As an additional benefit, this indirection allows scheduler plugins to be programmed assuming a simple, real-time-specific interface that changes only rarely. In contrast, Linux's scheduling class interface is more complex and changes frequently between versions.

The LITMUS$^{\mathrm{RT}}$ scheduling class does not implement Linux's full scheduling class interface, but only the features required for identical multiprocessors. In particular, LITMUS$^{\mathrm{RT}}$ currently does not support processor frequency scaling and other power-savings measures, and also does not support "hot-plugging," *i.e.*, the addition or removal of processors at runtime. Additionally, LITMUS$^{\mathrm{RT}}$ ignores processor affinity masks—whether scheduling occurs in a global, clustered, or partitioned fashion is determined by the currently active plugin and not by processes.

We note that early versions of LITMUS$^{\mathrm{RT}}$ that were based on Linux 2.6.20 (*i.e.*, before the introduction of the scheduling class hierarchy in Linux 2.6.23) used a very different approach to integrate with Linux. Since Linux 2.6.20 did not provide an easy method for adding additional scheduling policies, LITMUS$^{\mathrm{RT}}$ instead dynamically changed the priority of SCHED_FIFO processes. That is, LITMUS$^{\mathrm{RT}}$'s core infrastructure enacted a plugin's scheduling decisions by assigning the selected processes the highest SCHED_FIFO priority, and by assigning a lower priority to all other processes. As a result, Linux would then dispatch the SCHED_FIFO processes selected by the currently active LITMUS$^{\mathrm{RT}}$ plugin. This approach required each LITMUS$^{\mathrm{RT}}$ process to be queued in the SCHED_FIFO ready queue on the processor on which it was selected for execution by the plugin, which (slightly) increased migration overhead due to the need to dequeue and enqueue a process prior to dispatching it. In contrast, since LITMUS$^{\mathrm{RT}}$ 2008.1, the core infrastructure implements a proper scheduling class and LITMUS$^{\mathrm{RT}}$ processes are no longer *queued* in Linux's ready queues. Instead, LITMUS$^{\mathrm{RT}}$ processes are merely *assigned* (but not stored within) Linux runqueues as required by Linux for the purpose of serializing PCB state changes (recall Section 3.2.1). Each scheduler plugin allocates and manages ready queues individually as appropriate for the implemented policy.

With regard to purpose (ii) mentioned at the beginning of this section, the LITMUS$^{\mathrm{RT}}$ core infrastructure provides some frequently needed abstractions and utility code. Most of this functionality is Linux-related minutiae; two of the more interesting parts are discussed below, namely the "migration path" (Section 3.3.4) and an abstraction termed "real-time domains," which provide reusable ready and release queues (Section 3.3.5).

3.3.1.2 Scheduler Plugins

To facilitate scheduler development, LITMUS$^{\mathrm{RT}}$ provides a plugin interface that allows new scheduling policies to be implemented without being exposed to the "full" Linux kernel. Originally, the scheduler plugin interface was intended to simplify scheduler development so that researchers that are not Linux experts could implement scheduling policies inside a real OS kernel. However, experience over the last five years has shown that novice developers still struggle with the complexities of programming in a kernel even after much of Linux's complexity has been hidden—realistically, a certain degree of "kernel hacking" aptitude is required to work with LITMUS$^{\mathrm{RT}}$.

However, we have found the plugin interface to be beneficial for other reasons. For one, it provides (in our opinion) a conceptually cleaner interface than Linux's scheduling class interface, which reduces the mental burden during development. Second, LITMUS$^{\mathrm{RT}}$'s plugin interface has remained relatively stable over the years so that the maintenance burden associated with adding scheduling plugins has been reduced (compared to tracking changes in Linux in each plugin). Third, the plugin interface makes it possible to switch the active scheduling policy at runtime. We briefly elaborate on the benefits and limitations of the last point.

Active plugin. The *active plugin* is the scheduler plugin that is invoked by the LITMUS$^{\mathrm{RT}}$ scheduling class whenever a scheduling decision has to be made. By default (*i.e.*, after the system is booted), the active plugin is the "Linux plugin," which is merely a placeholder that does not allow any process to become a real-time process. The scheduling policy can later be switched at runtime by activating another of the included scheduler plugins.

There are several possible interpretations of what it means to "switch the scheduler at runtime." From a scheduling-theoretic point of view, it could be possible to switch the scheduling policy *during the execution of a task set*. This would require intricate schedulability analysis to show that the abrupt change in scheduling rules does not invalidate timing guarantees of pending jobs. To the best of our knowledge, such analysis has not been carried out for multiprocessor real-time schedulers, though a somewhat related topic, *mode change protocols*,[10] has received some attention lately; *e.g.*, see (Nelis *et al.*, 2009) for one recent example.

[10] A mode change protocol is required when a system may transition between operating modes with different task parameters (*e.g.*, a fault-tolerant system could have a "normal mode" and an "emergency mode" with reduced functionality).

Since there is no backing theory, LITMUS$^{\mathrm{RT}}$ does not allow switching the active plugin while there are real-time processes present. Instead, the active plugin may be changed only before a task set is configured, or after it has been terminated. Early versions of LITMUS$^{\mathrm{RT}}$ were even more restrictive and required selecting the active scheduler plugin at boot time. The ability to switch plugins at runtime has proved critical for our experiments: it reduces the need for manual intervention and thus saves time, which allows for larger, more extensive experiments.

The scheduler plugins in LITMUS$^{\mathrm{RT}}$ should not be confused with Linux's *loadable kernel modules* (LKMs), which, for example, are used to realize loadable device drivers in Linux. LKMs can be inserted into the kernel at runtime (like dynamically loaded libraries). In contrast, LITMUS$^{\mathrm{RT}}$ currently requires all scheduler plugins to be statically linked into the kernel. To support scheduler plugins to be loaded as LKMs, three features are required: **(i)** each scheduler plugin must be self-contained, **(ii)** scheduler plugins must be dynamically registered (*i.e.*, there is no hardcoded list of all plugins), and **(iii)** the LITMUS$^{\mathrm{RT}}$ core infrastructure must implement proper reference counting to prevent a scheduler plugin to be unloaded while it is in use. The current version of LITMUS$^{\mathrm{RT}}$ satisfies requirements (i) and (ii), but does not (yet) implement the required reference counting. This could be added with reasonable effort, but we have not found a pressing need for LKM support to date. One reason to support LKMs could be to enable the loading and unloading of a plugin during development and testing. However, the Linux kernel typically does not recover from faults in the scheduler, so frequent reboots (of virtual machines) are typical for scheduler development.

Plugin interface. The LITMUS$^{\mathrm{RT}}$ plugin interface consists of 13 methods that can be grouped into roughly four groups based on functionality (summarized in Table 3.1). We provide a brief overview to illustrate which events must be handled by each plugin, *i.e.*, how and when a scheduler plugin is invoked.

There are two main scheduling methods used to defer scheduling decisions to the active scheduler plugin that allow both event-driven and quantum-driven scheduling to be implemented.

- The method schedule() is called whenever a process must be selected for execution. The main scheduling logic of event-driven policies such as G-EDF is usually implemented in this method. In quantum-driven plugins, schedule() merely enacts the scheduling decision that was computed at the last quantum boundary.

Method	Purpose	Optional
schedule()	Select next process to be scheduled.	no
tick()	Called at quantum boundary to trigger scheduler.	yes
finish_switch()	Bookkeeping after context switch.	yes
release_at()	Prepare future timed job release.	yes
task_block()	Remove process from ready queue.	no
task_wake_up()	Add process to ready queue.	no
complete_job()	Prepare next periodic job release.	no
admit_task()	Check if process is correctly configured.	yes
task_new()	Allocate and initialize per-process scheduler state.	yes
task_exit()	Free per-process scheduler state.	yes
activate_plugin()	Allocate and initialize plugin state.	yes
deactivate_plugin()	Free plugin state.	yes
allocate_lock()	Allocate real-time semaphore for process.	yes

Table 3.1: List of methods in the LITMUS$^{\mathrm{RT}}$ plugin interface and their intended purpose. A method that is optional is not required to be implemented by every plugin. Default implementations are substituted for omitted methods as placeholders.

- The method tick() is called at each quantum boundary. While mostly unused in event-driven policies, it is the central scheduling method in quantum-driven plugins such as PD^2. If a preemption is required, then tick() can cause schedule() to be called at the end of the timer ISR by setting the rescheduling-flag of the currently scheduled process (see Section 3.2.1).

The schedule() method of the scheduler plugin interface should not be confused with Linux's main scheduler function of the same name. The plugin method's name is only valid as a member of a scheduler plugin instance; there is thus no name clash at the C language level. Further, there is a third method related to scheduling that is not used by all plugins.

- After a context switch has occurred, the plugin is notified by means of a call to its finish_switch() method. This is mainly used to record which process is currently scheduled; only global and clustered scheduler plugins handle this event (see Section 3.3.4 below).

The following four methods relate to the job lifecycle and job state changes. Jobs are an accounting abstraction in LITMUS$^{\mathrm{RT}}$. The purpose of these methods is to track whether the backing process is eligible to execute, *i.e.*, whether it should be considered when schedule() selects a process for execution. This is typically achieved by maintaining a ready queue of all eligible processes; these methods thus add and remove processes from the ready queue as needed.

- The method release_at() is called to notify the scheduler plugin at which time the next job release is going to occur. The method is not called for periodic job releases (each plugin should handle those automatically); rather, it is only required when a process requires a timed job release in the future. This is typically the case for the first job of a task.

- A process that suspends is reported with task_block(). There are no process scheduling decisions to be made at this point; the method mainly exists to allow the scheduler plugin to perform any cleanup that may be required (such as removing the suspended process from a ready queue).

- When a process resumes, it is reported to the active scheduler plugin by a call to task_wake_up(). The plugin must add the process to a ready queue, otherwise the scheduler "loses track" of the process.

- Finally, when a real-time process has finished executing the current job prior to exhausting its budget, the scheduler plugin's complete_job() method is called.

In our experience, handling suspending and resuming processes is a common source of errors. A process may suspend and be resumed again almost immediately thereafter before a context switch can occur (*e.g.*, this is the case for many page faults, and especially those due to copy-on-write semantics). In the past, novice developers have commonly disregarded the possibility of such "quickly resuming" processes, which may result in ready processes failing to be enqueued in a ready queue (in which case they are never scheduled again and become "stuck") or in being wrongly enqueued twice (which crashes the kernel eventually). Besides migration support (see Section 3.3.4 below), races related to short suspensions have in our experience been the most common cause of crashes in LITMUS$^{\mathrm{RT}}$.

Besides the methods corresponding to scheduling events discussed so far, there are further a number of resource management methods. In particular, there are three methods related to real-time process admittance, initialization, and cleanup (admit_task(), task_new(), and task_exits()) and two methods related to plugin initialization and cleanup when switching plugins (activate_plugin() and deactivate_plugin()). LITMUS$^{\mathrm{RT}}$ does not currently implement any schedulability tests within the kernel; admit_task() typically does little more than checking that a process's parameters are correct (*i.e.*, enforcing that $0 < e_i \leq p_i$). The "Linux" placeholder plugin active by default after the

system started rejects all processes and thus prevents real-time processes from being created until a scheduling policy has been selected.

Finally, there is a method named allocate_lock() that is invoked when processes attempt to allocate a semaphore, which allows each plugin to provide one or more locking protocols appropriate for the implemented scheduling policy. In the terminology of object-oriented design, the method allocate_lock() realizes what is called a factory method.[11]

Locking. In standard Linux, scheduling decisions on each processor are synchronized by the processor's runqueue lock. As discussed in Section 3.2.4, this may give rise to "migration glitches" and thus is problematic for global and clustered scheduling policies. In LITMUS$^{\mathrm{RT}}$, we thus use one or more additional locks in each scheduler plugin and do not rely on the runqueue locks to protect state specific to LITMUS$^{\mathrm{RT}}$.

Notably, the plugin interface does not make any assumptions about how the active plugin implements locking. That is, how synchronization is achieved is considered to be an internal plugin implementation detail that is not exposed; each plugin implementor may thus flexibly choose a different approach to locking that suits the implemented policy best. However, since the active plugin is invoked from the LITMUS$^{\mathrm{RT}}$ scheduling class, which in turn is automatically protected by Linux's runqueue locks, the processor holds a runqueue lock when the schedule(), tick(), task_block(), task_wake_up(), or task_new() method are being invoked. This must be taken into account in the implementation of any of these methods if another, second runqueue lock must be acquired (recall that runqueue locks must be acquired in order of lock address to avoid deadlock).

The scheduler plugins underlying the experiments presented in Chapter 4 are discussed in Section 3.3.6 below.

3.3.1.3 User-Space Interface

LITMUS$^{\mathrm{RT}}$ exposes a number of additional system calls and adds several virtual character devices (discussed below). The added system calls can be categorized according to their functionality as follows:

[11] A *factory method* allows objects to be instantiated without specifying their class. The method allocate_lock() allocates and initializes abstract semaphore objects with plugin-specific locking protocol implementations; the caller does not have to be aware which specific implementation is required.

1. getting and setting sporadic task parameters and processor assignments (*i.e.*, e_i, p_i, and P_i);
2. job control for real-time processes (*e.g.*, inquiring the current job sequence number, waiting for job releases, and signaling job completions);
3. measuring of system call overhead;
4. creating, locking, and unlocking of real-time semaphores; and
5. support for synchronous task set releases.

Categories 1–3 are straightforward in their implementation and purpose; we briefly discuss the latter two categories.

Real-time semaphores. Blocking-by-suspending requires kernel support, as does maintaining and enforcing priority ceilings and enacting priority inheritance. Thus, each shared resource is modeled as an object in kernel space, which contains state information such as the associated priority ceiling, unsatisfied requests, *etc.* The exception is spin-based waiting, which is unknown to the kernel, since it is realized almost entirely in user space (with the exception of non-preemptive sections, see Section 3.3.2 below).

All tasks that share a given resource must obtain a reference to the same in-kernel object. Since LITMUS$^{\mathrm{RT}}$ is committed to not unnecessarily restricting the application design space, references must be (transparently) obtainable across process boundaries (*i.e.*, address space separation should not preclude resource sharing). For performance reasons, resource references must be resolved by the kernel with as little overhead as possible. Further, in a multiuser OS such as Linux, security concerns such as visibility of resources and access control must also be addressed—the resource namespace must be managed by the kernel.

Prior to LITMUS$^{\mathrm{RT}}$ 2007.3, the kernel simply allocated a pre-defined number of resources statically and let real-time programs refer to objects by their offset. While this interim method had low overheads, it was also completely insecure and brittle. Further, the lack of flexibility inherent in static allocation also quickly proved to be troublesome. In (Brandenburg and Anderson, 2008b), we introduced a new solution to manage resources in a secure, reliable, and efficient matter. Instead of introducing a new namespace (which would require appropriate access policies and semantics to be defined), we opted to reuse the filesystem to provide access control by attaching LITMUS$^{\mathrm{RT}}$

resources at run-time to *inodes* (an inode is the in-kernel representation of a file). When a task attempts to obtain a reference to a resource, it specifies a file descriptor to be used as the naming context. By specifying the same file, synchronization across process boundaries is possible (but only if allowed by the file's permissions). If permitted, the kernel locates the requested resource and stores its address in a lookup table in the PCB. Similar to the concept of the file descriptor table, the resource lookup table enables fast reference-to-address translation in the performance critical path of synchronization-related system calls. With the new method, LITMUS$^{\mathrm{RT}}$ resources are created dynamically on demand.

The applications in our experiments share memory across process boundaries via a shared, memory-mapped file (using the mmap() system call) and synchronize by allocating real-time semaphores in the context of the shared file's inode. That is, from the point of view of the programmer, the real-time semaphores are attached to file descriptors. In the LITMUS$^{\mathrm{RT}}$ source code and documentation, real-time semaphores allocated in this manner are hence called *file-descriptor-attached shared objects* (FDSOs).

Another benefit of the FDSO approach is that priority ceilings do not have to be determined offline and specified manually at runtime. Instead, ceilings can be computed automatically when tasks obtain references to resources. When a real-time semaphore is first allocated, its associated priority ceiling is initialized to the lowest-possible priority (or largest-possible relative deadline under EDF-based schedulers). When a real-time process opens an FDSO handle to the real-time semaphore, the priority ceiling is raised if the priority of the process exceeds the current priority ceiling (or when the deadline of its reservation is shorter than the current priority ceiling under EDF-based scheduling). In our experience, automatic determination of priority ceilings facilitates task system setup greatly and eliminates the possibility for human error.

Synchronous releases. A task set is said to have been *synchronously released* if the first job of each task is released at exactly the same time (which is, by definition, time 0). A synchronous release could be approximated using standard Linux system calls using either a *pthreads* barrier or the nanosleep() system call, but it is difficult to achieve high precision with this approach—job release times could be slightly shifted.

A second problem during task set setup is that no job should be released until all processes have finished their initialization phase. This requires reliably detecting that each process is ready for real-time execution to commence.

LITMUS$^{\mathrm{RT}}$ solves both problems with the addition of a synchronous release API. To participate in a synchronous release, each process invokes a system call named wait_for_ts_release() once it has finished its initialization phase and transitioned into real-time mode. This system call causes it to suspend until the synchronous release occurs. Crucially, the kernel exports the number of waiting real-time processes by means of a virtual file in the *proc* file system. Another, non-real-time process monitors the number of waiting processes, and, once all real-time processes have "checked in," it invokes a system call to cause a synchronous task set release. By using the real-time plugin's method release_at(), it is ensured that the release time of each job is exactly the same.

Virtual devices. Besides exporting simple status information (*e.g.*, number of real-time tasks, number of tasks ready for synchronous release) with virtual files in Linux's standard *proc* file system, LITMUS$^{\mathrm{RT}}$ also adds four types of special-purpose virtual character devices. Of these, three are virtual trace devices: one for debug data, called the *TRACE() device*; one for overhead samples, called the *Feather-Trace device*; and one for scheduling events, called the *sched_trace() device*. The fourth device implements the "control page" that is discussed in detail in Section 3.3.2 below.

The TRACE() macro, which is part of the LITMUS$^{\mathrm{RT}}$ core infrastructure, is equivalent to the printk() function in purpose, namely to aid kernel development by exporting debugging information. Unfortunately, it is not possible to call printk() while holding a runqueue lock because it might attempt to call into the scheduling code and acquire a runqueue lock, which could result in recursive deadlock; technical details can be found in (Brandenburg *et al.*, 2007). Given that large parts of LITMUS$^{\mathrm{RT}}$ execute while the processor holds a runqueue lock, this effectively "blinds" LITMUS$^{\mathrm{RT}}$. Therefore, we implemented the TRACE() macro to log debug messages during development. The TRACE() device simply exports the stream of concatenated messages logged with the TRACE() macro. The fundamental difference that makes TRACE() runqueue-lock-agnostic is that the TRACE() device is polled, whereas printk() may attempt to resume a consumer waiting for messages. The TRACE() macro produces unstructured data intended for human consumption. It also creates high overheads

since it induces copious amounts of string formatting into every scheduling decision and must thus be disabled for all performance-relevant experiments.

Feather-Trace is a low-overhead tracing toolkit that is based on static instrumentation of the kernel (Brandenburg and Anderson, 2007a). Feather-Trace provides two components: static *triggers*, which can be used to embed custom probes, and a wait-free multi-writer, single-reader FIFO buffer, which can be used for low-overhead data collection. The key features that make it attractive for use in LITMUS$^{\mathrm{RT}}$ are that it is very simple (*i.e.*, it is easy to embed) and efficient. Feather-Trace works by directly rewriting the kernel's code; the cost of an inactive trigger is merely an unconditional jump. When a trigger is activated, then the jump is rewritten to call a user-provided function instead. The FIFO buffer is very efficient because it is lock-free, retry-loop-free, and, for writers, also copy-free. That is, a writer (*i.e.*, the instrumented code) proceeds without ever having to wait, and without having to perform expensive copy operations. Implementation specifics are not relevant to this dissertation; the interested reader is referred to (Brandenburg and Anderson, 2007a).

In LITMUS$^{\mathrm{RT}}$, Feather-Trace is used to record timestamps at various points during the execution of the scheduler. For example, there is a trigger just before a context switch, and one just after it. When enabled, these triggers call a simple function that records the current time (based on the TSC; recall the overview of clocks on page 40), the processor on which it executes, whether the currently scheduled process is a real-time process, and a unique number identifying the probe. These timestamps are exported to user space by means of the Feather-Trace device. Based on the recorded pairs of timestamps before and after an event, it is later possible to reconstruct how long each event took. Based on these overhead samples, it is then possible to compute overhead statistics such as average and maximum observed overheads (see Chapter 4). It is important that a low-overhead tracing toolkit such as Feather-Trace is used for this since otherwise the recorded scheduling overheads would be disturbed by the tracing overheads.

The purpose of the sched_trace() infrastructure is to record all scheduling events (*e.g.*, job releases, preemptions, job completions, *etc.*). In contrast to the TRACE() macro, the data exported by the sched_trace() device is structured and not human-readable. Instead, this data can be plotted to obtain a visual depiction of the recorded schedule, and is also used for the automated finding of scheduling errors (Mollison *et al.*, 2009). The sched_trace() infrastructure is implemented using Feather-Trace primitives; nevertheless, it should not be confused with the Feather-Trace device.

When development of LITMUS$^{\mathrm{RT}}$ started, Linux did not offer any of the tracing functionality that we required. Since then, Linux has gained the *debugfs* file system and *ftrace* tracing infrastructure, which serve similar purposes. It future versions of LITMUS$^{\mathrm{RT}}$, it may be desirable to re-implement LITMUS$^{\mathrm{RT}}$'s tracing facilities on top of Linux's now-standard tracing and debugging frameworks.

3.3.1.4 User-Space Library and Tools

Real-time tasks are regular Linux processes in LITMUS$^{\mathrm{RT}}$. A user-space library, named *liblitmus*, contains the required system call stubs and several "convenience" functions and macros that make programming real-time tasks easier. Additionally, there are a number of test and helper utilities based on liblitmus for interacting with the LITMUS$^{\mathrm{RT}}$ kernel; one commonly used example is the release_ts tool that is used to trigger synchronous releases. Furthermore, the LITMUS$^{\mathrm{RT}}$ distribution includes several tools that interact with the trace devices and post-process the collected data. While crucial to our experiments, most of this infrastructure is rather simple in nature.

Two parts worth pointing out are a unit-testing-like framework to test LITMUS$^{\mathrm{RT}}$ system calls that is aware of the differences between global, clustered, and partitioned plugins, and the tool rt_launch, which allows arbitrary executables to be launched as LITMUS$^{\mathrm{RT}}$ real-time tasks. Together with budget enforcement, the latter allows periodic execution to be forced on any Linux process. One use for this feature is debugging: by executing filesystem-bound applications such as find as real-time processes, it is often possible to quickly trigger race conditions in the task_wake_up() and task_block() scheduler plugin methods.

This concludes our high-level overview of LITMUS$^{\mathrm{RT}}$. In the remainder of this section, we examine specific challenges and the solutions that were adopted in detail. We begin with how non-preemptive sections are implemented in LITMUS$^{\mathrm{RT}}$.

3.3.2 Low-Overhead Non-Preemptive Sections

In a multiprocessor OS such as LITMUS$^{\mathrm{RT}}$, the main use of non-preemptive sections is to support spin-based locking protocols such as the MSRP. When using spin-based locking protocols, critical sections are typically short; entry and exit overheads can thus easily comprise a majority of the request length. The primary requirement for non-preemptive sections is thus low overheads.

In an RTOS without address space separation (or while executing a system call in the kernel), a scheduled process can become non-preemptable simply by disabling interrupt delivery, which will prevent the scheduler from being invoked (unless called directly by the scheduled process). For example, this approach is used in the Linux kernel itself. Other RTOSs that allow real-time tasks to disable interrupt delivery are VxWorks and RTEMS, which allow (in the case of VxWorks) or even force (in the case of RTEMS) real-time tasks to execute in kernel mode. Since interrupt delivery can be disabled with a single instruction in most processor architectures, this approach satisfies the low-overhead requirement well.

However, in a UNIX-like OS such as Linux where real-time tasks are implemented as processes, simply disabling interrupts is neither possible in all cases nor advisable.[12] When interrupts are disabled, the kernel cannot recover control from a mis-behaving process. For example, an accidental infinite loop could render the processor, and potentially the whole system, unresponsive. In deployed embedded systems, such a lack of robustness may be acceptable since real-time tasks are trusted to be correct and an accidental infinite loop in a user-space program likely causes system failure anyway. However, system lock-ups are a nuisance during application development and a reason for great concern in open (soft) real-time systems, where the code of the real-time tasks is neither known in advance nor trusted to be correct.

Under FP scheduling, non-preemptive sections can be emulated by reserving the highest-possible priority to cause effectively non-preemptable execution: to enter a non-preemptive section, a process simply raises its priority to the highest priority. This is virtually identical to priority boosting as it causes the process to not be preempted by later-resuming, higher-base-priority processes. In the case of JLFP scheduling, non-preemptive sections can similarly be indicated either by raising a job's priority temporarily to the highest-possible job priority (*e.g.*, time 0 under EDF), or by setting a flag in the PCB that prevents the scheduler from being invoked. Combined with budget enforcement, either method ensures that unresponsive, non-preemptable real-time tasks can be policed by the kernel. For example, the Linux console driver contains functionality that allows a developer to terminate all SCHED_FIFO and SCHED_RR processes, thereby enabling the system to be recovered if rendered otherwise unresponsive by runaway processes.

[12]The right to disable interrupts can be granted on a process-by-process basis in Intel's x86 architecture. This, however, is a non-portable peculiarity that is not available on other architectures (such as ARMv6).

However, a major disadvantage of this approach is that each process must execute two system calls for every request: one to become non-preemptable, and one to re-enable preemptions. Worse, these system calls are required unconditionally, *i.e.*, even if the resource in question is not contended. Since system calls are typically much slower than regular procedure calls, this approach to implementing non-preemptive sections can easily dominate request lengths and thus renders spin-based locking protocols considerably less attractive. In LITMUS$^{\mathrm{RT}}$, we solved this problem by moving the flag indicating that a process is non-preemptive from kernel space to the process address space

Control page. This was accomplished by means of the *control page device*, which is a virtual character device with a custom driver. The only system call that the control page device supports is mmap(), *i.e.*, mapping the "device memory" into the address space of a process. When a process maps the device, the driver allocates a page of kernel memory—the *control page*—and maps it into the address space of the process. The control page enables the kernel and each real-time process to share flags and information without requiring system calls. Since this memory can be modified by (potentially malicious) processes, kernel developers must take great care to interpret the contents of the control page as untrusted hints. Since the control page of each process is allocated in kernel space, it is never paged out (*i.e.*, page faults are impossible) and can be accessed by every processor, and even when a process is not currently scheduled (*i.e.*, when its address space is not in use).

The control page in LITMUS$^{\mathrm{RT}}$ is inspired by Mac OS X's *commpage* facility, which is used to map multiple pages of trusted code and data into each process under Mac OS X. In contrast to LITMUS$^{\mathrm{RT}}$'s control page, the Mac OS X's commpage is write-protected, shared among all processes, and directly inserted at a known, fixed address by the virtual memory management subsystem (*i.e.*, it is not a virtual device that is mmap()'ed). Linux similarly uses shared, write-protected pages to speed-up frequently called system calls such as gettimeofday(). Using a virtual device is easier to maintain but cannot guarantee a fixed location. Mac OS X's commpage is directly accessed from assembly code via hardcoded addresses and must thus reside at a fixed location. LITMUS$^{\mathrm{RT}}$ has no such requirements; we therefore chose the virtual device approach.

Non-preemptive section protocol. As part of its initialization, a real-time process maps the control page into its address space. There are two flags allocated for the purpose of supporting non-preemptive sections. The first flag, simply called the np_flag, is an integer variable that is initially zero and

```
1  variables: // allocated in per-process control page
2      integer np_flag initially 0
3      boolean delayed_preemption initially ⊥

5  enter_np_section():
6      set np_flag ← np_flag + 1

8  exit_np_section():
9      set np_flag ← np_flag − 1
10     if np_flag = 0 and delayed_preemption:
11         sched_yield()
12         // kernel resets delayed_preemption
```

Listing 3.3: Non-preemptive section entry and exit procedure.

incremented whenever the process enters a non-preemptive section and decremented again at the end of the non-preemptive section. The scheduler checks this flag when making scheduling decisions and does not preempt the currently scheduled process while the np_flag is positive. If a preemption is required, the scheduler instead sets a second boolean flag named $delayed\_preemption$ flag to true. At the end of a non-preemptive section, the process checks this flags and executes the sched_yield() system call (if np_flag $= 0$) to notify the scheduler that is now safe to preempt the process. Since np_flag is a counter, it is possible to nest non-preemptive sections. This straightforward protocol, summarized in Listing 3.3, is akin to how non-preemptable execution is implemented inside the Linux kernel. Its biggest advantage is that it avoids all system calls in the common case when no preemption is required, and requires only one system call if a preemption is required—a fifty percent reduction of overhead in the worst case, and virtually no overhead in the average case.

Robustness. A disadvantage of the protocol as described so far is that it does not solve the robustness problem if the scheduler relies on the process to clear its np_flag and to call sched_yield() when requested to do so. This can be easily addressed by enforcing a maximum non-preemptive section length. When the scheduler first observes the np_flag flag being non-zero, it records the current time in the corresponding process' PCB (which is *not* accessible to the process). At subsequent quantum boundaries (or by means of a one-shot timer), the kernel checks how much time has passed since the process was first observed as being non-preemptive. When the maximum non-preemptive section length has been exceeded, the process is either marked as defective and ineligible for non-preemptable execution, or forcefully terminated (a "temporal fault" analogous to a segmentation fault). In the

expected case, the timeout will not be triggered; the enforcement of maximum non-preemptive section lengths is thus virtually overhead-free.

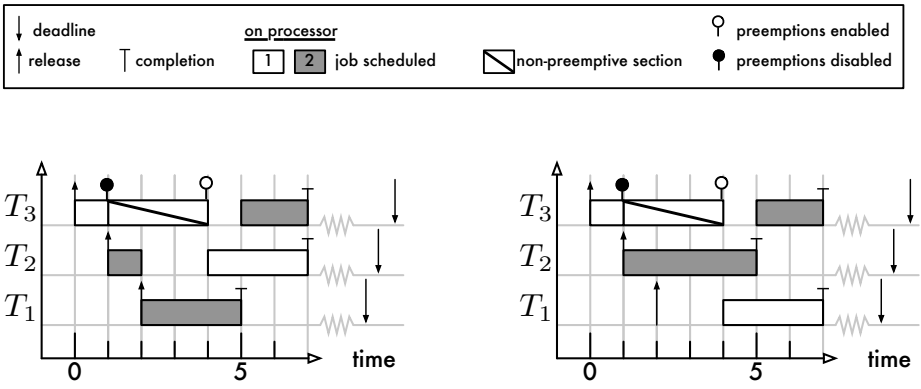
In a well-designed system, non-preemptive sections are short and should not exceed tens of microseconds; a timeout of one second thus seems reasonable to detect misbehaving non-preemptable processes. In the context of timesharing without temporal guarantees (*e.g.*, under CFS), a shorter timeout of, say, two milliseconds would cause non-preemptive section overruns to be (virtually) unnoticeable to users while still allowing for reasonable critical section lengths. We therefore believe the control-page approach to also be applicable in multiuser, throughput-oriented environments with untrusted processes and intend to investigate this in future work. Due to its nature as a research tool, the current LITMUS$^{\mathrm{RT}}$ implementation does not police non-preemptive section lengths.

Next, we discuss how non-preemptive sections can be supported in a predictable manner under global and clustered scheduling.

3.3.3 Non-Preemptive Sections with Predictable Pi-Blocking

As discussed in Section 2.4.3, non-preemptive sections can cause pi-blocking if a newly-released job cannot be scheduled because a lower-priority job is non-preemptive. Under uniprocessor JLFP scheduling, and hence also under partitioned JLFP scheduling, the maximum duration of pi-blocking is easy to bound: since (local) lower-priority jobs are not scheduled while a higher-priority job is ready, pi-blocking is only incurred at the time of release and is limited to the duration of one critical section (recall the analysis of the NCP on page 110). If jobs self-suspend due to accessing I/O devices or when participating in a suspension-based global locking protocol such as the MPCP, a job may also incur pi-blocking every time it resumes (as discussed on page 133). The key property that enables accurate analysis is that the blocked job controls the maximum number of times that it incurs pi-blocking. That is, a job J_i that suspends x times incurs pi-blocking at most $(x + 1)$ times, no matter how often lower-priority jobs execute non-preemptably.

Unfortunately, analysis of non-preemptive sections is not as straightforward under global scheduling (and hence also under clustered scheduling if $c > 1$; we focus on global scheduling in the following discussion). Bounding pi-blocking requires a careful analysis of when a job may be preempted, which is an interesting question if *some* but not *all* scheduled jobs are non-preemptable. Such a situation arises in the G-EDF schedule shown in Figure 3.4(a) at time 2 when J_1 is released.



(a) Eager preemption.

(b) Lazy preemption.

Figure 3.4: Illustration of eager and lazy preemptions. In this example, three jobs are scheduled on two processors under G-EDF. At time 2, J_1 should preempt J_3 but cannot since J_3 is non-preemptable during $[1, 4)$. **(a)** With eager preemptions, a preemption is enacted as soon as some lower-priority job is preemptable (J_2 in this case). **(b)** With lazy preemptions, a preemption is only enacted when the lowest-priority scheduled job (at the time of J_1's release) becomes preemptable.

Normally, J_1 would preempt J_3 and J_2 would continue to execute. However, J_3 is executing non-preemptably at time 2 and J_1's priority exceeds J_2's priority—should J_1 preempt J_2? The answer depends on whether preemptions are enacted "eagerly" or "lazily."

Eager preemptions. Consider the following naïve interpretation of event-driven global scheduling that allows jobs to have non-preemptable sections: at time t, if there are x non-preemptable ready jobs, then these jobs are scheduled at t. If there are further k additional preemptable jobs at t, then the $\min(k, m - x)$ highest-priority preemptable jobs are also scheduled at t. For example, this scheduling rule results if non-preemptive sections are emulated by temporarily raising a job's priority to the highest-possible priority. We refer to this preemption model as being *eager* because it enacts preemptions as soon as *some* lower-priority job is preemptable.

Example 3.3. The eager preemption rule is illustrated in Figure 3.4(a), where $m = 2$. J_1 is scheduled immediately upon its release because it preempts J_2, despite the fact that J_3 has a lower priority than J_2. J_2 is preempted because only the $\min(k, m - x)$ highest-priority preemptable jobs are scheduled at any time. At time 3, $k = 2$ and $x = 1$; therefore exactly one preemptable process is scheduled, and J_2's priority is only the second highest. $\Diamond$

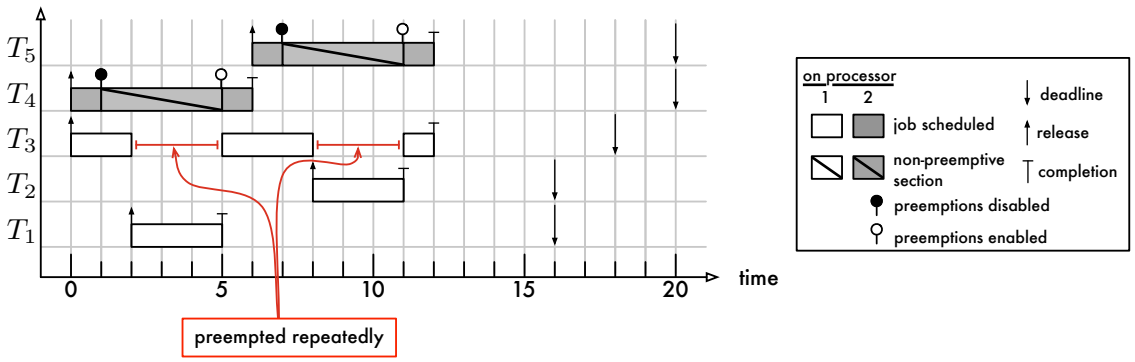


Figure 3.5: Illustration of repeated, eagerly enacted preemptions. Five jobs are scheduled under G-EDF on $m = 2$ processors. Job J_3 is preempted repeatedly even though it is among the two highest-priority ready jobs. This example demonstrates that jobs may be preempted repeatedly if preemptions are enacted eagerly.

Intuitively, it may seem appropriate to preempt J_2 immediately to allow the highest-priority job to commence execution without further delay. However, in the worst case, a newly-released job incurs pi-blocking anyway since all scheduled jobs may be non-preemptable; the *analytical* benefit of eagerly preempting J_2 is thus small. Worse, with eager preemptions, it is possible for a job J_i to incur pi-blocking whenever *other* jobs are released or resumed.

Example 3.4. Consider the G-EDF schedule of five jobs on $m = 2$ processors depicted in Figure 3.5. Even though J_3 is always among the two highest-priority runnable jobs (while it is pending) and should thus be scheduled, it is preempted whenever a higher-priority job arrives, because the lowest-priority scheduled jobs, J_4 and J_5, are non-preemptable. As a result, J_3 incurs pi-blocking *twice* even though it does not suspend. Further, J_3 incurs pi-blocking not immediately after its release (or when it resumes), but at arbitrary times during its execution. ◊

This example shows that, if preemptions are enacted eagerly under global (or clustered) scheduling, then the number of times that a job incurs pi-blocking cannot be bounded in terms of the number of times that it suspends. Rather, it could potentially incur pi-blocking *every time* that a higher-priority job is released. This greatly complicates deriving a bound on pi-blocking, and the resulting bound is more pessimistic by *at least* a factor of $(n - m)$ compared to the bound trivially achievable under

partitioned scheduling.[13] In other words, non-preemptive execution causes $O(1)$ pi-blocking under partitioned JLFP scheduling, but may cause $\Omega(n-m)$ pi-blocking for some tasks under eager global JLFP scheduling. This is clearly undesirable, and, as previously mentioned in Section 2.4.4.1, also does not satisfy the assumptions underlying Devi *et al.*'s analysis of non-preemptive spinlocks under G-EDF (Devi *et al.*, 2006).

Lazy preemptions. Such undesirable preemptions can be avoided by enacting preemptions *lazily* instead. That is, preemptions should be selectively delayed until the *lowest-priority* scheduled job becomes preemptable instead of preempting the *first available* lower-priority job. This can be realized by separating idealized job scheduling (where all jobs are preemptive) from actual process scheduling (where non-preemptivity is required) by means of *link-based scheduling*.

Link-based event-driven scheduling works as follows.

- A ready job is either *linked to a processor* or *unlinked*, independently of whether it is scheduled or not. Linking is a bijective mapping, *i.e.*, a job is linked to at most one processor and at most one job is linked to a processor at any time.
- At any time t, the m highest-priority ready jobs are linked to a processor.
- A processor schedules the job that is linked to it whenever possible, *i.e.*, unless the currently scheduled job is non-preemptable.
- If a job becomes unlinked while it is non-preemptable, then it is preempted as soon as it exits its non-preemptive section.

Intuitively, a link records where a job *should* be scheduled, *i.e.*, where it would have been scheduled if all jobs were preemptable at all times. Because the link is just an abstraction that can be established immediately (as opposed to an actual context switch), it is not hindered by non-preemptive sections. Further, an established link prevents a higher-priority job from being scheduled on another processor, and thus protects other lower-priority jobs from being preempted.

Example 3.5. The effect of link-based scheduling is illustrated in Figure 3.4(b). Here, the preemption necessitated by J_1's release is enacted lazily. Initially, J_3 and J_2 are linked to processors 1 and 2,

[13] Assuming $n > m$, under G-FP scheduling, a job of $T_{n-(m-1)}$ could be preempted and incur pi-blocking whenever a job of $T_1, \ldots, T_{n-m}$ is released while jobs of $T_{n-(m-2)}, \ldots, T_n$ are simultaneously non-preemptable. This is similarly possible under G-EDF if $T_{n-(m-1)}$ has one of the m longest relative deadlines.

respectively. When J_1 is released, it is linked to processor 1 and J_3 becomes unlinked (but remains scheduled). Since J_2 remains linked to processor 2, it is not preempted. J_3 is preempted and J_1 is scheduled when J_3 leaves its non-preemptive section at time 4. $\Diamond$

A generic link-based scheduler. A detailed pseudo-code specification of link-based scheduling is given in Listing 3.4. In addition to non-preemptive sections, the specification also takes suspending jobs and changes in effective priority into account.[14] There are three internal operations (lines 1–30) and four event handlers that are called on job state changes (lines 32–52). The event handlers are implemented in terms of the internal operations, which we discuss first.

The first internal operation is try_to_schedule() defined in lines 1–10. It simply enacts required preemptions as determined by the current linking. When invoked for a given processor, it checks whether the linked job J_l and the scheduled job J_s differ (lines 2–5). If a preemption is required and the scheduled job is preemptable (or if the processor was idle), then the linked job is scheduled (lines 4–9). Otherwise, there is either no need to preempt (if $J_l = J_s$) or the preemption must be deferred until J_s becomes preemptable.

The second internal operation is try_to_link() defined in lines 12–25. It links a given ready job J_i to a processor if possible. If there exists a processor that currently no job is linked to, then J_i can simply be linked to that processor (lines 13–16). Otherwise, the lowest-priority job that is currently linked, denoted J_l, may have to be displaced. If J_i's current priority exceeds that of J_l, then J_i is linked to J_l's processor and J_l becomes unlinked (lines 20–23). This corresponds to a preemption if all jobs were always preemptable. If J_l was indeed preemptable, then the preemption is enacted immediately (line 24).

The third internal operation, link_next(), defined in lines 27–30, selects the highest-priority currently unlinked job (if any) and links it to a processor (if possible). If all jobs are always preemptable, then this operation corresponds to a processor picking the next job off the ready queue and scheduling it.

The external interface of the link-based scheduler is comprised of four event handlers. These event handlers correspond to various changes in job states. From the point of view of enacting

[14] Link-based scheduling was first described in (Block *et al.*, 2007) in the context of G-EDF, where it was referred to as GSN-EDF ("G-EDF with support for **s**uspensions and **n**on-preemptive sections"). While the description given here closely matches the implementation of GSN-EDF in LITMUS$^{\mathrm{RT}}$, it is general in nature and can be applied to any priority-driven scheduling policy. The original specification of GSN-EDF did not take changes in priority into account.

```
1  try_to_schedule(proc_k):
2      let J_s ← job scheduled on proc_k or ⊥ if proc_k is idle
3      let J_l ← job linked to proc_k or ⊥ if none is linked
4      let can_schedule ← J_s = ⊥ or J_s is preemptable
5      if J_l ≠ J_s and can_schedule:
6          if J_s ≠ ⊥:
7              preempt J_s
8          if J_l ≠ ⊥:
9              schedule J_l on proc_k
10     else: // preempt lazily: nothing to do now

12 try_to_link(J_i):
13     let allocated ← number of linked jobs
14     if allocated < m:
15         let proc_f ← any unlinked processor
16         link J_i to proc_f
17     else:
18         let t ← current time
19         let J_k ← lowest-priority linked job at time t
20         if y(J_i, t) < y(J_k, t):
21             let proc_k ← processor that J_k is linked to
22             unlink J_k
23             link J_i to proc_k
24             try_to_schedule(proc_k)
25         else: // nothing to do

27 link_next():
28     let J_h ← highest-priority currently unlinked, ready job or ⊥ if none exists
29     if J_h ≠ ⊥:
30         try_to_link(J_h)

32 // Called when a job is released or resumed.
33 on_job_arrival(J_i):
34     try_to_link(J_i)

36 // Called when a job exits its non-preemptive section.
37 on_job_becomes_preemptable(J_i):
38     let proc_i ← processor that J_i is scheduled on
39     try_to_schedule(proc_i)

41 // Called when a job is completed or suspended.
42 on_job_departure(J_i):
43     deschedule J_i
44     if J_i is linked:
45         unlink J_i
46     link_next() // J_i is not ready and will not be linked again.

48 // Support for priority inheritance, priority boosting, and JLDP scheduling.
49 on_effective_priority_change(J_i):
50     if J_i is linked:
51         unlink J_i
52     link_next() // J_i may be linked again.
```

Listing 3.4: Pseudo-code definition of link-based global event-driven scheduling.

required preemptions, a job being resumed or suspended is equivalent to a job being released or completed—it either becomes available for scheduling, or ceases to be available. Therefore these operations are combined as on_job_arrival() and on_job_departure(), respectively (lines 33–34 and 37–40 in Listing 3.4, respectively).

The simplest event handler is on_job_arrival(). When a job becomes available for scheduling, it must be checked whether it can be linked to a processor, which may cause a preemption to take place. This check is encapsulated in the try_to_link() operation.

The event handler on_job_departure() is called when a previously scheduled job became unavailable for scheduling. If the departing job was previously linked, then this provides a chance for a lower-priority job to become linked (lines 44–46). The required linking update (if any) is handled by the link_next() operation.

The defining element of the algorithm, namely lazy preemptions, are triggered by the event handler on_job_becomes_preemptable(), which is defined in line 37–39. At this point, no link updates are required. Rather, it may be required to enact prior link changes if a necessary preemption was deferred. This case is detected and handled by the try_to_schedule() operation.

Finally, there is an event handler for the case that the effective priority of a job changed (lines 49–52). This is required to support priority inheritance, priority boosting, and JLDP scheduling. When the priority of a job is raised, it may become eligible to be linked. If the priority of a job is lowered, then it may have to be unlinked in favor of another job that now has a higher priority. For the sake of simplicity, both cases are reduced to an unlink-then-relink sequence (lines 50–52). In the case that the job was linked and may remain linked, it will be briefly unlinked and immediately linked again. Assuming that this operation is carried out atomically, *i.e.*, while holding the ready queue lock (see Section 3.3.5 below), this is not reflected in the actual schedule.

The key property of the algorithm given in Listing 3.4 is that try_to_link() is called on every job state change, which ensures that the m highest-priority ready jobs are linked at any time.

Example 3.6. As an example, consider the two-processor system in Figure 3.6, which depicts the same system as in Figure 3.5 assuming link-based G-EDF scheduling. When job J_1 is released at time 2, it is linked to processor 2 since the previously linked job, J_4, had the lowest priority of any linked job (line 19 in Listing 3.4). However, since J_4 is non-preemptable at time 2 (line 4), J_1

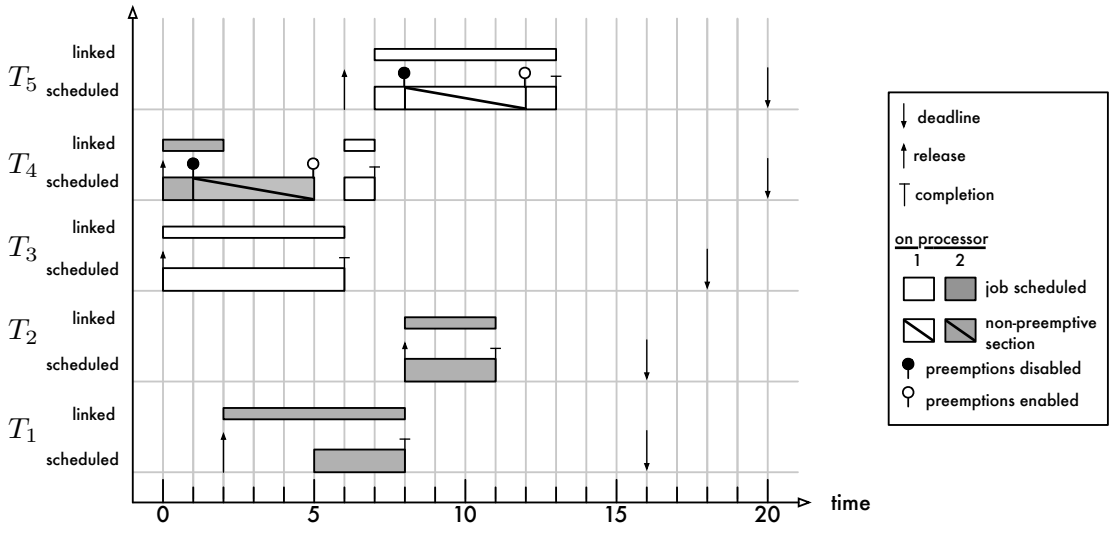


Figure 3.6: Example schedule demonstrating the benefit of link-based scheduling. The linking is illustrated as a narrow rectangle above each job. The scenario is the same as the one depicted in Figure 3.5. Under link-based scheduling, J_1 is linked to processor 2 when it is released at time 2. It therefore does not preempt J_3. Instead, J_4 is preempted when it becomes preemptable at time 5 since it became unlinked during its non-preemptive section.

is not scheduled until J_4 becomes preemptable (lines 37–39 and 1–10), at time 5. Notice that, in Figure 3.6, J_1 is delayed by non-preemptive execution immediately when it is released, and that the delay duration is upper-bounded by the maximum non-preemptive section length. ◊

Although it is not depicted in Figure 3.6, a job may be similarly delayed when it is resumed or when its effective priority is raised, *i.e.*, every time it may become eligible for execution due to a change to its state. Notably, in the absence of suspensions and priority changes, each job is pi-blocked due to a non-preemptive section at most once, which matches Devi *et al.*'s analysis of non-preemptive spinlocks under G-EDF scheduling (Devi *et al.*, 2006). That is, when preemptions are enacted lazily by means of link-based scheduling, non-preemptive sections cause at most $O(1)$ pi-blocking under global scheduling, just like under partitioned scheduling.

3.3.4 Safe Process Migrations

Link-based scheduling as just described is defined in terms of *jobs*, which are assumed to migrate if they become linked to a processor where they were not previously scheduled. However, in LITMUS$^{\mathrm{RT}}$, *process* migrations are required, which poses considerable challenges since **(i)** a process

is always assigned to some runqueue in Linux to serialize accesses to its PCB, and **(ii)** processes cannot be migrated instantly. We elaborate on each problem and describe our solution to it next.

Runqueue invariant. With regard to (i), recall that Linux uses runqueue locks to protect PCBs against concurrent updates: before a processor may change the state of a process, it must first acquire the lock of the runqueue that the process is currently assigned to. Linux further makes the assumption that a process that is currently executing on a processor is assigned to its runqueue. It is crucial that this invariant is maintained, because otherwise the process' PCB state might become corrupt.

However, in LITMUS$^{\mathrm{RT}}$, global plugins do not keep processes in per-processor runqueues; rather, a job may become linked to any processor, which results in the underlying process being selected for execution by the plugin's schedule() method. In other words, a global or clustered plugin may violate the invariant that a process executing on processor x is assigned to runqueue RQ_x.

The LITMUS$^{\mathrm{RT}}$ scheduling class contains a common code path that allows global (and clustered) plugins to be ignorant of Linux's invariant. After the active scheduler plugin has selected a process to be scheduled next, *i.e.*, after the schedule() method has returned, the LITMUS$^{\mathrm{RT}}$ scheduling class' pick_next_task() implementation checks whether the selected process is currently assigned to another, remote runqueue. If so, then a process migration is required: the selected process must be transferred from the remote runqueue to the local runqueue. This is similar in concept to Linux's pull operation.

Transferring the process, however, is unfortunately not straightforward. Recall that runqueue locks must be acquired in order of increasing lock addresses. It is possible for a process to be migrated from a higher-address runqueue to a lower-address runqueue. Linux's locking rule requires the local runqueue's lock to be dropped first before both locks can be acquired in the correct order. This creates great complications—when the current runqueue lock is dropped, the state of all processes may change. As a result, after the lock has been reacquired, no assumptions can be made about the current state of the to-be-migrated process. The LITMUS$^{\mathrm{RT}}$ scheduling class therefore contains code of considerable conceptual complexity to re-validate assumptions about this process.

The advantage of including this code in the joint LITMUS$^{\mathrm{RT}}$ scheduling class is that the implementation of global plugins is greatly simplified: from a plugin developer's point of view, any process can be assigned to any processor at any time.

Stack-safe migrations. With regard to (ii) above, consider that the Linux kernel requires a stack to execute on, *i.e.*, the processor's stack pointer register must contain a valid address while executing in kernel mode. Further, the stack contents are part of a process' processor state, *i.e.*, if a process is migrated from processor 1 to processor 2 while executing a system call inside the kernel, then processor 2 must continue using the *same* stack that processor 1 used prior to the migration since it contains the procedure arguments and return addresses that define the control flow of the process.

For these reasons, each process has a private kernel stack in Linux that the kernel uses while the process is scheduled. A context switch occurs when Linux changes kernel stacks—a process' kernel stack *is* the context that is being switched. Obviously, a kernel stack can only be in use on at most one processor at any time; otherwise, the sequential process would be executed on multiple processors at the same time, with unpredictable consequences.

Under global link-based scheduling, this represents a complication that the scheduler must be aware of. When a job becomes unlinked because it is going to be preempted by a higher-priority job, it is immediately available for linking to other processors. For example, suppose a job is unlinked due to line 22 in Listing 3.4. Then it could be immediately thereafter become relinked to another processor due to link_next(), *i.e.*, it could be selected by line 28 of Listing 3.4. On the level of jobs, this is essential to the correctness of link-based scheduling: once a ready job is no longer linked, it must be available for scheduling elsewhere, lest it be "overlooked" and incur a priority inversion. However, on the level of processes, the re-linking can race with the upcoming context switch.

Example 3.7. This is illustrated in Figure 3.7. Here, job J_3 is going to be preempted due to the release of J_1 at time 5. However, processor 1 incurs extreme overhead (indicated by the horizontal bars) between the time that it unlinked J_3 and the time that it could switch away from the kernel stack of the process implementing J_3. Concurrently, J_2 completes execution on processor 2. Processor 2 thus executes on_job_departure(), which in turn calls link_next()—which selects J_3 and links it to processor 2. At this point, processor 2 executes a context switch and starts using the kernel stack of the process implementing J_3 before processor 1 switched away from it. The result is a certain system crash. While the overhead in this example has been exaggerated for illustration purposes, such races do happen in practice (*e.g.*, process 1 might be servicing a long-running ISR). ◊

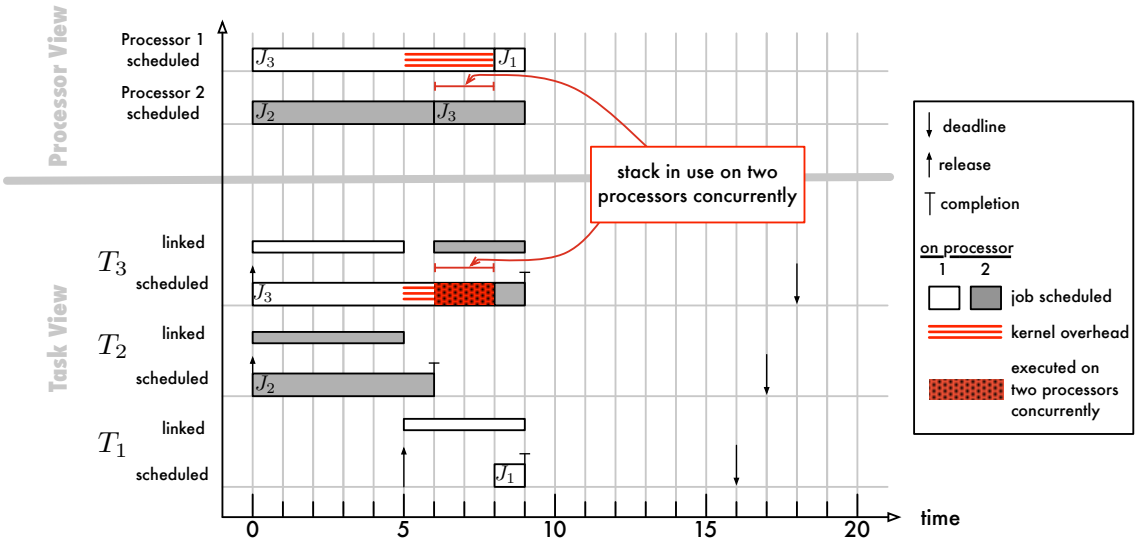


Figure 3.7: Example schedule demonstrating stack corruption. The figure depicts two views of the same schedule: the lower half shows a task-oriented view of the schedule, whereas the upper half shows a processor-oriented view of the schedule. As before, the linking is illustrated as a narrow rectangle above each job in the task-oriented view. Job J_3 is unlinked from processor 1 at time 5 and soon thereafter linked again to processor 2 at time 6. However, processor 1 does not finish the context switch from J_3 to J_1 until time 8. This results in job J_3's stack being used by two processors.

We briefly digress to note that diagnosing such stack corruption poses an interesting challenge. Once two processors execute on the same stack, it is virtually guaranteed that all debug diagnostics will be corrupt and misleading. In our experience, there are two common outcomes: either a processor triggers a page fault or illegal instruction exception on a return instruction (because the other processor concurrently overwrote the return address on the stack), or the system hangs due to a deadlock or locking invariant violation (since pointers to locks on the stack were corrupted). A reliable indicator can be found in the processor registers (which can be obtained from the virtual machine monitor when using a virtualized test environment such as Linux's *KVM*). If the stack register of two processors contains addresses less than one page apart, then the stack has been corrupted.

In LITMUS$^{\mathrm{RT}}$, we solved the kernel stack problem by adding an additional field to the PCB, named stack_in_use, that records on which processor a kernel stack is currently in use. The common migration path in the LITMUS$^{\mathrm{RT}}$ scheduling class busy-waits until the variable indicates that the stack is no longer in use. This realizes a kind of "migration barrier" that ensures that the context

switch on the target processor occurs only after the source processor has carried out its context switch, thereby releasing the process stack. Stack corruption is hence avoided.

3.3.5 Ready Queue and Release Queue

No matter the implemented scheduling policy, each scheduler plugin requires a mechanism to order ready jobs—a *ready queue*. Further, plugins require a mechanism to queue jobs for future time-based releases—a *release queue*. When a job is released, it must be transferred from the release queue to the corresponding ready queue, and the scheduler needs to be invoked if a preemption (or linking) is required. In LITMUS$^{\mathrm{RT}}$, to reduce code duplication, these three concerns are abstracted by a reusable component called a *real-time domain* (rt-domain).

There are many ways that these queues could be implemented. A ready queue could be implemented as a linked list, some kind of heap, a tree (as in Linux's CFS implementation), or any other realization of the priority queue abstract data type (such as Linux's bitfield-based ready queue for FP scheduling). In a quantum-driven implementation, the release queue can be implemented as a priority queue or timer wheel (Varghese and Lauck, 1987) that is polled by the scheduler_tick() method to transfer all jobs with release times in the preceding quantum to the ready queue. Alternatively, hardware timers can be programmed to trigger future job releases with interrupts if sufficiently high-resolution hardware timers are available. In event-driven plugins, using timers is preferable because it avoids increasing scheduling latency by up to a quantum (page 87). Since all platforms that LITMUS$^{\mathrm{RT}}$ has been used on support high-resolution timers, the release queue implementation that is part of the rt-domain abstraction uses Linux's *hrtimers* subsystem.

The implementation in LITMUS$^{\mathrm{RT}}$ is guided by the goal to reduce worst-case overhead when multiple job releases occur at the same time. Thus, instead of programming a timer for every job, our release queue implementation uses a timer per release time to avoid unnecessary overhead, *i.e.*, if multiple job releases coincide (*e.g.*, on a hyperperiod boundary), then only one timer interrupt is required. Efficient timer sharing is accomplished by looking up future release times in a hash table.

Additionally, timer sharing enables the use of *mergeable queues*. A mergeable queue supports adding multiple elements efficiently, *i.e.*, it is possible to insert k elements into the queue without invoking the *insert* operation k times. For example, if the ready queue were implemented as a simple binary heap, then releasing k jobs would take $O(k \log n)$ time (recall that n denotes the number of

tasks). In contrast, if the ready queue is implemented as a binomial heap (Vuillemin, 1978), which it is in LITMUS$^{\mathrm{RT}}$, then k jobs can be *merged* into the ready queue in $O(\log n)$ time—a potentially large reduction in overheads if many jobs are released at the same time. This, however, requires the k jobs to be organized in a binomial heap of size k prior to the release. As there is only one timer for each release time (in each rt-domain), this is achieved by simply associating a binomial heap with each timer, which is called the *release heap* for the timer's expiration time. When a job J_i is to be released at time t, the rt-domain implementation checks to see if a release heap already exists for time t. If there exists a release heap for time t, then J_i is simply enqueued; otherwise, a new release heap for time t is initialized and J_i becomes the initial element. The use of binomial heaps to efficiently merge multiple jobs into ready queues was proposed by Baruah *et al*. (1995) in the design of the PD scheduling algorithm. Besides LITMUS$^{\mathrm{RT}}$, we are not aware of other OSs that have adopted their approach.

Each rt-domain is protected against concurrent updates by two spinlocks, one for the ready queue and one for the release queue. Most scheduler plugins use the ready queue lock to serialize all scheduling decisions (see below). Internally, a third spinlock is used to serialize changes to the release queue with the timer ISR, but this is an implementation detail that is not exposed to the scheduler plugins. The rt-domain abstraction implements four main operations:

- add_release(), which adds a job to a release heap;
- add_ready(), which adds a job to the ready queue;
- take_ready(), which removes the highest-priority job from the ready queue (the priority comparison function is configurable); and
- peek_ready(), which returns the highest-priority job in the ready queue without removing it.

The semantics of the take_ready() operation implies that currently scheduled jobs are not enqueued in the ready queue. The peek_ready() operation is required to check whether a preemption is required—the highest-priority queued job must be compared to the lowest-priority scheduled job. If a preemption is required, then the peek_ready() operation is followed by a take_ready() operation; if no preemption is required at the time, then the peek_ready() operation will be called again. Finding the minimum element in a binomial heap requires $O(\log n)$ time. To avoid additional overhead,

each rt-domain caches the current minimum element, so that subsequent calls to peek_ready() and take_ready() require only $O(1)$ time after the first call to peek_ready() (until the ready queue is modified).

In a scheduler plugin implementing a global policy, only one shared rt-domain is required. In a partitioned (or clustered) scheduler plugin, each processor (or cluster) has a private rt-domain. This separation is required so that jobs are merged into the appropriate processor-local (or cluster-local) ready queue. That is, rt-domains cannot be shared among multiple instantiations of priority-driven scheduling; rather, clusters are defined by the shared rt-domain—hence the name "domain."

3.3.6 Scheduler Plugin Implementation

The rt-domain abstraction incorporates the core operation of priority-driven scheduling. Most real-time plugins are thus structured around one or several rt-domain instances and, from a high-level view, operate very similarly. We briefly sketch the operation of a "generic" plugin next and then discuss details of the actual plugins thereafter.

Generic plugin. Since each scheduling event (job release, job completion, job resumes, job suspends, beginning of a quantum) requires some job to be either added to or removed from the ready queue, or at least "peeking" at the ready queue, the ready queue lock is typically reused to serialize all scheduling decisions. Ready jobs not currently scheduled are stored in the ready queue. The currently scheduled (or linked) jobs are not queued. Jobs (*i.e.*, the PCBs of the implementing real-time processes) are added to the ready queue in one of three ways:

1. by the rt-domain when released by a timer ISR;
2. by the task_wake_up() plugin method when released by a device ISR or when resuming after a suspension; and
3. by the schedule() plugin method when preempted, or, under link-based global scheduling, by the try_to_link() operation when unlinked.

A job is dequeued from the ready queue once it is the highest-priority waiting job either by the schedule() plugin method or, under link-based scheduling, by the link_next() operation.

Each of the following plugins also supports dedicated interrupt handling. Recall from Section 2.5.1.4 that under dedicated interrupt handling one of the processors is a dedicated systems processor, which does not schedule real-time jobs and is used to handle all interrupts instead. Under partitioned scheduling, this is simple to realize (do not assign any tasks to the systems processors), whereas the systems processor must be explicitly marked as such under clustered and global scheduling. In LITMUS$^{\mathrm{RT}}$, this is accomplished by discarding the systems processor from the processor mapping, which ensures that no job will be linked to it.

3.3.6.1 The P-FP Plugin

The P-FP plugin is the exception to the generic implementation: it does not use the rt-domain's binomial heap as a ready queue. Instead, it follows Linux's bitfield-based approach to enable fast lookup of ready processes. Unlike Linux, LITMUS$^{\mathrm{RT}}$ supports 512 distinct priorities since we consider task sets consisting of several hundred tasks in Chapter 4. There is a tradeoff in the choice of ready queue. While using a bitfield to quickly identify the highest-priority ready job speeds up the task_ready() and peek_ready() operations considerably, it has the downside that jobs cannot be merged into the ready queue. Instead, each job must be added individually, which can result in higher overheads if many jobs are released at the same time. The P-FP plugin still uses a rt-domain instance on each processor for time-based releases, and uses each rt-domain's ready queue lock to serialize scheduling decisions (*i.e.*, similarly to how Linux uses runqueue locks).

Perhaps surprisingly, the LITMUS$^{\mathrm{RT}}$ P-FP plugin is somewhat simpler than Linux's FP implementation. As mentioned previously in Section 3.3.1.1, LITMUS$^{\mathrm{RT}}$ does not use processor affinity masks. Therefore, the P-FP plugin does not require the pull- and push operations that are invoked by Linux's FP implementation on every scheduling decision. Instead, a real-time process, once admitted, is not allowed to migrate to another partition. This restriction greatly simplifies the implementation of LITMUS$^{\mathrm{RT}}$'s P-FP plugin.

The P-FP plugin does not use link-based scheduling. Rather, it only keeps track of the currently scheduled process and does not have a separate linking step. This is sufficient since non-preemptable jobs executing on remote processors do not affect local scheduling decisions under partitioning; repeated preemptions are thus not possible.

3.3.6.2 The P-EDF Plugin

The P-EDF is the simplest plugin in LITMUS$^{\mathrm{RT}}$ and functions essentially like the "generic" plugin described above. There is one rt-domain instance for each processor, and the ready queue lock is used to serialize all scheduling decisions. Unlike the P-FP plugin, the P-EDF plugin uses the default binomial heap for its ready queue. Since the P-EDF plugin implements strictly partitioned scheduling, a process may not migrate among partitions after it has been admitted and link-based scheduling is not required.

Due to its simplicity, the P-EDF plugin has in the past served as the first test case when rebasing or porting LITMUS$^{\mathrm{RT}}$: when the P-EDF plugin crashes, the core infrastructure's Linux integration code and the LITMUS$^{\mathrm{RT}}$ scheduling class are likely incorrect.

3.3.6.3 The G-EDF Plugin

The G-EDF plugin implements link-based scheduling because LITMUS$^{\mathrm{RT}}$ supports non-preemptive spinlocks, and thus predictable non-preemptive sections must be supported.

Job scheduling. At the heart of our implementation of link-based scheduling is a data structure that we call the *processor mapping*. The processor mapping stores for each processor which *job* is currently linked to it, and which *process* is in fact scheduled. That is, the processor mapping realizes the separation of job scheduling from process scheduling. The difference is that job scheduling—updating links—can be carried out *on* any processor *for* any processor. In contrast, process scheduling can only be enacted locally: for example, if a timer ISR that is executed on processor 1 triggers the release of a job J_i, and if J_i is linked to processor 2, then processor 1 cannot enact a context switch on processor 2. Instead, processor 1 must send an IPI to processor 2, which then can detect the change in the linked job and initiate a context switch to the process that realizes J_i (*i.e.*, to the process that is attached to the reservation).

The separation of job scheduling from process scheduling makes link-based scheduling possible and greatly simplifies the implementation of global schedulers—since link changes can be enacted immediately, the current job-to-processor mapping is never in an inconsistent state (if link changes are serialized by means of a lock). This means that the scheduling logic matches the conceptual

notion of global, priority-driven scheduling much more closely as it does not have to take hybrid states into account (in contrast to Linux's push- and pull-based scheduling logic).

Since the most common operation involving the processor mapping is to check whether a preemption is required, which requires identifying the lowest-priority linked job (line 19 in Listing 3.4), the processor mapping is realized as a min-heap with processors ordered by the priority of their assigned jobs (if any—idle processors have the lowest priority).

Locking. A key question in the design of any global scheduler is how locking is realized. An important goal is to avoid lock contention as much as possible. Per-processor locks as used by Linux are preferable from this point of view. However, as demonstrated in Section 3.2.4, the lack of a consistent snapshot of the current scheduling state—the current job-to-processor mapping and the contents of the ready queue—can lead to "migration glitches" and greatly complicates the implementation of the scheduler. Further, link-based scheduling as described in Section 3.3.3 above requires atomicity of link updates, otherwise repeated preemptions cannot be ruled out. It is not obvious at all how link-based scheduling could be implemented with per-processor locks. Therefore, from a correctness point of view, the processor mapping must be protected by a global spinlock that serializes all changes, thus effectively serializing the execution of the link-based scheduler.

Global locking, of course, comes at the expense of increased contention. To the best of our knowledge, no other approach has been proposed to date that provably ensures the predictability of migrations and preemptions, *i.e.*, that is guaranteed to be "glitch free." It is an interesting opportunity for future work to reconcile the two conflicting needs at the heart of global scheduling—how can processors compute globally correct, atomic scheduling decisions based on mostly local or possibly inconsistent state?

A separate, but closely related question is how the ready queue should be implemented. In particular, is it acceptable to use a single, shared rt-domain (*i.e.*, a binomial heap with a coarse-grained lock), or would it be preferable to use some other queue implementation that allows some degree of parallelism? In the process of developing LITMUS$^{\mathrm{RT}}$, we investigated this question carefully (Brandenburg and Anderson, 2009a). In particular, we compared using a binomial heap with coarse-grained locking with two alternate ready queue implementations, one using a parallel heap based on fine-grained locking, and the other using a simple hierarchical queue-of-queues design,

where each processor has a local queue that it uses to enqueue jobs. We carefully evaluated versions of the G-EDF plugin based on each ready queue and found (to our surprise) that neither alternative resulted in a significant improvement over using a binomial heap. While allowing a higher degree of parallelism, the benefit of (partially) parallel ready queue updates is dwarfed by the contention for the processor mapping lock. This is because each access to the ready queue is typically accompanied by a scheduling decision. Parallelizing the ready queue does not help with the (so far unavoidable) sequential nature of link-based scheduling. Further, both fine-grained locking and the hierarchical queue introduced additional overheads due to an increased number of lock acquisitions. We refer the interested reader to (Brandenburg and Anderson, 2009a) for a detailed discussion of these results.

For these reasons, the G-EDF plugin underlying this dissertation uses a coarse-grained, global lock to protect the processor mapping and the ready queue and to serialize all scheduling decisions.

3.3.6.4 The C-EDF Plugin

The C-EDF plugin is a straightforward hybrid of the P-EDF and G-EDF plugins. As such, it implements link-based scheduling (for the case of $c > 1$) on a per-cluster basis. The processors in each cluster share an rt-domain instance and a processor mapping, but typically do not access the state of other clusters. This reduces contention of the ready queue lock considerably. Based on how clusters are configured, C-EDF further reduces overheads due to frequent migrations of cache lines that contain parts of the ready queue and processor mapping between processors that do not share a cache, *i.e.*, clusters that match the underlying processor topology reduce the intensity of cache-consistency traffic caused by each scheduler invocation.

The cluster size c is determined when the plugin is activated based on which level of cache sharing is desired. The user writes the desired cache level (*e.g.*, L1, L2, L3, *etc.*) to a virtual file in the *proc* file system. When the C-EDF's activate_plugin() method is called, it dynamically determines which processors form a cluster based on the processor topology, which the Linux kernel discovers during system startup.

One may wonder: if LITMUS$^{\mathrm{RT}}$ has a C-EDF scheduler plugin, why does it also contain dedicated P-EDF and G-EDF plugins? There are three reasons for this. Historically, the P-EDF and G-EDF plugins came first. Second, the P-EDF plugin is much simpler than the C-EDF plugin since it does not have to deal with the case of $c > 1$, which potentially results in lower overheads.

In particular, the P-EDF plugin neither requires nor uses link-based scheduling. And finally, as we discuss in Chapter 6, global and clustered scheduling (if $c < m$) differ in how locking protocols may be designed.

3.3.6.5 The PD^2 Plugin

Superficially, the PD^2 plugin resembles the C-EDF plugin. It supports configurable clustered scheduling based on the system's underlying cache topology and, in each cluster, it uses a shared rt-domain to implement the ready and release queues, uses a processor mapping to keep track of linked jobs, and serializes all scheduling decisions with the global ready queue lock. However, many differences become apparent when considering implementation details. For one, tasks carry extra state under PD^2. When a process is admitted, the PD^2 pre-computes all subtask parameters (*i.e.*, subtask deadline, successor bit, and group deadline) and checks that task parameters are a multiple of the quantum size. A bigger difference is how the scheduler is invoked.

Quantum update. PD^2 is an inherently quantum-driven scheduling policy, which implies that jobs are only linked and unlinked from within the scheduler_tick() method, *i.e.*, other scheduling events such as the arrival of jobs do not cause re-linking to occur. However, before consistent linking decisions can be made at the beginning of a new quantum, *all* jobs that were linked in the preceding quantum must advance to their next subtask, *i.e.*, their priority must be updated. This means that a processor cannot just update its locally linked job when it enters the scheduler_tick() method; rather, the first processor to execute the scheduler_tick() method for a given quantum updates the priority of all currently linked jobs, and then proceeds to check all currently existing links. While the first processor computes the job assignments for the next quantum, the other processors await their assignment. That is, the scheduler_tick() realizes a synchronization barrier that processors can only cross once the schedule for the next quantum has been computed. While this approach may seem overly serialized, such centralized scheduling is in fact preferable because it avoids cache-line bouncing and inconsistent scheduling decisions: if processors were to schedule before all job priorities have been updated, then inconsistent decisions could be made; if processors were to just locally schedule after all priorities have been updated, then the cache-lines holding the processor

mapping and ready queue would have to "visit" each processor at each quantum boundary. It is thus more efficient to just compute the schedule for the next quantum on one processor.

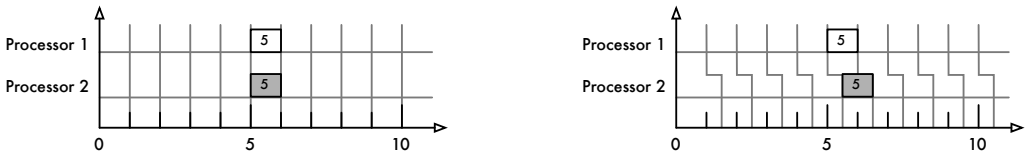
Quantum size. Recall from the review of pfair scheduling (Section 2.3.3.3) that it is only optimal (for implicit-deadline tasks) if the quantum size can be made sufficiently small. This, of course, is not possible in practice, where each quantum boundary creates overhead due to the need to service a timer interrupt, reschedule, and perform context switches.

Linux has a periodic *scheduler tick* that traditionally was used for timekeeping and scheduling purposes, and, prior to the introduction of *hrtimers*, also to process expired software timers. In modern Linux, with the introduction of accurate one-shot timers, the reliance on the scheduler tick has decreased considerably (in fact, Linux can now be configured to disable the scheduler tick while a processor is idle). Nonetheless, it still exists and LITMUS$^{\mathrm{RT}}$ plugins can use it to realize quantum-based scheduling by means of the accordingly named scheduler_tick() method.

Linux's scheduler tick occurs with a frequency that is configured at compile time. The highest supported frequency is 1000 Hz, which is equivalent to a quantum size of 1 ms. While lower frequencies are possible (*e.g.*, 100 Hz, 250 Hz), the resulting quantum sizes (*e.g.*, 10 ms, 4 ms) are too large for most real-time purposes. It would be possible to program a custom periodic timer with a very short period (say, 100 μs); however, the *effective quantum size*, *i.e.*, the time actually available for useful computation in each quantum would be unusably small in this case. Therefore, the quantum size under PD^2 is one millisecond in LITMUS$^{\mathrm{RT}}$. We discuss how to account for the resulting capacity loss in Section 3.5 below.

Quantum alignment. The analysis of PD^2 as discussed in Section 2.3.3.3 assumes that quanta are *aligned*, *i.e.*, that all processors cross a quantum boundary at the same time (with regard to physical time). From a hardware efficiency point of view, this is undesirable, since then all processors are going to execute a context switch and incur compulsory cache misses at roughly the same time, which creates peak memory bus contention. Similarly, lock contention is increased when all processors execute the scheduler_tick() method in parallel.

To overcome this contention, Holman and Anderson (2005) introduced *quantum staggering*, which is illustrated in Figure 3.8. Under staggered quantum-based scheduling, logical time is offset from processor to processor such that quantum boundary crossings are spaced out throughout



(a) Aligned quanta. (b) Staggered quanta.

Figure 3.8: Illustration of aligned and staggered quanta. The grey vertical lines indicate quantum boundaries. The fifth quantum is shown on each processor. **(a)** With aligned quanta, each processor crosses a quantum boundary at the same physical time. **(b)** With staggered quanta, quantum boundaries are shifted by $\frac{1}{m}$ time units from processor to processor.

one quantum. In other words, the beginning and end of each quantum on processor x, where $x \in \{1, \ldots, m\}$, is shifted by $\frac{x-1}{m} \cdot Q$ time units, where Q is the quantum size. For example, assuming $m = 24$ and $Q = 1000\mu s$, then a processor crosses a quantum boundary approximately every $41.67\mu s$. This helps to reduce contention since processors are unlikely to access scheduler state at the same time if quanta are staggered. In LITMUS$^{\mathrm{RT}}$ in particular, staggering masks the implicit barrier since the first processor to reach a quantum boundary (*i.e.*, processor $x = 1$, which is not shifted) has likely finished computing the schedule for the next quantum before the other processors reach their shifted quantum boundaries. However, staggering may also delay a job's completion by up to $\frac{m-1}{m} \cdot Q$ time units (if the job's last subtask is assigned to the most-shifted processor), which must be accounted for during schedulability analysis (see Section 3.5 below).

LITMUS$^{\mathrm{RT}}$ supports both aligned and staggered quanta as a boot-time option. The processor-local timer generating the scheduler tick on each processor is programmed during system startup. By default, Linux staggers quanta throughout half of a quantum. LITMUS$^{\mathrm{RT}}$ changes the calculation of the offset to realize aligned or staggered quanta, as specified by a command line option that is passed to the kernel by the boot loader.

This concludes our overview of the design and implementation of LITMUS$^{\mathrm{RT}}$. Next, we discuss how to account for runtime overheads as they arise in LITMUS$^{\mathrm{RT}}$. Exactly which overheads occur depends on whether the system uses partitioned scheduling and on whether the scheduler is event- or quantum-driven. We first consider event-driven scheduling in Section 3.4, and discuss quantum-driven schedulers thereafter in Section 3.5.

3.4 Overhead Accounting under Event-Driven Schedulers

In any real system, job execution is slowed down by various overheads and latencies. Frequently, such overheads are considered to be negligible in academic research. However, as we show in Chapter 4, overheads can actually have a significant impact on algorithm performance. This is especially true in the case of multiprocessors, where overheads tend to be higher than on uniprocessors, which is in large part due to increased synchronization requirements and cache affinity issues. It is therefore important to consider overheads both when comparing scheduling and locking choices for RTOS implementations and when establishing the temporal correctness of actual systems. To this end, overheads and latencies must be bounded and accounted for during schedulability analysis.

In the remainder of this chapter, we provide a comprehensive overview of the techniques that we apply to account for overheads in Chapter 4. The need to integrate overheads into schedulability tests is not new and a number of standard "textbook techniques" have long been known and used in the analysis of event-driven uniprocessor systems (Liu, 2000; Devi, 2003). Many of these accounting methods can also be applied to multiprocessor systems. However, we are not aware of prior efforts to document how these methods can be applied to event-driven multiprocessor real-time scheduling, which is not quite as straightforward as it may seem. In particular, some delays that arise under global scheduling and their interaction with pi-blocking analysis may not be obvious on first sight.

Jobs that do not self-suspend incur delays when they are released, when they are preempted, and when they are interrupted. We begin by discussing the fundamental safety property underlying overhead accounting (Section 3.4.1) and then discuss how job releases are delayed and how the resulting "jitter" must be accounted for (Section 3.4.2). Thereafter, we illustrate how standard techniques (Liu, 2000) can be used to account for preemption and migration overheads (Section 3.4.3), and then discuss how to account for interrupts (Section 3.4.4). Interrupts are especially difficult to account for under EDF-based schedulers. We first review *task-centric* interrupt accounting, which we previously introduced in (Brandenburg *et al.*, 2009, 2011). Task-centric interrupt accounting is safe and relatively straightforward, but can be severely pessimistic in some cases, *i.e.*, it can lead to significant algorithmic capacity loss. In Section 3.4.5, we introduce *preemption-centric* interrupt accounting, which is a novel interrupt accounting technique designed to overcome some

of the pessimism inherent in task-centric interrupt accounting. Finally, we summarize the overhead accounting techniques applicable to each scheduler in Section 3.4.6.

3.4.1 Approach

From a high-level perspective, there are two ways that overhead accounting can be integrated into schedulability analysis. The first method is to design a new schedulability test from first principles that recognizes overheads as first-class entities that are an integral part of the system model. However, such a model is much more complex and risks resulting in tedious, uninspiring analysis. Unsurprisingly, first-class analysis of overheads is rarely pursued.

The second method reuses the existing large body of schedulability results that are not cognizant of overheads. Given an overhead-affected implementation of a task set τ, applying such a schedulability test to τ does *not* guarantee temporal correctness due to the risk of unanticipated overhead-related delays. The key idea is to derive an *equivalent but overhead-free* task set τ' that is substituted for τ during schedulability analysis. That is, the idea is to construct an idealized task set that is amenable to existing analysis. For this approach to be safe, τ' must be a "lower bound" on schedulability in the sense that τ' *without* overheads is "at least as hard to schedule" as τ *with* overheads. This notion can be formalized as follows.

Definition 3.1. A task set τ' is a *safe HRT approximation* of τ (with respect to a given platform and implementation) if and only if the following assertion holds: if there exists a legal arrival sequence such that some task in τ misses a deadline in the presence of overheads, then there exists a legal arrival sequence such that a task in τ' misses a deadline in the absence of overheads.

The notion of a *safe SRT approximation* can be defined likewise with regard to exceeding a given tardiness bound B. All approximations used in this dissertation are safe for both HRT and SRT analysis; we therefore omit "HRT" and "SRT" in the following discussion of safe approximations.

Intuitively, Definition 3.1 requires that if τ can fail, then τ' can fail as well. This allows τ' to be substituted for τ in schedulability tests that assume overheads to be negligible: if τ' passes such a schedulability test, then the absence of temporal failures has been proven. This in turn implies that τ must be correct as well. The challenge of accounting for overheads during schedulability analysis can thus be reduced to the much simpler task of finding a suitable safe approximation τ'.

Obviously, the construction of τ' should introduce as little additional pessimism as possible: ideally, τ' should not fail the schedulability test if τ is in fact correct. For example, *any* infeasible task set is a safe approximation of any other task set, but such a pessimistic approximation is obviously devoid of practical value. Instead, to obtain a useful approximation, each task $T_i \in \tau$ is transformed into a corresponding task $T'_i \in \tau'$ such that the worst-case delays experienced by any J_i are reflected in the parameters of T'_i. Depending on the expressivity of the task model assumed by the underlying overhead-unaware schedulability test, overheads can be modeled in some or all of the following ways:

1. by inflating the execution requirement e_i,
2. by decreasing the period p_i,
3. by decreasing the relative deadline d_i,
4. by increasing the maximum time of self-suspension,
5. by increasing the bound on pi-blocking b_i, and
6. by adding tasks to represent overhead sources.

For conciseness, we let $susp_i$ denote the maximum time of self-suspension of any J_i in the following discussion. (However, few schedulability tests besides response-time analysis for FP scheduling support this parameter.) Note that these (offline) parameter changes are opposite to the online parameter changes that a sustainable scheduling algorithm must be resilient to (recall Section 2.2.3). To account for overheads, parameters must be overestimated; the use of sustainable schedulers and analysis ensures that this does not result in scheduling anomalies.

In the remainder of this section, we discuss how the scheduling overheads that arise under event-driven schedulers in LITMUS$^{\mathrm{RT}}$ can be accounted for with approximations that are safe for both HRT and SRT schedulability analysis.

3.4.2 Release Delay

We begin with *release delay*, which is the interval from the point in time that the event that triggered the release of a job J_i occurred (denoted t_0) until the first instruction of J_i is executed by a processor.

This corresponds to the metric commonly used to evaluate RTOS performance in industry, namely interrupt (or scheduling) latency. In the idealized sporadic task model, t_0 is J_i's release time and, if J_i has sufficient priority, also the time that J_i commences execution. In practice, the occurrence of the triggering event must first be relayed to the system (by means of an interrupt) and translated into a job release, which then triggers several implementation-related overheads.

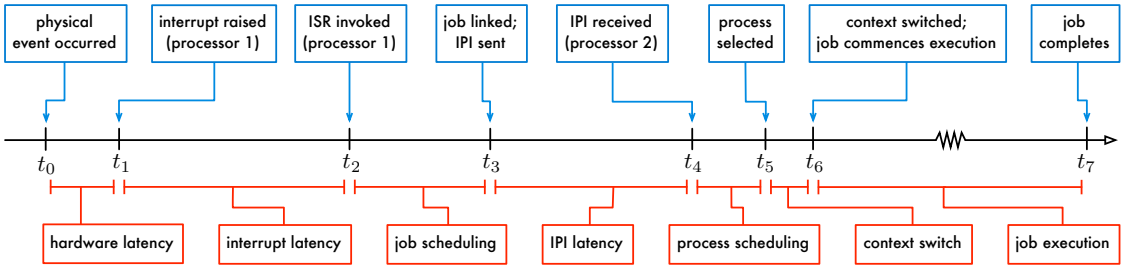
To simplify the following discussion, we assume in this section that jobs are not preempted and not interfered with by ISR execution once scheduled. Delays due to preemptions are considered in Section 3.4.3; the impact of interrupts is discussed in Section 3.4.4. We further assume that the released job J_i has sufficient priority to be scheduled immediately—otherwise, J_i is not affected by release delay since it is not scheduled anyway.

Global scheduling. The release delay under global event-driven scheduling consists of up to six contributing factors. These are illustrated in Figure 3.9(a), which depicts a timeline of a job's release assuming that its releasing interrupt is handled on processor 1 and that it should be scheduled on processor 2. We discuss each of the six sources of overhead in detail.

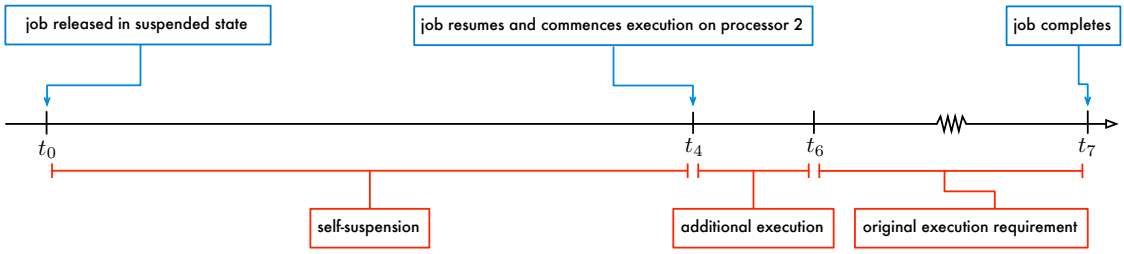
- At time t_0, the triggering event occurs. This is detected by some hardware device (*e.g.*, a photo sensor detects a change in luminosity, a hardware timer's counter matches a comparator register, *etc.*). The hardware device raises an interrupt line to communicate to the systems interrupt management hardware (*e.g.*, the IO-APIC in x86-based systems) that an interrupt should be generated. The interrupt hardware translates the physical interrupt signal to a packet describing the interrupt, determines which processor is responsible for handling this type of interrupt, and then routes the interrupt packet to a processor, which happens to be processor 1 in this case. If the bus is multiplexed among multiple uses, the delivery of the interrupt packet may be delayed. For example, there is no dedicated APIC bus in modern x86 systems; instead, the main memory bus, or *front side bus*, also carries APIC packets.

  The raising and propagation of the interrupt signal induces a delay that we refer to as *hardware latency*. The extent of hardware latency is generally not under control of the RTOS.

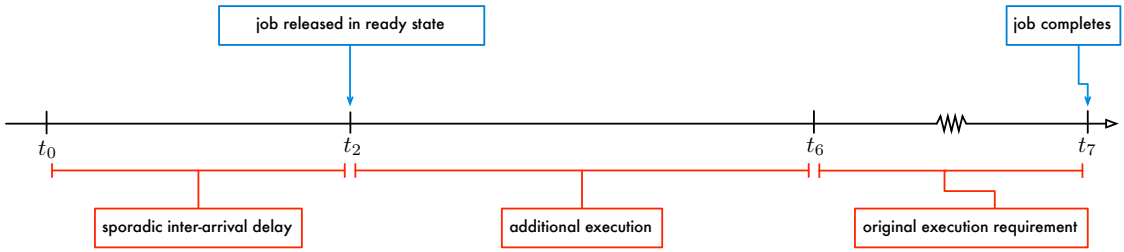
- At time t_1, the interrupt signal has reached processor 1 (*e.g.*, the local APIC in x86-based systems). However, the interrupt may not be processed immediately if processor 1 has currently disabled interrupt delivery (or has not yet acknowledged a previous, higher-priority interrupt).



(a) Actual latencies and overheads.



(b) Equivalent event sequence.



(c) Equivalent event sequence with shortened deadline and period.

Figure 3.9: Timeline illustrating release delay under global and clustered event-driven scheduling. **(a)** In a real system, a newly-released job is delayed by several sources of latencies and overhead before it commences execution. **(b)** The latencies and overhead can be equivalently modeled as an initial self-suspension followed by additional execution. **(c)** Under EDF-based schedulers, time t_2 is recorded as the release time and self-suspensions are analyzed as execution time instead.

As a result, *interrupt latency* accrues while the interrupt is pending. RTOS designers strive to minimize times during which interrupt delivery is disabled to reduce interrupt latency.

- At time t_2, processor 1 accepts the interrupt and transfers control to the appropriate ISR. The ISR determines that J_i has been released and updates the PCB of the process implementing T_i to reflect the parameters of the newly-released job. If the process was suspended, it is now resumed and enqueued in the ready queue. If J_i was instead released by a software timer, then the release heap corresponding to time t_0 is merged into the ready queue. The on_job_arrival() event handler in Listing 3.4 is executed, which links J_i to processor 2.

- At time t_3, processor 1 has finished the link-based job scheduling and sends an IPI to processor 2 to cause it to reschedule (if the process scheduled on processor 2 is currently preemptable; the case of a non-preemptable process is discussed below). The length of the interval $[t_2, t_3)$ is mainly determined by two factors: the efficiency of the ready queue implementation and the level of contention for the ready queue lock.

  IPIs are not delivered instantly. An IPI is typically relayed by means of the shared memory bus or an on-chip point-to-point network and may be delayed by the presence of memory transfers and other inter-processor messages. As is the case with regular interrupts, the IPI may not be processed immediately once it has reached processor 2 if interrupt delivery is currently disabled. Consequently, *IPI latency* accrues while the IPI is "in flight."

  If by chance J_i had been linked to processor 1 instead, then of course no IPI latency would have been incurred. In general, however, it is not possible to predict on which processor a job will be linked; therefore, IPI latency is part of the worst-case scenario.

- At time t_4, the IPI is received by processor 2. The IPI-handling code does little more than setting the rescheduling flag of the currently scheduled process to invoke the scheduler (recall Section 3.2.1). The Linux process scheduler invokes the pick_next_task() method of the LITMUS$^{\mathrm{RT}}$ scheduling class, which in turn calls the schedule() method of the active plugin. In a link-based scheduler, the schedule() method observes that J_i has been linked and selects the process implementing T_i to be scheduled next. The execution time of the process scheduler

is mainly determined by the level of lock contention for the ready queue lock, which is used to serialize scheduling decisions.

- After the next process to schedule has been selected, Linux enters the context switch code path at time t_5. Prior to the actual context switch, some statistics and maintenance code executes. Directly after the actual context switch, additional resource management tasks are carried out that could not be done while holding a runqueue lock. Finally, the kernel transfers control to the process and J_i commences to execute at time t_6.

Finally, if J_i is not preempted and not disturbed by interrupts, it will complete at most e_i time units after time t_6 at time t_7.

Safe approximation. An overhead-unaware schedulability test assumes that J_i is scheduled immediately at time t_0 on processor 2 when in fact processor 2 is unaware of J_i's arrival until time t_4. Further, J_i is not actually scheduled until time t_6 (in the sense of executing instructions accounted for by e_i). In an overhead-free environment, these delays can be modeled as illustrated in Figure 3.9(b).

- We refer to the delay during $[t_0, t_2)$ collectively as *event latency*, denoted as Δ^{ev}, to the delay during $[t_2, t_3)$ as *release overhead*, denoted as Δ^{rel}, and to the delay during $[t_3, t_4)$ as IPI latency, which we denote as Δ^{ipi}. While processor 2 is unaware of the release of a new job during $[t_0, t_4)$, J_i is effectively ineligible for scheduling. In an overhead-free model, this is equivalent to J_i having self-suspended immediately upon its release at time t_0 for $(\Delta^{ev} + \Delta^{rel} + \Delta^{ipi})$ time units.

- The time lost to process management tasks during $[t_4, t_6)$ is comprised of *scheduling overhead* during $[t_4, t_5)$, denoted as Δ^{sch}, and *context-switch overhead* during $[t_5, t_6)$, denoted as Δ^{cxs}. Since processor 2 is effectively executing code on behalf of J_i, this time can be added to J_i's execution requirement, *i.e.*, in an overhead-free model, J_i simply carried out additional computations for $(\Delta^{sch} + \Delta^{cxs})$ time units prior to executing for e_i time units.

Consequently, a transformed task T_i' is a safe approximation of T_i with regard to maximum release delay if

$$susp_i' \geq susp_i + (\Delta^{ev} + \Delta^{rel} + \Delta^{ipi}), \text{ and} \tag{3.1}$$

$$e_i' \geq e_i + (\Delta^{sch} + \Delta^{cxs}). \tag{3.2}$$

The period and relative deadline of T_i are not affected by this transformation, *i.e.*, $p_i' = p_i$ and $d_i' = d_i$ (pi-blocking is discussed below). Interrupt delivery is disabled during most of the interval $[t_4, t_6)$; nonetheless, the execution is not modeled as a non-preemptive section of J_i because we assume that any delays due to non-preemptable execution of the kernel are included in the interrupt latency and hence bounded by Δ^{ev}. Unfortunately, most G-EDF schedulability tests are not capable of analyzing self-suspensions, *i.e.*, the assumed task model requires $susp_i' = 0$. A different approximation is thus required for G-EDF.

Uncertain release times. There is in fact another discrepancy between model and reality that must be considered if a job's release time affects its priority. The question is: what *is* the release time of J_i that is used for prioritization purposes? In an overhead-free model, J_i is clearly released at time t_0. However, in a real system, if J_i was released by a device interrupt, then time t_0 is actually unknown to the kernel, as is time t_1 since hardware and interrupt latencies are not reported to the processor (at least in commodity hardware). Therefore, the kernel records time t_2 as the release time of J_i and defines its absolute deadline as $t_2 + d_i$. This creates a discrepancy between the analyzed and the actual schedule since the absolute deadline would have been $t_0 + d_i$ in the absence of overheads.

To compensate for this mismatch, the relative deadline and period have to be decreased by the maximum possible length of the interval $[t_0, t_2)$. The resulting approximation is illustrated in Figure 3.9(c). In this alternate idealized model, the job J_i' is not actually released until time t_2, which is equivalent to T_i' exhibiting increased inter-arrival delay. Further, to work around the lack of analysis for self-suspensions, the delay due to job scheduling and IPI latency are modeled as

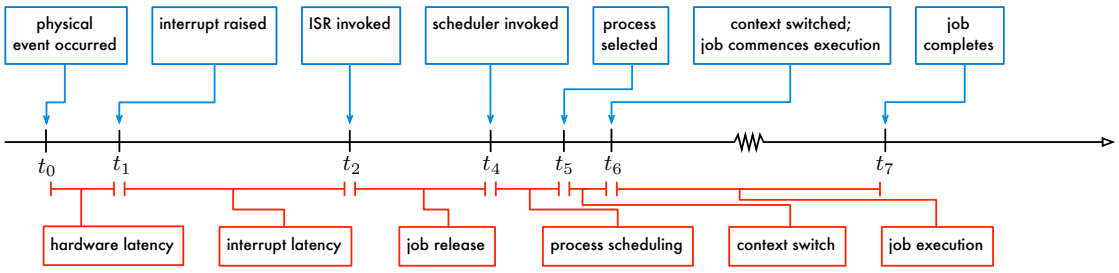


Figure 3.10: Timeline illustrating release delay under partitioned event-driven scheduling *without* dedicated interrupt handling. The points in time $t_0, \ldots, t_7$ match those in Figure 3.9. Under partitioned scheduling, interrupts can be routed such that jobs are released on the processor to which their corresponding task has been assigned. Therefore, jobs do not incur IPI latency, which is reflected by the absence of time t_3, *i.e.*, time t_4 follows time t_2 since scheduling occurs locally. If dedicated interrupt handling is used, then jobs can be delayed by IPI latency and Figure 3.10 applies instead.

additional execution time. This results in the following: task T_i' is a safe approximation of T_i if

$$p_i' \leq p_i - \Delta^{ev}, \tag{3.3}$$

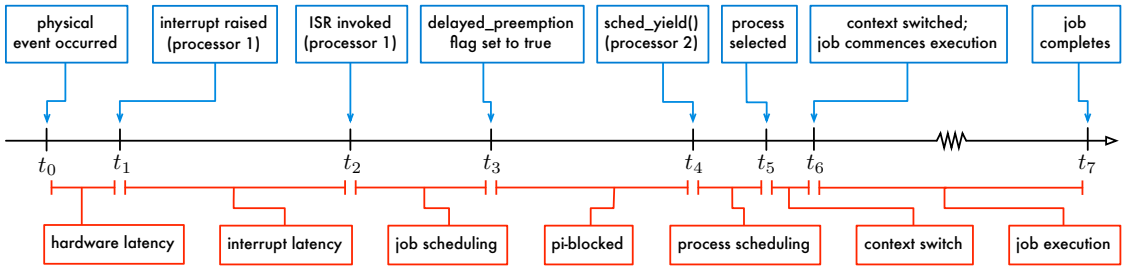
$$d_i' \leq d_i - \Delta^{ev}, \text{ and} \tag{3.4}$$

$$e_i' \geq e_i + (\Delta^{sch} + \Delta^{cxs} + \Delta^{rel} + \Delta^{ipi}). \tag{3.5}$$

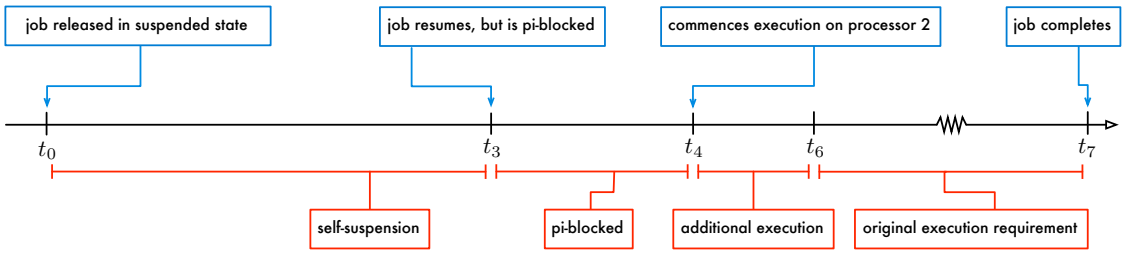
It is not sufficient to only decrease the relative deadline and leave p_i' unchanged. Since Δ^{ev} is an upper bound on the maximum event latency, it is theoretically possible for two job releases to be separated by only $(p_i - \Delta^{ev})$ time units (if the next job incurs zero event latency). This effect is commonly referred to as *release jitter* in the scheduling literature. In the case of FP scheduling, release jitter is equivalent to a self-suspension, which is expressed by Equation (3.1).

Note that uncertain release times affect EDF-based schedulers at runtime. The kernel thus needs to compute a job's absolute deadline based on d_i' and not based on d_i, *i.e.*, the parameters of T_i' should be used both during schedulability analysis and as the reservation parameters of T_i's process.

Partitioned scheduling. Releases are delayed largely in the same way under partitioned scheduling as in the global case. However, there is one notable difference, which is illustrated in Figure 3.10. If the system does *not* use dedicated interrupt handling, then interrupts can be routed such that each processor handles only local job releases. Consequently, J_i does not incur IPI latency ($\Delta^{ipi} = 0$). This renders Equation (3.1) and Equation (3.5) slightly less pessimistic.



(a) Actual latencies and overheads.



(b) Equivalent event sequence.

Figure 3.11: Timeline illustrating release delay under event-driven scheduling in the presence of non-preemptive sections. The depicted timeline corresponds to Figure 3.9(a), with the difference that the job on processor 2 is assumed to be non-preemptable. **(a)** No IPI latency is incurred; instead, processor 1 sets the delayed_preemption flag to notify the non-preemptable job on processor 2 that it should invoke the scheduler (recall Section 3.3.2). **(b)** As in Figure 3.9(a), the remaining latencies and overhead can be modeled as an initial self-suspension followed by additional execution. However, as it is in the actual case, the newly-released job incurs pi-blocking.

Non-preemptivity. In the scenario shown in Figure 3.9, we have assumed that the process scheduled on processor 2 is preemptable at time t_3. Figure 3.11 depicts a timeline where this is not the case. The event latency Δ^{ev} and release overhead Δ^{rel} during $[t_0, t_3)$ are not affected by the non-preemptive section. However, at time t_3, processor 1 does not send an IPI since processor 2 cannot preempt the current process anyway. Instead, processor 1 sets the delayed_preemption flag in the control page of the scheduled process to indicate that it should invoke the scheduler as soon as possible, as described in Section 3.3.2. This is possible since each control page is part of the kernel address space and thus accessible from all processors at all times. At the end of its non-preemptive section at time t_4, the scheduled process invokes the scheduler by means of the sched_yield() system call, which triggers scheduling and context-switch overhead before J_i commences execution. (Processor 1 must check the np_flag and set the delayed_preemption flag atomically; otherwise, it could race with the exit_np_section() protocol.)

This requires only a small change to the equivalent overhead-free model, which is shown in Figure 3.11(b). As previously in Figure 3.9(b), the event latency and release overhead can be modeled as a self-suspension. When J_i resumes in the overhead-free model at time t_3, it incurs pi-blocking, just as it does in the actual system. The scheduling and context-switch overhead that arise after the non-preemptive section can again be modeled as additional execution of J_i, as in Figure 3.11(b). This yields the following safe approximation for this scenario: T_i' is a safe approximation of T_i if $e_i' \geq e_i + (\Delta^{sch} + \Delta^{cxs})$ and $susp_i' \geq susp_i + (\Delta^{ev} + \Delta^{rel})$. Note that $susp_i'$ does not reflect for Δ^{ipi} in this case, *i.e.*, accounting for pi-blocking "absorbs" some of the worst-case release delay. This can be used to reduce pessimism in the construction of T_i'.

A given job may incur either IPI latency or pi-blocking due to a non-preemptive section upon release—but not both. Therefore, only the maximum of the two must be considered during schedulability analysis. This implies that (very) short non-preemptive sections are "free" in the sense that they do not add additional pessimism to the schedulability analysis required for global and clustered scheduling (and partitioned scheduling with dedicated interrupt handling). Alternatively, if the maximum non-preemptive section length exceeds Δ^{ipi}, then IPI latency becomes irrelevant and does not have to be accounted for. This argument applies equally when modeling IPI latency as increased execution requirement.

Formally, let b_i^{np} denote the maximum pi-blocking incurred by any J_i upon release due to non-preemptable execution, and let b_i^{other} denote the maximum pi-blocking due to other causes (such that $b_i = b_i^{other} + b_i^{np}$). If IPI latency is modeled as a self-suspension using Equation (3.1) under P-FP scheduling, or if IPI latency is modeled as execution time using Equation (3.5) under EDF-based schedulers, then T_i' remains a safe approximation of T_i as long as

$$b_i' \geq b_i^{other} + \max(0, b_i^{np} - \Delta^{ipi}). \tag{3.6}$$

So far, we have assumed that jobs incur release delay but are not preempted or disturbed by interrupts once scheduled. Next, we consider delays that arise due to preemptions and migrations. For the sake of clarity, we first consider preemption and migrations in isolation, too, and thereafter integrate release delay, preemption and migration delays, and interrupt delays in Section 3.4.4.

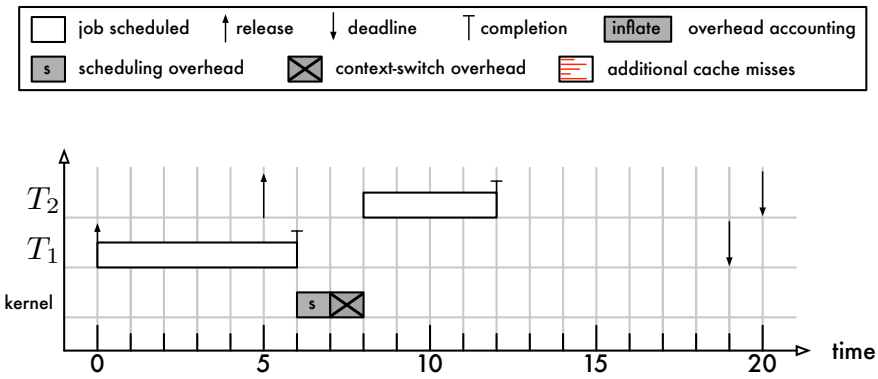
3.4.3 Preemption and Migration Delays

From the point of view of overhead accounting, preemptions and migrations are equivalent. In either case, a ready job is preempted in favor of a higher-priority job and continues execution at a later time. Whether a job continues execution on the same or on a different processor impacts the magnitude of overheads, but it does not change the kind of overheads that are incurred. We therefore focus on preemptions first in the following discussion and note that it applies equally to migrations.

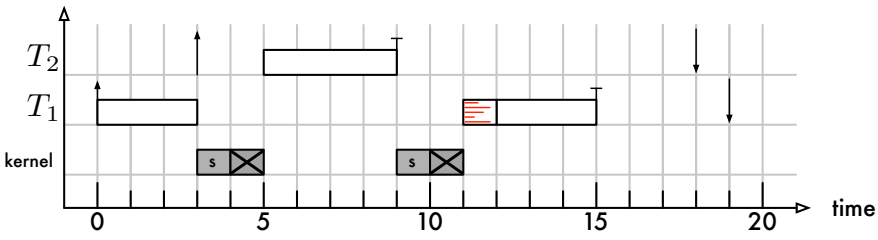
Conveniently, the standard techniques used to account for preemptions under JLFP schedulers on uniprocessors (Liu, 2000) also apply to both global and partitioned JLFP schedulers. To illustrate the additional overhead caused by a preemption, we first consider a schedule *without* preemptions as a baseline scenario. Figure 3.12(a) shows a uniprocessor EDF schedule of two jobs that execute in sequence. When J_2 is released at time 5, it has a later deadline than the already scheduled J_1, which is hence not preempted. When J_1 completes after six time units of execution at time 6, the scheduler is invoked to select the next process. The process implementing J_2 is chosen and commences execution after the context switch completes at time 8. There are two important things to note. First, J_1 is not affected by J_2's execution. Even though scheduling and context-switch overhead arise before J_2 is scheduled, these delays occur only *after* J_1's completion and thus do not affect its temporal correctness. Second, J_2 is delayed by these overheads. However, the resulting delay of $(\Delta^{sch} + \Delta^{cxs})$ time units is eclipsed by the worst-case release delay, which J_2 did not actually incur since it was not the highest-priority job at the time of release. No additional overhead accounting is thus required in this scenario.

In Figure 3.12(b), J_2's release occurs already at time 3, which results in J_2 having a higher priority than J_1. A preemption is thus required, which causes additional *direct* and *indirect* overheads. The direct overhead is due to the invocation of the scheduler, which is invoked twice. Before J_2 commences execution at time 5, the scheduler must identify J_2's implementing process and a context switch must be performed. Similarly, after J_2 completes at time 9, the scheduler must be invoked again to identify J_1 as the next-highest-priority job that should continue to execute, and another context switch is required.

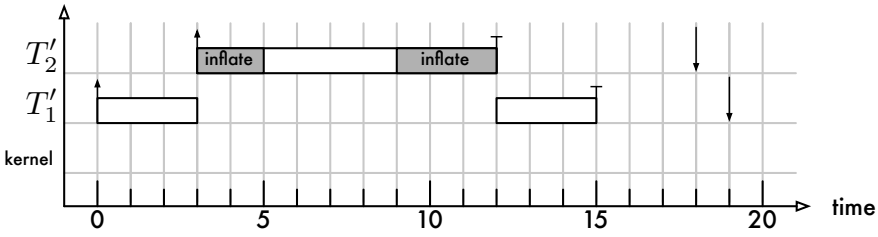
The indirect overhead is due to a loss of cache affinity experienced by the preempted job J_1. Recall from Section 2.1.2 that a process develops cache affinity when most of its working set has



(a) Schedule with sequential execution.



(b) Schedule with preemption.



(c) Equivalent overhead-free schedule.

Figure 3.12: Uniprocessor EDF schedules illustrating overheads caused by preemptions. For the sake of clarity, overhead due to release delay and interrupts is considered negligible, whereas scheduling and context-switch overheads are exaggerated. **(a)** If jobs execute sequentially, then only one scheduler invocation and context switch are required to select the next job. J_1 executes for six time units before completing. **(b)** In the case of a preemption, additional direct and indirect overheads are caused. J_1's execution requirement increased to seven time units due to the intermittent loss of cache affinity. **(c)** Both direct and indirect overheads can be modeled by inflating the execution requirement of the preempting job.

been loaded into cache(s). When a job commences execution, it must be necessarily expected to be cache-cold, so it is reasonable to assume that the initial burst of compulsory misses on release is accounted for by the execution requirement e_i. However, when a job is preempted, it loses cache affinity as the state of the cache(s) is perturbed by the preempting job. Consequently, it suffers a renewed burst of compulsory cache misses when it reestablishes cache affinity. This has the effect of increasing its execution requirement. For example, in Figure 3.12(b), J_1's execution requirement increased from six to seven time units due to the additional cache misses.

From the point of view of J_2, this scenario is an example of release delay and all overheads are adequately accounted for by the analysis of release delay presented in the preceding section. In particular, J_2 is not delayed by the additional scheduling and context-switch overheads that follow its completion.

However, J_1 experiences significant delays that are unrelated to J_1's release delay, and that must be accounted for separately. Because both scheduler invocations occur before J_1 completes, it is delayed both times, and its completion is further delayed by the loss of cache affinity. On first sight, it may seem intuitive to account for these overheads by increasing e'_1 to reflect the additional scheduling and cache-related overheads. However, to construct a safe approximation of T_1 based on this approach, an upper bound on the number of preemptions must be derived. In the general case, this is both difficult and very pessimistic. For example, it is easy to find task sets where a job is preempted more than n times, *i.e.*, jobs can incur $\Omega(n)$ preemptions in general.

Luckily, indirect accounting can be used to obtain a safe $O(1)$ approximation under JLFP schedulers: instead of charging the costs of a preemption to the preempted, lower-priority job, they are charged to the preempting, higher-priority job (Liu, 2000). This is illustrated in Figure 3.12(c). Here, J'_2's execution requirement has been inflated by the cost of both scheduler invocations and context switches, and also by an amount of time that matches the cache-related execution time increase of J_1. As shown in Figure 3.12(c), this results in a safe approximation since overhead-unaware schedulability analysis will consider T'_2's execution requirement when bounding T'_1's response time, which then implicitly accounts for the preemption-related overheads. Since each job is released only once and thus causes at most one preemption in a JLFP scheduler, it is sufficient to charge each task for the cost of one preemption with this "accounting trick."

Recall that Δ^{sch} denotes scheduling overhead and that Δ^{cxs} denotes context-switch overhead. We further let Δ^{cpd} denote *cache-related preemption and migration delay* (CPMD), *i.e.*, the maximum cost of re-establishing cache affinity after a preemption and migration. This results in the following transformation: the overhead-free model τ' is a safe approximation of τ with regard to preemptions and migrations if

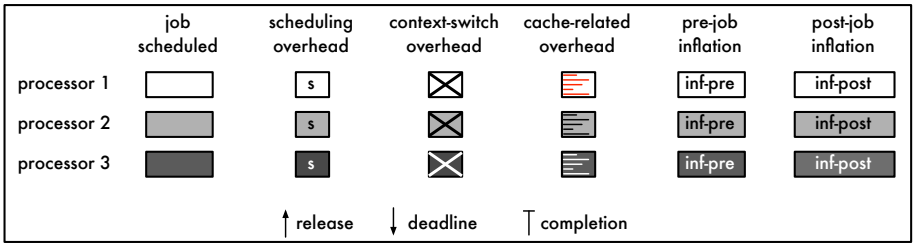
$$e'_i \geq e_i + 2 \cdot (\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd} \tag{3.7}$$

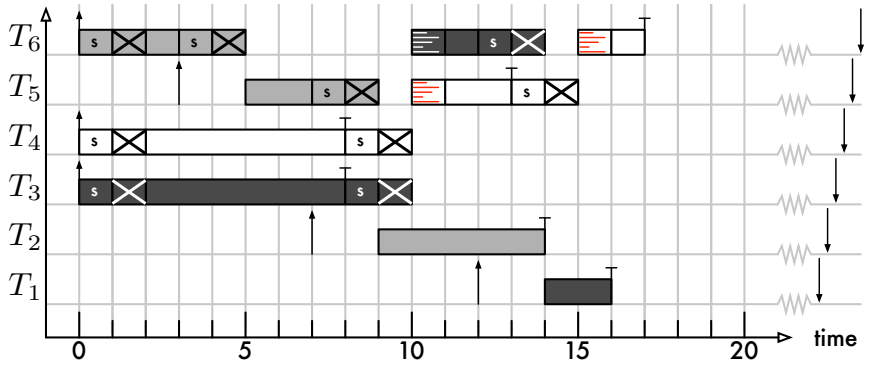
for all $T_i \in \tau$ and each corresponding $T'_i \in \tau'$. Note that while release delay is accounted for on a task-by-task basis, this transformation is only correct if it is applied to all tasks in the task set.[15] Further note that the term Δ^{cpd} is *not* a bound on the maximum CPMD incurred by any job of T_i—rather, it is the maximum CPMD incurred by any *other* job, *i.e.*, any potential preemption "victim." In this dissertation, we make the simplifying assumption that Δ^{cpd} denotes the maximum CPMD incurred by any job of any task.

Migrations. Equation (3.7) is also sufficient under global and clustered JLFP scheduling. This is demonstrated in Figure 3.13, which shows two schedules that illustrate that migrations can be accounted for similarly to preemptions.

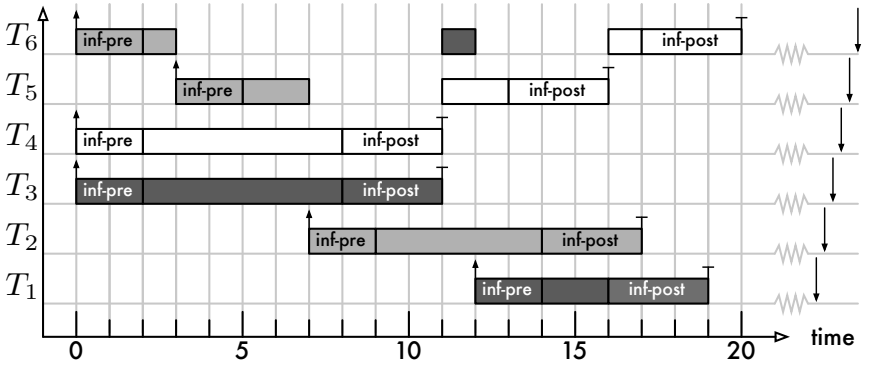
Example 3.8. Inset (a) shows a G-EDF schedule of six jobs on three processors. Two of the jobs, J_5 and J_6, migrate in the depicted schedule. Interrupts and latencies are assumed negligible for the sake of clarity. At time 0, jobs J_3, J_4, and J_6 are released and commence execution at time 2 after preempting the background or idle processes on each of the three processors. At time 3, J_5 is released and linked to processor 2, which causes the lowest-priority job J_6 to be preempted. J_5 incurs release delay until the context switch is complete at time 5. At time 7, J_5 is in turn preempted by the higher-priority J_2, which commences execution at time 9. In the mean time, J_3 and J_4 complete at time 8. Since J_5 and J_6 are unlinked at the time, they become linked to processors 1 and 3, and corresponding migrations are initiated. When J_5 and J_6 continue to execute at time 10, they both incur CPMD, which delays their completion by one time unit. J_6 is preempted a second time at time 12 when J_1 arrives, which results in J_6 migrating to processor 1 when J_5 completes.

[15] In general, any sporadic job release could potentially cause a preemption to occur under JLFP scheduling. As an exception, jobs of the lowest-priority task under FP scheduling never preempt other real-time jobs.





(a) Schedule with preemptions and migrations.



(b) Equivalent overhead-free schedule.

Figure 3.13: G-EDF schedules illustrating overheads caused by preemptions and migrations. For the sake of clarity, overhead due to interrupts and latencies is considered negligible, whereas scheduling and context-switch overheads are exaggerated. **(a)** Newly released jobs must first preempt a previously scheduled job (at times 3, 7, and 12) or a background process (at time 0). A job that continues execution is similarly delayed by a scheduler invocation, a context switch, and CPMD. Note that J_6 is linked to processor 1 at time 13 when J_5 completes, but that it is not scheduled until time 15. Concurrently, J_2 completes on processor 2 at time 14. In theory, processor 2 is idle thereafter; in practice, processor 2 executes a context switch to a background process (not shown). **(b)** An equivalent overhead-free schedule can be constructed since each job's execution requirement is inflated to account for a scheduler invocation and context switch before it commences execution, and additionally for a scheduler invocation, context switch, and CPMD after it finishes execution. Note that J_6 does not incur any priority inversion during $[14, 16)$ because the post-job inflation of both J_5 and J_2 account for its delay.

Technically, a priority inversion arises during $[14, 15)$: after J_2 completes at time 14, processor 2 is idle from a real-time scheduling point of view, but J_6 is effectively not scheduled until time 15, when the context switch from J_5 to J_6 is complete. This is unavoidable in practice because processor 1 has already selected J_6 at time 14, and because processor 2 still needs to switch away from J_2.

Figure 3.13(b) shows an equivalent, overhead-free schedule where the execution requirements have been inflated according to Equation (3.7). The increase in a job J_i's execution requirement can be split into two components: the *pre-job* charge is $\Delta^{sch} + \Delta^{cxs}$, wheres the *post-job* charge is $\Delta^{sch} + \Delta^{cxs} + \Delta^{cpd}$. The pre-job charge accounts for the overhead that arises when J_i commences execution; the post-job charge accounts for the delay that a preempted job incurs when it continues execution after J_i completed. Both charges are shown in Figure 3.13(b). Note that each preemption causes a preempted job to be delayed by $2 \cdot (\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd}$ time units, *i.e.*, one pre-job and one post-job charge, but that the accounting for one job's delay is effectively split across up to two higher-priority jobs. For example, consider J_5, which is preempted by J_2 at time 7. In Figure 3.13(a), J_5 is delayed by overheads during $[7, 11)$. In the overhead-free schedule depicted in Figure 3.13(b), this delay can be equivalently attributed to J_2's pre-job charge during $[7, 9)$ and to J_4's post-job charge during $[9, 11)$. Similarly, J_6's delay during $[14, 16)$ can be attributed to the post-job charges of both J_5 and J_2 without introducing any priority inversion (the post-job charge attributed to J_2 rules out any priority inversion). This highlights that it is key that Equation (3.7) is applied to *all* tasks. Response times in inset (b) exceed the actual response times in inset (a), which illustrates that the jobs in inset (b) are a safe approximation of those in inset (a). $\Diamond$

To summarize, Equation (3.7) provides an efficient way to safely account for preemption and migration costs under clustered event-driven JLFP schedulers. Next, we discuss how to bound interference from interrupts, which is the most pessimistic part of overhead accounting.

3.4.4 Interrupt Delays

ISRs execute with a statically higher priority than any real-time task in the system and cannot be scheduled. That is, while interrupts can be temporarily masked by the OS, they cannot be selectively delayed in favor of a higher-priority job. We say that a scheduled job is *stopped* when a processor transfers control to an ISR and the scheduled job's completion is delayed. In contrast to a regular

preemption, a stopped job cannot migrate to another processor while the interfering ISR executes since the ISR uses the kernel stack of the interrupted process.

Delays due to ISRs are fundamentally different from preemption overheads and release delay. Both preemption and release delay occur *synchronously* in the sense that they arise at a known point in time in relation to the job that is charged for them, namely prior to and after the job's execution. In contrast, delays due to ISRs arise *asynchronously* in the sense that ISRs may be triggered at any point during a job's execution.

Bounding ISR activations can be challenging in practice. The number of distinct interrupts is often limited (to reduce hardware costs), and hence interrupts may be shared among multiple devices. Further, devices and interrupts are commonly multiplexed among many logical tasks. For example, in Linux (and hence LITMUS$^{\mathrm{RT}}$ as well), a single HW timer is shared among multiple real-time tasks and potentially even best-effort tasks. As a result, it is difficult to characterize a system's worst-case interrupt behavior by modeling the individual (hardware) interrupts. Instead, it is more illuminating to consider logical *interrupt sources* that cause one or more ISRs to be invoked in some pattern, but do not necessarily correspond to any particular device.

In this dissertation, we consider two specific interrupt sources directly related to scheduling: sporadic *release interrupts*, which trigger job releases, and periodic *timer ticks*, which signal the beginning of a new quantum. The presented analysis techniques, however, can easily be transferred to integrate other sporadic and periodic interrupt sources. A more general interrupt model that allows for arbitrary interrupt sources (including "bursty" interrupt patterns) can be found in (Brandenburg *et al*., 2011).

Under dedicated interrupt handling, release interrupts are exclusively handled on the systems processors and thus never stop jobs, which are only scheduled on application processors. In contrast, timer ticks occur on all processors, even under dedicated interrupt handling. Under global scheduling without a dedicated systems processor, it can in general not be predicted on which processors a job may execute. Therefore, jobs may be delayed by *all* interrupt sources in this case. Under clustered scheduling, this applies similarly on a per-cluster basis.

Integrating sporadic and periodic interrupt sources is straightforward under FP and P-FP scheduling since ISRs can be modeled as jobs (Liu, 2000). Interrupt accounting is more difficult under EDF-based schedulers and global scheduling in general (where ISRs and jobs differ since ISRs

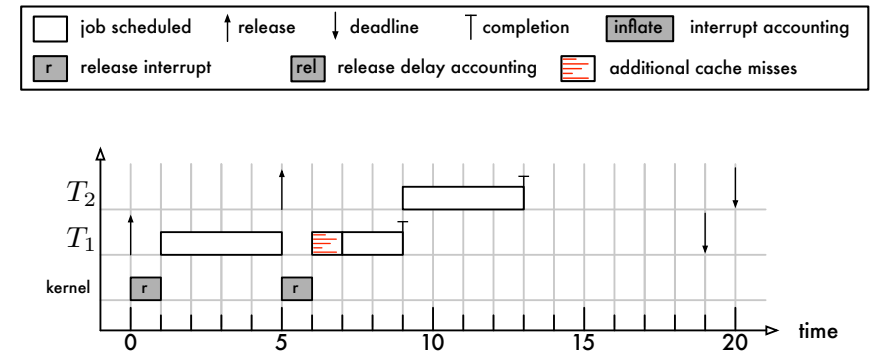
stop the currently scheduled job instead of preempting it). In this section, we discuss previously-published interrupt accounting techniques for FP and EDF-based scheduling. In the case of FP scheduling, we use the approach described by Liu (2000), and in the case of EDF-based scheduling, we first discuss an approach called "task-centric" interrupt accounting (Brandenburg *et al.*, 2011), which we subsequently extend in Section 3.4.5.

3.4.4.1 Release Interrupts

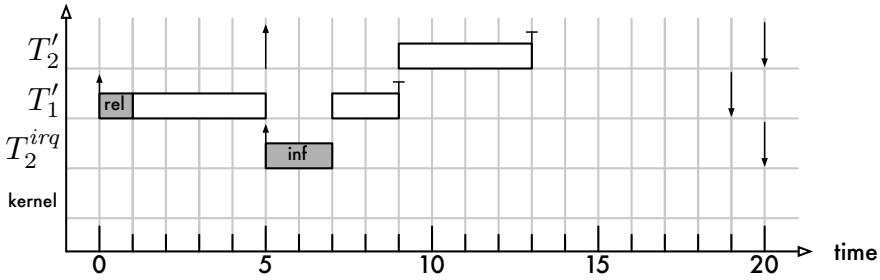
Recall the timeline depicted in Figure 3.11(a), which depicts the events that precede the release of a job J_i. After incurring hardware and interrupt latency, the ISR that releases J_i executes on processor 1 during $[t_2, t_3)$. Release delay already accounts for the delay that J_i experiences due to the execution of its *own* release interrupt, but it does not reflect delays due to the release of *other* jobs. The job that is scheduled on processor 1 at time t_2 (if any) is delayed while it is stopped during $[t_2, t_3)$. This must be accounted for separately.

Each task T_i represents an interrupt source that causes a release interrupt at most once every p_i time units (subject to a worst-case jitter of Δ^{ev} time units). Further, recall that a release ISR executes for at most Δ^{rel} time units. There is thus a strong resemblance between release interrupts and sporadic tasks. However, a sporadic interrupt source is not the same as a sporadic task because interrupts are not subject to scheduling.

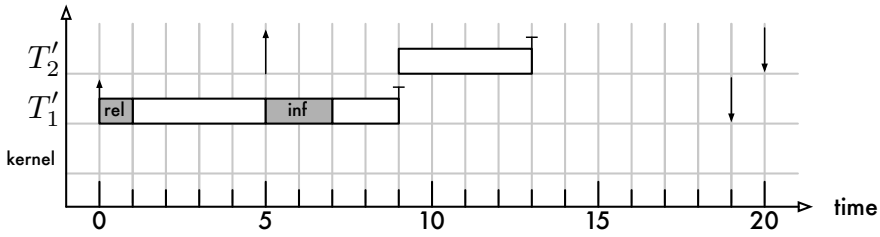
Example 3.9. The lack of control of the scheduler is illustrated in Figure 3.14, which shows the same uniprocessor EDF schedule of the two jobs from Figure 3.12(a), but this time subject to non-negligible delays due to release interrupts. At time 0, J_1 is released and delayed by its own release interrupt. This is accounted for by J_1's release delay and does not qualify as asynchronous interrupt delay. However, J_1 is delayed by another release interrupt at time 5 when J_2 is released. Since J_2 has a later deadline than J_1, J_2 does not affect J_1 in theory. In practice, the interrupt management hardware is unaware of J_2's deadline and the ISR executes immediately, which delays the higher-priority job in two ways: first, J_1 is stopped during $[5, 6)$, and second, parts of J_1's working set are evicted from cache(s) by the footprint of the ISR. Neither delay can be charged to J_2 because J_2's execution requirement does not impact J_1's response time under overhead-free analysis



(a) Schedule with release interrupts.



(b) Equivalent overhead-free FP schedule.



(c) Equivalent overhead-free EDF schedule.

Figure 3.14: Uniprocessor schedules illustrating interrupt delay accounting techniques. Scheduling and context-switch overheads have been omitted for clarity; the impact of release interrupts has been exaggerated. The schedule depicts the same scenario as shown in Figure 3.12(a). **(a)** In theory, J_1 should not be delayed by J_2. In practice, J_1 is stopped while the ISR that releases J_2 executes, and must reload parts of its working set when it continues execution. **(b)** Under FP and P-FP scheduling, J_2's release interrupt can be modeled as an additional higher-priority task T_2^{irq} with the same period as T_2. J_1's own release interrupt is already included when accounting for worst-case release delay. **(c)** Under EDF-based schedulers, J_2's release interrupt is accounted for by inflating J_1's execution requirement.

of EDF. This is also the case if the system were scheduled using FP instead since J_1 has a higher priority than J_2. ◊

The way interrupts are accounted for differs depending on the scheduling policy.

P-FP scheduling. In the case of FP and P-FP scheduling, each source of sporadic interrupts can be modeled as an additional higher-priority task. This is illustrated in Figure 3.14(b). The lower-priority task T_2 represents an interrupt source that can delay jobs of T_1. This source of delay is accounted for by adding an imaginary interrupt task T_2^{irq} with the same period as T_2. During response-time analysis, imaginary interrupt tasks are considered to have a higher priority than T_1 such that T_1's response-time bound depends on the execution carried out by all ISRs. The execution requirement of each interrupt task consists of two parts, namely the execution of the ISR itself and the delay caused by a partial loss of cache affinity of the scheduled job. Since ISRs typically have small cache footprints, a job loses cache affinity to a much lesser degree when interrupted by a release interrupt than when preempted by another job.

Let Δ^{cid} denote an upper bound on the *cache-related interrupt delay*, *i.e.*, the delay incurred by any job of any task when restoring cache affinity after being stopped by a release interrupt, and recall that Δ^{rel} denotes an upper bound on the execution requirement of a release ISR. A task set τ' is a safe approximation of a task set τ with regard to release interrupts if it includes an additional interrupt task T_i^{irq} for each $T_i \in \tau$ with the following parameters.

$$e_i^{irq} = \Delta^{rel} + \Delta^{cid} \qquad p_i^{irq} = p_i \qquad susp_i^{irq} = \Delta^{ev} \tag{3.8}$$

Setting $susp_i^{irq} = \Delta^{ev}$ accounts for the jitter in ISR invocation caused by hardware and interrupt latencies. The relative deadline and pi-blocking parameters are irrelevant for interrupt tasks.

When applying response-time analysis, each interrupt task is considered to have higher priority than any real task. As an exception, T_1's own interrupt does not have to be accounted for in this way since it is already accounted for by the self-suspension used to model release delay (see Section 3.4.2).

EDF-based scheduling. Accounting for interrupts is much more challenging under EDF-based schedulers since (global) EDF schedulability tests do not account for tasks that are *always* of higher priority, regardless of their release times. Liu and Layland (1973) discussed mixed EDF/FP

scheduling on a uniprocessor. Based on Liu and Layland's mixed EDF/FP analysis, Jeffay and Stone (1993) presented schedulability analysis for uniprocessor EDF in the presence of interrupts. However, their analysis assumes strictly periodic (and not sporadic) execution of both interrupts and tasks and assumes quantum-driven scheduling and thus does not apply to our implementation of P-EDF. Additionally, neither Liu and Layland's nor Jeffay and Stone's approach takes the effect of jitter into account, which we require for our purposes. To the best of our knowledge, interrupt accounting techniques for global event-driven scheduling had not been considered prior to our work presented in (Brandenburg *et al*., 2009, 2011).

In this dissertation, we discuss two interrupt accounting techniques for EDF-based schedulers in detail. We begin by summarizing task-centric interrupt accounting, which was first presented in (Brandenburg *et al*., 2009, 2011), and then introduce preemption-centric interrupt accounting, which is a refinement of the task-centric method.

Interrupt sources are not modeled as additional tasks under task-centric accounting. Instead, each task's execution requirement is inflated to directly account for the worst-case interrupt-related delay that a job may incur. This approach is illustrated in Figure 3.14(c). Release interrupts are charged for by inflating J_1's execution requirement to account both for its own release (as part of accounting for release delay) and for the release of J_2.

In general, to correctly inflate the execution requirement of a task T_i requires bounding the number of interrupts that can possibly stop any of its jobs. This in turn depends on its maximum response time—the longer a job J_i is pending, the more interrupts may occur during its execution. That is, even though a job can only be stopped by interrupts while it is scheduled, its exposure to interrupts increases when it is preempted. Since release interrupts are sporadic (and not periodic), interrupt sources could potentially experience inter-sporadic arrival delays to fire at exactly the time when J_i continues execution after a preemption. For example, suppose that J_i is preempted at time t_a and scheduled again at time t_b. In the worst case, no interrupt source fires during $[t_a, t_b)$, but each interrupt source fires at time t_b.

Task-centric interrupt accounting is based on the following coarse-grained upper bound on the number of times that a release occurs while a job J_i is pending, which is denoted as $nirq(T_i)$ and

defined as

$$nirq(T_i) \triangleq \sum_{1 \leq j \leq n} \left\lceil \frac{R_i + \Delta^{ev}}{p_j} \right\rceil. \tag{3.9}$$

Recall that R_i denotes a bound on the maximum response time of any J_i. Akin to response-time analysis for FP scheduling, this formula computes how many consecutive jobs of each T_j can overlap with an interval of length $R_i + \Delta^{ev}$, and hence bounds the maximum number release interrupts due to each T_j. The term "$+\Delta^{ev}$" is required to account for any jitter experienced by jobs of T_j (Audsley *et al.*, 1993).

Note that Equation (3.9) also counts release interrupts due to T_i itself, which is required if $R_i + \Delta^{ev} > p_i$, *i.e.*, if jobs may be tardy or if $d_i > p_i$. However, this overcharges by one release interrupt since J_i is not yet pending prior to the ISR that triggers its release. This means that the charge of Δ^{rel} in Equation (3.5) can be omitted when using task-centric interrupt accounting since it is implicitly counted by Equation (3.9).

A practical issue in the computation of Equation (3.9) is that R_i is likely unknown prior to accounting for interrupts (since interrupts affect the response time). As in the case of FP response-time analysis, this can be resolved using an iterative approach; in the case of HRT constraints, it is also possible to substitute d_i for R_i (Brandenburg *et al.*, 2009, 2011).

Assuming an upper bound on Equation (3.9) is known, a task set τ' is a safe approximation of τ with regard to asynchronous release interrupts if the execution requirement of each $T'_i \in \tau'$ satisfies:

$$e'_i \geq e_i + nirq(T_i) \cdot (\Delta^{rel} + \Delta^{cid}). \tag{3.10}$$

Note that this transformation must be applied to *each* task in τ; it is not sufficient to only inflate the execution requirement of a subset of the tasks. This is because interrupts prior to J_i's release can push back higher-priority demand and thus indirectly delay J_i. If each job is inflated for the interrupts that occur while it is executing, then the pushed-back higher-priority demand is implicitly accounted for by regular EDF schedulability analysis.

Task-centric interrupt accounting can be very pessimistic. In fact, it can be shown that task-centric interrupt accounting is subject to $\Omega(n^2)$ algorithmic capacity loss (Brandenburg *et al.*, 2009,

2011), which renders it inappropriate for large n. Intuitively, this quadratic capacity loss results since each of the n tasks' execution requirement is inflated to account for $nirq(T_i) \geq n - 1$ releases.

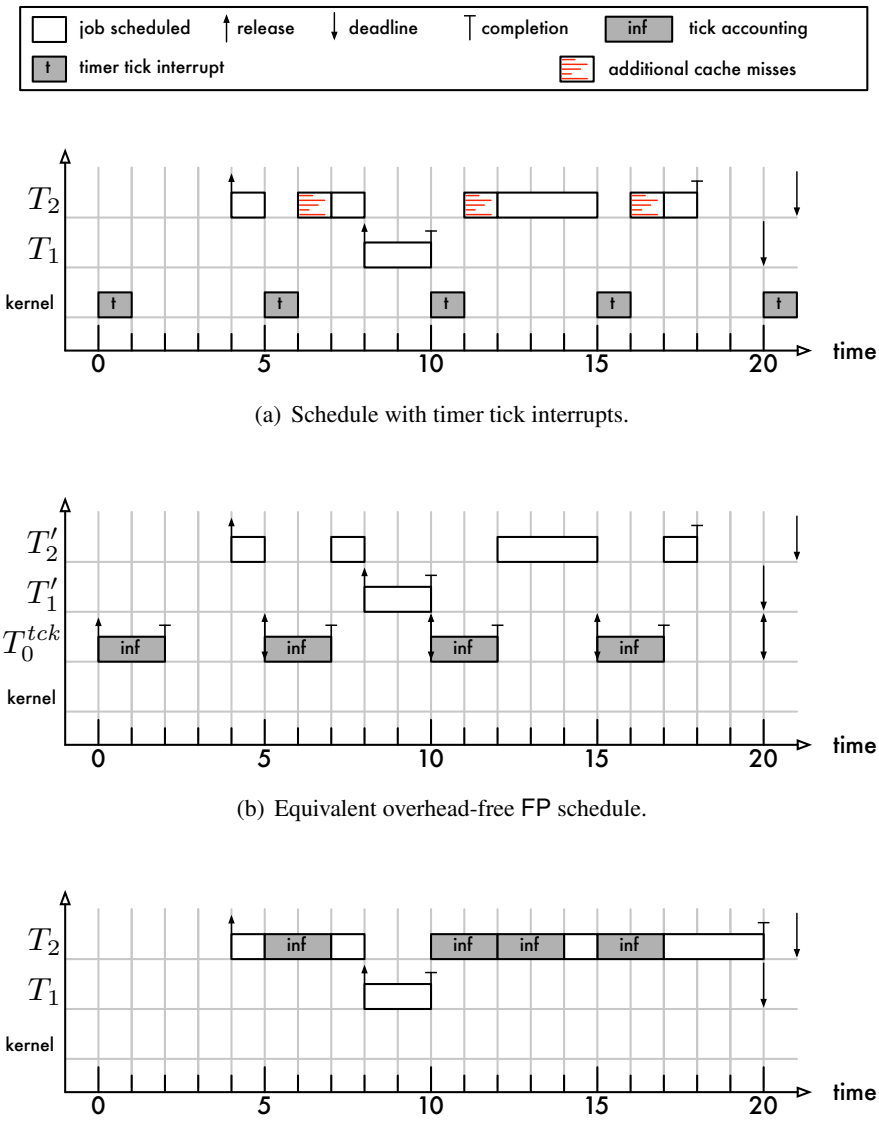
An alternate accounting technique called *processor-centric* interrupt accounting was also introduced in (Brandenburg *et al.*, 2011). Under processor-centric accounting, the execution requirement of jobs is not inflated to account for ISR execution. Intuitively speaking, ISRs are instead considered to "slow down" the effective rate of execution of scheduled jobs. More accurately, processor-centric accounting is based on special *restricted-supply* schedulability analysis, where processors are assumed to only be partially available to a task set (*e.g.*, this is the case under hierarchical scheduling). ISR execution can be modeled as a supply restriction under such analysis.

Processor-centric accounting is sometimes less pessimistic than task-centric accounting, but suffers from two limitations: first, processor-centric accounting is very pessimistic in the presence of high-utilization tasks, and second, it is not compatible with ordinary schedulability analysis. For these reasons, we do not use processor-centric interrupt accounting in this dissertation. The interested reader is referred to (Brandenburg *et al.*, 2011) for an empirical comparison of processor-centric and task-centric interrupt accounting, and to (Leontyev, 2010) for an in-depth discussion of the underlying restricted-supply schedulability analysis.

3.4.4.2 Timer Ticks

Timer ticks mark the beginning of a new quantum. They are implemented as periodic per-processor timers with a period of Q time units. In recent versions of Linux, this timer is implemented as a software high-resolution timer (*i.e.*, by means of the *hrtimers* subsystem) that is typically backed by the local APIC timer on x86 platforms. This implies that the timer tick is a processor-local interrupt source that occurs on each processor. While not strictly required for event-driven schedulers, the version of Linux underlying LITMUS$^{\mathrm{RT}}$ does not support deactivating the periodic tick while a process is scheduled. It thus creates overhead that must be accounted for.

Example 3.10. Figure 3.15(a) shows a schedule of two jobs in the presence of timer ticks that occur every $Q = 5$ time units. All other sources of overheads have been omitted for clarity. At time 4, job J_2 is released and commences execution. It is stopped during $[5, 6)$ when the tick ISR executes and incurs cache-related interrupt delay thereafter while it re-establishes cache affinity. It is then



(a) Schedule with timer tick interrupts.

(b) Equivalent overhead-free FP schedule.

(c) Equivalent overhead-free EDF schedule.

Figure 3.15: Uniprocessor schedules illustrating timer tick delay accounting techniques. All other overheads are omitted for clarity; the impact of tick interrupts has been exaggerated. Timer ticks occur every $Q = 5$ time units. J_2 would require the processor for $e_2 = 6$ time units were it not delayed by timer ticks. **(a)** J_2 is delayed by a timer tick three times during its execution. While the tick ISR executes, J_2 is stopped. Afterwards, J_2 needs to reestablish cache affinity. By chance, J_1 is not affected by timer ticks due to its "lucky" release time. **(b)** Under FP and P-FP scheduling, the timer tick can be modeled as a higher-priority periodic task T_0^{tck} with a period of Q time units. **(c)** Under EDF-based schedulers, J_2 is inflated to account for two timer ticks that it incurs unconditionally, *i.e.*, even when executing in isolation, and also two additional timer ticks for the preemption due to J_1, which safely, but pessimistically, overestimates the additional delay due to the preemption.

preempted by J_1 at time 8. J_1 happens to be "lucky" in that it executes in between two quantum boundaries and thus is not delayed by a timer tick. However, the preemption causes J_2 to incur two more timer ticks at times 10 and 15.

Had J_2 not been preempted, it would have completed before time 15 and thus would have been spared a third delay. This is shown in Figure 3.16(a), which depicts J_2 assuming that it executes in isolation, *i.e.*, without being preempted. This example shows that higher-priority jobs can cause lower-priority jobs to incur additional timer ticks. ◊

P-FP scheduling. The recurring timer tick interrupt can be trivially modeled as the highest-priority task under FP and P-FP scheduling (Liu, 2000), which is illustrated in Figure 3.15(b). The possibility that a preemption can "push" a lower-priority job into additional timer ticks is implicitly accounted for when conducting response-time analysis.

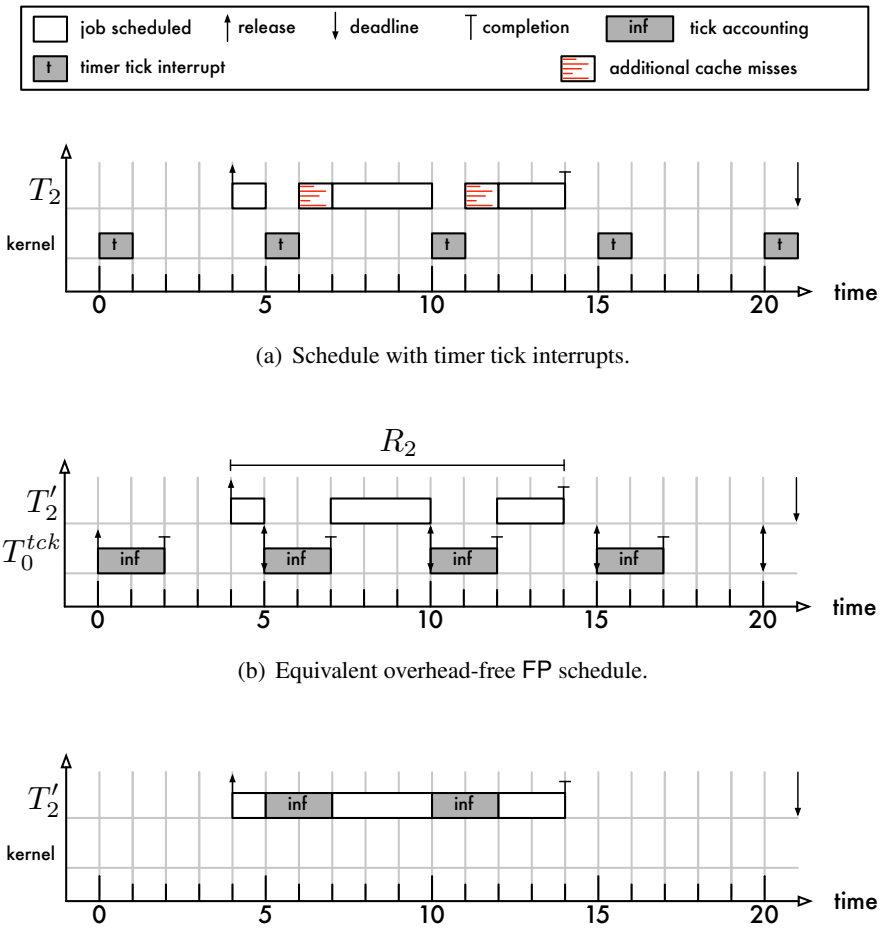
We let T_0^{tck} denote the task that models the periodic timer tick. Its execution requirement accounts for both the actual *timer tick ISR overhead*, which is denoted as Δ^{tck}, and the maximum cache-related interrupt delay (Δ^{cid}). In practice, it may be possible to derive values of Δ^{cid} specific to each interrupt source since ISRs likely differ in cache footprint. For the sake of simplicity, however, we do not differentiate between the loss of cache affinity due to a release interrupt and that due to a timer tick. This yields the following parameters for T_0^{tck}.

$$e_0^{tck} = \Delta^{tck} + \Delta^{cid} \qquad p_0^{tck} = Q \qquad susp_0^{tck} = \Delta^{ev} \tag{3.11}$$

As before in the case of release interrupts, a maximum self-suspension duration of Δ^{ev} is used to account for jitter.

EDF-based scheduling. The task-centric approach used to analyze release interrupts could be applied to timer ticks as well. The resulting bound, however, would be overly pessimistic since Q is typically much shorter than task periods. In (Brandenburg *et al.*, 2011), we presented task-centric analysis specific to timer ticks, which we summarize in detail since we use the same proof strategy in the derivation of preemption-centric accounting in Section 3.4.5 below.

The analysis consists of two steps. In the first step, it is assumed that the job under analysis executes in isolation, *i.e.*, without preemptions, which yields the number of timer ticks that are



(a) Schedule with timer tick interrupts.

(b) Equivalent overhead-free FP schedule.

(c) Equivalent overhead-free EDF schedule.

Figure 3.16: Illustration of timer tick overhead if J_2 is not preempted. The depicted scenario corresponds to Figure 3.15 assuming that T_1 is inactive. **(a)** When scheduled in isolation (*i.e.*, not preempted), J_2 is delayed by the effects of timer ticks twice. **(b)** If J_2 is not preempted, then the schedule is equivalent to a two-task FP schedule amenable to response-time analysis. **(c)** Since J_2 is preempted twice in the equivalent FP schedule, it is charged the cost of two timer tick interrupts.

unavoidable given its execution requirement. In the second step, the number of additional timer ticks that a job may incur per preemption is integrated into the bound derived in the first step.

In the absence of preemptions, the scheduling policy is irrelevant and the schedule can be converted to an equivalent FP schedule of two tasks T_2' and T_0^{tck}, as illustrated in Figure 3.16(b). Note that J_2' executes for exactly e_2 time units since the cache-related interrupt delay has been modeled as execution time of the timer tick ISR. Based on this equivalent overhead-free model, J_2's response-time increase due to timer ticks can be determined by applying regular FP response-time analysis to T_2'.

Recall that R_i denotes an upper bound on the worst-case response time. If a job J_i is not preempted by other jobs, *i.e.*, while pending, J_i is either executing or stopped by a timer tick, as illustrated in Figure 3.16(b), then R_i is given by the smallest value that satisfies

$$R_i = e_i + \left\lceil \frac{R_i + \Delta^{ev}}{Q} \right\rceil \cdot (\Delta^{tck} + \Delta^{cid}). \tag{3.12}$$

Equation (3.12) follows directly from Audsley *et al.*'s jitter-aware response-time bound for FP scheduling (Audsley *et al.*, 1993) since $p_0^{tck} = Q$ and $e_0^{tck} = \Delta^{tck} + \Delta^{cid}$.

For the special case of jobs that are not preempted, the response-time bound R_i can be used as a safe approximation of T_i by setting $e_i' = R_i$.

Example 3.11. In Figure 3.16(b) where $\Delta^{ev} = 0$, J_2's execution requirement of $e_2 = 6$ is inflated by the cost of two timer ticks since

$$\begin{aligned} R_2 &= e_2 + \left\lceil \frac{R_2 + \Delta^{ev}}{Q} \right\rceil \cdot (\Delta^{tck} + \Delta^{cid}) \\ &= 6 + \left\lceil \frac{10 + 0}{5} \right\rceil \cdot (1 + 1) = 6 + 2 \cdot 2 = 10. \end{aligned}$$

The resulting overhead-free task T_2' is illustrated in Figure 3.16(c). ◊

Next, we consider the impact of preemptions. As Figure 3.15(a) demonstrates, Equation (3.12) is no longer a safe approximation if jobs may be preempted. However, the number of additional timer ticks is bounded by the number of preemptions, as we illustrate in the following proof for the special case of a single preemption.

If a job J_i is preempted by other jobs once, then it can be thought of being comprised of two *subjobs* that execute without being preempted. For example, in Figure 3.15(a), J_2 consists of two subjobs that are scheduled during $[4, 8)$ and $[10, 18)$, respectively (recall that a job is stopped and not preempted when interrupted by a timer tick, and thus considered to be scheduled).

Let $e_{i,1}$ and $e_{i,2}$, where $e_i = e_{i,1} + e_{i,2}$, denote the execution times of the two subjobs in the equivalent FP schedule where all cache-related interrupt delay has been modeled as the execution of T_0^{tck}. For example, in Figure 3.15(b), $e_{2,1} = 2$ and $e_{2,2} = 4$.

Since each subjob is not preempted, we can apply Equation (3.12) to determine its response-time increase due to timer ticks. Let $R_{i,1}$ and $R_{i,2}$ denote the response times of the subjobs. In the scenario shown in Figure 3.15(b), $R_{2,1} = 4$ and $R_{2,2} = 8$.

Let E_i denote the total time that J_i is scheduled (either executing or stopped), which is equivalent to the sum of the response times of the two subjobs in the equivalent two-task FP schedule.

$$E_i = R_{i,1} + R_{i,2} \tag{3.13}$$

This yields the following recurrence.

$$\begin{aligned}
E_i &= R_{i,1} + R_{i,2} \\
&\quad \{ \text{ expanding Equation (3.12) } \} \\
&= e_{i,1} + \left\lceil \frac{R_{i,1} + \Delta^{ev}}{Q} \right\rceil \cdot (\Delta^{tck} + \Delta^{cid}) + e_{i,2} + \left\lceil \frac{R_{i,2} + \Delta^{ev}}{Q} \right\rceil \cdot (\Delta^{tck} + \Delta^{cid}) \\
&\quad \{ \text{ since } e_i = e_{i,1} + e_{i,2} \} \\
&= e_i + \left(\left\lceil \frac{R_{i,1} + \Delta^{ev}}{Q} \right\rceil + \left\lceil \frac{R_{i,2} + \Delta^{ev}}{Q} \right\rceil \right) \cdot (\Delta^{tck} + \Delta^{cid}) \\
&\quad \{ \text{ since } \lceil a \rceil + \lceil b \rceil \leq \lceil a + b \rceil + 1 \text{ for any } a, b \} \\
&\leq e_i + \left(\left\lceil \frac{R_{i,1} + R_{i,2} + \Delta^{ev}}{Q} + \frac{\Delta^{ev}}{Q} \right\rceil + 1 \right) \cdot (\Delta^{tck} + \Delta^{cid}) \\
&\quad \{ \text{ by Equation (3.13) } \} \\
&= e_i + \left(\left\lceil \frac{E_i + \Delta^{ev}}{Q} + \frac{\Delta^{ev}}{Q} \right\rceil + 1 \right) \cdot (\Delta^{tck} + \Delta^{cid})
\end{aligned}$$

The smallest E_i satisfying the above recurrence is hence a safe upper bound on the total time that J_i was scheduled or stopped by an interrupt. With this definition, it is possible to compute E_i directly without knowing at which point J_i was preempted during its execution. Note that the above recurrence is essentially Equation (3.12) with one additional timer tick and $\frac{\Delta^{ev}}{Q}$ additional latency accounted for.

The derivation of E_i can be generalized to an arbitrary number of subjobs, which results in the cost of one additional timer tick and $\frac{\Delta^{ev}}{Q}$ latency being charged for each preemption. Let η_i denote the maximum number of times that any J_i is preempted. A safe approximation with regard to timer ticks is then given by the smallest e'_i that satisfies

$$e'_i = e_i + \left(\left\lceil \frac{e'_i}{Q} + (\eta_i + 1) \cdot \frac{\Delta^{ev}}{Q} \right\rceil + \eta_i \right) \cdot (\Delta^{tck} + \Delta^{cid}). \tag{3.14}$$

Under EDF-based schedulers, J_i is only preempted by another job J_h if J_h arrived later than J_i and has a shorter deadline, which implies that $d_h < d_i$. If d_h is not an integer multiple of d_i, then at most $\left\lfloor \frac{d_i}{d_h} \right\rfloor$ jobs of T_h with earlier absolute deadlines are released while J_i is pending (Liu and Layland, 1973). This also holds if J_i is tardy since newly-released jobs cannot preempt a tardy job (since a tardy job's absolute deadline is in the past). Otherwise, if d_h is an integer multiple of d_i, then only $\left\lfloor \frac{d_i}{d_h} \right\rfloor - 1$ jobs of T_h can preempt J_h. The two cases can be equivalently expressed as $\left\lceil \frac{d_i}{d_h} \right\rceil - 1$, which yields the following bound on η_i:

$$\eta_i \leq \sum_{T_h \in \tau}^{i \neq h} \left(\left\lceil \frac{d_i}{d_h} \right\rceil - 1 \right). \tag{3.15}$$

Note that if $\Delta^{ev} = 0$ and $\Delta^{cid} = 0$, then Equation (3.14) reduces to $e'_i = e_i + \left(\left\lceil \frac{e'_i}{Q} \right\rceil + \eta_i \right) \cdot \Delta^{tck}$, which corresponds to the bound stated in (Brandenburg *et al.*, 2011).

Example 3.12. Figure 3.15(c) illustrates the equivalent, overhead-free T'_2 based on Equation (3.14). In the depicted scenario, $e_2 = 6$, $d_2 = 17$, and $d_1 = 12$. J_2 is thus preempted at most $\eta_2 = \left\lceil \frac{17}{12} \right\rceil - 1 = 1$ times, which is indeed the case in Figure 3.15. This yields the following safe

approximation:

$$\begin{aligned} e_2' = 14 &= e_i + \left(\left\lceil \frac{e_i'}{Q} + (\eta_i + 1) \cdot \frac{\Delta^{ev}}{Q} \right\rceil + \eta_i \right) \cdot (\Delta^{tck} + \Delta^{cid}) \\ &= 6 + \left(\left\lceil \frac{14}{5} + 2 \cdot \frac{0}{5} \right\rceil + 1 \right) \cdot (1 + 1) \\ &= 6 + 4 \cdot 2 \end{aligned}$$

That is, e_2 is inflated to account for up to four timer ticks. Note that J_2' is actually scheduled for only 12 time units in Figure 3.15(a) since J_2 is only delayed by three timer ticks. This illustrates that the approximation is safe but not exact. $\Diamond$

A disadvantage of Equation (3.14) is that it requires knowledge of the maximum number of preemptions that any job of a task can incur. Since η_i is part of the recurrence, it is not immediately possible to charge it to the preempting task as is the case with the scheduling- and cache-related costs inherent in a preemption. Therefore, Equation (3.15) is used to bound the maximum number of preemptions per job. Unfortunately, a tighter bound than Equation (3.15) is not possible in general as one can construct schedules where it is accurate for some T_i.

However, charging every task for all job releases of shorter-deadline tasks is severely pessimistic. To illustrate this, suppose that tasks are indexed in order of increasing relative deadlines. Then each T_i is inflated to account for $\Omega(i)$ preemptions (unless some relative deadlines are equal). This effectively assumes that *each* lower-priority job is preempted every time that a higher-priority job is released. Since a higher-priority job causes at most one job to be preempted upon release, this is overly pessimistic.

In fact, charging each T_i for the effect of $\Omega(i)$ preemptions implies that $\Omega(n)$ tasks are charged for $\Omega(n)$ preemptions. Across all tasks, this in turn results in an $\Omega(n^2)$ increase in task set utilization. To summarize, the task-centric analysis of timer ticks is less pessimistic than the task-centric analysis of release interrupts because J_i is charged only for timer ticks that occur while it is executing, and not for all interrupts that occur while it is pending. That is, Equation (3.14) is based on e_i, whereas Equation (3.9) is based on R_i (and $R_i \geq e_i$). However, the improved analysis specific to timer ticks is asymptotically just as wasteful as the general task-centric method since the number of preemptions

η_i must be bounded explicitly, which gives rise to pessimism. As we show next, this limitation can be overcome by using a more coarse-grained response-time approximation.

3.4.5 Preemption-Centric Interrupt Accounting

In this section, we introduce *preemption-centric* interrupt accounting, a novel interrupt accounting method that modifies the above-described task-centric approach to be less pessimistic with regard to the number of preemptions. In particular, the key property of the preemption-centric approach is that preemption-related interrupt delays are accounted similar to CPMD, that is, the preempting job is charged for additional delays and not the preempted job.

Preemption-centric interrupt accounting is structurally very similar to the task-centric timer tick analysis. It uses the same approach, namely, to analyze the response time of subjobs that execute preemption-free in an equivalent FP schedule. There are only two significant differences: first, we apply the subjob analysis to both release interrupts and timer ticks at the same time (to avoid having to inflate twice), and second, we use a simple, less-accurate upper bound on response time, which simplifies the algebraic manipulation of subjob response times.

The proof is analogous to the argument underlying the preceding analysis of timer ticks. In the first step, we consider a job J_i that is not preempted and bound its response time. In the second step, we assume that J_i is preempted η times and bound the total time that it is executing or stopped by combining the response times of its $\eta + 1$ subjobs. Finally, we charge the costs of additional interrupts to the preempting task, which results in an $O(1)$ charge per interrupt source (instead of the $\Omega(n)$ inherent in task-centric accounting).

Step 1. As before, let J_i denote a job that is not preempted by other jobs. While J_i is pending, it is thus either executing or stopped due to an interrupt. In total, there are $n + 1$ interrupt sources that can delay J_i: n release interrupt sources (one for each task) and one timer tick. While there may be additional timer tick sources on other processors, J_i does not migrate since J_i is not preempted. Only the processor-local timer tick is thus relevant.

Consider an equivalent overhead-free FP schedule of T_i and $n + 1$ ISR tasks in which all cache-related interrupt delays are modeled as execution times of ISRs. Let T_0^{tck} denote the task corresponding to the timer tick, and let T_j^{irq}, where $1 \leq j \leq n$, denote the task modeling the interrupt

source that releases jobs of T_j. In other words, consider the task set $\tau^{\mathsf{FP}} = \{T_0^{tck}, T_1^{irq}, \ldots, T_n^{irq}, T_i\}$, where T_i is the lowest-priority task and the relevant parameters of ISR tasks are as follows.

$$e_0^{tck} = \Delta^{tck} + \Delta^{cid} \qquad p_0^{tck} = Q \qquad u_0^{tck} = \frac{\Delta^{tck} + \Delta^{cid}}{Q}$$

$$e_j^{irq} = \Delta^{rel} + \Delta^{cid} \qquad p_j^{irq} = p_j \qquad u_j^{irq} = \frac{\Delta^{tck} + \Delta^{cid}}{p_j}$$

Applying Audsley *et al.*'s jitter-aware response-time bound for FP scheduling (Audsley *et al.*, 1993) to τ^{FP} yields the following response-time bound for J_i since T_i is the lowest-priority task.

$$R_i = e_i + \left\lceil \frac{R_i + \Delta^{ev}}{Q} \right\rceil \cdot e_0^{tck} + \sum_{1 \le j \le n} \left\lceil \frac{R_i + \Delta^{ev}}{p_j} \right\rceil \cdot e_j^{irq} \tag{3.16}$$

Next, we bound R_i with a non-recurrent approximation by substituting each ceiling with an upper bound, which is a well-known method (Sjödin and Hansson, 1998) but included here for the sake of completeness.

Consider the rate at which J_i accumulates processor service. While J_i is scheduled and executing, it receives service at a rate of 1. However, while J_i is stopped, it does not receive any service, *i.e.*, the rate of service accumulation is 0. When the total amount of service accumulated by J_i reaches e_i, then J_i completes (at the latest). Upper-bounding the response time of J_i requires thus lower-bounding the amount of service that J_i receives over time.

This can be accomplished with a linear lower bound on the amount of processor service received by J_i that is offset by some constant c^{pre} from the origin, as illustrated in Figure 3.17. A linear function (the dashed curve in Figure 3.17) bounds the curve representing actually received service from below. Let s denote the slope of the linear lower bound on service, *i.e.*, on average, J_i receives at least s time units worth of service during each time unit. Then J_i completes after at most $\frac{e_i}{s}$ time units after c^{pre}. For example, Figure 3.17, the offset is 4, $e_i = 8$, and $s = \frac{2}{3}$. J_i thus completes after at most $4 + \frac{8 \cdot 3}{2} = 4 + 12 = 16$ time units. Bounding R_i thus consists of determining s and c^{pre}.

In the following, we first formally define $\frac{e_i}{s}$ and c^{pre} and then show why they arise in the following derivation.

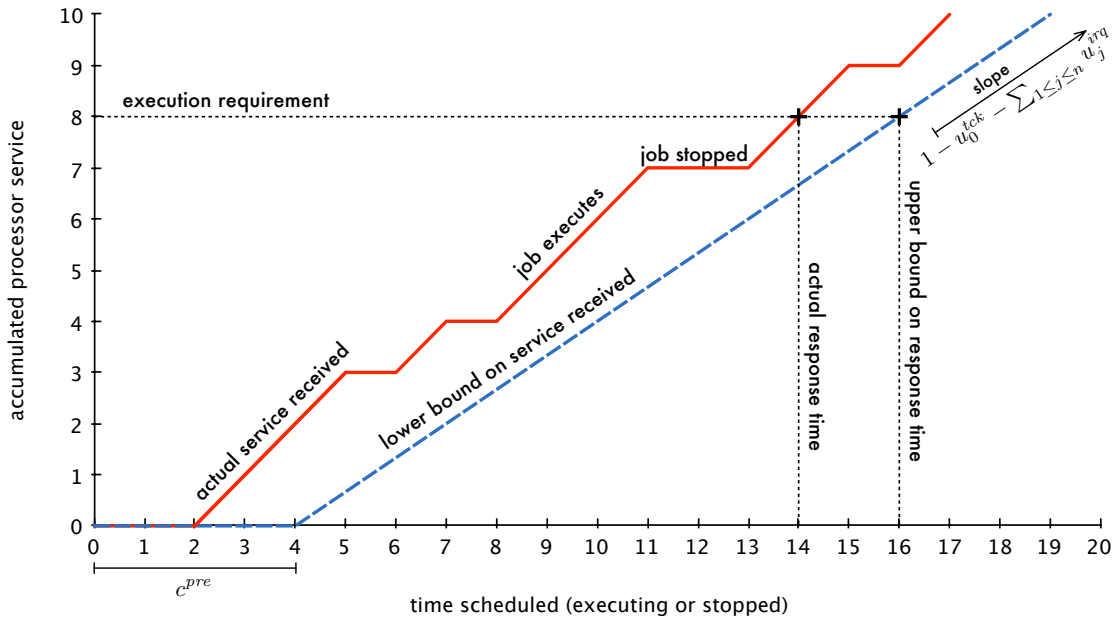


Figure 3.17: Illustration of response-time bound for preemption-centric interrupt accounting. The graph illustrates the amount of processor service received by a job J_i that is not preempted but stopped by interrupts. The solid curve depicts the cumulative processor service actually received by J_i. It has a slope of 1 when J_i is executing, and a slope of 0 while J_i is stopped. The dashed curve depicts the lower bound on processor service used for preemption-centric interrupt accounting. The lower bound has a slope of $1 - u_0^{tck} - \sum_{1 \le j \le n} u_j^{irq}$. The actual response time and the response-time bound are determined by the intersection of the service curves and the execution requirement $e_i = 8$. The additional charge c^{pre} corresponds to the offset of the lower bound curve and is required to account for the possibility that J_i is stopped immediately.

Definition 3.2. Let e_i^{inf} denote T_i's inflated execution requirement, defined as

$$e_i^{inf} \triangleq \frac{e_i}{1 - u_0^{tck} - \sum_{1 \le j \le n} u_j^{irq}}, \tag{3.17}$$

and let c^{pre} denote the cost of one preemption (with regard to interrupts), defined as

$$c^{pre} \triangleq \frac{e_0^{tck} + \Delta^{ev} \cdot u_0^{tck} + \sum_{1 \le j \le n} \left(\Delta^{ev} \cdot u_j^{irq} + e_j^{irq} \right)}{1 - u_0^{tck} - \sum_{1 \le j \le n} u_j^{irq}}. \tag{3.18}$$

Note that c^{pre} is independent of T_i, *i.e.*, it is a task-set-specific constant but not task-specific. This will allow us to charge preempting jobs for the additional interrupts incurred by the preempted

job, as will become apparent in Step 2 of the proof below. The definitions of e_i^{inf} and c^{pre} are chosen such that Lemma 3.1 below holds.

Definition 3.3. Let E_i denote the total amount of time that J_i is either executing or stopped.

Lemma 3.1. *If J_i is not preempted, then $E_i \leq e_i^{inf} + c^{pre}$.*

Proof. Since J_i is not preempted, E_i corresponds to J_i's response time, *i.e.*, $E_i = R_i$. The stated bound can be derived in a straightforward manner starting with Equation (3.16).

$$R_i = e_i + \left\lceil \frac{R_i + \Delta^{ev}}{Q} \right\rceil \cdot e_0^{tck} + \sum_{1 \leq j \leq n} \left\lceil \frac{R_i + \Delta^{ev}}{p_j} \right\rceil \cdot e_j^{irq}$$

{ converting ceiling to upper bound }

$$\Rightarrow \quad R_i \leq e_i + \left(\frac{R_i + \Delta^{ev}}{Q} + 1 \right) \cdot e_0^{tck} + \sum_{1 \leq j \leq n} \left(\frac{R_i + \Delta^{ev}}{p_j} + 1 \right) \cdot e_j^{irq}$$

{ multiplying out and converting to utilization }

$$\Leftrightarrow \quad R_i \leq e_i + R_i \cdot u_0^{tck} + \Delta^{ev} \cdot u_0^{tck} + e_0^{tck} + \sum_{1 \leq j \leq n} \left(R_i \cdot u_j^{irq} + \Delta^{ev} \cdot u_j^{irq} + e_j^{irq} \right)$$

{ factoring out R_i }

$$\Leftrightarrow \quad R_i \cdot \left(1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq} \right) \leq e_i + \Delta^{ev} \cdot u_0^{tck} + e_0^{tck} + \sum_{1 \leq j \leq n} \left(\Delta^{ev} \cdot u_j^{irq} + e_j^{irq} \right)$$

{ rearranging }

$$\Leftrightarrow \quad R_i \leq \frac{e_i + e_0^{tck} + \Delta^{ev} \cdot u_0^{tck} + \sum_{1 \leq j \leq n} \left(\Delta^{ev} \cdot u_j^{irq} + e_j^{irq} \right)}{1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}}$$

{ separating costs }

$$\Leftrightarrow \quad R_i \leq \frac{e_i}{1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}} + \frac{e_0^{tck} + \Delta^{ev} \cdot u_0^{tck} + \sum_{1 \leq j \leq n} \left(\Delta^{ev} \cdot u_j^{irq} + e_j^{irq} \right)}{1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}}$$

{ applying Equation (3.17) and Equation (3.18) }

$$\Leftrightarrow \quad R_i \leq e_i^{inf} + c^{pre}$$

And thus also $E_i \leq e_i^{inf} + c^{pre}$. □

Step 2. When J_i is preempted, it may incur additional interrupts. As we show next, the additional delay is limited to c^{pre} per preemption. The argument is analogous to the task-centric analysis of timer ticks in that each subjob is analyzed individually as a job that is not preempted.

Let η_i denote the number of times that J_i is preempted. J_i's execution is then comprised of $(\eta_i + 1)$ subjobs. Let $e_{i,x}$ denote the amount of execution carried out by the x^{th} subjob in the overhead-free equivalent FP schedule, where $1 \leq x \leq \eta_i + 1$ and

$$e_i = \sum_x e_{i,x}, \tag{3.19}$$

and let $E_{i,x}$ denote the total duration that the x^{th} subjob is scheduled, *i.e.*, either executing or stopped.

Lemma 3.2. *If J_i is preempted at most η_i times, then $E_i \leq e_i^{inf} + (\eta_i + 1) \cdot c^{pre}$.*

Proof. The total duration that J_i is either scheduled or stopped is comprised of the sum of the response time of each subjob, *i.e.*, $E_i = \sum_{x=1}^{\eta_i+1} E_{i,x}$. This yields the following inequality.

$$\begin{aligned}
E_i &= \sum_{x=1}^{\eta_i+1} E_{i,x} \\
&\quad \{ \text{ by Lemma 3.1 with respect to each } E_{i,x} \text{ since subjobs are not preempted } \} \\
&\leq \sum_{x=1}^{\eta_i+1} (e_{i,x}^{inf} + c^{pre}) \\
&\quad \{ \text{ by Equation (3.17) and pulling out } c^{pre}, \text{ which is independent of } x \} \\
&= \sum_{x=1}^{\eta_i+1} \left(\frac{e_{i,x}}{1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}} \right) \quad + \quad (\eta_i + 1) \cdot c^{pre} \\
&\quad \{ \text{ pulling out divisor } \} \\
&= \frac{\sum_{x=1}^{\eta_i+1} e_{i,x}}{1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}} \quad + \quad (\eta_i + 1) \cdot c^{pre} \\
&\quad \{ \text{ by Equation (3.19) } \} \\
&= \frac{e_i}{1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}} \quad + \quad (\eta_i + 1) \cdot c^{pre} \\
&\quad \{ \text{ by Equation (3.17) } \} \\
&= e_i^{inf} + (\eta_i + 1) \cdot c^{pre}
\end{aligned}$$

□

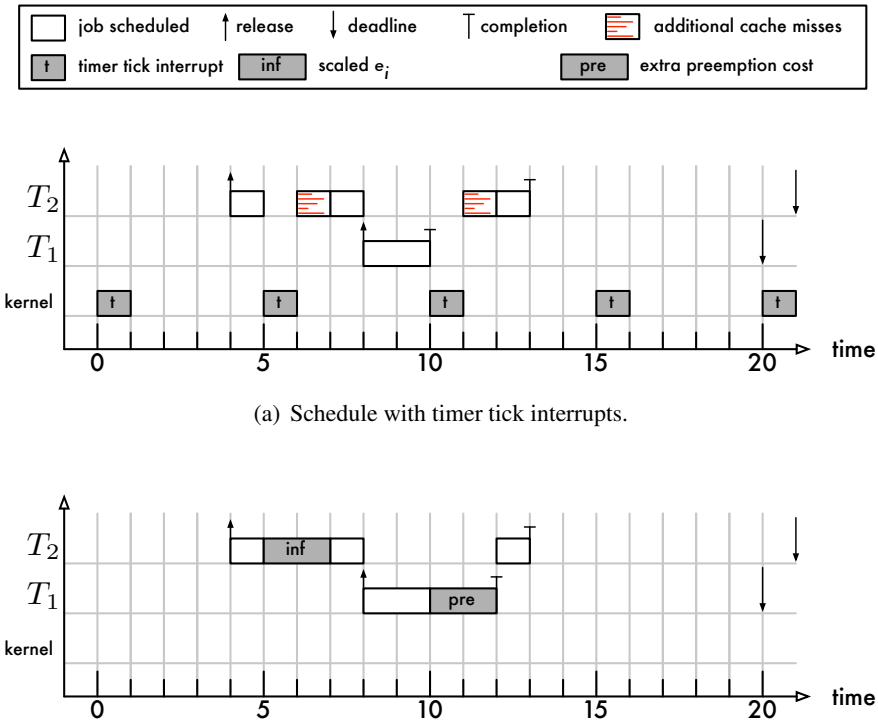
In other words, each time that J_i is preempted, its effective execution requirement increases by at most a constant amount c^{pre}. It is important to realize that c^{pre} is not task-specific, *i.e.*, it does not depend on the identity of J_i. This allows the increase in execution time to be attributed to the preempting job, analogously to how scheduling and context-switch overheads are charged to the preempting job. That is, the preempting job is charged c^{pre} time units to account for additional delays incurred by the preempted job. If jobs migrate, then c^{pre} functions similar to the post-job charge in Figure 3.13. We first consider the uniprocessor case; an example with migrations is presented thereafter.

Example 3.13. Consider the uniprocessor example of timer tick overhead shown in Figure 3.18, where $n = 2$, $e_1 = 2$, $e_2 = 3$, $\Delta^{tck} = 1$, $\Delta^{cid} = 1$, and $Q = 5$ (and all other overheads are omitted for clarity). This implies $u_0^{tick} = \frac{2}{5} = 0.4$. Using preemption-centric accounting, the delay due to timer ticks is bounded as follows. By Equations (3.17) and (3.18), we have

$$e_1^{inf} = \frac{2}{1 - \frac{2}{5}} = \frac{10}{3}, \qquad e_2^{inf} = \frac{3}{1 - \frac{2}{5}} = 5, \quad \text{and} \qquad c^{pre} = \frac{2}{1 - \frac{2}{5}} = \frac{10}{3} \geq 2.$$

In the schedule shown in Figure 3.18(a), J_2 is delayed by two timer ticks. Note that e_2^{inf} is inflated by $e_2^{inf} - e_2 = 2$ time units, which is sufficient to account for one timer tick. This inflation is labeled in Figure 3.18(b) as "inf." The second timer tick delay, however, is charged to J_1 since it preempted J_2. An additional charge of c^{pre} is sufficient to model J_2's additional delay of two time units. In Figure 3.18(b), the latter preemption-related charge is labeled "pre." ◊

In general, suppose a job J_i preempts a lower-priority job J_l at the time of J_i's release. J_i requires a budget of $e_i^{inf} + c^{pre}$ time units for its own execution (Lemma 3.1), plus an additional c^{pre} time units to compensate for the delay induced in J_l by J_i's release. At the time of its release, J_i is charged c^{pre} to account for the additional delay that is incurred by J_l, just like J_i is also charged for additional scheduling and context-switch overheads. This leaves a budget of $e_i^{inf} + c^{pre}$ time units for J_i to finish. By Lemma 3.1, J_i executes for at most $e_i^{inf} + c^{pre}$ time units if it is not preempted; the budget is thus sufficient in this case. If J_i is preempted $\eta_i > 0$ times, then there exist η_i jobs that



(a) Schedule with timer tick interrupts.

(b) Equivalent overhead-free EDF schedule under preemption-centric accounting.

Figure 3.18: Uniprocessor schedules illustrating preemption-centric interrupt accounting. The impact of tick interrupts has been exaggerated; all other overheads have been omitted for clarity. Timer ticks occur every $Q = 5$ time units. J_2 would require the processor for $e_2 = 3$ time units were it not delayed by timer ticks. **(a)** J_2 is delayed by a timer tick two times during its execution because it is preempted by J_1. **(b)** Under preemption-centric accounting, J_2's inflated execution cost e_2^{inf} accounts for the first timer tick delay. The second timer tick delay is charged to J_1's execution time, which has been inflated by c^{pre} time units to account for this possibility.

are each charged c^{pre} time units to J_i's budget each time that J_i is preempted. The total budget for J_i is thus $e_i^{inf} + c^{pre} + \eta_i \cdot c^{pre}$, which by Lemma 3.2 is sufficient for J_i to complete.

Lemma 3.3. *A task set τ' is a safe approximation of a task set τ with regard to release interrupts and timer ticks if $e'_i \geq e_i^{inf} + 2 \cdot c^{pre}$ (and $p'_i \leq p_i$ and $d'_i \leq d_i$) for each $T'_i \in \tau'$*

Proof. Follows from the preceding discussion. □

Compared to task-centric interrupt accounting, preemption-centric accounting has two major advantages. First, the required inflation is straightforward to compute, *i.e.*, there is no fix-point iteration required. Second, ignoring the effect of latency, *i.e.*, assumnig $\Delta^{ev} = 0$, c^{pre} amounts to charging one additional ISR execution for each interrupt source (the sum of which is then scaled).

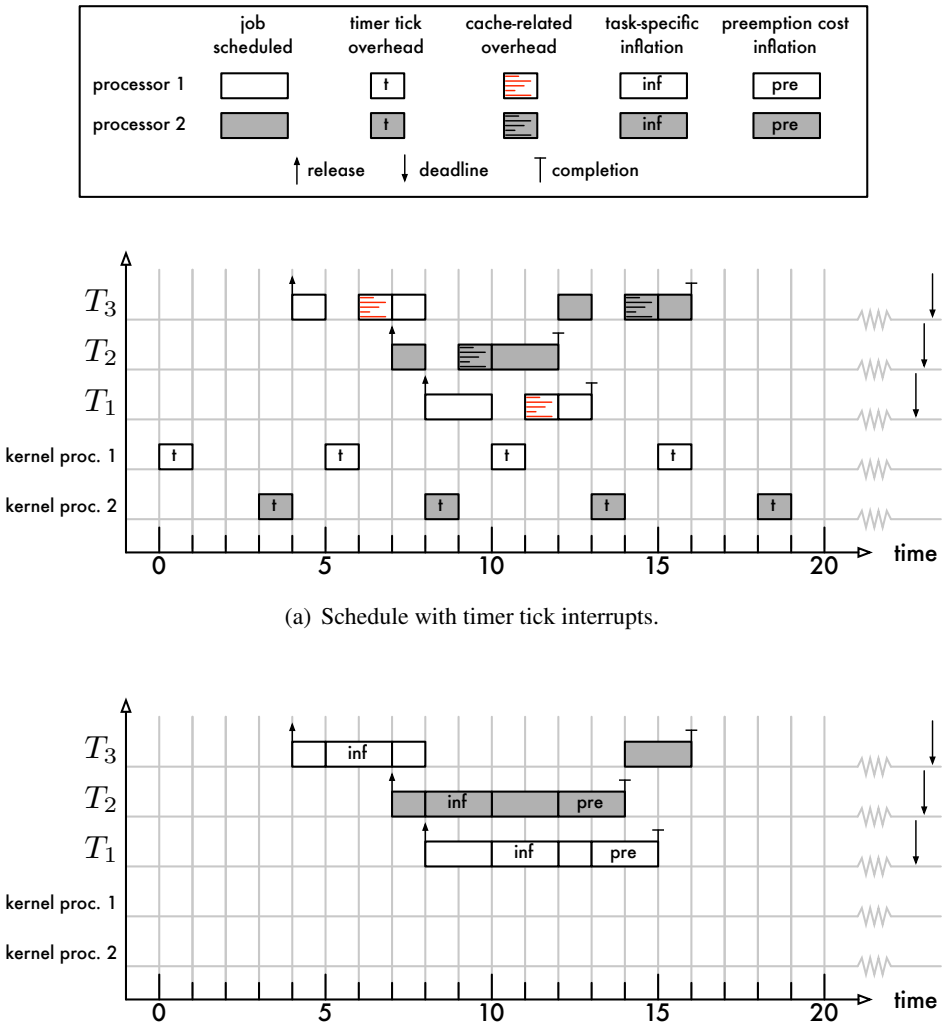
It is trivial to construct schedules where a job J_i is indeed delayed by an additional ISR from each interrupt source each time that it is preempted. The derived bound is thus fairly tight.

Some pessimism, however, arises due to the inaccuracy of the upper bound on subjob response times introduced by Lemma 3.1. That is, the scaling of the additional ISR charges overestimates the maximum delay. This could be reduced by re-deriving E_i using a tighter response-time bound for each R_i. For example, Bini and Baruah (2007) recently presented an improved response-time bound that is compatible with preemption-centric interrupt accounting. Unfortunately, Bini and Baruah's response-time bound does not take jitter into account and is thus not suitable for our overhead-aware approach (though it would likely be possible to extend their result to take latencies into account).

Migrations. While we have only discussed preemptions so far, the derived bound does not depend on whether J_i continues execution on the same processor. Since each subjob is analyzed individually and since subjobs never migrate, Lemma 3.2 applies also to global schedulers (and also to staggered quanta, *i.e.*, timer ticks do not have to occur synchronously on all processors). This is illustrated in Figure 3.19, which shows schedules of three jobs on two processors under G-EDF.

Example 3.14. In Figure 3.19(a), job J_3 is delayed by timer ticks twice. It is first delayed at time 5 when the timer tick fires on processor 1. Note that J_1 is stopped even though processor 2 is idle at the time. This illustrates that interrupts cannot simply be modeled as tasks under global scheduling (including G-FP). At time 8, J_3 is preempted by J_1; at time 12, J_3 migrates to processor 2 when J_2 completes. As a result, J_3 incurs additional timer tick delay during $[13, 15)$ when the timer tick fires on processor 2.

An equivalent overhead-free G-EDF schedule is depicted in Figure 3.19(b). Each of the three jobs incurs at least one timer interrupt over the course of its execution, regardless of preemptions. This is adequately accounted for by each e_i^{inf}. The delayed completion of J_3 due to the second timer tick can be equivalently attributed to the additional charge of c^{pre} time units that is added to *each* job. (In fact, c^{pre} exceeds the two time units shown in Figure 3.19(b) since preemption-centric interrupt accounting is somewhat pessimistic.) This example illustrates that interrupt costs can be charged similarly to CPMD as a post-job charge (recall Figure 3.13), even if jobs migrate, provided that *all* tasks are charged c^{pre} time units to account for interrupt-related preemption costs. $\Diamond$



(a) Schedule with timer tick interrupts.

(b) Equivalent overhead-free EDF schedule under preemption-centric accounting.

Figure 3.19: G-EDF schedules of three jobs on two processors illustrating preemption-centric interrupt accounting in the presence of migrations. The impact of tick interrupts has been exaggerated; all other overheads have been omitted for clarity. Timer ticks occur every $Q = 5$ time units and are not aligned. **(a)** J_3 is twice delayed by a timer tick; once on the processor on which it started execution, and once after migrating to processor 2. **(b)** J_3's inflated execution cost accounts for the first timer tick delay. The second timer tick delay can be attributed to the additional charges of c^{pre} time units to both J_1 and J_2's execution requirements.

Dedicated interrupt handling. As discussed in Section 2.5.1.4, delays due to ISR execution can be avoided by employing a systems processor for dedicated interrupt handling (Stankovic and Ramamritham, 1991). This has the benefit that release interrupts manifest only as release delay, but not as asynchronous interrupts since executing jobs are never stopped by them (recall that jobs are only scheduled on application processors and not the systems processor). Consequently, preemption-centric interrupt accounting is only required for timer ticks under dedicated interrupt handling, which greatly reduces the pessimism inherent in interrupt accounting (since timer ticks are typically much shorter ISRs than release interrupts), albeit at the cost of making the capacity of the systems processor unavailable to real-time tasks and, in the case of partitioned schedulers, also at the cost of introducing IPI latency. We explore this tradeoff in our case study in Chapter 4.

Another issue that is possibly exacerbated by dedicated interrupt handling is that if the systems processor is handling many interrupt sources, then the execution of one ISR can delay the handling of a release interrupt of a high-priority job. We assume that such delays are reflected in the worst-case event latency; a detailed discussion in the context of arbitrary interrupt sources can be found in (Brandenburg *et al*., 2011).

This concludes our discussion of overhead accounting techniques for event-driven scheduling. We summarize the presented safe approximations next.

3.4.6 Schedulability Analysis

Prior to testing whether a task set τ is schedulable on a given platform, it is transformed into a safe approximation τ' based on the overhead accounting techniques discussed in the preceding sections. If τ' passes an overhead-unaware schedulability test, then τ is schedulable in the presence of the modeled overheads. Table 3.2 summarizes the major sources of scheduling overhead.

Four combinations of event-driven scheduling are considered in this dissertation: P-FP scheduling with and without dedicated interrupt handling, and C-EDF scheduling with and without dedicated interrupt handling, which subsumes the special cases of P-EDF and C-EDF. For each of these combinations, we next summarize how to derive the safe approximation τ' prior to applying one of the overhead-unaware schedulability tests reviewed in Section 2.3. In the case of partitioned and clustered scheduling, overhead-accounting is applied after the task assignment phase on a cluster-by-cluster

Notation	Overhead	Accounts for
Δ^{ev}	event latency	delay until ISR starts execution
Δ^{ipi}	IPI latency	delay until IPI is received
Δ^{rel}	release overhead	execution of a release ISR
Δ^{tck}	timer tick overhead	execution of a timer tick ISR
Δ^{sch}	scheduling overhead	process selection
Δ^{cxs}	context-switch overhead	process switching
Δ^{cpd}	cache-related preemption and migration delay	loss of cache affinity due to job
Δ^{cid}	cache-related interrupt delay	loss of cache affinity due to ISR

Table 3.2: Summary of the overheads that arise in event-driven schedulers and our notation.

basis. To avoid notational clutter, we assume in the following that $\tau = \{T_1, \ldots, T_n\}$ represents one of the clusters.

A possible effect of inflating for overheads is that tasks become ill-defined. For example, if $e'_i > p'_i$ for some T'_i or if $u_{\text{sum}}(\tau') \geq m$, then τ' is clearly not feasible. As a result, τ is claimed unschedulable. Another issue arises with preemption-centric interrupt accounting, which by Equation (3.17) scales execution costs by the factor $\left(1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}\right)^{-1}$. In the unlikely case that $\left(u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}\right) \geq 1$, a processor could be continuously servicing ISRs. In this case, it cannot be ruled out that jobs starve due to interrupt overload and the scaled execution time e_i^{inf} becomes nonsensical or undefined, which causes τ to be claimed unschedulable.

Recall that non-preemptive sections can mask the effect of IPI latency (page 229). To account for this beneficial effect, we let b_i^{np} denote maximum pi-blocking incurred by any J_i upon release due to non-preemptive sections, and let b_i^{other} denote maximum pi-blocking due to other causes (such that $b_i = b_i^{other} + b_i^{np}$).

P-FP. Each task $T_i \in \tau$ is transformed to account for release delay and preemption costs. Since interrupts are modeled as additional tasks, T_i is not directly changed to account for them. This yields the following parameters for task $T'_i \in \tau'$.

$$e'_i = e_i + 2 \cdot (\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd} \qquad \{ \text{ Inequality (3.7) } \}$$

$$p'_i = p_i$$

$$d'_i = d_i$$

$$susp'_i = susp_i + (\Delta^{ev} + \Delta^{rel} + \Delta^{ipi}) \qquad \{ \text{Inequality (3.1)} \}$$

$$b'_i = b_i^{other} + \max(0, b_i^{np} - \Delta^{ipi}) \qquad \{ \text{Inequality (3.6)} \}$$

If dedicated interrupt handling is used, then $\Delta^{ipi} > 0$. Otherwise, if each processor is involved in interrupt handling, then IPI latency is irrelevant and the T'_i's parameters are determined assuming $\Delta^{ipi} = 0$.

To account for ISR execution, tasks T_0^{tck} and $T_1^{irq}, \ldots, T_n^{irq}$ are added to τ' to model the execution of timer tick and release ISRs. These ISR tasks have the following parameters.

$$e_0^{tck} = \Delta^{tck} + \Delta^{cid} \qquad p_0^{tck} = Q \qquad susp_0^{tck} = \Delta^{ev} \qquad \{ \text{Equations (3.8)} \}$$

$$e_i^{irq} = \Delta^{rel} + \Delta^{cid} \qquad p_i^{irq} = p_i \qquad susp_i^{irq} = \Delta^{ev} \qquad \{ \text{Equations (3.11)} \}$$

Pi-blocking and deadlines are irrelevant for ISR tasks. When using dedicated interrupt handling, the resulting safe approximation is given by

$$\tau' = \{T_0^{tck}, T'_1, \ldots, T'_n\}.$$

Otherwise, if each processor handles release interrupts of the tasks assigned to it, then

$$\tau' = \{T_0^{tck}, T_1^{irq}, \ldots, T_n^{irq}, T'_1, \ldots, T'_n\}.$$

In both cases, ISR tasks are presumed to have higher priority than regular tasks.

C-EDF. Under EDF-based schedulers, no tasks are added and each T_i's parameters are transformed to account for release delay, preemption delays, and interrupts. We employ preemption-centric interrupt accounting. We first consider the case that each processor handles release interrupts, *i.e.*, C-EDF without dedicated interrupt handling.

Preemption-centric interrupt accounting works by scaling each e_i to account for the slow-down due to interrupts. When integrating preemption-centric interrupt accounting with the accounting for CPMD, the scaling of e_i is performed *after* accounting for CPMD. This is required to reflect that CPMD increases the time that a job executes, and thus increases the exposure to interrupts. The

scaling for interrupt costs is compatible with charging the preempting job instead of the preempted job since the scaling factor $\left(1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}\right)$ does not depend on the identity of the executing task. Therefore, the cost of a preemption, namely scheduling overhead, context-switch overhead, and CPMD, can still be charged to the preempting job under preemption-centric interrupt accounting. IPI latency that is part of the release delay is not affected by this since it does not correspond to actual execution.

Formally, let $u_0^{tck} = \frac{\Delta^{tck} + \Delta^{cid}}{Q}$ and $u_j^{irq} = \frac{\Delta^{tck} + \Delta^{cid}}{p_j}$, and define c^{pre} as specified in Definition 3.2 on page 251. In the first step, e_i is inflated to account for preemption-related non-interrupt overheads, *i.e.*, to satisfy Inequality (3.7), let

$$e_i^{[1]} \triangleq e_i + 2 \cdot (\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd}.$$

In the second step $e_i^{[1]}$ is then scaled to account for interrupts, *i.e.*, by Lemma 3.3 (page 256), let

$$e_i^{[2]} \triangleq \frac{e_i^{[1]}}{1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}} + 2 \cdot c^{pre}.$$

Finally, $e_i^{[2]}$ is inflated to account for release delay according to Inequality (3.5), *i.e.*, $e_i' = e_i^{[2]} + \Delta^{ipi}$. Note that Δ^{rel} is not explicitly added as in Inequality (3.5) since it is already implicitly included in $e_i^{[1]}$: the preemption-centric approach considers *all* release interrupts that occur while a job J_i is pending, including the one that released J_i.

These considerations lead to the following definition of $\tau' = \{T_i' \mid T_i \in \tau\}$, where each T_i' is defined by the following parameters.

$$e_i' = \frac{e_i + 2 \cdot (\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd}}{1 - u_0^{tck} - \sum_{1 \leq j \leq n} u_j^{irq}} + 2 \cdot c^{pre} + \Delta^{ipi} \qquad \{ \text{ Inequalities (3.7) and (3.5) and Lemma 3.3 } \}$$

$$p_i' = p_i - \Delta^{ev} \qquad \{ \text{ Inequality (3.3) } \}$$

$$d_i' = d_i - \Delta^{ev} \qquad \{ \text{ Inequality (3.4) } \}$$

$$b_i' = b_i^{other} + \max(0, b_i^{np} - \Delta^{ipi}) \qquad \{ \text{ Inequality (3.6) } \}$$

Since the underlying overhead-unaware EDF schedulability tests do not support analysis of self-suspensions, the parameter $susp_i$ is irrelevant. As a special case, P-EDF is not affected by IPI latency if each task's release interrupts are serviced by the processor to which it has been assigned; it is therefore safe to assume $\Delta^{ipi} = 0$ in this case.

Under C-EDF with dedicated interrupt handling, release interrupts no longer contribute to (asynchronous) interrupt delay, but are still reflected by (synchronous) release delay and by timer ticks. Consequently, the execution times are inflated for preemptions as before (*i.e.*, $e_i^{[1]}$ remains unchanged) and preemption-centric interrupt accounting is still applied, but release interrupts are not accounted for (since they do not occur on application processors). This changes the inflated execution cost and the interrupt-related cost of a preemption as follows.

$$c^{pre} = \frac{e_0^{tck} + \Delta^{ev} \cdot u_0^{tck}}{1 - u_0^{tck}}$$
$$e_i' = \frac{e_i + 2 \cdot (\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd}}{1 - u_0^{tck}} + 2 \cdot c^{pre} + \Delta^{ipi} + \Delta^{rel}$$

The other parameters of each T_i' remain unchanged. Note that Δ^{rel} is charged once to account for release delay, as required by Inequality (3.5), since it is no longer implicitly reflected by preemption-centric interrupt accounting (since only timer ticks are considered). When using dedicated interrupt handling, IPI latency must be accounted for under P-EDF, too.

This concludes our discussion of overhead accounting techniques for event-driven scheduling. Next, we summarize how to integrate overheads under quantum-driven scheduling.

3.5 Overhead Accounting under Quantum-Driven Schedulers

Holman (2004) and Devi (2006) studied how to account for overheads under quantum-driven schedulers in great detail. We provide a summary here for the sake of completeness and relate their techniques to how scheduling events occur in LITMUS$^{\mathrm{RT}}$.

Incorporating scheduling overheads into schedulability analysis is much simpler under quantum-driven scheduling than under event-driven scheduling. The main reason for this difference is the regular structure of a quantum-based schedule: since preemptions are only allowed at well-defined

times, it is easier to account for them. Further, the quantum size Q is typically much shorter than the shortest deadline, which greatly simplifies accounting for interrupts.

A quantum-driven scheduler requires that each task parameter is a multiple of the quantum size Q. In theory, the quantum size is chosen such that it divides the parameters of every task in a given task set. In practice, however, the quantum size Q is fixed and the task parameters must be *quantized* by rounding e_i up and p_i down to the nearest quantum multiple. Quantization is a source of capacity loss, but not strictly an overhead. Parameters should be quantized after accounting for actual overheads since overhead accounting requires non-quantum-sized adjustments.

There are fundamentally two sources of delay in a quantum-based scheduler. Initially, a newly-released job is not eligible to be scheduled since preemptions are only enacted at quantum boundaries, which adds to its release delay. Once a job is scheduled, it should receive Q time units worth of service from the processor until the next quantum boundary in theory, so that a job J_i requires at most $\left\lceil \frac{e_i}{Q} \right\rceil$ quanta of service. In practice, however, the *effective quantum length* Q' is less than Q since some processor cycles are lost to overheads and timer inaccuracy during each quantum. The central element of overhead accounting under quantum-driven scheduling is to determine a lower bound on Q', which implies an upper bound on J_i's actual processor demand of $\left\lceil \frac{e_i}{Q'} \right\rceil$. If quantum boundaries are staggered across processors, then jobs can be additionally delayed by the staggering offset. We account for staggering as part of the release delay.

In the following discussion, we focus on PD^2 since it is the sole quantum-driven scheduler considered in this dissertation. We previously evaluated quantum-driven and event-driven implementations of G-EDF, and found event-driven G-EDF implementations to be generally preferable (Brandenburg and Anderson, 2009a). In contrast, any implementation of PD^2 is necessarily quantum-driven since PD^2 is defined in terms of discrete subtasks.

3.5.1 Release Delay

Accounting for release delays under quantum-driven scheduling differs in two important ways from the event-driven case. First, quantum-driven scheduling creates additional sources of latency since scheduling events are not processed immediately by the scheduler. Second, scheduling and context-switch related overheads are accounted for differently under quantum-based scheduling (since they occur at every quantum boundary) and are hence not considered to be part of the release delay.

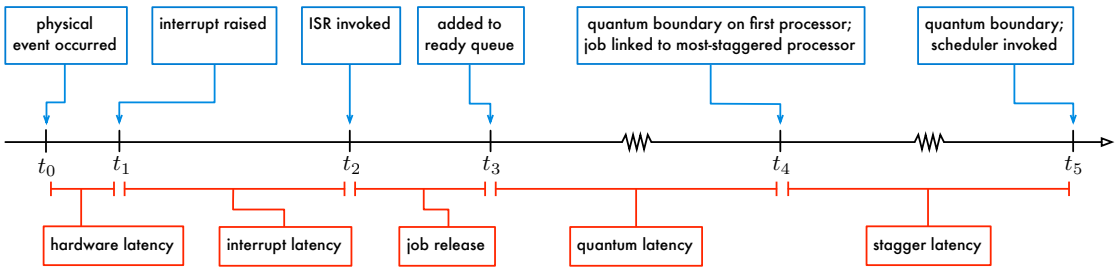


Figure 3.20: Timeline illustrating release delay under quantum-driven scheduling. In addition to hardware latency and interrupt latency, a job is also subject to quantum latency while it waits for the scheduler to react to its arrival. From the point of view of the scheduler, the job is not released until time t_4. This additional delay can be modeled by shortening the period of each task.

Figure 3.20 illustrates sources of release delay under quantum-based scheduling.

- The events during $[t_0, t_3)$ are the same as in the case of event-driven scheduling shown in Figure 3.9(a). This is the case because differences in the invocation of the scheduler do not affect the functioning of the hardware and interrupt delivery. Schedulability analysis of quantum-driven scheduling must thus take both Δ^{ev} and Δ^{rel} into account. With respect to Figure 3.20, these two delays are given by $\Delta^{ev} = t_2 - t_0$ and $\Delta^{rel} = t_3 - t_2$.

- At time t_3, the newly-released J_i is added to the ready queue. No scheduling (neither process nor job scheduling) takes place at this point—under quantum-based scheduling, J_i will not be considered for scheduling until the next quantum boundary. Consequently, quantum-based schedulers typically do not use IPIs to initiate rescheduling.

- At time t_4, the next quantum begins and job scheduling (or subtask scheduling, in the case of PD^2) takes place and J_i is assigned to a processor. We refer to the delay during $[t_3, t_4)$ as *quantum latency*. Recall that Q denotes the quantum size, which is one millisecond in LITMUS$^{\mathrm{RT}}$. In the worst case, t_3 occurs just after a new quantum started, so that $t_4 - t_3 \approx Q$, *i.e.*, J_i may be delayed up to a whole quantum before it is considered by the scheduler.

- Finally, if the system is configured to use staggered quanta, then J_i may additionally be delayed by *stagger latency* while it waits for its assigned processor to reach the next quantum. In the worst case, J_i has been assigned to the processor with the maximum offset, in which case

$t_5 - t_4 = (m-1) \cdot \frac{Q}{m}$. If the system is configured to use aligned quanta, then $t_4 = t_5$, *i.e.*, there is no additional delay.

After time t_5, J_i is still further delayed by the unavoidable process scheduling and context-switch overheads; however, these decrease the effective quantum size under quantum-driven scheduling and thus are not considered to part of the release delay. A notable difference from event-driven scheduling is that there is no ambiguity with regard to J_i's release time: from the point of view of the scheduler, J_i is not released until time t_4, when it first becomes eligible for scheduling.

Safe approximation. Since J_i is effectively not released until time t_4, its relative deadline must be shortened to compensate for the shifted release time. Additionally, J_i is effectively self-suspended during $[t_0, t_4)$ for the purpose of computing the minimum separation of job releases.

The task model underlying the analysis of PD^2 does not consider self-suspensions, but requires that all deadlines are implicit. Therefore, the period is shortened instead, which accounts for both J_i's release delay and the potentially reduced minimum job separation. Additionally, T_i''s period must be quantized by rounding down the shortened period to the nearest integer multiple of Q.

$$p_i' = Q \cdot \left\lfloor \frac{p_i - \Delta^{ev} - \Delta^{rel} - Q}{Q} \right\rfloor. \tag{3.20}$$

In the case of staggered quanta and HRT constraints, p_i must further take the maximum stagger latency of $(m-1) \cdot \frac{Q}{m}$ time units into account; formally,

$$p_i' = Q \cdot \left\lfloor \frac{p_i - \Delta^{ev} - \Delta^{rel} - Q(1 + \frac{m-1}{m})}{Q} \right\rfloor. \tag{3.21}$$

In the case of SRT constraints, the delay due to staggered quanta is simply a bounded source of tardiness and can thus be ignored.

3.5.2 Effective Quantum Size

Once a job J_i has been assigned a processor for a slot, it will execute until at least the start of the next quantum. Under schedulers that do not reschedule at every quantum boundary (such as a quantum-based G-EDF implementation), it is preferable to consider intervals longer than one quantum to avoid pessimism associated with narrow analysis intervals. Under PD^2, however, a

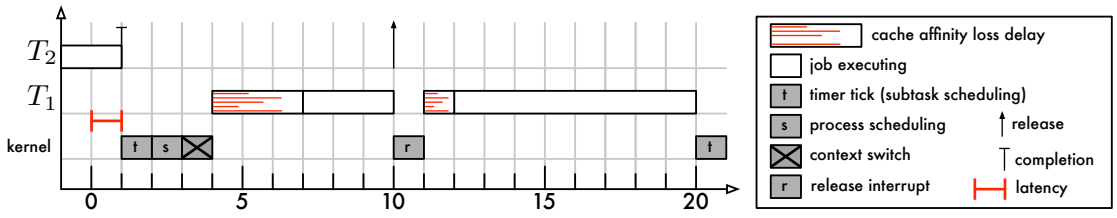


Figure 3.21: Schedule illustrating effective quantum length for $Q = 20$. Job J_1 should be scheduled at time 0, but the effective quantum length is reduced due to event latency during $[0, 1)$, subtask scheduling overhead during $[1, 2)$, process scheduling overhead during $[2, 3)$, context-switch overhead during $[3, 4)$, and cache-cold execution during $[4, 7)$. Finally, it is stopped by a release interrupt during $[10, 11)$, after which it incurs additional cache-related interrupt delay. In total, J_1 effectively receives only $Q' = 11$ of its nominally $Q = 20$ time units of allocated processor time.

job is typically preempted after each quantum (unless a task has a high utilization). It is therefore necessary to bound the minimum amount of useful execution carried out on behalf of J_i during a given quantum, assuming that J_i was not scheduled in the previous quantum.

There are four categories of overheads that cause J_i to receive less than Q time units of processor service during a given quantum. Each of the following sources of overhead are illustrated in Figure 3.21, which depicts events during a single quantum assuming $Q = 20$.

1. The timer tick that signals the beginning of J_i's quantum may be delayed due to hardware and interrupt latency. Let $t_q = 0$ denote the start time of J_i's assigned quantum, and let $t_{q+1} = 20$ denote the start time of the next quantum. In the worst case, the timer tick signaling the start of J_i's quantum is not serviced until $t_q + \Delta^{ev}$, and the timer tick signaling the beginning of the next quantum is serviced latency-free, *i.e.*, occurs exactly at time t_{q+1}. This effectively reduces the quantum length available to J_i by up to Δ^{ev} time units.[16]

2. The timer tick ISR carries out subtask scheduling (which is included in Δ^{tck}) and triggers process scheduling, which is followed by a context switch to J_i's implementing process. This requires up to $(\Delta^{tck} + \Delta^{sch} + \Delta^{cxs})$ time units before J_i commences execution.

3. Once scheduled, J_i must reestablish cache affinity since it was not scheduled in the preceding quantum, which increases J_i's execution requirement by up to Δ^{cpd} time units. This increase

[16] Holman (2004) also considers the possibility that timers may fire earlier than requested. In Linux, this does not happen unless the underlying hardware is faulty.

in execution requirement can be understood as decreasing the amount of processor service available to J_i during the quantum, as illustrated in Figure 3.21.

4. And finally, release ISRs may stop J_i at any time during the quantum (unless dedicated interrupt handling is employed). In the case of PD^2, $2Q \leq p_i$ for any T_i (otherwise Equation (3.20) results in $p'_i \leq 0$, which renders τ unschedulable). Each release interrupt source thus fires at most once during each quantum. This limits the total delay due to release ISRs to $n \cdot (\Delta^{rel} + \Delta^{cid})$ per quantum. In the case of HRT constraints, J_i is no longer pending when the release interrupt for its successor job occurs, which reduces the maximum delay slightly to $(n-1) \cdot (\Delta^{rel} + \Delta^{cid})$. This approach is called *quantum-centric* interrupt accounting in (Brandenburg *et al*., 2009, 2011).

Charging for n or $(n-1)$ release interrupts in each quantum is very pessimistic. Clearly, not every quantum will be shortened by that amount. However, it is difficult to avoid this pessimism in general since it is unknown which quanta a job will receive, and since release interrupts can occur during any quantum due to their sporadic nature. This emphasizes the pessimism inherent in analyzing multiple intervals of short duration. In the case of sporadic tasks, interrupt-related pessimism can be avoided by employing dedicated interrupt handling; in the case of periodic tasks, no release interrupts are required (see Section 3.5.3 below).

Taken together, if every processor potentially services release interrupts, the four overhead categories reduce the useful processor service time available to J_i during a quantum to

$$Q' = Q - \Delta^{ev} - \Delta^{tck} - \Delta^{sch} - \Delta^{cxs} - \Delta^{cpd} - n \cdot (\Delta^{rel} + \Delta^{cid}) \tag{3.22}$$

time units in the worst case. Under dedicated interrupt handling, release interrupts never delay subtasks and the effective quantum length is hence reduced to

$$Q' = Q - \Delta^{ev} - \Delta^{tck} - \Delta^{sch} - \Delta^{cxs} - \Delta^{cpd} \tag{3.23}$$

time units in the worst case. If this results in $Q' \leq 0$, then τ is claimed not schedulable. Additionally, J_i's inflated execution requirement must be quantized by rounding it up to the nearest integer multiple

of Q; formally,

$$e'_i = Q \cdot \left\lceil \frac{e_i}{Q'} \right\rceil . \tag{3.24}$$

The task set τ is claimed schedulable under PD^2 if $u_{\text{sum}}(\tau') \leq m$ and $p'_i \geq e'_i$ for each $T'_i \in \tau'$. The latter constraint implies a maximum feasible per-task utilization for any task $T_i \in \tau$. That is, since Equation (3.24) effectively scales e_i, it limits the maximum utilization that can be supported by the implementation.

$$\begin{aligned}
& p'_i \geq e'_i \\
\Rightarrow \quad & p_i \geq e'_i && \{ \text{ since Equation (3.20) implies } p_i \geq p'_i \ \} \\
\Leftrightarrow \quad & p_i \geq Q \cdot \left\lceil \frac{e_i}{Q'} \right\rceil && \{ \text{ by Equation (3.24) } \} \\
\Rightarrow \quad & p_i \geq Q \cdot \frac{e_i}{Q'} && \{ \text{ dropping the ceiling } \} \\
\Leftrightarrow \quad & u_i \leq \frac{Q'}{Q} && \{ \text{ rearranging } \}
\end{aligned}$$

The ratio $\frac{Q'}{Q}$ reflects the efficiency of the implementation; if it is low, then any task set that contains at least one high-utilization task is rendered unschedulable.

3.5.3 Periodic Task Sets

As noted above, charging for every release interrupt during each quantum is grossly pessimistic. This is particularly the case if all tasks are periodic (and not sporadic), as such tasks do not have to be released with asynchronous interrupts. Instead, the scheduler can automatically merge newly-released jobs into the ready queue at every quantum boundary—that is, job releases can be synchronously *polled* if tasks execute strictly periodically. Since the timer tick is required anyway, this causes virtually no additional overhead. The primary advantage of polled releases is that it avoids all delays due to release interrupts and may thus dramatically increase the effective quantum size, thereby improving the efficiency ratio $\frac{Q'}{Q}$.

From the point of view of accounting for overheads, polled releases are virtually identical to dedicated interrupt handling. The only difference is that all m processors are available for job scheduling if releases are polled, whereas only $m - 1$ processors service jobs under dedicated interrupt handling. That is, if job releases are polled by the scheduler, then the effective quantum size

is given by Equation (3.23) and not Equation (3.22). Additionally, event latency and release overhead do not affect release delay if job releases are polled; it is hence safe to assume $\Delta^{ev} = 0$ and $\Delta^{rel} = 0$ when evaluating Equation (3.20) or Equation (3.21). Event latency can still reduce the effective quantum size and hence should not be assumed to be negligible when evaluating Equation (3.23). This is because event latency still affects the accuracy of timer ticks and hence can cause a quantum boundary to be signaled up to Δ^{ev} time units late (*e.g.*, this happens during $[0, 1)$ in Figure 3.21).

When implementing a periodic workload, it is beneficial to choose a quantum size Q and to align timer ticks with the system's start of real-time operation such that job releases always occur exactly on quantum boundaries (this implies that each p_i is an integer multiple of Q). In this case, no job incurs quantum latency since it is immediately considered for scheduling upon release, *i.e.*, $t_3 = t_4$ in Figure 3.20. As a result, periods do not have to be shortened by Q time units: in the case of aligned quanta, Equation (3.20) is hence reduced to $p'_i = Q \cdot \left\lfloor \frac{p_i}{Q} \right\rfloor = p_i$, and Equation (3.21) is reduced to $p'_i = Q \cdot \left\lfloor \frac{p_i - Q \cdot (m-1)/m}{Q} \right\rfloor = p_i - Q$ in the case of staggered quanta.

Release polling could similarly be implemented in event-driven schedulers. Analogously to the quantum-driven case, release polling in event-driven plugins resembles dedicated interrupt handling from an analysis point of view. Additionally, if job releases are not aligned with quantum boundaries, then quantum latency of up to Q time units must be considered when accounting for release delay, just as under quantum-driven schedulers.

Given that the main appeal of PD^2 is that it is HRT optimal for implicit deadline tasks, and that HRT applications are more likely to be of a periodic nature than SRT applications, implementing support for polled releases is likely more useful under PD^2 than under event-driven plugins. Nonetheless, it is worth noting that release interrupts can be eliminated entirely if all tasks are periodic.

3.6 Summary

We have discussed the design and implementation of LITMUS$^{\mathrm{RT}}$, which extends the Linux kernel's scheduling class hierarchy with a flexible scheduler plugin framework. A key difference between stock Linux and LITMUS$^{\mathrm{RT}}$ is that LITMUS$^{\mathrm{RT}}$ contains explicit support for global scheduling and safe process migrations (Section 3.3.4), whereas Linux's implementation of global scheduling is not correct in all cases (Section 3.2). Another important distinction is that LITMUS$^{\mathrm{RT}}$'s G-EDF and

C-EDF plugins implement link-based scheduling (Section 3.3.3), which ensures that non-preemptive sections cause only $O(1)$ pi-blocking.

We have further discussed the various overheads that affect the execution of real-time tasks in LITMUS$^{\mathrm{RT}}$ under event- and quantum-driven schedulers, and have provided corresponding overhead accounting methods. Based on the notion of safe approximations, task parameters are adjusted to reflect the worst-case delay that any job may incur due to kernel- and cache-related overheads (Sections 3.4 and 3.5). Next, we demonstrate how these overhead accounting techniques can be used to compare real-time schedulers under consideration of overheads as they arise in LITMUS$^{\mathrm{RT}}$.

CHAPTER 4

OVERHEAD-AWARE EVALUATION OF REAL-TIME SCHEDULERS*

In the preceding two chapters, we have explored the challenges involved in ensuring HRT and SRT constraints for sporadic workloads on multiprocessors. In Chapter 2, we focused on algorithmic questions and surveyed several partitioned, global, and clustered scheduling algorithms that apply to the sporadic task model and reviewed their corresponding schedulability analysis. In Chapter 3, we examined the practical challenges involved in supporting the sporadic task model at the RTOS level and described how LITMUS$^{\mathrm{RT}}$ implements the sporadic task model and the considered scheduling algorithms, and explained which sources of overheads sporadic tasks are exposed to and how they can be accounted for during schedulability analysis.

The provided overview makes it clear that scheduling approaches differ radically in terms of capability, analysis, and implementation. Given this wide range of options, it is not obvious which scheduler is the best choice for ensuring HRT or SRT constraints in practice. Which scheduler(s) should the next generation of multiprocessor RTOSs implement? Does there exist a scheduler that always "performs best," or should future RTOSs be designed to support multiple schedulers? Should RTOSs have separate SRT and HRT schedulers, or can one algorithm handle both kinds of workloads? Are some algorithms indeed impractical? These are challenging questions to answer, but they are of great importance to the design and implementation of future RTOSs.

In this chapter, we first present our overhead-aware methodology for investigating such questions and explain how it differs from prior approaches. Thereafter, we present a large-scale case study of LITMUS$^{\mathrm{RT}}$ on a 24-core Intel Xeon system. This study demonstrates that interesting, non-

* Contents of this chapter previously appeared in preliminary form in the following paper:
Bastoni, A., Brandenburg, B., and Anderson, J. (2010a). Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–44.

obvious, and sometimes counterintuitive conclusions can be derived using the proposed approach. Throughout this chapter, we assume that tasks are independent. Locking-related issues are the subject of Chapter 7.

4.1 Methodology

There are many ways that real-time performance could be assessed. From the point of view of a real-time application developer, the "performance" of a real-time scheduler is defined by whether a specific application's temporal constraints can be supported. That is, in the case of HRT constraints, the scheduler performs well if the application is schedulable; in the case of SRT constraints, the scheduler performs well if tardiness is bounded and the bound is suitably small. This perspective, however, does not lend itself as a quantifiable performance metric.

While tardiness bounds allow for some degree of quantification, small differences in tardiness are likely irrelevant in an SRT setting. Further, in the context of a specific application, there is little benefit to having a "better" scheduler if a "good enough" scheduler is available (where "better" and "good enough" are interpreted from an analytical viewpoint). For example, if a task set is schedulable under FP scheduling, there is little benefit to be gained from employing EDF instead. As a secondary concern, schedulers may differ with regard to the throughput obtained by a system's best-effort background workload (if any). That is, among two schedulers that both ensure temporal correctness of a given real-time workload, it may be preferable to choose the one that leaves more processor time to best-effort processes. However, throughput considerations do not impact a system's "real-time performance": if the application's temporal constraints are satisfied, the system's "real-time performance" is good enough.

In contrast, from the point of view of an RTOS developer, the "performance" of the implemented scheduler is not tied to a specific application. Rather, an RTOS vendor aims to license its product to customers in as many markets and application domains as possible (an open-source RTOS similarly gains utility when it appeals to a larger user base). Driven by such economic considerations, a primary RTOS design goal is to be as versatile as possible. The criterion for choosing an RTOS scheduler is hence to *maximize the set of schedulable task sets*, *i.e.*, to preclude as few applications as possible.

In the latter interpretation, the real-time performance of a scheduler can be quantified as the fraction of task sets that are schedulable (either HRT or SRT) under it. In the real-time literature, this metric, the *fraction of schedulable task sets*, is also called a scheduler's *schedulability* and empirical comparisons based on it are called *schedulability experiments*.[1] Schedulability experiments are a pervasive method of evaluation in the real-time community and we adopt it as the primary performance evaluation methodology in this dissertation as well.

In short, a scheduler $\mathcal{A}_1$ is superior to another scheduler $\mathcal{A}_2$ if more task sets can be claimed schedulable under $\mathcal{A}_1$ than under $\mathcal{A}_2$. Equivalently, the objective is to minimize capacity loss.

4.1.1 Sources of Capacity Loss

Capacity loss, which we formalize in Section 4.1.5 below, measures to which degree a scheduler's performance diverges from that of an optimal, overhead-free scheduler. Intuitively, a scheduler exhibits capacity loss when a feasible task set cannot be shown to be schedulable under it. Capacity is being "lost" in this case since the failure to schedule a feasible task set implies inefficient use of the available resources (since, by definition, it is possible to schedule a feasible task set).

To motivate our methodology, we begin by discussing why capacity loss arises in practice. Suppose the temporal constraints of a feasible task set τ can be validated on a given platform under scheduling algorithm A_1 (*i.e.*, τ can be shown to be schedulable), but not under another scheduling algorithm A_2. There are four major causes of validation failure.

1. *Fundamental algorithmic differences.* If algorithm A_2 has inherent limitations that prevent τ from being schedulable, then it is irrelevant how efficiently A_2 is implemented. For example, if τ cannot be partitioned onto m processors, then a partitioned scheduler's low implementation overheads are immaterial.

2. *Pessimistic analysis.* It may be the case that τ is in fact schedulable under A_2, but that it is not possible to verify this with existing schedulability tests. For example, none of the G-EDF schedulability tests published to date is exact.

3. *Differences in overhead.* If jobs incur higher scheduling overhead under A_2 than under A_1, then this may render τ unschedulable under A_2. For example, lock contention is higher in

[1] Schedulability *experiments* should not be confused with schedulability *tests*.

global schedulers than in partitioned schedulers. Large differences also exist between event-driven and quantum-driven scheduling. Differences in preemption and migration overhead under partitioned and global scheduling fall within this category as well.

4. *Overhead accounting differences.* Even when scheduling overheads have similar magnitudes under A_1 and A_2, it may be impossible to validate τ's timing correctness under A_2 due to differences in how overheads are accounted for. For example, timer tick ISRs execute virtually identical code under P-FP and P-EDF scheduling, but delays due to ISRs are very differently accounted for under each scheduler. Similarly, under PD^2 scheduling, release interrupts are considered to decrease the effective length of every quantum, which is likely more pessimistic than the preemption-centric interrupt accounting employed under G-EDF.

Causes 1 and 2 comprise *algorithmic capacity loss*, while Causes 3 and 4 together yield *overhead-related capacity loss*. Unfortunately, the relative impact of algorithmic and overhead-related capacity loss is very difficult to anticipate. In particular, it is virtually impossible to foresee how algorithmic properties and implementation choices interact.

For example, since EDF is optimal on a uniprocessor, whereas FP is subject to algorithmic capacity loss, P-EDF can sustain a higher utilization on each processor than P-FP. However, one might reasonably suspect P-FP to have much lower overheads since it is built around an efficient, bitfield-based ready queue. Does this make P-FP the better scheduler in practice? This depends on many factors, including whether ready queue operations actually constitute a significant part of the overall scheduling overhead and the maximum number of queued tasks.

Similarly, P-EDF is impacted by bin-packing limitations more than C-EDF if $c > 1$. Thus, algorithmic capacity loss should be less under C-EDF, but locking overheads are likely higher under C-EDF. Does the simplification of the bin-packing problem justify the increase in locking overhead? This, again, depends on the actual magnitude of overheads and whether the to-be-scheduled task set is actually hard to partition (*i.e.*, whether it contains many high-utilization tasks).

We maintain that the magnitude of each overhead is impossible to anticipate in the absence of an actual, working implementation. To yield results that are meaningful in practice, a proper evaluation of real-time schedulers must be based on a real, in-kernel implementation of each evaluated algorithm.

It is further crucial to employ an evaluation methodology that is capable of reflecting each of the four causes of capacity loss. We present our approach next.

4.1.2 Evaluation Approach

The method of evaluation underlying this dissertation has been developed and refined over the course of several published studies (Brandenburg *et al.*, 2008; Brandenburg and Anderson, 2009a; Bastoni *et al.*, 2010a,b, 2011), which in turn were inspired by the original LITMUS$^{\mathrm{RT}}$ study (Calandrino *et al.*, 2006). However, the methodology described herein differs in some details from each of these studies since its current version incorporates a number of improvements and refinements based on lessons learned in the preparation of the just-cited studies.

Our evaluation methodology consists of eight steps that can be categorized into an OS phase and an analytical phase. The flowchart shown in Figure 4.1 depicts each of the steps and phases. In this section, we provide a high-level overview of the individual steps, what they entail, and why they are necessary. In the following sections, we define the employed metrics (schedulability, weighted schedulability score, and relative tardiness) and explain how to interpret the resulting graphs. Finally, the case study in the latter part of this chapter provides concrete examples for each step.

OS phase. In the first step, each of the schedulers that are to be evaluated is implemented in the kernel. The RTOS underlying our work is LITMUS$^{\mathrm{RT}}$, as discussed in the preceding chapter. Obviously, there is little benefit to evaluating incorrect implementations, so great care should be taken in testing and debugging the implemented schedulers. Unsurprisingly, this step is very time consuming; the process of implementing new schedulers in LITMUS$^{\mathrm{RT}}$ typically takes several weeks from the first design until the plugin is stable, *i.e.*, until it passes all stress tests and produces valid schedules. LITMUS$^{\mathrm{RT}}$'s TRACE() and sched_trace() tracing infrastructure are crucial during this step to observing and debugging how scheduling decisions are computed.

In the second step, the kernel is instrumented to record the duration of each scheduling decision, context switch, timer tick, *etc.* In LITMUS$^{\mathrm{RT}}$, the Feather-Trace infrastructure is used for this purpose. A large number of benchmark task sets of different composition (*i.e.*, with varying numbers of tasks and task parameters) are executed under each of the schedulers and the overhead samples are recorded. In a separate set of experiments, similar overhead data is collected to assess cache-related

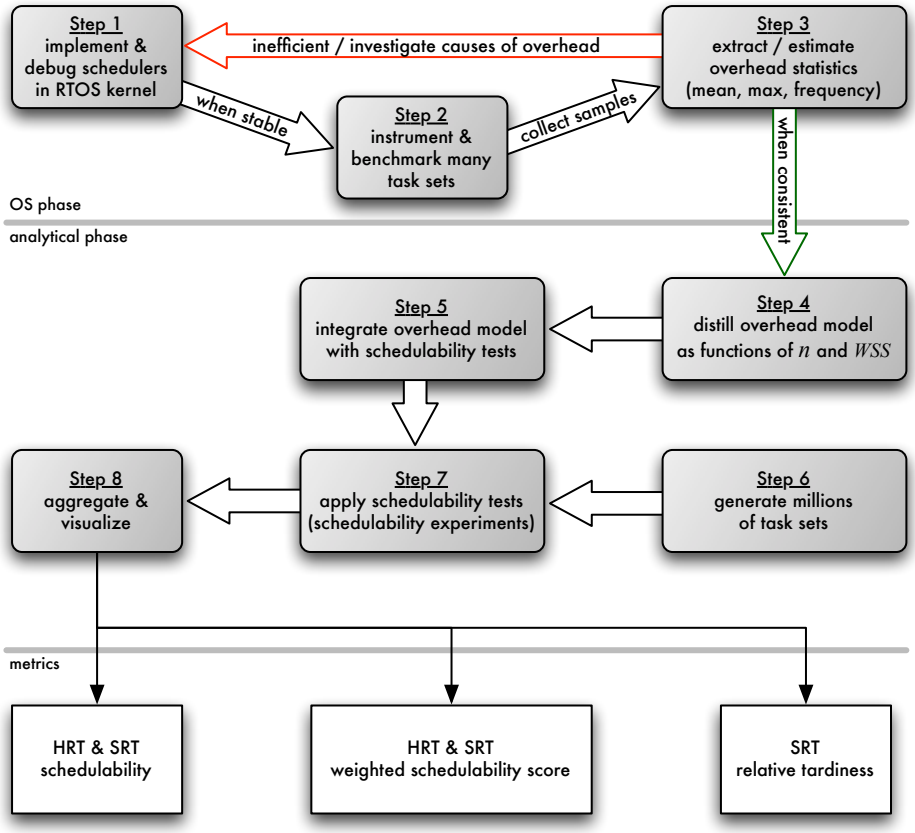


Figure 4.1: Flowchart illustrating the employed overhead-aware evaluation method.

delays after preemptions and migrations. In a moderately-sized study, Step 2 typically requires 20–60 hours of execution and produces tens to hundreds of gigabytes of overhead samples. We discuss this step in more detail in the context of the case study in Sections 4.3 and 4.4 below.

In the final step of the OS phase, the raw collected overheads are processed to estimate relevant statistics. In particular, the maximum and average values for each overhead and each benchmark task set are determined, as are histograms of the observed overhead magnitude. If the measurements were disturbed by noise sources such as interrupts, then some filtering of outliers may be required, which is discussed in Section 4.3 in more detail. Step 3 serves a dual purpose. For one, the determined statistics are needed to incorporate overheads in the analytical model. However, a close examination of overhead trends commonly reveals issues in the underlying implementation such as avoidable inefficiencies or inexplicable inconsistencies. In such cases, additional debugging and improvement of the implementation are required.

Analytical phase. Once the recorded overheads are consistent and reasonable, the analytical phase commences. The analytical phase is not tied to LITMUS$^{\mathrm{RT}}$, *i.e.*, the steps are the same and the tools and techniques could be reused in the context of other RTOSs.

In Step 4, the large data set of recorded overheads for specific values of n is compressed into a much smaller model that also yields interpolated values for arbitrary values of n. Additionally, the model also masks any unwanted non-monotonicity in the recorded overhead trends; this is explained in detail in Section 4.1.3 below. Once the overhead model has been derived it is integrated into the existing schedulability analysis in Step 5, which we discuss shortly.

Steps 6 and 7 comprise what are commonly called schedulability experiments. As noted above, schedulability experiments are an established way to quantify scheduler performance that is frequently found in the real-time literature. Schedulability tests for each evaluated scheduling algorithm are applied to a large body of task sets. By counting the task sets that are schedulable (either HRT or SRT), the fraction of task sets that is schedulable under each algorithm is empirically estimated. Additionally, we also record tardiness bounds in the case of SRT constraints. We explain how task sets are generated in Section 4.1.4 and discuss schedulability experiments in detail in Section 4.1.5. Step 7 can be very time-consuming, depending on the number of generated task sets. We typically run schedulability experiments on UNC's research cluster *Topsail* over night (due to the large number of task sets, it is trivial to parallelize schedulability experiments).[2]

Step 5 is the defining step of our methodology that connects theory and practice: in contrast to prior work, we account for overheads in our schedulability experiments. In Step 5, starting from the standard, overhead-unaware schedulability experiment setup (Section 4.1.5), we integrate the overhead model based on real, measured overheads (Section 4.1.3) using the overhead accounting techniques discussed in Chapter 3. The resulting overhead-aware schedulability experiment setup, used in Step 6, is discussed in Section 4.1.6 below.

Finally, in Step 8, the resulting schedulability data is aggregated and visualized. We employ three metrics: schedulability (the fraction of schedulable task sets), a weighted schedulability score (to aggregate results), and relative tardiness (a normalized variant of tardiness). We define each metric and explain how to interpret each plot in Sections 4.1.5, 4.1.7, and 4.1.8, respectively.

[2] *Topsail* is a distributed-memory cluster consisting of 512 compute nodes. Each node consists of eight 2.3 GHz processors with 12 GB of memory. Our schedulability experiments ran on up to 64 nodes, depending on availability.

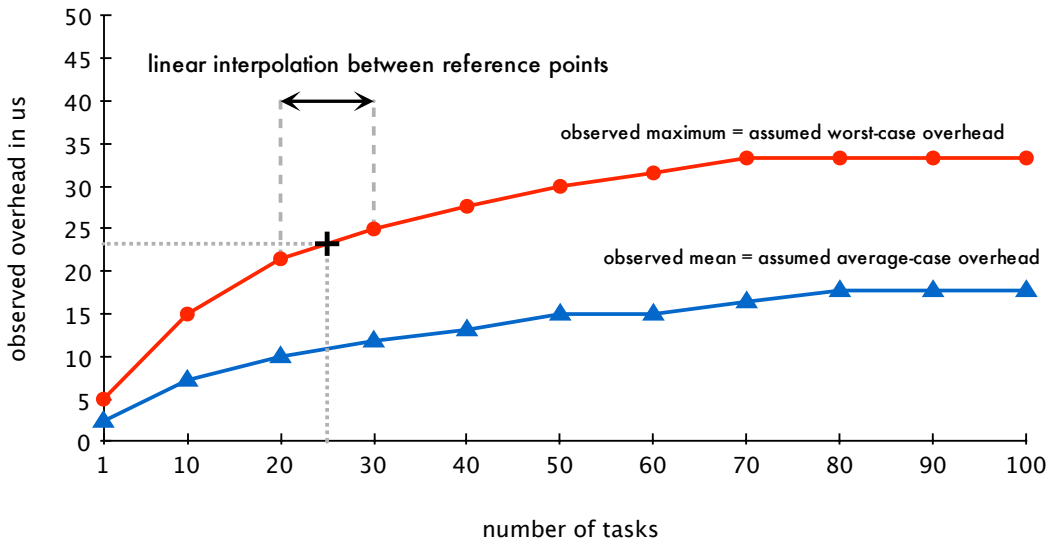


Figure 4.2: Illustration of an overhead model. Worst-case and average-case overheads are modeled as a piece-wise linear function of the task set size n. The reference points are the observed maximum and mean overheads for the tested task set sizes, which are typically a multiple of m (in this example, $m = 10$). An exception is $n = 1$ since at least one task is required to collect overhead samples.

In addition to this overview, the individual steps are described in more detail in other sections. The implementation issues arising in Steps 1 and 2 are discussed at length in Chapter 3, as are the overhead accounting techniques employed in Step 5. Real data for Steps 2 and 3 is presented in our case study in the latter part of this chapter, as are the results from Step 8.

In the following, we describe the employed overhead model, review how regular, overhead-unaware schedulability experiments are performed, and then explain how we integrated the former with the latter. Finally, we introduce a weighted schedulability score to combine large schedulability data sets into aggregate results to expose certain trends more clearly.

4.1.3 Overhead Model

Instead of using the recorded overheads directly (of which there are several gigabytes), the recorded maximum and average overhead values are distilled into a model of worst-case and average-case overheads where each overhead is represented by a piece-wise linear function. Overheads must be represented as a function since many overheads are not scalar, but rather depend on the task set size n, as illustrated in Figure 4.2. For example, scheduling decisions and job releases are typically

affected by the length of the ready queue. Therefore, it is necessary to measure maximum and average overheads for different task set sizes.

A piece-wise model is required because it is not feasible to measure each possible value of n due to the effort and resources involved in such experiments; rather, overheads are recorded for $n = km$, where $k \in \{1, 2, 3, \ldots\}$, up to some reasonable upper bound (*e.g.*, $k = 20$). In other words, overheads are recorded only at reference points of k tasks per processor. For task set sizes that fall between reference points, maximum and average overheads are interpolated linearly. For example, Figure 4.2 depicts the interpolated assumed worst-case overhead for $n = 25$.

Monotonicity. A perhaps counterintuitive, but nonetheless frequently occurring effect is that observed overheads may *decrease* with increasing task set size. For example, if the scheduler is invoked more frequently due to a larger number of tasks, then kernel data structures are more likely to remain cached between invocations. This can reduce observed overheads significantly. However, it makes little sense to reflect non-monotonic trends in the overhead model because not all sporadic tasks may have a job pending at the same time. That is, some tasks may exhibit inter-sporadic arrival delays, in which case pending jobs incur overheads as if they were part of a smaller task set.

If such phases of inactivity may cause an *increase* in overheads, then it should be reflected by the model. Therefore, we model overheads as monotonically increasing functions, *i.e.*, the overhead model discards decreasing trends. This is illustrated in Figure 4.3, where the circular points indicate observed overheads and the piece-wise linear curve indicates the corresponding overhead model. Starting at $n = 70$, the observed overhead values exhibit a decreasing trend. In the corresponding piece-wise linear model, this trend is discarded and monotonicity of the overhead model is enforced.

Formally, let O_n denote the observed overhead magnitude for n tasks. When interpolating the overhead for $n = 85$, instead of interpolating between O_{80} and O_{90}, which would yield a decreasing trend, the overhead model interpolates between $\max\{O_1, O_{10}, O_{20}, \ldots, O_{70}, O_{80}\}$ and $\max\{O_1, O_{10}, O_{20}, \ldots, O_{70}, O_{80}, O_{90}\}$, which forces the desired monotonicity. In the extreme, if O_1 is larger than any observation for larger n, this reduces the overhead to a constant function. In the case that the sequence of observations O_n is itself monotonically increasing, this method reduces to normal piece-wise linear interpolation. We require each modeled overhead to be monotonic.

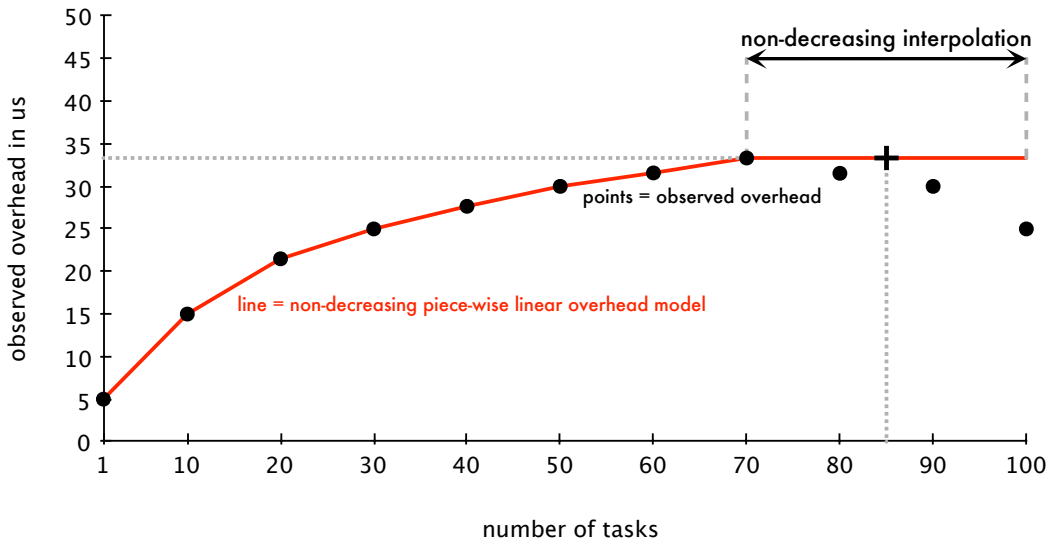


Figure 4.3: Illustration of forced monotonicity in the overhead model. While the observed overhead (either maximum or average values) exhibits a decreasing trend for $n \geq 70$, the overhead model uses piece-wise linear non-decreasing interpolation such that the resulting overhead model is monotonic.

An alternative to our approach is to extrapolate based on the last positive slope. That is, instead of forcing monotonicity with a slope of zero when the observed samples decrease (as shown in Figure 4.3), one could enforce strict monotonicity by linearly extrapolating the last segment with an increasing slope. However, this would create increasing trends (and induce additional pessimism for large n) in the overhead model that are not consistent with the observed data.

Most overheads can be expressed as a function of n (or are constant). However, an exception to the rule are cache-related overheads that jobs incur due to the (partial) loss of cache affinity caused by preemptions, migrations, and interrupts. For these overheads, the number of tasks in the system is irrelevant. Instead, their magnitude increases with larger working sets, *i.e.*, the more cache lines a job uses, the more compulsory cache misses are caused by a loss of cache affinity. Cache-related overheads are therefore modeled as a function of WSS, and not as a function of n. As in the case of overheads dependent on n, we use linear interpolation for WSSs that fall between reference points.

Note that we use *observed* maximum values to estimate worst-case overheads since WCET analysis tools are currently not available for our platforms (recall Section 3.1). In the future, it may become possible to derive true worst-case overhead bounds analytically if practical WCET analysis tools become available. However, even then Steps 3 and 4 would still be required to compute

average-case overheads (which we use for SRT analysis—see below). When estimating worst-case overheads based on measured data, it is important to use consistent sample sizes to avoid introducing a bias (see Section 4.3.1 below).

We note that measured maxima and interpolation do not necessarily bound the *true* worst case. As such, this methodology is not applicable to the certification of safety-critical applications. However, in the context of our work—when comparing schedulers under consideration of *realistic* overheads—the derived overhead models offer much greater accuracy than "guesstimating" overheads without data (or, worse, not considering overheads at all). See Section 3.1 for a discussion of formal guarantees in the context of inexact overhead estimates.

4.1.4 Task Set Generation

The schedulability metric, *i.e.*, the fraction of schedulable task sets, is necessarily defined with respect to a given *testing set* of task sets that serve as the benchmark. This poses the problem of obtaining a suitable testing set. In particular, testing sets should contain hundreds or thousands of task sets to avoid under-sampling (*e.g.*, just testing two cherry-picked task sets does not yield representative results). Unfortunately, sufficiently large data sets of representative tasks are currently not available (at least publicly, as companies commonly consider application details to be trade secrets). Instead, testing sets are typically created by randomly choosing task parameters until a desired utilization cap is reached. With this approach, a large number of task sets can be generated quickly, which enables large-scale experiments covering wide ranges of possible task parameters.

Listing 4.1 provides the task set generation procedure as pseudo-code. Given two random distributions from which to draw task utilizations and periods, $udist$ and $pdist$, implicit-deadline tasks are generated until a specified cap on total utilization, denoted $ucap$, is reached. Each task's execution budget e_i is computed based on the generated utilization and period. Generally speaking, this allows testing sets to be generated with any parameter distributions provided that $u_i \in [0, 1]$ and $p_i > 0$ for each T_i. The choice of parameter distributions is of course crucial to the significance of the experiments; we consider this topic in Section 4.2 below.

When the utilization cap is reached, the last-generated task is not included in the generated task set to ensure that $ucap$ is not exceeded (lines 4–7 in Listing 4.1). The generated task set τ hence likely does not match $ucap$ exactly, *i.e.*, the task set generation procedure leaves some slack on average.

```
1  generate_taskset(udist, pdist, ucap):
2      set τ ← ∅
3      for i ← 1, 2, 3, ...
4          allocate T_i
5          set u_i ← prng_draw_next(udist)
6          if u_sum(τ) + u_i > ucap:
7              break
8          else:
9              set p_i ← prng_draw_next(pdist)
10             set e_i ← p_i · u_i
11             set τ ← τ ∪ {T_i}
12     return τ
```

Listing 4.1: Task set generation pseudo-code.

Alternatively, the utilization of the last-generated task could be reduced such that $ucap = u_{\text{sum}}(\tau)$. However, this could result in a task being generated with a utilization that is not in accordance with $udist$. For example, when generating "heavy" tasks with utilizations in the range $[0.5, 1)$, scaling the utilization of the last-generated task could introduce an unexpected "light" task with much lower utilization. We therefore discard the last-generated task to avoid biasing the specified parameter distributions.

4.1.5 Schedulability Experiments

A schedulability experiment estimates the ratio of schedulable tasks for given task parameter distributions by repeatedly generating and testing task sets. A typical example setup is given in Listing 4.2. For a given $ucap$ and parameter distributions, the procedure generates a fixed number of task sets (a common choice is $samples = 100$) and tests each generated task set to determine whether it is schedulable under the specified algorithm $\mathcal{A}_x$, where schedulable means either HRT or SRT schedulable depending on the context. In the case of clustered or partitioned scheduling, the invoked schedulability test procedure also attempts to assign tasks to clusters, and fails if no valid assignment can be found. Finally, the schedulability experiment returns the ratio, which ranges from 0 to 1, of the generated task sets that were claimed schedulable. Naturally, the accuracy of the estimate improves with increasing sample size.

Alternatively, a schedulability experiment can be equivalently understood as a stochastic experiment. As an analogy, the tested scheduler $\mathcal{A}_x$ can be considered to be a coin, with generating and

```
1  sample_schedulability(A_x, udist, pdist, ucap):
2      set count ← 0

4      for j ← 1, 2, 3, ..., samples
5          set τ_j ← generate_taskset(udist, pdist, ucap)
6          if τ_j is schedulable under A_x on m processors:
7              set count ← count + 1

9      return ( count / samples )
```

Listing 4.2: Overhead-unaware schedulability experiment.

testing a task set corresponding to a coin flip, and sample_schedulability() giving an estimate of the coin's bias. A scheduling algorithm performs well if it is heavily biased towards success.

More accurately, let X_j denote a binary random variable such that $X_j = 1$ if τ_j is schedulable, and $X_j = 0$ otherwise. Determining the value of each X_j is a Bernoulli trial. Since each trial is independent, the procedure given in Listing 4.2 implements a Bernoulli process. The sampled ratio of schedulable task sets thus estimates the probability that a randomly chosen task set, with respect to the specified parameter distributions, is schedulable under A_x. That is, sample_schedulability() approximates $P(X_j = 1)$. This matches the intuitive notion of schedulability: the higher an algorithm's schedulability, the more likely it is to correctly schedule a given task set.

Visualizing capacity loss. For fixed parameter distributions $udist$ and $pdist$, an algorithm's schedulability can be pictured as a function of $ucap$. In the following, we let $S(\mathcal{A}_x, ucap)$ denote the schedulability of $\mathcal{A}_x$ for a given $ucap$, where $S(\mathcal{A}_x, ucap) \in [0, 1]$ and $ucap \leq m$.

The *capacity loss* of a potentially non-optimal algorithm $\mathcal{A}_x$ in the presence of overheads describes how much $\mathcal{A}_x$'s performance diverges from that of an optimal, overhead-free algorithm $\mathcal{A}_{opt}$. Formally, we define capacity loss as $S(\mathcal{A}_{opt}, ucap) - S(\mathcal{A}_x, ucap)$, with respect to given parameter distributions.

Since we assume implicit deadlines, all generated task sets are feasible if $ucap \leq m$. Therefore, the schedulability of an optimal, overhead-free algorithm is by definition always 1, *i.e.*, $S(\mathcal{A}_{opt}, ucap) = 1$ for any $ucap \leq m$. The capacity loss of $\mathcal{A}_x$ is hence given by

$$S(\mathcal{A}_{opt}, ucap) - S(\mathcal{A}_x, ucap) = 1 - S(\mathcal{A}_x, ucap).$$

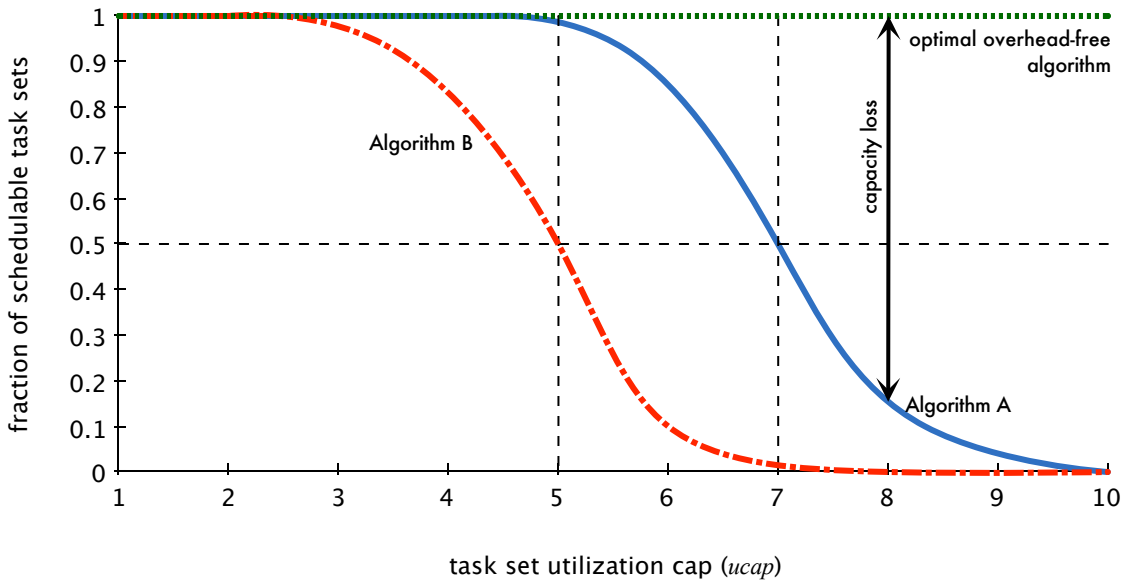


Figure 4.4: Illustration of a standard schedulability plot assuming $m = 10$. Each curve depicts the ratio of schedulable task sets as a function of task set utilization, *i.e.*, $S(A, ucap)$ and $S(B, ucap)$. In the absence of overheads, an optimal scheduler guarantees a schedulability of 1 until the total utilization exceeds m. Algorithmic and overhead-related capacity loss cause schedulability to gradually decrease in practice.

Schedulability and capacity loss can be easily visualized as illustrated Figure 4.4, which depicts schedulability curves for two schedulers, labeled Algorithm A and Algorithm B, respectively. Since any task set τ with $u_{\text{sum}}(\tau) > m$ is necessarily infeasible, the x-axis is limited to $1 \leq ucap \leq m$. The y-axis ranges from 0 to 1 and gives schedulability (and hence, implicitly, an estimate of $P(X_j = 1)$) for each tested algorithm. As $ucap$ increases, the number of tasks increases and the amount of available slack decreases, which makes the scheduling problem more difficult. Consequently, the schedulability of each (potentially non-optimal) algorithm decreases, *i.e.*, it becomes less likely that a generated task set is schedulable. The widening gap between each curve and the line $y = 1$ reflects capacity loss. This allows for a quick assessment of algorithm performance: the higher a curve, the better the corresponding algorithm's performance.

In Figure 4.4, Algorithm A is clearly preferable to Algorithm B since Algorithm B never exceeds Algorithm A's schedulability at any point. For instance, consider the points where each scheduler crosses the line $y = 0.5$ (marked by dashed lines), *i.e.*, the utilization caps at which fewer than 50% of the generated task sets can be claimed schedulable under each algorithm. Algorithm A crosses

$y = 0.5$ at $ucap = 7$, whereas Algorithm B has 0.5 capacity loss already at $ucap = 5$. This indicates that Algorithm A can sustain a higher processor utilization before its performance deteriorates; it is hence superior to Algorithm B because it uses the available processing capacity more efficiently.

It is worth pointing out that each schedulability plot shows scheduler performance for fixed $udist$ and $pdist$ distributions. That is, each schedulability plot of this kind corresponds to a specific task set composition. To draw valid conclusions, many different parameter distributions should be considered. The search for the "best" scheduler can thus be rephrased as follows: we seek to identify the scheduler that exhibits the least capacity loss (*i.e.*, that has a schedulability curve closest to $y = 1$) for most parameter distributions.

4.1.6 Integrating Overhead Accounting

The standard schedulability experiment setup discussed so far can be used to study algorithmic capacity loss. For example, setups similar to the above description have been used to quantify improvements in schedulability tests (Bertogna and Baruah, 2011) and to compare priority assignment algorithms for G-FP scheduling (Davis and Burns, 2011a), among many other examples. However, standard schedulability experiments cannot reveal implementation-related capacity loss since overheads are not considered. To overcome this limitation, we extend the basic schedulability experiment setup from Listing 4.2 to integrate overhead analysis using the safe-approximation approach detailed in Chapter 3.

Recall from Section 4.1.3 that most overheads depend on the number of tasks n, but that cache-related overheads depend on the maximum WSS of any task. The number of tasks n is implicitly determined by the task set generator and thus is readily available to compute appropriate bounds on scheduling overheads. However, the maximum WSS is independent of the utilization cap and thus unknown. Even though large WSSs likely correspond to large execution times in practice, from the point of view of implementing a schedulability experiment, the maximum WSS is an arbitrary parameter that must be provided. Overhead accounting thus introduces a second variable wss besides $ucap$ that must be either chosen arbitrarily or varied across some range.

Earlier LITMUS$^{\mathrm{RT}}$-based studies did indeed simply pre-determine one or a few fixed WSSs such as 64KB (Calandrino *et al.*, 2006; Brandenburg *et al.*, 2008; Brandenburg and Anderson, 2009a). However, this is somewhat unsatisfactory since there certainly exist workloads with larger WSSs,

```
1  sample_schedulability(A_x, udist, pdist, ucap, wss):
2      set count ← 0
3      set (Δ^cpd, Δ^cid) ← estimate cache-related overheads for working set of size wss under A_x

5      for j ← 1, 2, 3, ..., samples
6          set τ_j ← generate_taskset(udist, pdist, ucap)
7          set (Δ^rel, Δ^sch, Δ^tck, Δ^cxs, Δ^ev, Δ^ipi) ← overheads for |τ_j| tasks
8          set τ'_j ← derive safe approximation of τ_j (see Chapter 3)
9          if τ'_j is schedulable under A_x on m processors:
10             set count ← count + 1

12     return ( count / samples )
```

Listing 4.3: Overhead-aware schedulability experiment for global schedulers.

just as there exist workloads with smaller WSSs. Results derived with a fixed WSS hence beg the inevitable question: if the WSS were just slightly higher or lower, would the results change significantly? We therefore consider the WSS to be a variable similar to $ucap$ that is varied across its domain from 0 KB (*i.e.*, no significant WSS) to some platform-determined maximum value (in our case study, the L2 cache size provides a natural upper limit—see Section 4.4). The schedulability of an algorithm $\mathcal{A}_x$ with respect to given parameter distributions is hence a function of *two* variables: $S(\mathcal{A}_x, ucap, wss)$. As a result, an algorithm's schedulability must be explored along two axes, which greatly magnifies the required effort but yields more illuminating results. We successfully applied this approach in recent LITMUS$^{\mathrm{RT}}$-based studies (Bastoni *et al.*, 2010a,b, 2011).

Since overhead accounting is carried out after assigning tasks to clusters or partitions (otherwise the overhead in each cluster is overestimated),[3] overhead accounting is slightly simpler under global scheduling than under clustered scheduling. The overhead-aware schedulability procedure for global schedulers is illustrated in Listing 4.3.

Based on the overhead model, appropriate bounds on cache-related preemption and migration overhead (Δ^{cpd}) and cache-related interrupt overhead (Δ^{cid}) are determined for the specified wss (line 3). The remaining scheduler overheads are determined specifically for each task set based on the number of generated tasks $|\tau_j|$ (line 7). Once all overhead bounds are known, a safe approximation of the generated task set is derived as discussed in Chapter 3 (and as appropriate for A_x). Finally, the

[3]The employed bin-packing heuristics likely assign different numbers of tasks to each cluster. Accounting for overheads after the task assignment phase ensures that the correct number of tasks to account for is known in each cluster. For example, this is crucial when bounding the delay due to release interrupts.

```
1  sample_schedulability(A_x, udist, pdist, ucap, wss):
2      set count ← 0
3      set (Δ^cpd, Δ^cid) ← estimate cache-related overheads for working set of size wss under A_x

5      for j ← 1, 2, 3, ..., samples
6          set τ_j ← generate_taskset(udist, pdist, ucap)
7          set (τ_{j,1}, ..., τ_{j,m/c}) ← partition(τ_j)
8          if τ_j could be partitioned:
9              set schedulable ← ⊤
10             foreach k ∈ {1, ..., m/c}:
11                 set (Δ^rel, Δ^sch, Δ^tck, Δ^cxs, Δ^ev, Δ^ipi) ← overheads for |τ_{j,k}| tasks
12                 set τ'_{j,k} ← derive safe approximation of τ_{j,k} (see Chapter 3)
13                 if τ'_{j,k} is not schedulable under A_x on c processors:
14                     set schedulable ← ⊥
15             if schedulable:
16                 set count ← count + 1

18     return ( count / samples )
```

Listing 4.4: Overhead-aware schedulability experiment for partitioned and clustered schedulers.

overhead-unaware schedulability test(s) for $\mathcal{A}_x$ is applied to the safe approximation τ'_j to determine whether τ can be claimed schedulable in the presence of overheads. As a result, the estimated schedulability value reflects all four possible sources of capacity loss and not just algorithmic capacity loss, as is the case with Listing 4.2.

The overhead-aware schedulability setup for clustered and partitioned schedulers is given in Listing 4.4. It is conceptually similar and only differs in the way that overheads are accounted for after partitioning the generated task set τ_j. Recall that c denotes the cluster size. After the tasks have been assigned to $\frac{m}{c}$ clusters (line 7), each cluster is checked individually (line 10). The scheduling overheads are determined based on the number of tasks assigned to the cluster (line 11). Only if the safe approximation of the tasks assigned to each cluster is deemed schedulable is the task set as a whole considered schedulable (lines 9 and 13–16). As in Listing 4.3, the estimated schedulability reflects both overhead-related and algorithmic capacity loss, including failure to partition (line 8).

The setup for schedulability experiments shown in Listings 4.2, 4.3, and 4.4 generates task sets "on the fly." In our implementation, this is not actually the case. Instead, we generate all task sets ahead of time (*i.e.*, Step 6 in Figure 4.1), which ensures that all tested algorithms are tested against identical task sets (and not just identical parameter distributions).

Average vs. worst-case overheads. When conducting overhead-unaware schedulability experiments, there is no difference between SRT and HRT schedulability (aside the use of different schedulability tests). However, when accounting for overheads, the question arises which overheads to use in each case. In the HRT case, it clearly makes sense to assume worst-case overheads. However, assuming maximum overhead is likely excessively pessimistic in the context of SRT systems. We therefore use *average-case* overheads when conducting SRT schedulability experiments. This is acceptable because even if a job incurs above-average overheads, it only incurs a constant amount of additional tardiness, and likely a very small amount at that (if any). Assuming average-case tardiness thus does not violate the SRT schedulability criterion. Further, since tardiness bounds are typically on the order of tens of *milli*seconds, whereas scheduling overheads are in the range of (at the very most) a few hundred *micro*seconds, any additional tardiness due to above-average scheduling overheads is (at least) an order of magnitude smaller than algorithmic sources of tardiness. Depending on a task's WSS, cache-related overheads can reach several milliseconds in the worst case and may thus have a somewhat larger impact on overhead-related tardiness.

The chance of additional tardiness could further be reduced by choosing any approximation between the observed average and the observed maximum (*e.g.*, the 75^{th} or 90^{th} percentile), which could be appropriate for "firm" real-time systems that do not quite require the rigor of HRT analysis but should also have close to zero tardiness. However, assuming *less* than average-case overheads is not safe since then the additional tardiness due to under-estimated overheads could build up slowly and theoretically grow without bound. In practice, however, overheads are unlikely to cause unbounded tardiness because any accumulated tardiness is compensated for when the system experiences phases of idleness. That is, underestimated overheads may result in tardiness being accrued while the system is continuously servicing real-time jobs, whereas slack in the schedule counteracts a build-up of tardiness.

4.1.7 Weighted Schedulability Score

Defining schedulability as a function of two variables, as required when accounting for WSS-dependent overheads, introduces a practical problem. While it is possible to sample each algorithm's schedulability for each combination of $ucap$ and wss over their respective domains (for some finite step size), visualizing the resulting trends becomes difficult. Plotting a function of two variables

requires a 3D projection, and such projections are hard to interpret. Given that schedulability plots commonly include four or more curves (*e.g.*, see Section 4.5 below), the resulting graphs are virtually unintelligible due to the resulting visual clutter and occlusion.

Partial results. A straightforward way to visualize $S(\mathcal{A}_x, ucap, wss)$ is to simply fix one of the two parameters while varying the other over its domain, in which case the resulting 2D plots look like a regular schedulability plot such as the one shown in Figure 4.4. Unfortunately, this approach has three problems that make it undesirable.

The first issue is a question of scale. Generating a schedulability plot for each tested value of wss (and each combination of $udist$ and $pdist$, which are also fixed) quickly results in a very large number of graphs. For example, in the case study presented in the latter part of this chapter, this approach results in several thousand schedulability plots. With such an "unwieldy" data set, understanding and communicating results becomes difficult.

The second issue is related to observing WSS-dependent trends. For example, if Algorithm B is more resilient to WSS increases than Algorithm A (for example, this could be the case because migrations occur less frequently under Algorithm B than under Algorithm A), then Algorithm B could become preferable for large WSSs even if Algorithm A is preferable for small WSSs. When looking at thousands of schedulability plots, each for a fixed wss value, it becomes difficult to notice such trends simply due to the sheer number of graphs that have to be considered in unison.

The third issue is that fixing $ucap$ can introduce an unintentional bias. That is, simply plotting schedulability as a function of wss for a fixed $ucap$ is not a good solution to study WSS-dependent trends. To illustrate this risk, consider the schedulability graph of Algorithms A and B shown in Figure 4.4, and suppose that it depicts only algorithmic capacity loss (*i.e.*, that overheads have not been considered). In this example, Algorithm B performs acceptably only if $ucap \leq 5$. Now suppose overheads are accounted for and schedulability is visualized as a function of wss, and the utilization cap is fixed at $ucap = 6$. This would give Algorithm A an unfair advantage, since it already achieves much higher schedulability than Algorithm B for $ucap = 6$ in the absence of overheads. The resulting graph would thus *not* be representative of Algorithm B's performance with regard to increasing WSSs. In other words, fixing $ucap$ thus could easily lead to incorrect conclusions since then any individual graph does not reflect all utilization caps.

Result aggregation. For these reasons, we devised a simple aggregate metric—the *weighted schedulability score*—that lends itself to visualizing WSS-dependent trends in a small number of 2D graphs. The underlying idea is to collapse each "standard" schedulability curve for a fixed wss value into a scalar—the score—such that *all* considered $ucap$ values are reflected in the aggregate score, which avoids introducing a $ucap$ bias. We first formally define the weighted schedulability score and then explain its benefits and how to interpret the resulting graphs.

Definition 4.1. Let $ustep$ denote the discrete step size used to vary $ucap$ across $[1, m]$, and let $Q = \{1, 1 + ustep, 1 + 2 \cdot ustep, \ldots, m\}$ denote the set of all sampled values of $ucap$. The *weighted schedulability score* $W(wss)$ for a given WSS is defined as

$$W(\mathcal{A}_x, wss) = \frac{\sum_{ucap \in Q} S(\mathcal{A}_x, ucap, wss) \cdot ucap}{\sum_{ucap \in Q} ucap}. \tag{4.1}$$

For a given wss, an algorithm's schedulability $S(\mathcal{A}_x, ucap, wss)$ is summed across all tested $ucap$ values, where each schedulability result is weighted by the corresponding utilization cap. The resulting score is normalized to the range $[0, 1]$ by dividing it by the sum of all tested $ucap$ values. Weighting individual schedulability results by $ucap$ reflects the intuition that high-utilization task sets have higher "value" since they are more difficult to schedule.

The weighted schedulability score has several advantages that make it suitable for visualizing WSS-dependent trends without introducing a $ucap$ bias.

1. Since it has only one parameter (for given task parameter distributions), it can be easily visualized as a function of wss in a 2D plot. Figure 4.5 illustrates how weighted schedulability scores can be visualized.

2. The resulting plot is influenced by each combination of wss and $ucap$, *i.e.*, each graph represents *all* of the generated data for a particular task parameter distribution.

3. Since many schedulability graphs are aggregated into fewer graphs, larger studies become manageable. For example, in our case study, 59,940 individual schedulability graphs were reduced to 1,620 weighted schedulability graphs

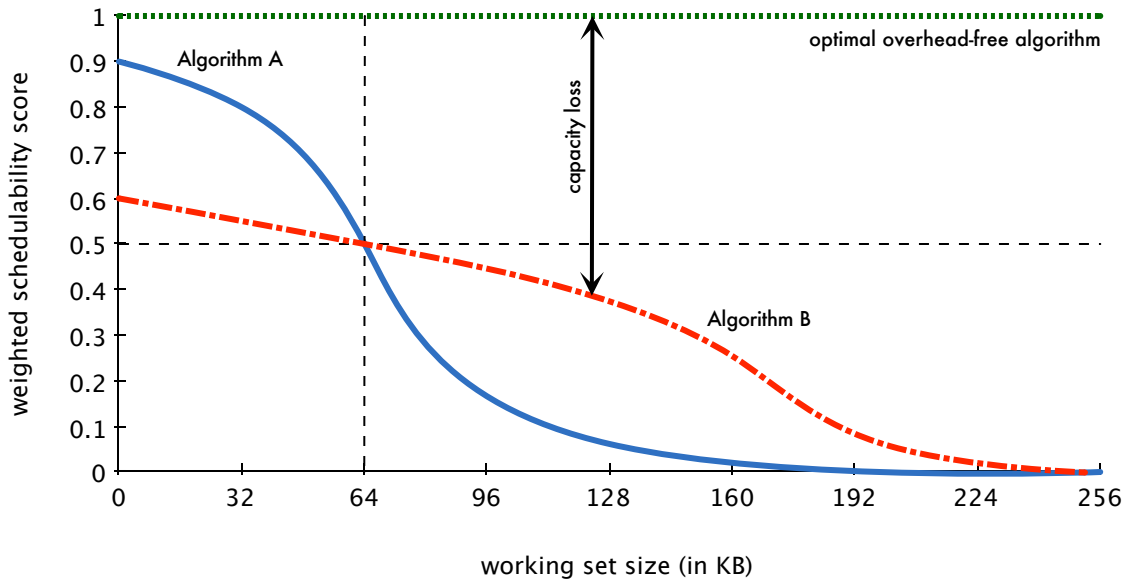


Figure 4.5: Illustration of a weighted schedulability score plot. The two curves show $W(A, wss)$ and $W(B, wss)$. Each numeric value does not directly correspond to a number of scheduled task sets. However, the score is normalized such that an optimal algorithm scores 1 in the absence of overheads. The higher an algorithm scores, the closer it comes to scheduling all task sets. Each point in this plot corresponds to a graph of the type shown in Figure 4.4. This example illustrates that an algorithm that is not preferable for small WSS may in fact become preferable for larger WSSs if it is more resilient to WSS increases. The weighted schedulability score is designed to expose such trends without introducing a $ucap$-dependent bias.

4. WSS-dependent trends are readily apparent. For example, in Figure 4.5, Algorithm A scores higher if $wss < 64$, but Algorithm B is more resilient to cache-related overheads and thus is preferable for larger WSSs.

5. Due to the normalization, the weighted schedulability score of an optimal, overhead-free algorithm is 1, just as in the case of regular schedulability. An algorithm that fails to schedule any task sets has a weighted schedulability score of 0.

6. Consequently, increasing capacity loss corresponds to decreasing the weighted schedulability score. Figure 4.5 illustrates the "gap" between Algorithm B and an optimal, overhead-free scheduler that is caused by capacity loss.

The weighted schedulability score is a synthetic construct in the sense that it does not correspond to any "real" quantity.[4] One could easily devise alternate ways to aggregate large schedulability data sets. Nonetheless, we have found weighted schedulability scores to be very useful in understanding WSS-dependent trends and employ it in our case study.

It should be noted, however, that since the weighted schedulability score *aggregates* data, it loses some detail and is not a substitute for regular schedulability graphs. Instead, it is intended to augment schedulability graphs that show schedulability for a fixed WSS with an indication of how different algorithms are affected by changes in cache-related overheads.

4.1.8 Tardiness

Besides the metrics schedulability and weighted schedulability score, schedulers can also be compared on the basis of the magnitude of tardiness bounds. Naturally, a lower tardiness bound is preferable to a larger one. However, absolute tardiness bounds cannot be easily compared across different task sets and task parameter distributions. For example, a maximum tardiness of $10ms$ likely has a different qualitative impact on a task with a relative deadline of $5ms$ than on a task with a relative deadline of $200ms$. Instead, tardiness bounds should be considered in relation to the urgency of each task. We therefore report the following normalized measure of tardiness.

Definition 4.2. Let B_i denote the maximum tardiness that any job of an implicit-deadline task T_i may incur. The maximum *relative tardiness* of T_i is given by $\frac{B_i}{p_i}$.

Relative tardiness has practical significance. Suppose that each job processes one work item such as a video frame, sensor reading, etc. Then a buffer of $\left\lceil \frac{B_i}{p_i} \right\rceil$ work items is sufficient to mask the effects of tardiness. That is, task T_i lags at most $\left\lceil \frac{B_i}{p_i} \right\rceil$ jobs behind, and this value corresponds to the maximum number of unprocessed work items that must be stored to mask the effects of tardiness (additional storage may be required to account for differences in the rate of execution of producers and consumers; such buffering is unrelated to tardiness concerns). We consider this task-specific characterization of deadline misses to be a more useful measure than absolute tardiness values.

Another tardiness-related issue arises when reporting average and maximum tardiness bounds (either relative or absolute) for multiple task sets. If the tardiness of at least one task set cannot

[4] In (Bastoni *et al.*, 2010a,b, 2011), we referred to the weighted schedulability score simply as "weighted schedulability." We added the "score" moniker in this dissertation to emphasize its synthetic nature and to avoid ambiguities.

be bounded, *i.e.*, if there is potentially no finite bound on maximum tardiness, then average and maximum tardiness are ill-defined. We hence excluded task sets with unbounded tardiness when reporting average and maximum relative tardiness bounds.

It is important to note that our methodology is based on analytical tardiness *bounds* and not on *observed* tardiness. This is consistent with the use of schedulability tests in the HRT case. While maximum tardiness as observed during simulations is likely lower than analytical bounds, no guarantees can be derived from finite simulations. The use of analytical bounds is thus preferable from a predictability point of view.

Summary. Our overhead-aware evaluation methodology consists of an OS phase and an analytical phase. Schedulers are compared based on three metrics: schedulability, which reflects sensitivity to utilization increases, weighted schedulability score, which reflects sensitivity to WSS increases, and relative tardiness, which reflects buffer requirements. In the remainder of this chapter, we report on a case study that demonstrates how the proposed methodology was used to compare the schedulers implemented in LITMUS$^{\mathrm{RT}}$.

4.2 Case Study

We conducted a case study to answer Question Q1 from Chapter 1 for our test platform, namely to identify the LITMUS$^{\mathrm{RT}}$ plugins and configurations that are best suited to satisfying HRT and SRT constraints on a 24-core Intel Xeon platform. In the remainder of the chapter, we first discuss the general setup of the case study and then report in Sections 4.3 and 4.4 on the scheduling and cache-related overheads that we measured during the OS phase. Besides discussing the most relevant trends in the overhead data, we also discuss the process of how they were measured to augment the high-level discussion from the preceding section. Finally, in Section 4.5, we discuss the schedulability study that we conducted and highlight key results that answer Question Q1 for our platform.

4.2.1 Platform

The hardware platform underlying our case study, a 24-core Intel Xeon L7455 system, was already presented in detail in Section 2.1. To summarize, the L7455 is a 24-core 64-bit uniform memory access (UMA) machine with four physical sockets. Each socket contains a chip with six cores

running at 2.13 GHz. All cores in a socket share a unified 12-way set associative 12 MB L3 cache, while groups of two cores each share a unified 12-way set associative 3 MB L2 cache. Every core also includes an 8-way set associative 32 KB L1 data cache and an identical L1 instruction cache. All caches have a line size of 64 bytes (*i.e.*, 8 words). The system is equipped with a 64 GB main memory; the memory bus operates at an effective speed of 1066 MHz.

4.2.2 Tested Schedulers

In total, we evaluated 22 configurations of LITMUS$^{\mathrm{RT}}$ scheduler plugins in our case study. We define each of these configurations next and introduce a naming convention to uniquely identify each in the subsequent discussion.

There are four cluster sizes that align well with the topology of our test platform: partitioned scheduling ($c = 1$) to ensure L1 cache affinity, clustered scheduling based on L2 cache affinity ($c = 2$), clustered scheduling based on L3 cache affinity ($c = 6$), and global scheduling ($c = 24$). Recall from Section 3.3.6 that we implemented two plugins that support clustered scheduling with arbitrary cluster sizes (C-EDF and PD^2), and three global or partitioned plugins (P-EDF, P-FP, and G-EDF). We instantiated C-EDF with cluster sizes $c = 2$ and $c = 6$, and PD^2 with cluster sizes $c = 2$, $c = 6$, and $c = 24$. This yields a total of five event-driven plugin configurations (P-FP, P-EDF, G-EDF, and two C-EDF variants) and three quantum-driven plugin configurations (three PD^2 configurations). When referring to plugin configurations, we prefix the plugin name with C2- and C6- when referring to clustered scheduling with $c = 2$ and $c = 6$, respectively, and let P- and G- indicate partitioned and global scheduling, respectively.

Each of the plugins supports both dedicated interrupt handling (one processor is reserved for interrupt handling and $m - 1$ processors service jobs) and global interrupt handling (all m processors service both interrupts and jobs). In the case of partitioned scheduling, dedicated interrupt handling simply reduces the number of partitions by one. In the case of global scheduling, dedicated interrupt handling reduces the system to a $(m - 1)$-processor platform. In the case of clustered scheduling with $1 < c < m$, however, dedicated interrupt handling could be either implemented on a per-cluster basis (thereby losing capacity in each cluster) or by reserving only a single processor from one of the clusters. The latter has the disadvantage that clusters are no longer of uniform size; however, the former causes unacceptable capacity loss (*i.e.*, if $c = 2$, then half of the available processors

Scheduler	Plugin	clusters count $\times$ c	processors servicing release interrupts	jobs	quanta	cache affinity
P-FP-Rm	P-FP	24×1	24	24	—	L1
P-FP-R1	P-FP	23×1	1	23	—	L1
P-EDF-Rm	P-EDF	24×1	24	24	—	L1
P-EDF-R1	P-EDF	23×1	1	23	—	L1
C2-EDF-Rm	C-EDF	12×2	24	24	—	L2
C2-EDF-R1	C-EDF	11×2 & 1×1	1	23	—	L2
C6-EDF-Rm	C-EDF	4×6	24	24	—	L3
C6-EDF-R1	C-EDF	3×6 & 1×5	1	23	—	L3
G-EDF-Rm	G-EDF	1×24	24	24	—	none
G-EDF-R1	G-EDF	1×23	1	23	—	none
C2-aPD^2-Rm	PD^2	12×2	24	24	aligned	L2
C2-sPD^2-Rm	PD^2	12×2	24	24	staggered	L2
C2-aPD^2-R1	PD^2	11×2 & 1×1	1	23	aligned	L2
C2-sPD^2-R1	PD^2	11×2 & 1×1	1	23	staggered	L2
C6-aPD^2-Rm	PD^2	4×6	24	24	aligned	L3
C6-sPD^2-Rm	PD^2	4×6	24	24	staggered	L3
C6-aPD^2-R1	PD^2	3×6 & 1×5	1	23	aligned	L3
C6-sPD^2-R1	PD^2	3×6 & 1×5	1	23	staggered	L3
G-aPD^2-Rm	PD^2	1×24	24	24	aligned	none
G-sPD^2-Rm	PD^2	1×24	24	24	staggered	none
G-aPD^2-R1	PD^2	1×23	1	23	aligned	none
G-sPD^2-R1	PD^2	1×23	1	23	staggered	none

Table 4.1: List of evaluated schedulers ($m = 24$).

would be reserved for interrupt handling). We therefore accept non-uniform cluster sizes in clustered schedulers when applying dedicated interrupt handling. This, of course, must be considered during the task assignment phase. When referring to plugin configurations with global interrupt handling, we use the suffix -Rm to indicate that release interrupts are handled by all processors; when referring to plugin configurations that use dedicated interrupt handling, we similarly use the suffix -R1.

Finally, when using quantum-driven plugins, LITMUS$^{\mathrm{RT}}$ can be configured to use either aligned or staggered quanta. Either choice can be combined with both dedicated and global interrupt handling, which results in four configurations for each of the three considered PD^2 cluster sizes. When referring to PD^2 with aligned quanta, we denote the scheduler as aPD^2; analogously, when referring to PD^2 with staggered quanta, we denote the scheduler as sPD^2.

As listed in Table 4.1, these options yield ten event-driven and twelve quantum-driven schedulers.

4.2.3 Parameter Distributions

After deriving overhead models (see Section 4.1.3) for each of the 22 evaluated schedulers, we conducted a large-scale schedulability study to assess each scheduler's suitability to ensuring HRT and SRT constraints on the test platform. Recall that a task set generation procedure is the core of a schedulability experiment (Section 4.1.5), and that task sets are generated by randomly choosing each utilization u_i and each period p_i from two distributions $udist$ and $pdist$ (Section 4.1.4). The results of the schedulability experiment thus fundamentally depend on the employed parameter distributions, which should include a wide range of possible parameter choices. If the system under evaluation is intended for specific use cases, then it makes sense to choose distributions to resemble known real-world applications (*e.g.*, plant controllers for HRT constraints, video games for SRT constraints, *etc.*). Alternatively, task sets can be generated using synthetic distributions that are known to stress specific sources of algorithmic and overhead-related capacity loss (*e.g.*, by generating task sets that are difficult to partition). Since we are interested in covering a large parameter space with our case study, we chose the latter approach.

In our case study, we combined three period and nine utilization distributions for a total of 27 tested *scenarios* similar to those used by Calandrino *et al.* (2006), which in turn were chosen based on parameter distributions proposed by Baker (2005). We generated implicit-deadline periodic tasks by first generating task utilizations using three uniform, three bimodal, and three exponential distributions. The ranges for the uniform distributions were [0.001, 0.1] (*light*), [0.1, 0.4] (*medium*), and [0.5, 0.9] (*heavy*). For the bimodal distributions, utilizations uniformly ranged over [0.001, 0.5) or [0.5, 0.9] with respective probabilities of 8/9 and 1/9 (*light*), 6/9 and 3/9 (*medium*), and 4/9 and 5/9 (*heavy*). For the exponential distributions, utilizations were generated with a mean of 0.10 (*light*), 0.25 (*medium*), and 0.50 (*heavy*). With exponential distributions, we discarded any points that fell outside the allowed range of $[0, 1]$. Integral task periods were then generated using three uniform distributions with ranges [$3ms$, $33ms$] (*short*), [$10ms$, $100ms$] (*moderate*), and [$50ms$, $250ms$] (*long*). Besides the case study reported herein, we have also employed these parameter distributions in several published studies (Brandenburg *et al.*, 2008; Brandenburg and Anderson, 2009a; Brandenburg *et al.*, 2009, 2011; Bastoni *et al.*, 2010b, 2011). As mentioned in Section 4.1.6, we pre-generated all task sets to ensure that each scheduler is tested with the same task sets. Task parameters were

generated with the Mersenne Twister pseudo-random number generator (Matsumoto and Nishimura, 1998) as implemented in the Python 2.6 standard library (van Rossum and contributors, 2010).

The rationale for the choice of these distributions is the following. When using light utilization distributions, many tasks "fit" into the given utilization cap. The resulting large number of tasks in each task set exposes overhead-related capacity loss (*e.g.*, there are a large number of interrupt sources, ready queues are long, *etc.*). When using heavy utilization distributions, only few, high-utilization tasks are being generated, which results in task sets that are difficult to partition but relatively overhead insensitive. Medium utilization distributions represent a compromise that contains elements of both challenges.

The three uniform utilization distributions further emphasize these properties since they disallow the generation of non-light or non-heavy tasks. The bimodal utilization distributions are employed because they generate task sets that contain both low-utilization and high-utilization tasks. Such task sets are challenging to schedule and thus expose algorithmic capacity loss (especially under G-EDF). Finally, the exponential distributions are included because they also generate difficult to schedule task sets (similar to the bimodal distributions), and because they create the most "realistic" utilization distributions (*i.e.*, many low-utilization tasks, a few high-utilization tasks).

The choice of period distribution affects the outcome in two ways. First, period distributions with a large ratio of longest-possible period to shortest-possible period stress interrupt accounting since jobs of long-period tasks may incur a large number of release interrupts. Second, the period length influences the impact of overheads: the shorter the period, the larger the relative magnitude of overheads. The short period distribution is thus taxing for high-overhead plugins, whereas the long period distribution deemphasizes implementation efficiency. The moderate period distribution contains the timing constraints of most multimedia applications, as determined by human sensory capabilities.

Taken together, the 27 tested distribution scenarios cover a wide range of task set compositions. It should be noted, however, that since these parameter distributions are synthetic, they do not correspond to actual applications. This schedulability study thus does not directly predict performance of the evaluated schedulers for any particular application. Rather, the performance evaluation is indirect: a scheduler that performs well in each of the tested scenarios likely also performs well for real-world applications (implemented on our hardware platform), and, conversely, a scheduler that

performs badly in all or most cases is also likely not a good choice for most use cases (again, on our hardware platform). Nonetheless, in future work, it would beneficial to augment our results with realistic parameter distributions obtained from representative multiprocessor real-time applications.

4.3 Scheduling Overheads

To construct the overhead model required for overhead-aware schedulability experiments, a large number of overhead samples must be collected to determine statistically valid average-case and worst-case estimates of overheads discussed in Chapter 3.

There are three groups of overheads. *Direct kernel overheads* arise when the kernel is executing and the scheduled job is not (*e.g.*, when a release interrupt is serviced). *Cache-related* overheads manifest indirectly as a slowdown of the executing job. Finally, *latencies* are intervals during which a job is delayed that do not correspond to execution times (of either the kernel or jobs). Of these, direct overheads and latencies are easy to measure since they have a well-defined start and end time. That is, direct overheads and latencies can be sampled simply by recording when intervals of overhead begin and end. In contrast, cache-related overheads are of a diffuse nature and cannot be directly measured since there is no clear end time, *i.e.*, the restoration of cache affinity is interwoven with normal execution. In this section, we discuss how direct overhead and latencies are measured in LITMUS$^{\mathrm{RT}}$ and discuss notable overhead trends from our case study. Cache-related overheads and trends are discussed thereafter in Section 4.4.

4.3.1 Tracing, Post-Processing, and Statistics

In LITMUS$^{\mathrm{RT}}$, overhead samples are collected using the Feather-Trace framework (Brandenburg and Anderson, 2007a). Recall from Section 3.3 that the Feather-Trace framework provides event triggers and a wait-free, multi-writer, single-reader buffer implementation. Direct overheads are measured by placing event triggers at the beginning and end of overhead-causing code segments. For example, there is an event trigger at the beginning of Linux's schedule() function. Similarly, there is an event trigger at the beginning of the timer tick ISR, and another event trigger after the last statement of the timer tick ISR. When activated, each trigger invokes a simple function that places an

event record into a shared wait-free buffer instance. Each event record consists of 15 bytes that store the following information:

- the current time (in cycles, as measured by the TSC),
- a unique event ID that describes the code path (*e.g.*, "schedule() entered"),
- the processor ID,
- whether the currently-scheduled job is a real-time task, and
- a unique 32-bit sequence number (subject to overflow).

The wait-free trace buffer is periodically flushed to disk by a non-real-time background process.

Latencies are recorded similarly. To record IPI latency, the sending processor stores an event record when the IPI is generated, and the receiving processor records when the IPI is processed by the kernel. Note that this requires time stamps from different processors to be comparable, which is not necessarily the case in modern multiprocessors. In our test platform, however, time stamps are comparable because all TSCs are driven by the same clock signal, and because processors are prohibited from entering sleep states during the experiments.

Recall from Section 3.4.2 that a job release is delayed by both hardware and interrupt latency, which together comprise event latency (Δ^{ev}). It is not possible to directly measure this latency from within the system itself since the physical time at which the release interrupt was raised is unknown to the kernel (*i.e.*, time t_0 in Figure 3.9). However, when jobs are released by programmable timers, then event latency can be computed since the time when the ISR *should* have occurred is known. In LITMUS$^{\mathrm{RT}}$, event latency is recorded in the release timer ISR by storing two event records: one for the occurrence of the physical event that is backdated to the programmed release time, and one corresponding to the beginning of the ISR execution, which stores the actual current time.

As a special case, scheduling overhead is measured in two parts in LITMUS$^{\mathrm{RT}}$. This is because the Linux scheduler executes some deferred resource management code *after* a context switch has occurred (this is due to locking concerns). The corresponding Feather-Trace events are denoted SCHED and SCHED2 in the source code; the sum of both overhead sources yields Δ^{sch}.

Overhead tracing. Broadly speaking, an overhead experiment is carried out as follows. First, m background processes are launched that create memory bus and cache contention by repeatedly accessing large arrays. This is required so that the recorded overheads correspond to a system under heavy load (otherwise, worst-case overheads would be considerably underestimated). Thereafter, all overhead-related Feather-Trace triggers are activated and the background process responsible for flushing the trace buffer to disk is launched. Finally, an experimental task set τ is launched. After all tasks have finished their initialization phase, the task set is synchronously released, after which tasks execute periodically. Job releases are triggered by timer interrupts. After a pre-determined time (60 seconds in our case study), all tasks in τ terminate, the event record buffer is emptied, and all background processes are terminated.

Post-processing. This results in a large binary log of event records. In a post-processing step, the binary log is sorted according to sequence number (due to the wait-free nature of the buffer, events can be recorded out of order). The sorted event log is then split into overhead samples by finding "matching" overhead start and end event records. Let E_x and E_y denote two event records with sequence numbers x and y, respectively, where $x < y$. E_x and E_y are considered to be *matching* if and only if

1. E_x and E_y occurred on the same processor or are IPI latency event records (which are necessarily recorded on two separate processors),
2. E_x is a start event record of type X (*e.g.*, scheduler invocation),
3. E_y is an end event record of type X,
4. no other event records of type X exist in between E_x and E_y (with respect to the recording processor), and
5. all sequence numbers between x and y are present in the log.

Condition 5 is required to detect overflows of the trace buffer (if any), which could result in incorrect measurements if there is a "hole" in the log between E_x and E_y. The trace buffer can overflow because the system generates event records at a very high rate for large task counts, which is problematic if the buffer-flushing background process is starved by real-time processes.

Let t_x and t_y denote the times at which E_x and E_y were recorded (according to the log). If all five conditions are met, then E_x and E_y match and represent a recorded overhead sample of type X and length $t_y - t_x$. The difference $t_y - t_x$ includes some overhead due to Feather-Trace; however this additional cost is small in comparison to the measured intervals and hence not compensated for. Event records for which no matching event record can be found are discarded.

Not all recorded overhead samples are necessarily relevant. There are two groups of overheads: **(i)** those that are only relevant if a scheduled real-time job is directly affected by them, and **(ii)** those that affect real-time jobs even if the currently scheduled process is a background process. An example of the former is the cost of a scheduling decision that does not involve real-time tasks (*e.g.*, if an idle processor schedules a background process); an example of the latter is a release interrupt that stops a background process since the to-be-released job experiences a delay regardless of the identity of the stopped process. An overhead sample belonging to group (i) is considered *valid* only if either E_x or E_y recorded that a real-time task was involved (*i.e.*, scheduled at the time of the event, or immediately thereafter). Overhead samples belonging to group (ii)—which includes IPI latency, event latency, and release interrupt overhead samples—are always considered valid. All invalid overhead samples are discarded to avoid influencing overhead experiments with data stemming from background activity unrelated to real-time tasks.

The process described so far yields overhead samples for a single task set τ. However, the magnitude of some overhead sources such as scheduling overhead indirectly depend on the number of tasks in τ. Therefore, many overhead traces covering a wide range of n should be collected for several task set compositions (and under each tested scheduler). This allows worst-case and average-case overheads under each tested scheduler to be estimated as a function of n when constructing the overhead model (as discussed in Section 4.1.3).

Unbiased estimators. Tracing task sets of different sizes and with different task parameters almost certainly results in traces with a variable number of valid samples. From a statistics point of view, comparing observed maxima from populations of different sizes is problematic. To obtain an unbiased estimator for the worst-case overhead, the same number of samples should be used in the analysis of each trace. One method to achieve this is to simply discard extraneous samples prior to analysis.

However, truncating all traces to the length of the shortest trace is not desirable since this would bias the results to reflect only jobs that executed early during the trace interval. We therefore randomly shuffle each trace before truncation. This ensures that each job is reflected with equal probability, regardless of whether it was released toward the beginning or end of the experiment. As a result, we obtain statistically unbiased estimators of worst-case and average-case overheads.

Outlier filtering. In repeated measurements of some overhead, a small number of samples may be "outliers," *i.e.*, some samples appear to not match the overall trend. While outliers typically do not significantly affect average-case estimates (due the large number of correct samples), large outliers can dramatically alter the estimated maximum.

An example of this effect is shown in Figure 4.6, which depicts the distribution of scheduling overhead samples measured under P-EDF-Rm when scheduling one task on each processor (*i.e.*, $n = 24$). The main body of overhead samples is in the range from $1\mu s$ to $27\mu s$. However, a single outlier increases the observed maximum to more than $129\mu s$. This sample likely does not reflect true scheduling overhead. Rather, the measurement was almost certainly disturbed by an inopportune interrupt since interrupt delivery is disabled in the scheduler only *after* the initial timestamp is recorded. Since interrupts are accounted for separately, outliers caused by interrupts should not be included in the estimate of Δ^{sch} and must thus be filtered.

Statistical outliers are either caused by measurement error (such as the just-discussed interference from interrupts), or by the actual occurrence of rare, high-overhead events. Ideally, it would be desirable to only remove outliers of the first kind without affecting outliers of the second kind. Unfortunately, this is not practical.

For one, statistical methods such as interquartile range (IQR) outlier filters[5] or percentile-based cutoffs cannot discern between the two kinds. Detecting measurement error after the fact is fundamentally a subjective effort, which can be challenging even for humans. For example, in the case of long-tail distributions, any notion of a "correct" cutoff between the tail of true measurements and interrupt-related outliers is ambiguous at best.

A second reason is related to the use of Linux and x86 hardware as the test platform, neither of which was designed with real-time systems in mind. This is reflected by inopportune, long-

[5]The US National Institute of Standards and Technology (NIST) suggests IQR as a standard technique to remove outliers (NIST/SEMATECH, 2010). We apply IQR filters to traces affected by interrupt interference; see Section 4.3.2.

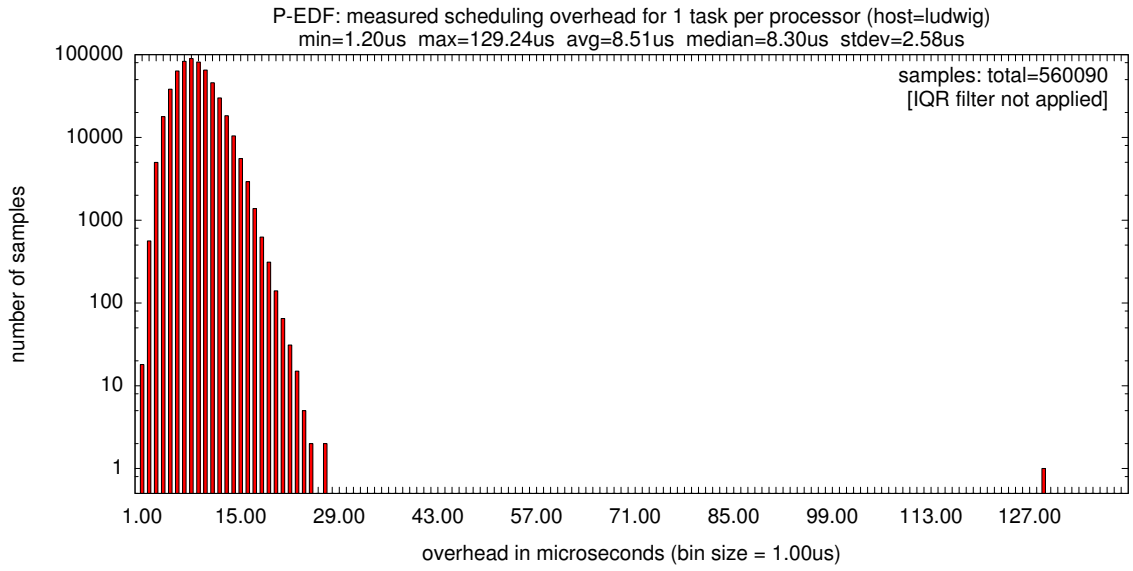


Figure 4.6: Example of an outlier in overhead measurements. The histogram shows the distribution of scheduling overhead samples under P-EDF-Rm when scheduling one task per processor. A single sample, out of more than half a million valid samples, raises the measured maximum from $27\mu s$ to $129\mu s$. The outlier was likely disturbed by an interrupt and does not reflect true scheduling overhead. (Note the logarithmic scale.)

running interrupt handlers (such as network-related packet processing or cycle-stealing SMIs—recall Section 2.1.3), which can cause "true outliers," *i.e.*, such samples reflect infrequent events that are not caused by measurement error. In actual, production-quality real-time systems, such events are cause for significant concern and should not be ignored. In the context of our research goal, however, we deem it acceptable to implicitly filter such rare events when discarding outliers due to measurement error because they are entirely unrelated to scheduler implementation concerns and tradeoffs.

In essence, our measurements confirm that Linux and the underlying x86 hardware are not truly HRT-capable, which is hardly surprising. As discussed in Chapter 3, the choice of stock Linux as the starting point of LITMUS$^{\mathrm{RT}}$, and the use of x86 hardware, represent a compromise between availability and real-time capability. In some cases, this compromise necessitates the use of statistical outlier filters to remove both measurement errors and statistically infrequent events. In future work, we plan to rebase LITMUS$^{\mathrm{RT}}$ on top of the Linux PREEMPT_RT patch to benefit from its reduced latencies, and to focus our effort on embedded multicore platforms more suited to

real-time computing (once such platforms become widely available). Additionally, it would also be interesting to replicate our experiments in other, non-Linux-based RTOSs.

4.3.2 Experiments

We applied the above-described experimental setup to collect overhead samples under each of the 22 evaluated schedulers listed in Table 4.1. We traced task sets ranging in size from one to 20 tasks per processor, *i.e.*, task set sizes ranged from $n = 24$ to $n = 480$ in steps of $m = 24$.

Each task set of size $n = k \cdot m$, where $1 \leq k \leq 20$, was generated by creating m partitions of k tasks each. Assembling tasks sets from individual partitions ensures that the experimental process is independent of bin-packing considerations and that there is an equal number of tasks assigned to each processor. Partitions were simply merged as required under clustered and global schedulers.

When generating a partition τ_i of k tasks, we used moderate periods and randomly chose one of the following utilization distributions: light uniform, light bimodal, light exponential, medium uniform, and medium bimodal. Based on the selected task parameter distributions, k tasks $\tau_i = \{T_1, \ldots, T_k\}$ were generated. If this resulted in $u_{\text{sum}}(\tau_i) > 1$, then τ_i was discarded and regenerated to ensure that generated task sets do not over-utilize the system. Heavier utilization distributions were not considered because they are not suited to generating feasible partitions for large k. Additionally, the period distribution was constrained to ensure that the least-common multiple of all generated periods, *i.e.*, the *hyperperiod*, did not exceed 60 seconds.

For each task set size $n = k \cdot m$, we generated ten task sets. Each of the resulting 200 task sets was executed and traced for 60 seconds under each of the 22 tested schedulers as described above, which corresponds to 73 hours of continuous execution. (The actual experiments were more time consuming since each task set requires a set-up and tear-down phase.) In total, the overhead experiments resulted in 510 GB of event records, which contained more than 11 billion valid overhead samples after event record matching. For each task set size, scheduler, and type of overhead, we merged the samples from each of the ten task sets. The resulting overhead data sets were shuffled, truncated to a common length, and filtered for outliers before computing the observed average- and worst-case overheads.

Interrupts and outliers. Besides release interrupts and timer ticks, our system was also frequently servicing long-running ISRs related to disk and network I/O during overhead tracing. It would be desirable to avoid such I/O during the experiments, but unfortunately it was unavoidable since trace data had to be written to disk. Network connectivity was required to administer the experiments (our Xeon L7455 system is a "headless" system, *i.e.*, it does not have its own display or keyboard).

In our case study, we observed outliers *only* in data sets from overhead sources that can be disturbed by interrupts. In fact, outliers occurred comparatively frequently in measurements of event latency, IPI latency, and context-switch overhead, which are all strongly affected by interrupt delivery. In contrast, outliers occurred rarely in the measurements of scheduling overhead since interrupt delivery is disabled throughout most parts of the measured scheduling code path. Further, *all* measurements of release interrupt overhead and timer tick overhead were completely outlier-free since interrupt delivery is disabled entirely while these ISRs execute. Overall, our data indicates that long-running ISRs are the primary (or even sole) cause of outliers on our platform. This highlights the need for split interrupt handling in future versions of LITMUS$^{\mathrm{RT}}$.

We addressed outliers in our case study as follows. Data sets without apparent outliers, namely, timer tick overhead, release interrupt overhead, and parts of the scheduling overhead (*i.e.*, SCHED2), were not filtered. In data sets with few outliers (such as the first part of scheduling overhead, *i.e.*, SCHED), we identified outliers using histograms plotted on a logarithmic scale as shown in Figure 4.6 and removed them manually. Finally, data sets affected by more frequent outliers were filtered using an IQR filter with extent 12.[6] As a special case, measurements of IPI latency under dedicated interrupt handling were disturbed by ISRs to such an extent that IPI latency measurements from global interrupt handling were used in all cases. The reason for this is that the processor dedicated to interrupt handling was frequently servicing long-running I/O-related ISRs.

4.3.3 Results

After obtaining and processing average-case and worst-case overheads for each of the 22 schedulers, we plotted the resulting trends as function of task set size n and derived corresponding non-decreasing, piece-wise linear overhead models (Section 4.1.3). In the following, we discuss select trends that

[6] An IQR filter with extent 12 considers a sample x_i to be an outlier if $x_i > q_{50} + 12 \cdot (q_{75} - q_{25})$, where q_{25}, q_{50}, and q_{75} denote the 25$^{\text{th}}$, 50$^{\text{th}}$, and 75$^{\text{th}}$ percentile, respectively. NIST/SEMATECH (2010) provides a succinct tutorial.

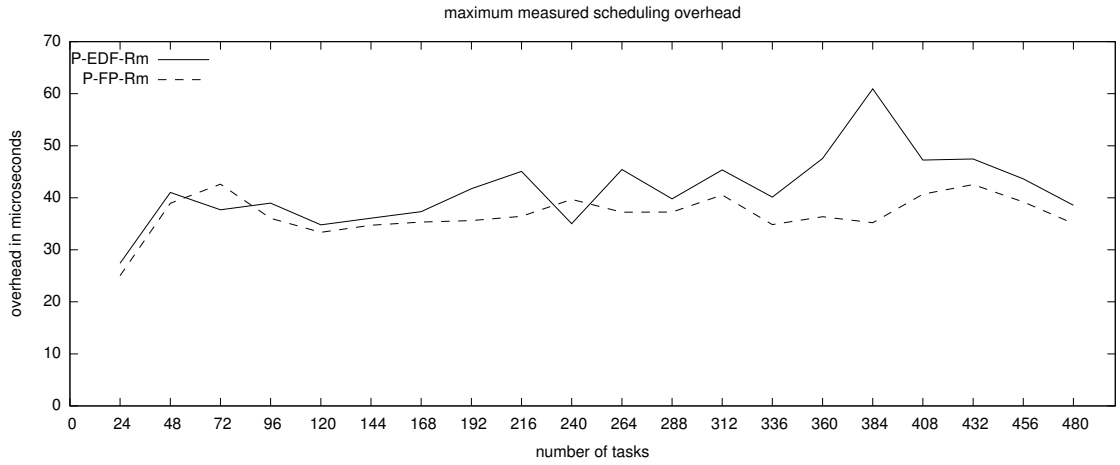
highlight implementation tradeoffs of particular relevance to our case study. The entire data set of recorded average-case and worst-case overheads, graphs visualizing all overhead distributions, and graphs visualizing overhead trends as functions of task count can be downloaded from the companion web site to this dissertation (Brandenburg, 2011).

P-FP vs. P-EDF. One reason for the popularity of FP and P-FP scheduling is that it can be efficiently implemented using bitfield-based ready queues (recall Section 3.3.6.1). As a result, the runtime complexity of finding the next highest-priority ready job does not depend on the number of ready tasks. In contrast, the P-EDF plugin uses a binomial heap to implement each per-processor ready queue. One might hence expect scheduling overhead to be higher under P-EDF than under P-FP (under either dedicated or global interrupt handling).

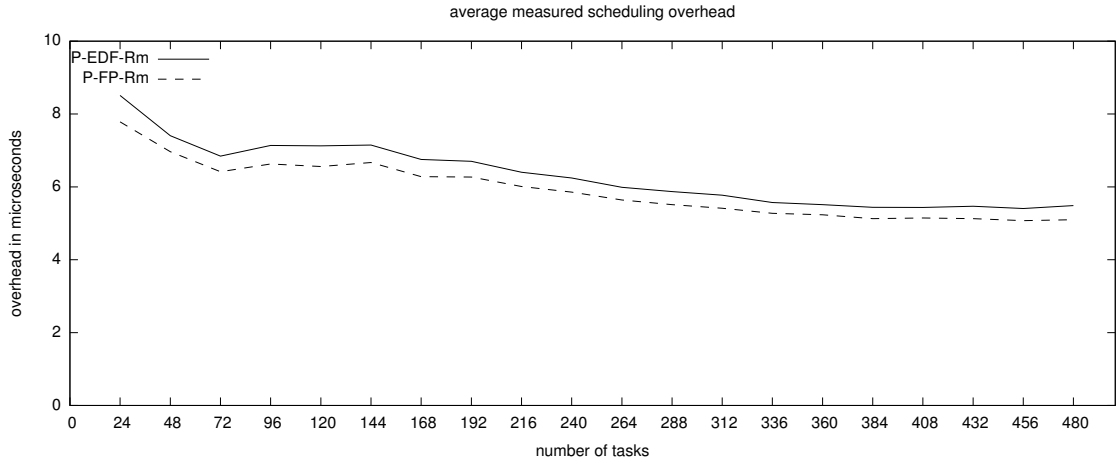
As can be seen in Figure 4.7, this is indeed the case, but the difference in overhead is rather small. Inset (a) shows worst-case scheduling overhead under P-EDF-Rm and P-FP-Rm as a function of n. Two trends are apparent. First, worst-case scheduling overhead is slightly higher under P-EDF-Rm than under P-FP, with the exception of $n = 72$ and $n = 240$, which we suspect is due to variance inherent in empirical measurements. Second, worst-case scheduling overhead under either scheduler does not appear to be strongly correlated to the task set size.

Both trends manifest more clearly in the average case, which is shown in Figure 4.7(b). Under P-EDF-Rm, average-case scheduling overhead is consistently higher than under P-FP-Rm, although only by a fraction of a microsecond. This illustrates that the bitfield-based ready queue used in the P-FP plugin is indeed more efficient than the binomial heap used in the P-EDF plugin; however, this also illustrates that ready queue manipulation comprises only a small faction of the total overhead.

Surprisingly, average-case scheduling overhead *decreases* under both schedulers with increasing task count. At least in the case of P-EDF-Rm, this is very counterintuitive since dequeuing the highest-priority job from the ready queue requires $O(\log n)$ time in a binomial heap. However, the more-frequent invocation of the scheduler likely results in an increased cache hit rate, which lowers the cost of a scheduler invocation on average. Another contributing factor is that task sets with high task counts also have a high utilization, which means that the background processes that create memory contention execute less frequently. The lowered scheduling overhead is clearly apparent in Figure 4.8, which shows the distribution of scheduling overhead samples for $n = 24$ and $n = 480$.



(a) Maximum observed scheduling overhead.

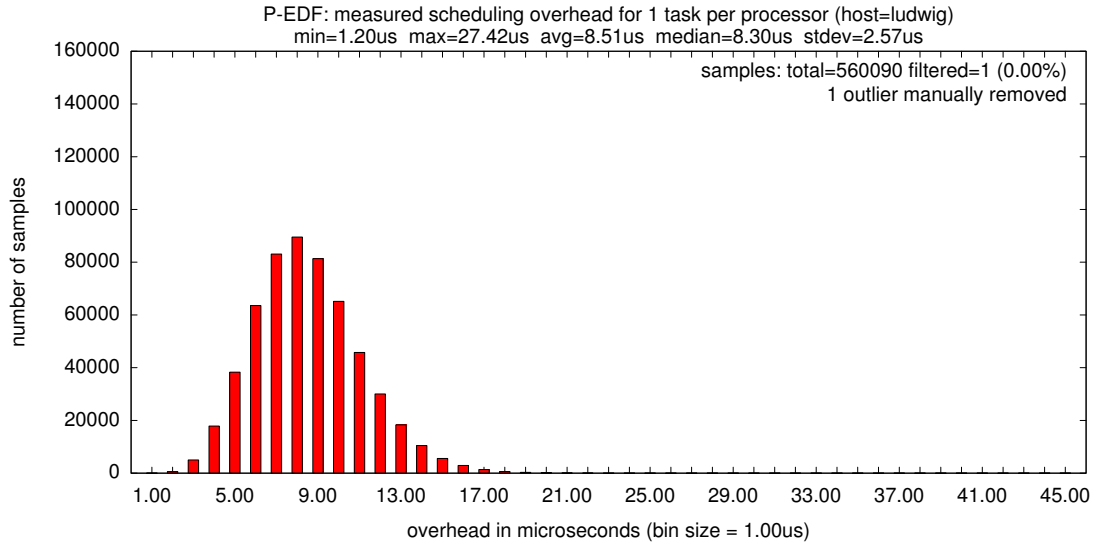


(b) Average observed scheduling overead.

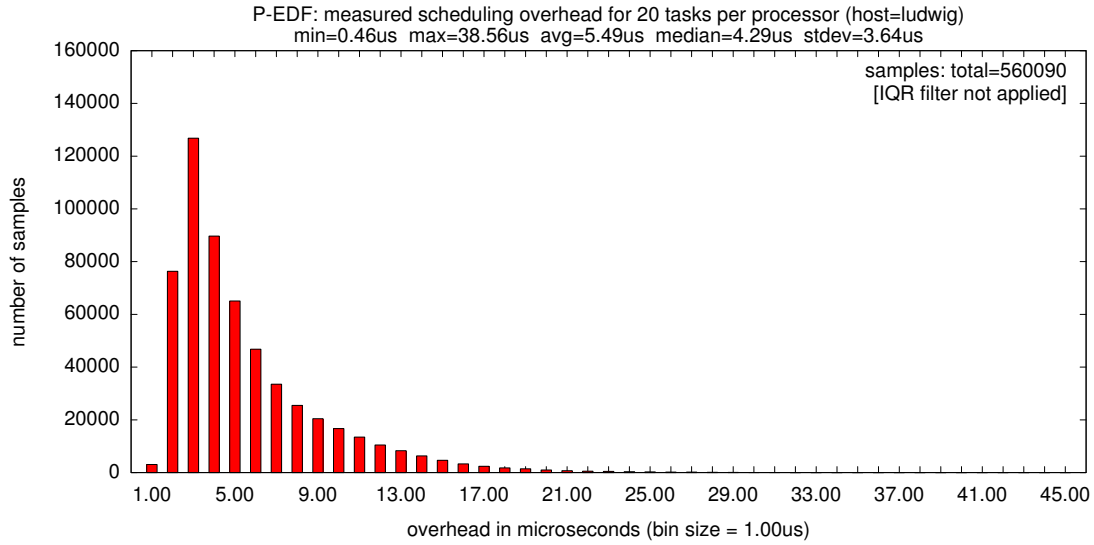
Figure 4.7: Scheduling overhead under P-FP-Rm and P-EDF-Rm.

In inset (a), the distribution of recorded samples is centered around the median of approximately $8.5\mu s$. In comparison, the whole distribution is skewed to lower overheads in the case of $n = 480$, which is shown in inset (b). In fact, the median overhead is almost halved to approximately $4.3\mu s$ despite the twenty-fold increase in tasks. This suggests that, in the case of partitioned scheduling, the runtime complexity of the implemented priority queue algorithm may be less relevant than its cache footprint.

Overall, our overhead data shows that the P-FP plugin is not at a significant advantage to the P-EDF plugin from an overhead point of view.



(a) Distribution of scheduling overhead samples for $n = 24$ (one task per processor).



(b) Distribution of scheduling overhead samples for $n = 480$ (20 tasks per processor).

Figure 4.8: Distribution of recorded scheduling overhead samples under P-EDF-Rm for **(a)** one task per processor and **(b)** 20 tasks per processor. Average and median observed scheduling overhead is lower in inset (b) despite a twenty-fold increase in task count. Inset (a) depicts the same data previously shown in Figure 4.6 with the single outlier removed. No outliers were filtered from the data depicted in inset (b).

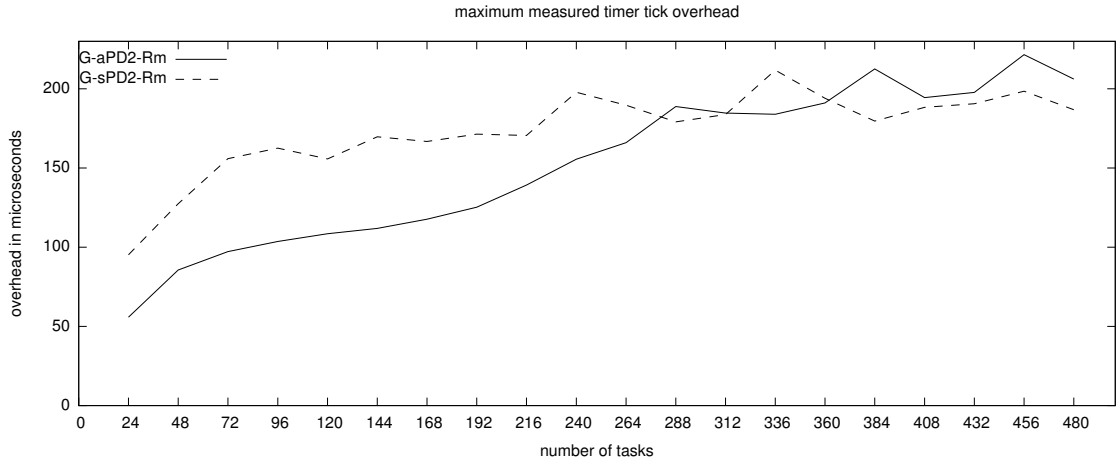
Staggered vs. aligned quanta. Another interesting trend concerns quantum staggering. Recall that quantum staggering under PD^2 is a tradeoff between the theoretic optimality of PD^2, which requires aligned quanta, and the presumed lower overhead of staggered quanta, which require periods to be shortened in the HRT case. Our data for clusters of size 6 and 24 shows that staggered quanta, in fact, do not reduce worst-case overhead at quantum boundaries. This can be seen in Figure 4.9(a), which shows worst-case observed timer tick overhead under G-aPD^2-Rm and G-sPD^2-Rm. As is evident in the depicted graph, the worst-case overhead is not worse when quanta are aligned. To our surprise, we did observe somewhat lower maximum overheads under C2-sPD^2-Rm than under C2-aPD^2-Rm, though it is not entirely clear whether this is caused only by staggering. In future work, it would be interesting to re-investigate the benefit of quantum staggering on a platform with more predictable interrupt delivery and memory bus arbitration.

In contrast, staggered quanta are clearly very effective at lowering average-case overheads, which can be seen in Figure 4.9(b). The average-case tick overhead under G-sPD^2-Rm barely increases across the tested range of task set sizes and stays below $10\mu s$, whereas the average-case tick overhead under G-aPD^2-Rm exhibits a rapidly increasing trend and exceeds $70\mu s$ for large task sets, which is more than seven times the average-case overhead under G-sPD^2-Rm.

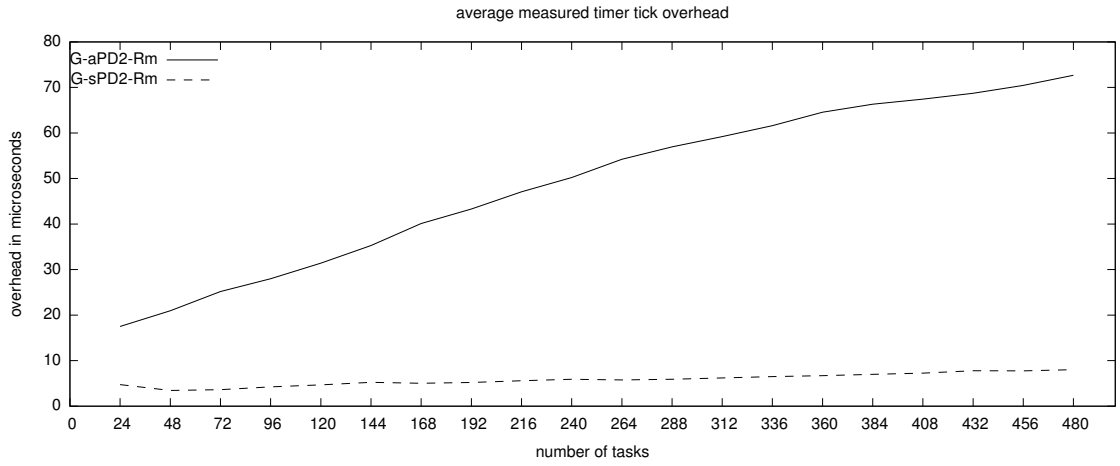
These overhead trends suggest that, on the platform underlying the case study, staggered quanta may not be worthwhile in the HRT case, but could be beneficial in the SRT case. The schedulability experiments presented in Section 4.5 confirm this to be indeed the case.

Cluster size. Note that the worst-case timer tick overhead in Figure 4.9(a) exceeds $200\mu s$ for large task sets, and $100\mu s$ even for task sets of moderate size. Given that LITMUS$^{\mathrm{RT}}$ uses a quantum size of $1000\mu s$, this shows that up to 20% of a quantum may be lost to subtask scheduling at quantum boundaries. Of course, such overheads are unlikely to yield high schedulability due to the implied overhead-related capacity loss.

Clustered scheduling has been proposed to overcome exactly this kind of limitation. By introducing modest algorithmic capacity loss in the form of bin-packing constraints, the goal is to alleviate most of the overhead-related capacity loss. However, just as with quantum staggering, it is not obvious that clustered scheduling necessarily results in significant improvements.



(a) Maximum observed timer tick overhead.



(b) Average observed timer tick overhead.

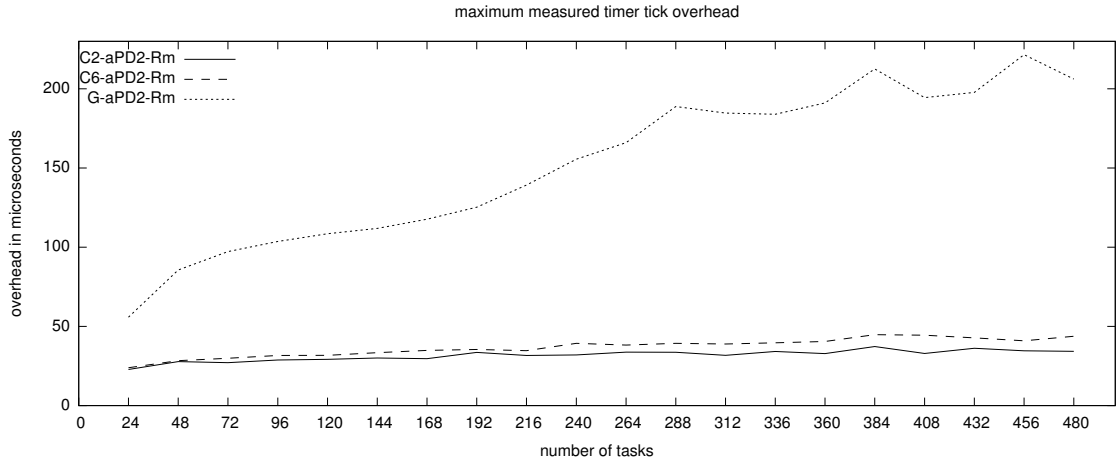
Figure 4.9: Timer tick overhead under G-aPD^2-Rm and G-sPD^2-Rm. Quantum staggering is effective and lowering average-case overheads, but does not lower worst-case overheads.

Our overhead data shows that clustered scheduling with smaller clusters is in fact very effective in lowering both worst-case and average-case overheads on our platform. Figure 4.10 shows this for the case of PD^2-based schedulers. Inset (a) shows worst-case tick overhead for C2-aPD^2-Rm, C6-aPD^2-Rm, and G-aPD^2-Rm. Both variants with smaller cluster sizes incur much lower overhead than the global configuration of the PD^2 plugin. Whereas worst-case timer tick overhead exceeds $200\mu s$ in the global case, worst-case timer tick overhead under C6-aPD^2-Rm stays below $50\mu s$. Further, worst-case overhead under C2-aPD^2-Rm is even less, but the difference between $c = 2$ and $c = 6$ is much less pronounced than the difference between $c = 6$ and $c = 24$. This reflects a large increase in cost if scheduler state is shared across sockets. Figure 4.10(b), which depicts average-case timer tick overhead for the same schedulers, shows that the relative trends occur similarly in the average case. Trends based on aligned and staggered quanta shown in Figure 4.9 arise in smaller clusters just as they do under global scheduling, albeit at smaller magnitudes.

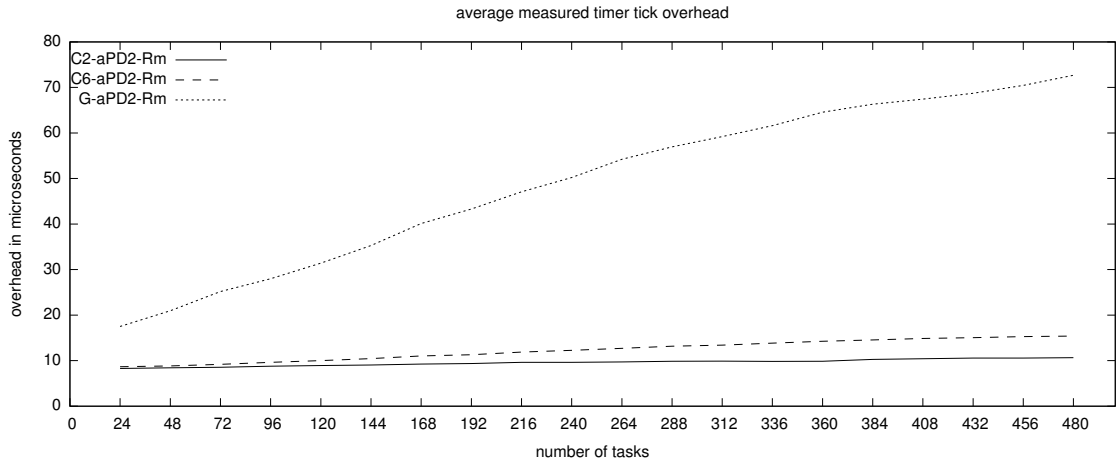
The benefit of smaller clusters in terms of reduced overhead also extends to EDF-based schedulers. This is apparent in Figure 4.11, which shows worst-case and average-case scheduling overhead under P-EDF-Rm, C2-EDF-Rm, C6-EDF-Rm, and G-EDF-Rm. As in the case of PD^2-based clusters, overheads are significantly lessened when using smaller cluster sizes. Notably, the two C-EDF variants C2-EDF-Rm and C6-EDF-Rm exhibit overheads that are much closer to partitioned scheduling than to global scheduling. This is both the case with worst-case overheads, which are depicted in inset (a) of Figure 4.11, and with average-case overheads, which are depicted in inset (b) of the same figure.

The significantly higher overhead under G-EDF-Rm can be explained by the increased cost of off-chip shared state and by increased lock contention—in the worst case, a processor locking a ready queue must wait more than four times longer under G-EDF-Rm (up to $m - 1 = 23$ preceding critical sections) than under C6-EDF-Rm (up to $c - 1 = 5$ preceding critical sections).

The negative effects of high lock contention is not limited to scheduling overhead, but also affects release interrupt overhead since the ready queue lock must be acquired to process a job release. Release interrupt overhead under each of the EDF-based schedulers (with global interrupt handling) is shown in Figure 4.12. As can be seen in inset (a), worst-case release interrupt overhead is similarly afflicted by severe lock contention, with worst-case overheads exceeding half a millisecond for large

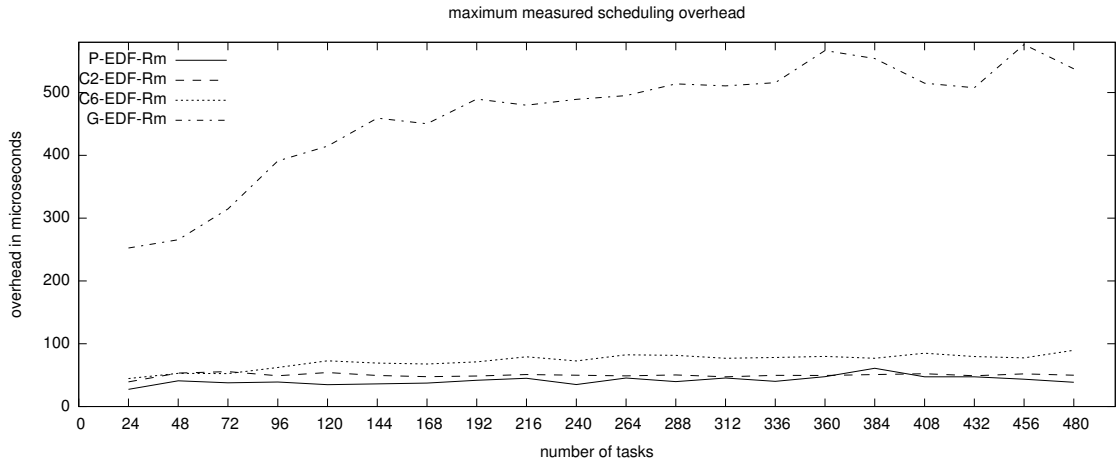


(a) Maximum observed timer tick overhead.

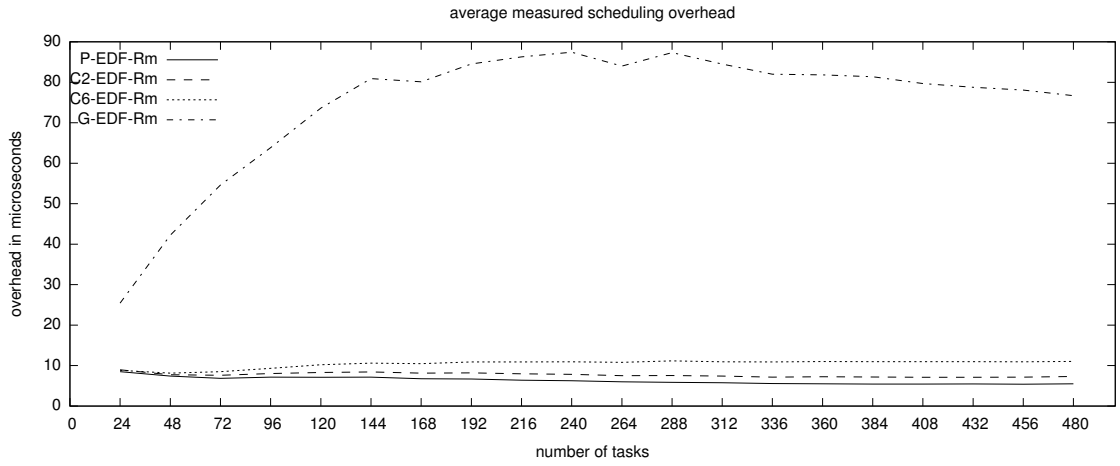


(b) Average observed timer tick overhead.

Figure 4.10: Timer tick overhead under C2-aPD^2-Rm, C6-aPD^2-Rm, and G-aPD^2-Rm. Smaller cluster sizes result in lower maximum and average-case overheads.



(a) Maximum observed scheduling overhead.



(b) Average observed scheduling overhead.

Figure 4.11: Scheduling overhead under P-EDF-Rm, C2-EDF-Rm, C6-EDF-Rm, and G-EDF-Rm. Smaller cluster sizes result in lower maximum and average-case scheduling overhead.

n under G-EDF-Rm. Note that the other plugins with cluster sizes $c < m$ are ordered roughly proportional to cluster size, and that they do not exhibit noticeably increasing trends.

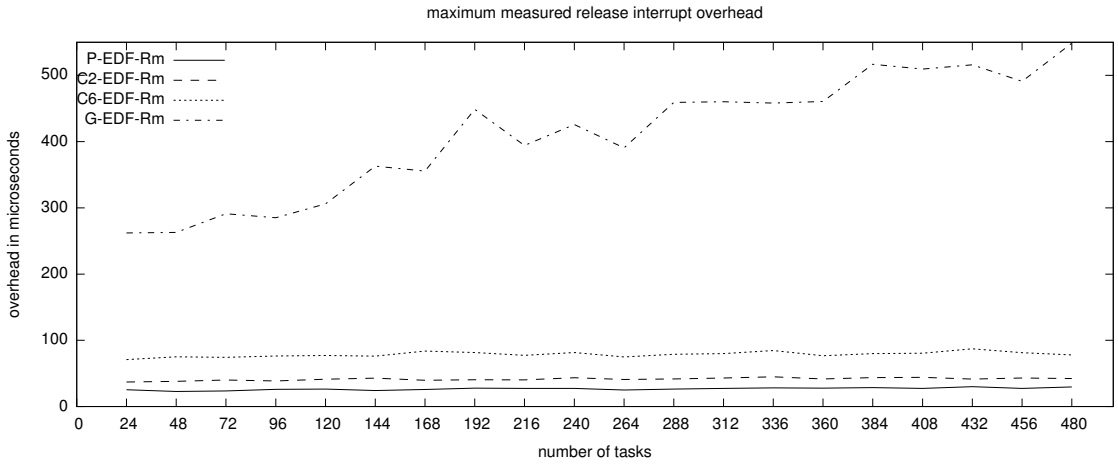
Figure 4.12(b) illustrates that the curves showing average-case release interrupt overheads compare similarly. Notably, the C6-EDF-Rm does exhibit an increasing trend in the average case.

Overall, our overhead data shows that clustered scheduling is highly effective at lowering overheads on our platform (compared to global scheduling). Further, the overheads in the case of $c = 2$ and $c = 6$ are much closer to the overheads exhibited by partitioned scheduling than those exhibited by global scheduling. This shows that link-based scheduling, which is implemented in C-EDF and currently requires coarse-grained locking to be implemented, is viable at these cluster sizes from an overhead point of view.

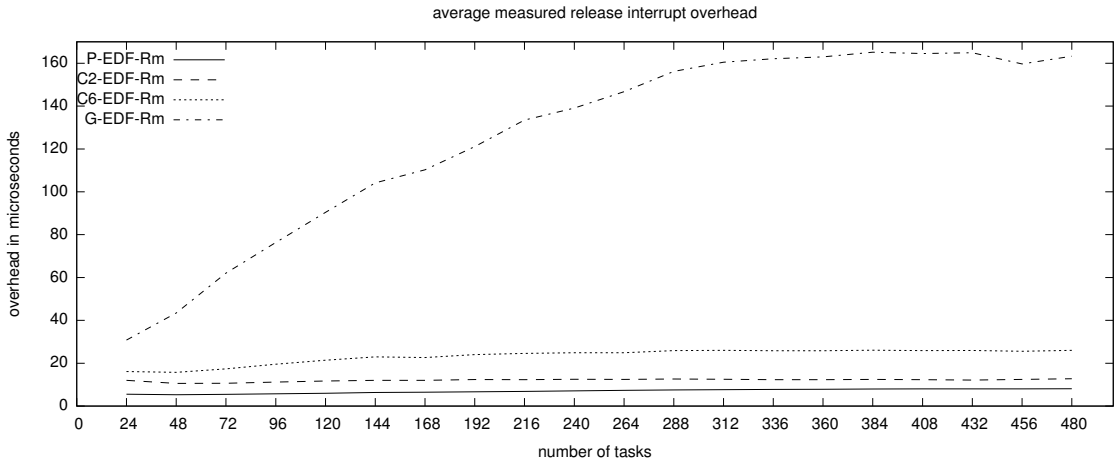
Event-driven vs. quantum-driven. Figure 4.13 illustrates how differences in implementation lead to significantly different overhead trends under event-driven and quantum-driven schedulers. Inset (a) of Figure 4.13 depicts average-case timer tick overhead under the event-driven scheduler C6-EDF-Rm and under the quantum-driven scheduler C6-aPD^2-Rm; inset (b) shows average-case release interrupt overhead under the same schedulers. Interestingly, the two graphs show reverse trends. Timer ticks cause very little overhead in event-driven schedulers since no job scheduling takes place at quantum boundaries (the ready queue lock is thus not acquired by the timer tick ISR). In contrast, timer ticks are a main source of overhead under PD^2-based schedulers since subtask scheduling is carried out in the timer tick ISR. The converse is the case with release interrupt overhead. Under event-driven schedulers, job scheduling takes place in reaction to a job release, which requires the release ISR to acquire the appropriate ready queue lock. In contrast, under PD^2, a newly-released job is simply queued for consideration at the next quantum boundary, which avoids lock contention.

This example highlights that overheads collected under one particular scheduler (and one particular platform) cannot be "reused" for the evaluation of other schedulers, which may require a different implementation approach. Counterintuitive trends such as overheads that decrease with an increasing number of tasks further call attention to the importance of measuring overheads as they occur in real systems, as opposed to extrapolating or estimating "reasonable" overheads.

This concludes our discussion of direct overheads. In this section, we did not exhibit graphs depicting overheads under schedulers that employ dedicated interrupt handling. The reason for this is

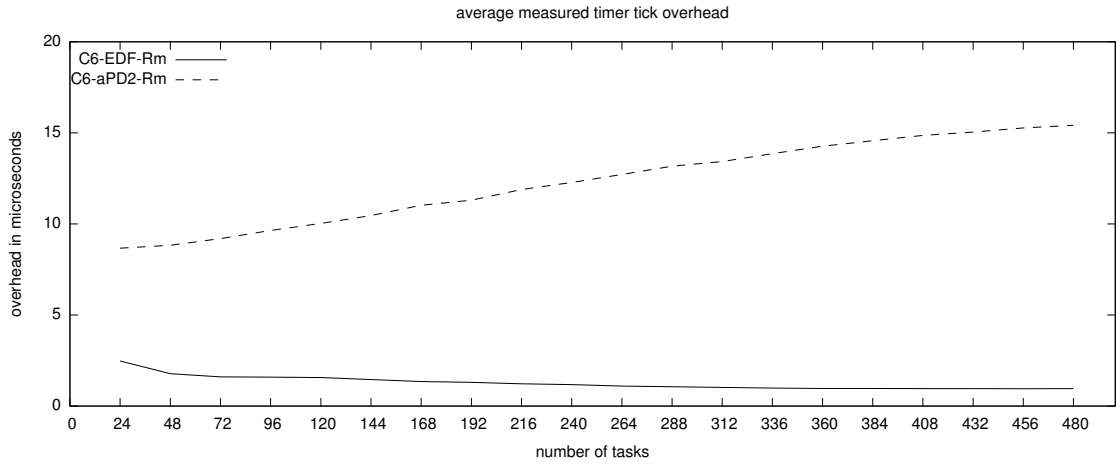


(a) Maximum observed scheduling overhead.

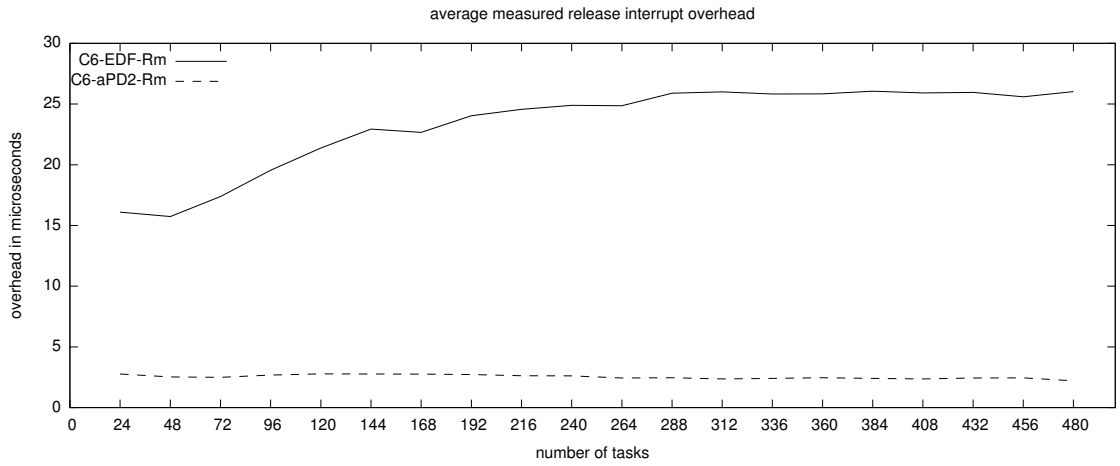


(b) Average observed scheduling overhead.

Figure 4.12: Release interrupt overhead under P-EDF-Rm, C2-EDF-Rm, C6-EDF-Rm, and G-EDF-Rm. Global scheduling is affected by lock contention and cache line bouncing.



(a) Average observed timer tick overhead.



(b) Average observed release interrupt overhead.

Figure 4.13: Average observed **(a)** timer tick and **(b)** release interrupt overhead under C6-aPD^2-Rm and C6-EDF-Rm. As no job scheduling is required, timer ticks create only little overhead in event-driven schedulers and release ISRs cause only little overhead in quantum-driven schedulers.

that dedicated interrupt handling by itself does not change the observed overhead trends significantly in most cases (with the exception of global schedulers, where contention is somewhat reduced). We refer the interested reader to the full data set with all graphs, which can be obtained online (Brandenburg, 2011).

4.4 Cache-Related Overheads

Accurately assessing cache-related delays is a classical component of WCET analysis (Wilhelm *et al.*, 2008). However, as discussed in Section 3.1, current WCET analysis techniques have not yet matured to the point that current multicore platforms with shared caches can be effectively analyzed. Hence, we rely on empirical measurements to estimate cache-related overheads instead.

The magnitude of cache-related delays caused by preemptions and migrations (and interrupts) is primarily determined by two factors:

1. *The number of cache lines that must be reloaded.* This is determined by the preempted job's WSS, the preempting job's cache footprint, and the length of preemption (the sooner a job continues execution, the higher the likelihood that parts of its working set are still present in some cache). However, in the worst case, the duration of preemption is sufficiently long and the cache footprint of the preempting job is large enough to evict the *complete* working set of the preempted job. Hence, the number of cache lines to be reloaded is mainly dependent on the preempted job's WSS.

2. *The cost of reloading an individual cache line.* This depends on where the evicted cache contents are loaded from (*i.e.*, if they can be retrieved from a lower-level cache, or if they have to be refetched from memory), and on the level of contention for memory bandwidth. In a shared-memory system, the central memory bus is a key bottleneck. If contention for the main memory bus is high, then reloading a cache line from main memory can incur significant delays while the processor is waiting to access the bus. Similarly, if cache lines are loaded from a lower-level shared cache (*e.g.*, if L2 cache contents are reloaded from the L3 cache), contention for cache access can cause the processor to stall. Therefore, the cost of a preemption is determined by the workload on *all* processors, not just the processor(s) involved in a job's preemption or migration.

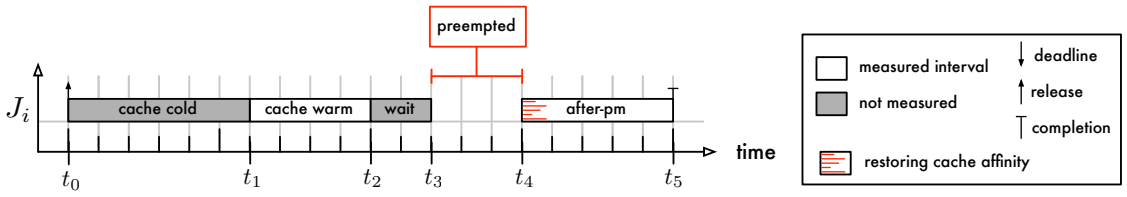


Figure 4.14: Illustration of CPMD measurement. The job J_i measures the time taken for a complete, cache-warm access to its entire working set during $[t_1, t_2)$. During $[t_2, t_3)$, J_i maintains cache affinity while it waits to be preempted. After the preemption (or migration), J_i times how long it takes to re-access its entire working set. The difference $(t_5 - t_4) - (t_2 - t_1)$ yields the delay incurred due to reloading cache contents.

In a real system, the actual magnitude of cache-related delays caused by preemptions and interrupts is further influenced by a particular application's working set and memory reference pattern. However, in a schedulability study, no specific application is assumed. Our primary intent is hence to determine *realistic* ranges of CPMD to assume in schedulability studies (and not to measure CPMD in actual applications). In particular, we are interested in *simple*, yet effective, methods for measuring CPMD that can be realistically performed as part of scheduling research and by practitioners during early design phases (*i.e.*, when selecting platforms and algorithms).

In this section, we first summarize two methods for measuring cache-related overheads and then report CPMD as determined on the platform underlying this case study. The techniques and data reported in this section are the result of joint work with Bastoni *et al.* (2010a).

4.4.1 Measurement Approach

Recall that a job is delayed after a preemption or a migration due to a (partial) loss of cache affinity. To measure such delays, we consider jobs that access their working set as illustrated in Figure 4.14: a job J_i starts executing cache-cold at time t_0 and experiences compulsory misses until time t_1, by which time its working set is completely loaded into the cache. After t_1, each subsequent memory reference by J_i is cache-warm. At time t_2, the job has successfully referenced its *entire* working set in a cache-warm context. From t_2 onward, the job repeatedly accesses single words of its working set (to maintain cache affinity) and checks after each access if a preemption or migration has occurred. Suppose that the job is preempted at time t_3 and not scheduled until time t_4. When J_i continues execution, it detects the preemption (or migration) and re-accesses its complete working set, which it

finishes at time t_5. As J_i lost cache affinity during the interval $[t_3, t_4]$, the length of the interval $[t_4, t_5]$ (*i.e.*, the time needed to reference again its *entire* working set) includes the time lost to additional cache misses.

Let D denote the cache-related delay suffered by J_i. After the working set has been fully accessed for the third time (at time t_5), D is given by the difference $D = (t_5 - t_4) - (t_2 - t_1)$. The interval $[t_2, t_3]$ is not reflected in D since jobs are simply waiting to be preempted while maintaining cache affinity during this interval. After collecting a trace $D_1, \ldots, D_k$ from a sufficiently large number of jobs k, $\max_l\{D_l\}$ can be used to approximate Δ^{cpd} (recall that Δ^{cpd} is a bound on the maximum CPMD incurred by any job). Similarly, average delay and standard deviation can be readily computed during off-line analysis.

On multiprocessors with a hierarchy of shared caches, migrations are categorized according to the level of cache affinity that is preserved (*e.g.*, a job migration between two processors sharing an L2 cache is an *L2-migration*). A *memory migration* does not preserve any level of cache affinity. The type of migration can be identified by recording at time t_3 the processor on which J_i was executing and at time t_4 the processor on which J_i resumes execution.

Each sample D_l can be obtained either directly or indirectly. A low-overhead clock can be used to directly measure the working set access times $[t_1, t_2]$ and $[t_4, t_5]$, which immediately yield D. Alternatively, some platforms include *hardware performance counters* that can be used to indirectly measure D by recording the number of cache misses. The number of cache misses experienced in each interval is then multiplied by the time needed to service a single cache miss. In the reported case study, we measured D directly using the TSC since it can be accessed from user space without incurring large overhead (recall that the TSC is a cycle counter available in every modern x86 processor; see Section 2.1.4).

Both direct and indirect measurements can be perturbed by ISR execution. These disturbances can be avoided by disabling interrupts while measuring working set access times. Although this does not prevent NMIs from being serviced, NMIs are infrequent events that likely only have a minor impact on CPMD approximations. We note, however, that our methodology currently cannot detect interference from NMIs. Any outliers due to NMIs could be removed using IQR filters as described in Section 4.3.1. On our test platform, NMIs did not appear to pose a problem (*i.e.*, no outliers were apparent) and hence no outlier filtering was performed on the CPMD data used in this dissertation.

Cache-related delays clearly depend on the WSS of a job, and possibly also on the scheduler and on the task set size n. Hence, to detect such dependencies (if any), each trace $D_1, \ldots, D_k$ should ideally be collected as part of a task set that is executing under the scheduler that is being evaluated without altering the implemented policy. We next describe a method that realizes this idea.

4.4.1.1 Schedule-Sensitive Method

Under the *schedule-sensitive* method, CPMD samples are recorded on-line while scheduling a task set under the algorithm of interest. Performing these measurements without changing the regular scheduling of a task set poses the question of how to efficiently distinguish between a cold, warm, and post-preemption (or migration)— denoted *post-pm*—working set access. In particular, detecting the first post-pm memory access (*i.e.*, time t_4 in Figure 4.14) is tricky, as jobs running under OSs with address space separation such as Linux are generally not aware of being preempted or migrated.

Solving this issue requires a low-overhead mechanism that allows the kernel to inform a job of every preemption and migration. As part of our case study, we modified LITMUS$^{\mathrm{RT}}$ to export all relevant data to real-time tasks by means of the control page (Section 3.3.2). A job can infer whether it has been preempted or migrated as follows: when it is selected for execution, the kernel updates the task's control page by increasing a preemption counter and the job sequence number, storing the preemption length, and recording on which core the task will start its execution. This allows the job to record all pertinent information without invoking system calls.

Delays should be recorded by executing many task sets with a wide range of sizes and working sets. Since samples are collected from a valid schedule, the advantage of this method is that it can identify dependencies (if any) of CPMD on scheduling decisions and on the number of tasks. However, this implies that it is not possible to control *when* a preemption or a migration will happen, since these decisions depend exclusively on the scheduling algorithm (which is not altered). Therefore, the vast majority of the collected samples are likely invalid, *e.g.*, a job may not be preempted at all or may be preempted prematurely, and only samples from jobs that execute exactly as shown in Figure 4.14 can be used in the analysis. Thus, large traces are required to obtain few valid samples. Worse, for a given scheduling algorithm, not all combinations of WSS and task set sizes may be able to produce the execution pattern needed in the analysis (*e.g.*, some arrival sequences may simply fail to trigger preemptions under event-driven schedulers such as G-EDF). We therefore

developed a second method that achieves finer control over the measurement process by artificially triggering preemptions and migrations of a single task.

4.4.1.2 Synthetic Method

Under the *synthetic method*, CPMD measurements are collected by a single instrumented process that repeatedly accesses working sets of different sizes. No LITMUS$^{\mathrm{RT}}$-specific real-time features are required. Instead, the measurement process is scheduled using Linux's SCHED_FIFO policy and assigned the highest priority. It therefore cannot be preempted by other processes.

In contrast to the schedule-sensitive method, preemptions and migrations are explicitly triggered in the synthetic method. In particular, the destination core and the preemption length are chosen randomly (a preemption arises if the same core is chosen twice in a row). In order to trigger preemptions, L2-migrations, L3-migrations, *etc.* with the same frequency (and thus to obtain an equal number of samples; recall Section 4.3.1), proper probabilities must be assigned to each core.

A preemption of a given length is enacted by invoking Linux's nanosleep() system call, which gives background tasks a chance to execute, *i.e.*, to "preempt" the measurement process. Migrations are enacted by changing the processor affinity mask prior to carrying out the post-pm working set access. As the task execution is tightly controlled, post-pm working set accesses do not need to be detected, and no kernel interaction or modification is required.

The synthetic method avoids the main limitation of the schedule-sensitive approach as it generates only valid CPMD samples. This allows a statistically meaningful number of samples to be rapidly obtained. However, as preemption and migration scheduling decisions are externally imposed, this methodology cannot identify correlations between CPMD and the scheduling policy or task set size.

A limitation shared by both the schedule-sensitive and synthetic methods is that they only reveal CPMD related to the TLB and unified and data caches, but not instruction caches, as only a small measurement loop is executed. Measuring CPMD caused by instruction cache misses requires instrumenting an actual application. However, given that dedicated instruction caches are typically much smaller than data caches, the additional CPMD is likely small and thus not particularly relevant to our goal (which is to determine realistic ranges of CPMD to assume in schedulability experiments and *not* to determine exact upper bounds for a specific application). In future work, it would be

interesting to employ performance counters to study the impact of real-time scheduling policies on instruction cache misses in real applications.

4.4.2 Experiments

We measured CPMD as it arises on our 24-core test platform with three levels of shared caches using both the schedule-sensitive and the synthetic method (Bastoni *et al*., 2010a). The main result from applying the schedule-sensitive method is that CPMD does *not* depend on the number of tasks n. Since the schedule-sensitive method is much more time consuming than the synthetic method (applying the schedule-sensitive method to a *single* scheduler required more than 24 hours of tracing to collect a sufficient number of valid samples; we consider 22 schedulers in this case study), we rely exclusively on data collected with the synthetic method in this dissertation. The interested reader is referred to (Bastoni *et al*., 2010a) for a detailed discussion of experiments conducted using the schedule-sensitive method.

Setup. We applied the synthetic method to our test platform as follows. A measurement process was scheduled with SCHED_FIFO as described above. To collect a sample, the process sequentially accessed[7] its working set (a large array) three times in a cache-hot context, and then simulated a preemption or migration, after which it recorded the length of the post-pm working set access. The WSS was chosen from $\{2^2, 2^3, 2^4, \ldots, 2^{13} = 8192\}$ KB. We further tested WSSs of 3 MB and 12 MB, as they correspond to the sizes of L2 and L3 caches, respectively. In these experiments, several per-WSS write ratios were used. In particular, we considered write ratios ranging over $\{0, 2^{-7}, 2^{-6}, 2^{-4}, 2^{-2}, 2^{-1}, 1\}$. All write ratios are given with respect to individual words (and not cache lines). For each WSS we ran the test program until 5,000 CPMD measurements were collected (for each preemption/migration category). Preemption lengths were uniformly distributed in $[0\ \mathrm{ms}, 50\ \mathrm{ms}]$.

The experiments were repeated using two configurations. In the first configuration, we measured CPMD as it arises in an otherwise idle system, *i.e.*, the measurement process is the only active user process (aside from mostly idle system daemons). In the second configuration, we added a background process to each processor that continuously accesses a large array (≈ 38 MB) in

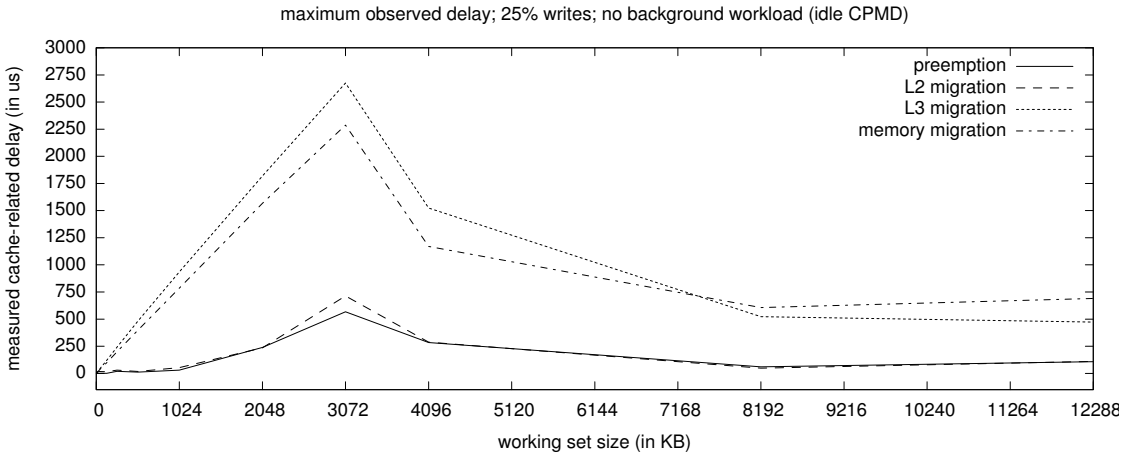
[7]Cache-line pre-fetching was disabled on the test platform, so the access pattern is irrelevant to these experiments.

a sequential manner to thrash all caches. The cache-polluting background load has the effect of simulating heavy cache and memory bus contention as it might arise in a heavily loaded system. In particular, a cache-polluting background process will be scheduled whenever the measurement process invokes the nanosleep() system call to simulate a preemption and thus likely evict its working set from the cache. We refer to measurements obtained from the first configuration as *idle CPMD*, and to measurements observed in the presence of cache-polluting background tasks as *load CPMD*. In total, more than 3.5 million valid samples were obtained under both configurations during more than 50 hours of tracing. As noted above, interrupts were disabled during measurements and no outliers were removed from the collected data.

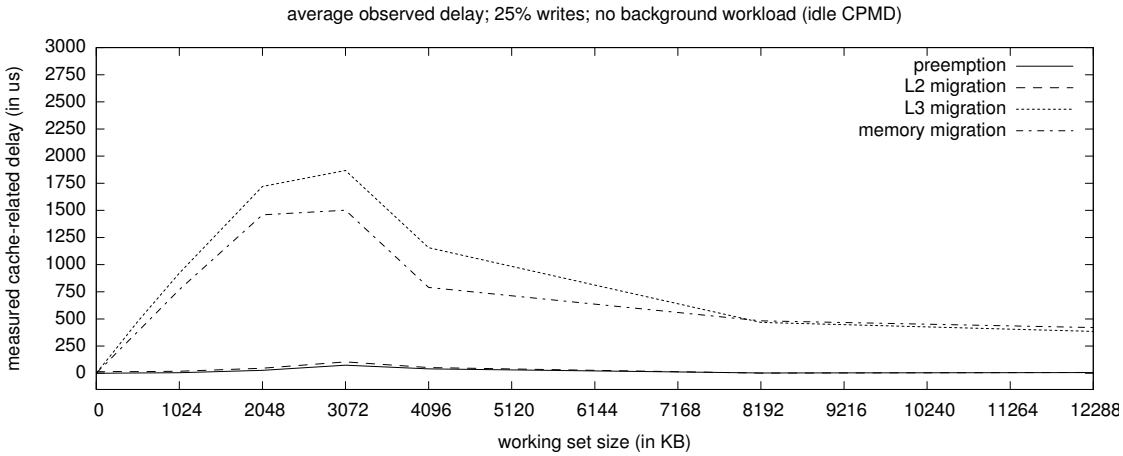
In principle, both the synthetic and the schedule-sensitive method could be adapted to estimate cache-related interrupt delays (Δ^{cid}). However, given the relatively large caches in our platform, it is unlikely that briefly executing ISRs displaces large parts of a stopped job's working set. The resulting delays are hence likely minuscule compared to CPMD and it is not clear whether they could be discerned from other slowdowns such as cache interference and bus contention. As major changes to both LITMUS$^{\mathrm{RT}}$ and the instrumented tasks would be required to detect release interrupts and timer ticks, we did not measure cache-related interrupt delays and assumed $\Delta^{cid} = 0$ in the schedulability experiments.

CPMD results. Write ratios of 1/2 and 1/4 showed the highest worst-case overheads, with 1/4 performing slightly worse. There are eight words in each cache line, thus each task updated every cache line in its working set multiple times. Tests with write ratios lower than 1/8, under which some cache lines are only read, exhibited reduced overheads. Since we are interested in estimating worst-case delays, we focus on a write ratio of 1/4 in the following. The complete set of graphs can be found online (Brandenburg, 2011).

Figure 4.15 shows maximum and average idle CPMD caused by preemptions and each kind of migration (L2, L3, memory) as a function of WSS. Maximum and average load CPMD is shown in Figure 4.17. In the manner discussed earlier, these graphs display the difference between a post-pm and a cache-warm working set access. Declining trends with increasing WSSs thus indicate that the cache-warm working set access cost is increasing more rapidly than the post-pm working set access cost.



(a) Maximum idle CPMD.



(b) Average idle CPMD.

Figure 4.15: Graphs showing **(a)** maximum observed CPMD and **(b)** average observed CPMD as determined by the synthetic method in an otherwise idle system.

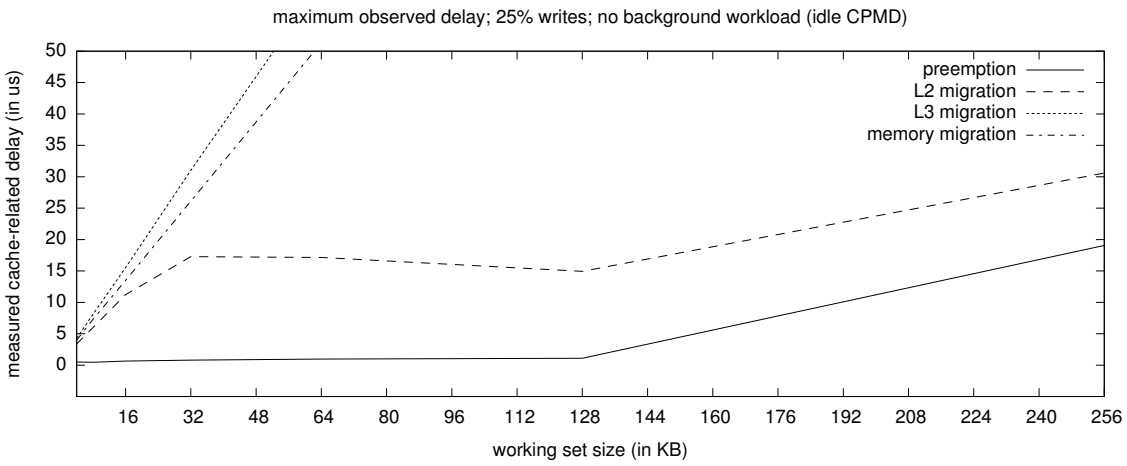


Figure 4.16: Maximum observed CPMD as determined by the synthetic method in an otherwise idle system. This graph shows a subset of the data depicted in Figure 4.15(a) at a higher resolution.

We consider idle CPMD first. Both worst-case and average-case idle CPMD exhibit a decreasing trend for WSSs exceeding 3 MB, which corresponds to the size of the L2 cache. In other words, jobs with WSSs that lead to L2 thrashing make only inefficient use of the memory hierarchy. The cost of losing cache affinity is reduced in this case because the benefit of using the cache was low to begin with, *i.e.*, a trashing job develops only little affinity.

Another major trend is that L2-migrations and preemptions incur virtually identical CPMD, whereas L3- and memory migrations incur much higher costs. This can be seen in both inset (a) and inset (b) of Figure 4.15, where the curves for L2-migrations and preemptions overlap in large parts. The lack of a major difference between L2-migrations and preemptions is due to the small size of the L1 cache (32 KB), which implies that most of the tested WSSs thrash the L1 cache anyway and thus derive little benefit from L1 affinity.

As an exception, if the working set is very small ($wss \leq 128$ KB), then preemptions are negligible (around $1\mu s$), whereas L2 migrations incur somewhat higher CPMD (around $15\mu s$). This can be seen in Figure 4.16, which shows a subset of the data depicted in Figure 4.15(a) at a smaller scale. While one might expect preemptions to become as expensive as L2-migrations when the WSS exceeds the L1 cache size (32 KB), the cost of preemptions remains virtually negligible until $wss = 128$ KB. The reason for this is that CPMD for such small WSSs is dominated by cache coherency overhead (and not the cost of reloading cache contents). If a job resumes quickly after

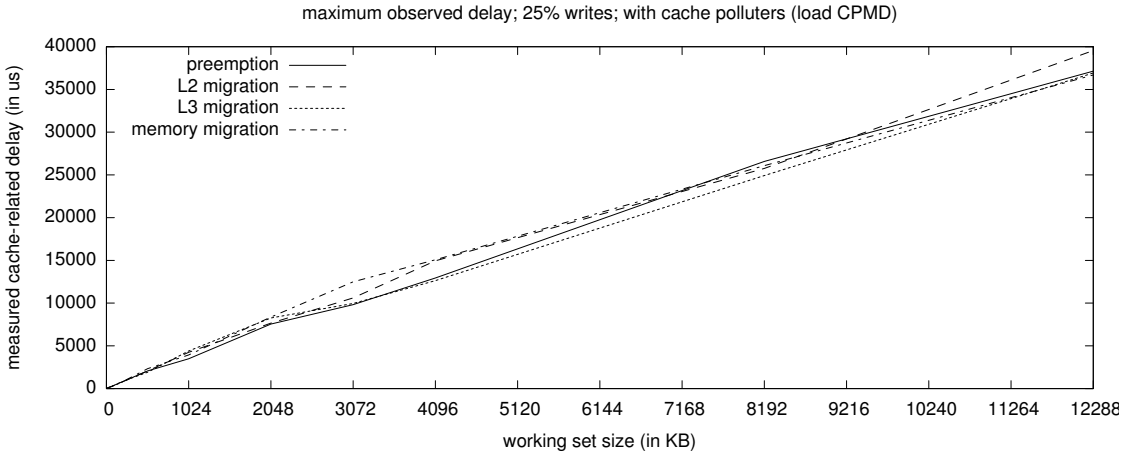
being migrated, parts of its working set are still present in the previously-used L1 cache and thus need to be evicted prior to being updated. In the case of preemptions, cache line evictions are not required. Similarly, if tasks never write to their working set (*i.e.*, if cache lines are not exclusive to any processor), then L2-migrations are indeed similar to preemptions and the effect disappears.

Surprisingly, jobs incur *higher* CPMD after L3-migrations than after memory migrations. This shows that cache affinity is not *always* beneficial, as the large L3 caches are not significantly faster than reloading data from main memory (in an idle system). We suspect that the counterintuitive trend of increased costs in the case of L3-migrations is also related to the cache-coherency protocol. Indeed, in the case of a write ratio of 0 (*i.e.*, only reads), L3-migrations become significantly cheaper than memory migrations. However, it is not clear why cache line evictions would be more expensive after an L3-migration than after a memory migration.[8] This illustrates the difficulty in correctly anticipating realistic overheads, hints at the challenges involved in bounding WCETs on a multicore platform, and also highlights the need to use fully transparent hardware platforms for safety-critical applications. (Intel does not publicize all details of the implemented cache coherency protocol.)

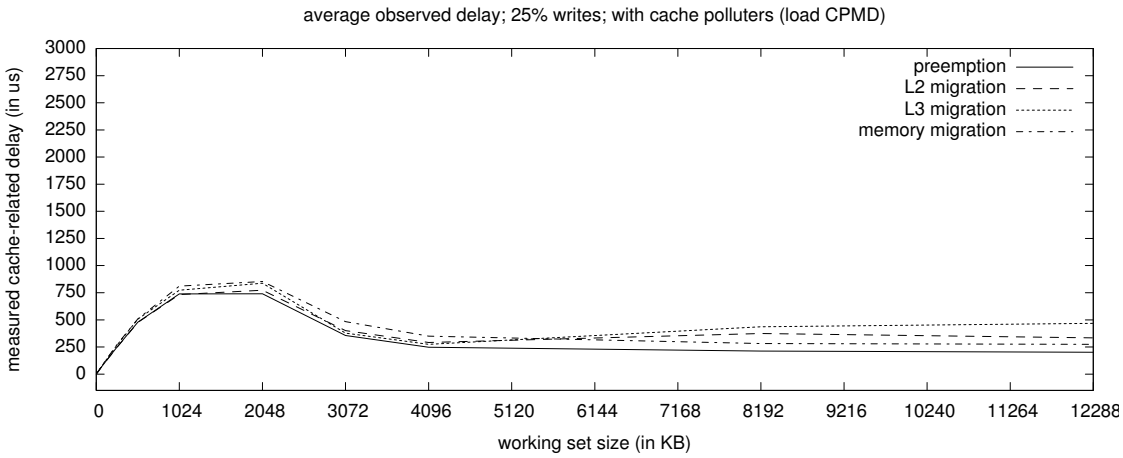
Remarkably, the observed worst-case and average-case idle CPMD have comparable magnitudes (insets (a) and (b) of Figure 4.15 use the same y-axis scale). This is not the case with load CPMD, which is depicted in Figure 4.17. In the presence of cache polluters, the observed worst-case load CPMD is an order of magnitude larger than in the average case, and also an order of magnitude larger than idle CPMD. Further, there are *no substantial differences* between preemption and migration costs, both in the worst and average case. When a job is preempted or migrated in the presence of heavy background activity, its cache lines are likely evicted quickly from all caches and thus virtually every post-pm access reflects the high overhead of refetching the entire working set from memory. This is evident in Figure 4.17(b) since all curves are very close together (in fact, within one standard deviation).

Another counterintuitive trend is that average-case load CPMD, shown in Figure 4.17(b), is generally *lower* than average-case idle CPMD, shown in Figure 4.15(b). This again implies reduced

[8]We repeated the experiments several times to rule out measurement error; the counterintuitive difference persisted. One possibly cause is a difference in memory bandwidth. Each of the test platform's four chips is connected to the main memory by an independent bus (Intel Corporation, 2007). Compared to an L3-migration, a memory migration thus involves twice the memory bandwidth (cores on the same chip share an L3 cache). Memory bandwidth could be a limiting factor if post-migration cache line evictions require a significant number of dirty cache lines to be written back to memory.



(a) Maximum load CPMD.



(b) Average load CPMD.

Figure 4.17: Graphs showing **(a)** maximum observed CPMD and **(b)** average observed CPMD as determined by the synthetic method in the presence of m cache-polluting background tasks. Note that inset (a) is plotted using a different y-axis scale due to the magnitude of the measured CPMD.

cache efficiency. In the presence of heavy cache contention, the measurement process experienced cache interference through the shared L2 and L3 cache, which reduces the degree of cache affinity that it can develop. Further, due to the thrashing cache-polluter tasks, bus access times are highly unpredictable and L3 cache interference is very pronounced. In fact, our traces show that jobs frequently incur "negative CPMD" for WSSs exceeding 2–3 MB because the "cache-warm" access itself is strongly interfered with. This implies that, from the point of view of schedulability analysis, CPMD is not well-defined for such WSSs, since a true WCET must account for worst-case cache interference and thus is already more pessimistic than CPMD, *i.e.*, actual CPMD effects are likely negligible compared to the required bounds on worst-case interference.

Interpretation. The setup used in the experiments depicted in Figure 4.17 simulate *worst-case scenarios* in which a job is preempted by a higher-priority job with a large WSS that (almost) completely evicts the preempted job's working set while activity on other processors generates significant memory bus contention. In contrast, the graphs shown in Figure 4.15 correspond to situations in which the preempting job does not cause many evictions (which is the case if it has a virtually empty working set or its working set is already cached) and the rest of the system is idle, *i.e.*, Figure 4.15 depicts *best-case scenarios*. Hence, Figure 4.17(a) shows the observed worst-case CPMD in a worst-case situation, whereas Figure 4.15(a) shows the observed worst-case CPMD in a best-case situation. Similarly, Figure 4.17(b) shows the observed average-case CPMD in a worst-case situation and Figure 4.15(b) shows the observed average-case CPMD in a best-case situation.

Further note that, even though the synthetic method relies on a background workload to generate memory bus contention, the data shown in Figure 4.17 also applies to scenarios in which the background workload is absent if the real-time workload itself generates significant memory bus contention. This has profound implications for empirical comparisons of schedulers. If it is possible that a job's working set is completely evicted by a preempting job, then this (possibly unlikely) event must be reflected in the employed schedulability tests. Thus, unless it can be shown (or assumed) that *all* tasks have only small WSSs and there is *no* background workload (including background OS activity), then bounds on CPMD should be estimated based on the high-contention scenario depicted in Figure 4.17.

Based on our CPMD data, it is *not warranted* to consider migrations to be more costly than preemptions when making worst-case assumptions (*e.g.*, when applying hard real-time schedulability tests). This implies that, in the worst case, there is only little benefit to maintaining cache affinity, which calls into question one of the claimed advantages of partitioned scheduling. However, does this render clustered or global scheduling preferable to partitioned scheduling? We next discuss results from our overhead-aware schedulability study, which answer this and other questions.

4.5 Schedulability Experiments

After distilling the model of scheduling and cache-related overheads as they occur on our test platform under each of the scheduler plugins and configurations, we carried out large-scale schedulability experiments. For each of the 27 combinations of utilization and period distributions (see Section 4.2.3), we generated task sets by varying $ucap$ from 1 to $m = 24$ in steps of 0.25, and by varying wss from 0 KB to 3072 KB (*i.e.*, the size of each L2 cache). Regarding the latter, we used a step size of 16 KB in the range from 0 KB to 256 KB, a step size of 64 KB in the range from 256 KB to 1,024 KB, and a step size of 256 KB in the range from 1024 KB to 3072 KB. This allows for a greater resolution in the range of small WSSs where most relevant trends occur. For each combination of $ucap$ and wss, we generated 100 task sets. This resulted in distinct 3,441 combinations of $ucap$ and wss for each of the 27 task parameter distributions. In total, we generated and tested 92,907,000 task sets.

Schedulability analysis. We evaluated each of these task sets under ten event-driven scheduler variants (P-FP and four EDF-based schedulers with either global or dedicated interrupt handling) and twelve quantum-driven scheduling approaches (three PD^2-based schedulers with either global or dedicated interrupt handling, and either aligned or staggered quanta, as listed in Table 4.1). For each of the 22 scheduler configurations and each tested task set τ, we determined whether τ is schedulable under four sets of analysis assumptions:

1. Requiring HRT correctness while accounting for worst-case overheads and worst-case load CPMD as shown in Figure 4.17(a).

2. Requiring HRT correctness while accounting for worst-case overheads and worst-case idle CPMD as shown in Figure 4.15(a).

3. Requiring SRT correctness while accounting for average-case overheads and average-case load CPMD as shown in Figure 4.17(b).

4. Requiring SRT correctness while accounting for average-case overheads and average-case idle CPMD as shown in Figure 4.15(b)

The benefit of considering both idle and load CPMD under both HRT and SRT analysis is that it provides a range of results in which real applications likely fall. That is, real applications will incur no less CPMD than idle CPMD, whereas load CPMD approximates worst-case cache contention. Prior to accounting for preemption-related overheads, each task's execution cost was increased by one CPMD charge to reflect the cost of the initial cache-cold working set access at the beginning of each job's execution.

We tested schedulability after partitioning (if required) and accounting for overheads under each of the four analysis assumptions as follows. If the worst-fit decreasing heuristic failed to partition the task set (if required) or if inflating for overheads caused any task to become infeasible (*i.e.*, $e_i > p_i$), then the task set was claimed unschedulable. Otherwise, we applied appropriate HRT and SRT schedulability tests. In the case of P-EDF, P-FP, and PD^2, the same test was applied to establish both HRT and SRT schedulability. HRT and SRT schedulability under these schedulers hence differs only in the use of worst-case or average-case overheads.

- Under P-FP-Rm and P-EDF-Rm, tasks were assigned priorities according to the RM policy and claimed schedulable if each partition passed the response-time test (Theorem 2.2).

- Under P-EDF-Rm and P-EDF-R1, a task set was claimed schedulable if none of the partitions exceeded a total utilization of 1 since EDF is optimal on a uniprocessor (Theorem 2.3).

- Under PD^2-based schedulers, a task set was claimed schedulable if none of the clusters was over-utilized since PD^2 is optimal on a multiprocessor for implicit-deadline tasks (Theorem 2.11). (Quantum staggering is accounted for when considering overheads).

In the case of EDF-based schedulers with $c > 1$, different HRT and SRT schedulability tests must be applied.

- Under EDF-based schedulers with $c > 1$, a task set was claimed HRT schedulable if each cluster passed either the density test (Theorem 2.7), Bertogna and Cirinei's multiprocessor response-time test (Theorem 2.8), or Baruah's test (Theorem 2.9).

- Under EDF-based schedulers with $c > 1$, a task set was claimed SRT schedulable if none of the clusters was over-utilized since G-EDF is SRT optimal on a multiprocessor (Theorem 2.10). The tardiness bound for each task was determined on a cluster-by-cluster basis. If the tasks assigned to the j^{th} cluster τ_j passed one of the three employed G-EDF HRT schedulability tests (Theorems 2.7–2.9), then the maximum tardiness of each task in τ_j is zero. Otherwise, maximum tardiness was bounded using Devi and Anderson's analysis (Theorem 2.10).

We note that Theorem 2.10 is no longer the most-accurate tardiness bound for G-EDF. Erickson *et al.* (2010a) recently proposed an improved, iterative method for computing less-pessimistic tardiness bounds under G-EDF. However, while Devi and Anderson's tardiness bound (Theorem 2.10) can be computed in a straightforward manner, Erickson *et al.*'s iterative method is of (at least) pseudo-polynomial time complexity.[9] Erickson *et al.* further reported that improvements are greatest for small processor counts (or smaller cluster sizes), and that runtimes grow super-linearly with increasing values of m. In empirical experiments, they found an average reduction in tardiness bounds of 4% or less for $m = 16$ at the cost of 1,000–10,000 iterations in their algorithm per task set (no results for larger processor counts were reported). In the case of $m = 4$, which yielded the largest reductions, the relative improvement only rarely exceeded 15%. Given that our platform consists of $m = 24$ processors, that reductions in tardiness bounds would likely be small using Erickson *et al.*'s method, and that more than 90 million task sets had to be tested twice, we chose to use Devi and Anderson's faster bound instead of Erickson *et al.*'s more-accurate bound.

Results. Since there are 22 tested scheduler configurations and four sets of analysis assumptions, our experiments generated 88 sets of schedulability data (in addition to tardiness bounds under SRT analysis). When visualized as regular schedulability plots (*i.e.*, schedulability as a function of $ucap$ for a fixed wss for each scenario), this data corresponds to 999 schedulability plots and 999 relative tardiness plots, each showing 88 curves. When aggregated using the weighted schedulability score,

[9] Erickson *et al.* (2010a) presented two variants of their iterative method: one with pseudo-polynomial complexity, and one slightly less pessimistic with unknown runtime complexity.

27 graphs (each still showing 88 curves) result, one for each parameter distribution. However, it is not helpful to include all 88 curves in a single graph since relevant trends would be obscured by clutter. Consequently, we grouped curves by analysis type (HRT or SRT constraints, CPMD in a loaded or idle system) and generated more than 60,000 graphs showing subsets of the curves to enable individual comparisons (*e.g.*, to compare SRT schedulability under all EDF-based schedulers, to compare HRT schedulability under both P-FP variants, *etc.*).

Clearly, it is infeasible to include all graphs in this dissertation. Instead, we describe the major trends that our experiments revealed and illustrate each point with select graphs showing only curves relevant to the discussed trend. We further only show HRT schedulability results corresponding to Analysis Assumption 1, and SRT schedulability results (and relative tardiness results) corresponding to Analysis Assumption 4. Additionally, in graphs showing weighted schedulability score, we restrict our attention to $wss \leq 1024$ KB because all major trends manifest in this range. We refer the interested reader to the companion website to this dissertation (Brandenburg, 2011), which makes all graphs, all task sets, the entire schedulability data set, and the underlying LITMUS$^{\mathrm{RT}}$ version and tools available for download.

To make the case study's results more accessible, the remainder of the discussion is structured as follows. First, we discuss for each scheduler plugin and cluster size whether dedicated or global interrupt handling is preferable, and, in the case of PD^2-based schedulers, whether to use aligned or staggered quanta (Section 4.5.1). Second, we consider which cluster size works best for EDF-based schedulers and PD^2-based schedulers on our test platform (Section 4.5.2). Finally, we compare the remaining schedulers to identify which scheduler is best suited to statisfying HRT and SRT constraints, and yielding low tardiness, on our platform (Section 4.5.3).

4.5.1 Interrupt Handling and Quantum Alignment

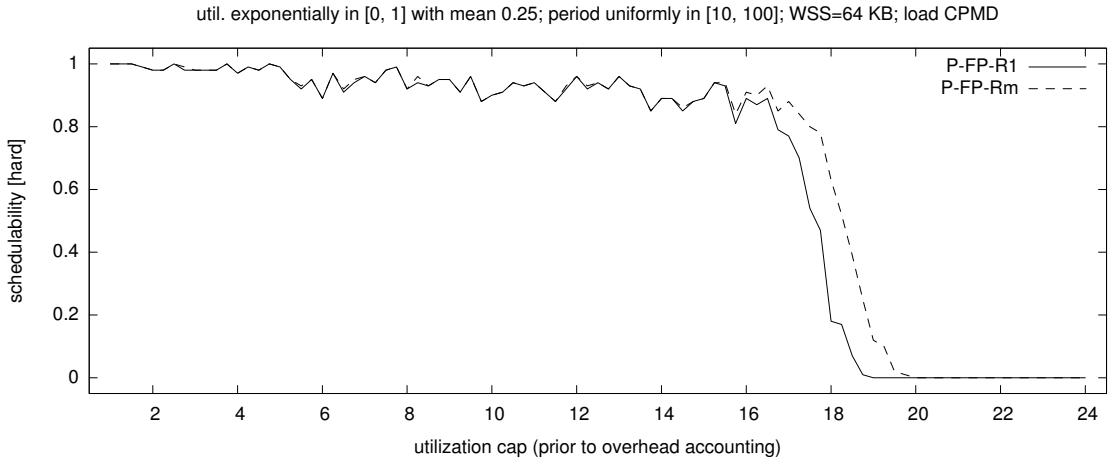
In LITMUS$^{\mathrm{RT}}$, event-driven plugins can be configured to use either dedicated interrupt handling or global interrupt handling. Quantum-driven schedulers may further use staggered or aligned quanta (with either interrupt handling option). In this section, we determine for each implemented event-driven scheduler and, in the case of C-EDF and C-PD^2, for each cluster size, the configuration that is preferable from a schedulability point of view. We consider interrupt handling in event-driven schedulers first.

Dedicated interrupt handling is a tradeoff. On the one hand, jobs are not disturbed by release interrupts; on the other hand, the system is essentially reduced to $m - 1$ processors. Further, under partitioned schedulers, it increases worst-case release delay since an IPI is added to the critical path from release interrupt to context switch. Intuitively, dedicated interrupt handling is worthwhile to implement if the overhead-related capacity loss due to release interrupts under global interrupt handling exceeds the loss of one processor, which may be the case if interrupt accounting is severely pessimistic, if release ISRs cause high overhead (*i.e.*, if Δ^{rel} is large), or if there are many tasks.

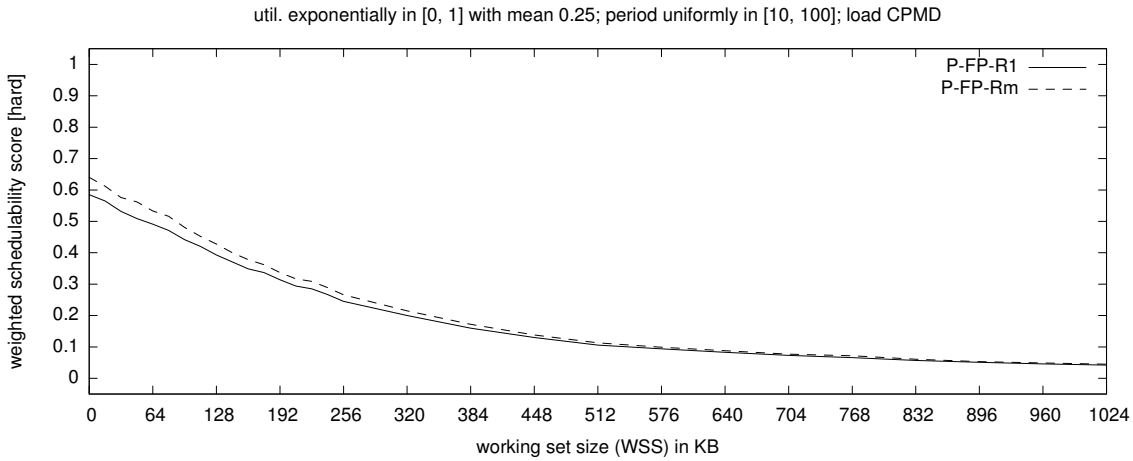
P-FP variants. This is not the case with the P-FP plugin, under which we found global interrupt handling (*i.e.*, P-FP-Rm) to be uniformly preferable in *all* tested scenarios. There are three contributing factors. First, release interrupts are relatively short under the P-FP plugin since locks are only rarely contended. Second, interrupts are modeled as tasks under P-FP scheduling, which is the least pessimistic method of accounting for interrupts. Third, P-FP scheduling is already affected by bin-packing limitations, which are exacerbated if a processor is reserved for interrupts.

An example schedulability graph illustrating this observation is shown in Figure 4.18(a), which depicts HRT schedulability under P-FP-R1 and P-FP-Rm for the medium exponential utilization distribution, moderate periods, and a fixed WSS of 64 KB. The curves of P-FP-R1 and P-FP-Rm overlap in most parts and only diverge in the range $16 \leq ucap \leq 20$. Recall from Section 4.1.5 that schedulability is the ratio of schedulable task sets, and that an optimal, overhead-free scheduler achieves a schedulability of 1. The graph shows that both configurations suffer some capacity loss already for utilization caps as low as 4, but that more than 80% of all generated tasks can be scheduled under either configuration if $ucap \leq 16$. For higher utilization caps, global interrupt handling yields higher schedulability, until none of the task sets are schedulable under either configuration. However, the difference is rather small.

Figure 4.18(b) shows the weighted HRT schedulability score for the same task parameter distributions. The curves shown in Figure 4.18(a) correspond to the points at $wss = 64$ KB in Figure 4.18(b). The small difference in Figure 4.18(a) is reflected by the fact that both curves are close to each other in Figure 4.18(b) for $wss = 64$ KB. The weighted schedulability score reveals that the difference between P-FP-Rm and P-FP-R1 decreases continuously as the WSS increases. However, even for $wss = 0$ KB, the difference remains small.



(a) HRT schedulability for $wss = 64$ KB.



(b) Weighted HRT schedulability score.

Figure 4.18: Graphs showing **(a)** HRT schedulability and **(b)** weighted HRT schedulability score for exponential medium utilizations and moderate periods.

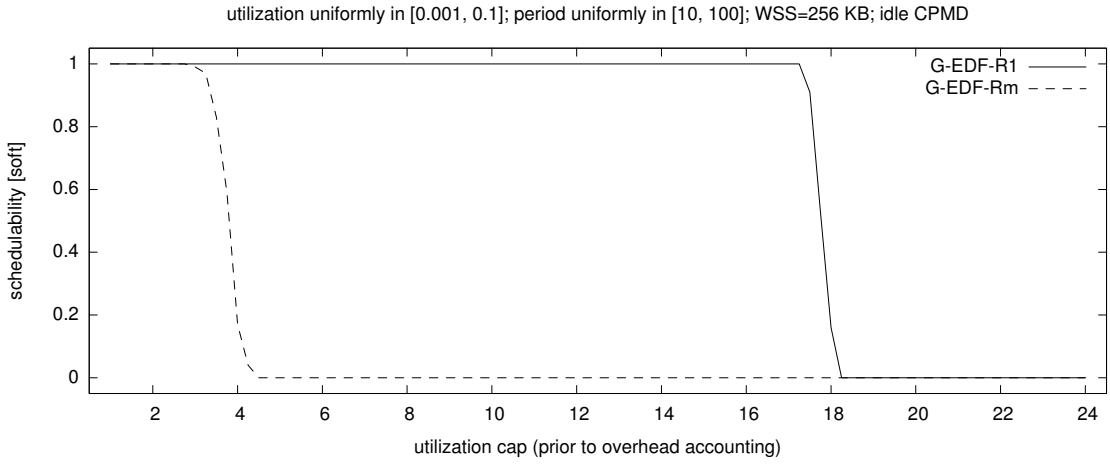
In summary, P-FP-Rm is always preferable to P-FP-R1, if only slightly so. The difference diminishes as cache-related overheads dominate the difference in interrupt handling. These trends are apparent in all tested scenarios.

EDF-based schedulers. Under G-EDF, the exact opposite is the case since G-EDF suffers from high release overhead (due to lock contention). Interrupt accounting also has a greater impact than under partitioned scheduling since each job may be delayed by all other tasks (instead of only tasks local to its partition). Therefore, G-EDF-R1 is preferable to G-EDF-Rm in all but three of the tested scenarios, and usually by a large margin. The few exceptions in which G-EDF-Rm is preferable involve scenarios with only few, high-utilization tasks where interrupt handling is not the deciding factor (*e.g.*, SRT schedulability for uniform heavy utilizations and long periods).

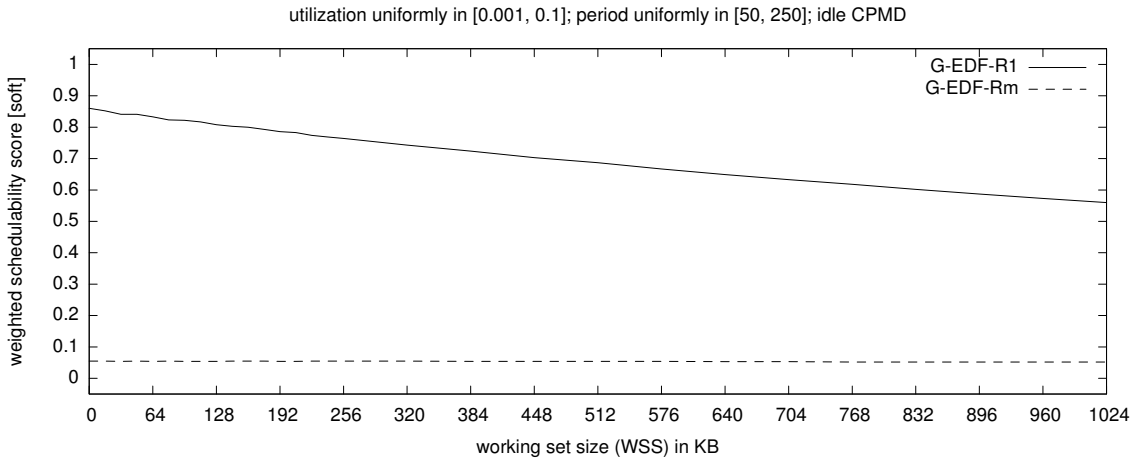
A typical example with a large gap in performance can be seen in Figure 4.19(a), which shows SRT schedulability under G-EDF-Rm and G-EDF-R1 for uniform light utilizations, moderate periods, and a WSS of 256 KB. Under the light uniform utilization distribution, a large number of tasks is generated (in excess of 400 tasks per task set for large utilization caps), which causes major overhead-related capacity loss. This is evidenced by the performance of G-EDF-Rm, which deteriorates quickly at $ucap \approx 4$, where there are about 80 tasks in each task set. In contrast, G-EDF-R1 ensures bounded tardiness to all generated task sets until $ucap \approx 17.5$. Figure 4.19(b), which depicts the weighted SRT schedulability score of G-EDF-R1 and G-EDF-Rm for the same task parameter distributions, shows that the gap in performance increases with decreasing WSSs, but remains relative large even for a WSS of 1 MB.

G-EDF-R1 also achieves higher schedulability than G-EDF-Rm in the case of HRT constraints. Similar trends are also apparent to a large extent when comparing C6-EDF-Rm to C6-EDF-R1. As an exception, if there are few, heavy tasks and long periods, then C6-EDF-Rm is preferable to C6-EDF-R1 since overhead due to release interrupts is much lower. Graphs supporting these conclusions can be found online (Brandenburg, 2011).

The choice between dedicated and global interrupt handling is less clear in the cases of P-EDF and C-EDF with clusters defined by shared L2 caches. In both cases, bin-packing issues are magnified by the loss of one processor. The release overhead is also lower than under G-EDF-Rm and C6-EDF-Rm since there are fewer tasks per cluster and locks are less frequently contended



(a) SRT schedulability for $wss = 256$ KB.



(b) Weighted SRT schedulability score.

Figure 4.19: Graphs showing **(a)** SRT schedulability and **(b)** weighted SRT schedulability score for light uniform utilizations and moderate periods.

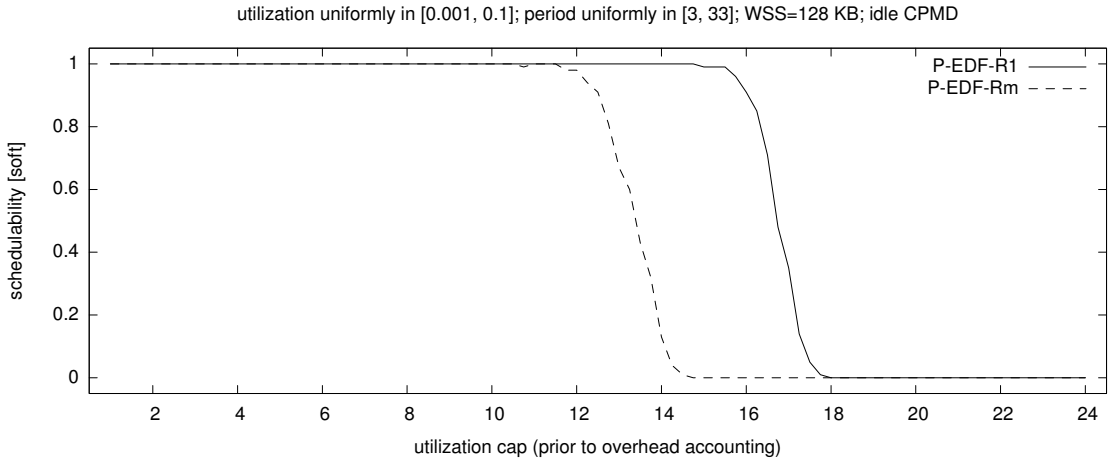
(and by fewer processors). Nonetheless, release overhead causes significant capacity loss when there are many tasks. Therefore, dedicated interrupt handling is preferable in some scenarios with light utilizations and moderate or short periods, whereas global interrupt handling is preferable in the other cases where there are fewer tasks or if tasks have long periods, which reduces the impact of overheads.

An example of the latter case is shown in Figure 4.20, which shows SRT schedulability and weighted SRT schedulability of P-EDF-R1 and P-EDF-Rm for uniform light utilizations, short periods, and a WSS of 128 KB. Of all considered task parameter distributions, this scenario creates the most tasks with the most-stringent time constraints. Consequently, it is most susceptible to overhead-related capacity loss. Not surprisingly, dedicated interrupt handling is preferable in this case: P-EDF-R1 can support task sets with total utilization up to 16, whereas P-EDF-Rm's schedulability decreases sharply around $ucap = 12$. Figure 4.20(b) shows that P-EDF-R1 is generally preferable for all WSSs, though differences become negligible for larger WSSs.

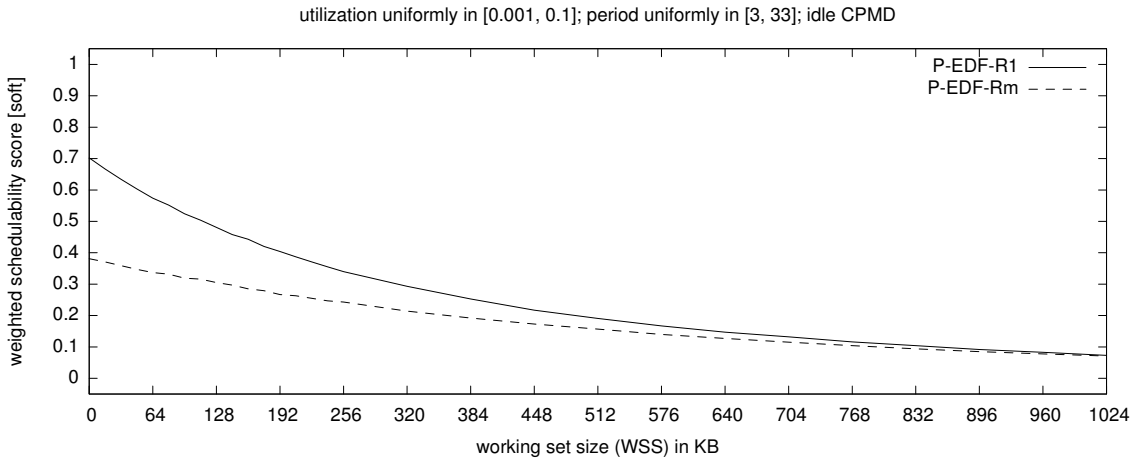
In contrast, Figure 4.21 depicts a case where P-EDF-Rm is preferable to P-EDF-R1. Figure 4.21(a) shows HRT schedulability for medium bimodal utilizations, long periods, and a WSS of 512 KB. Long periods imply that release interrupts cause only little capacity loss since the magnitude of Δ^{rel} is small in comparison to most periods. Further, bimodal task sets are more difficult to partition due to the likely presence of high-utilization tasks. Therefore, dedicating a processor to interrupt handling actually results in lower schedulability than under global interrupt handling. Figure 4.21(b) shows that this is the case for all considered WSSs.

To summarize, under P-EDF, dedicated interrupt handling is preferable only if there are many light tasks with short or moderate periods that cause high overheads (and that are easy to partition), and global interrupt handling is preferable otherwise. In the SRT case, global interrupt handling is preferable in the majority of the tested scenarios since average release overheads are much lower than worst-case release overheads. These observations also apply to C2-EDF-Rm and C2-EDF-R1, which exhibit trends similar to P-EDF-Rm and P-EDF-R1.

Note that this characterization is hardware-specific: on faster processors, release overhead could be lower and P-EDF-Rm and C2-EDF-Rm could be preferable in all cases. This is a good example of a tradeoff between algorithmic and overhead-related capacity loss that can only be revealed by an overhead-aware evaluation methodology. Without considering real, measured overheads based on

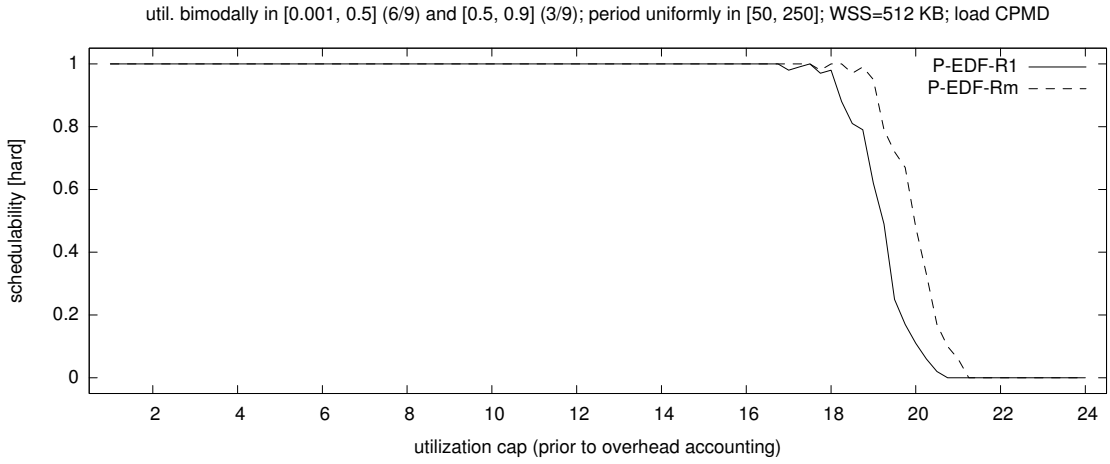


(a) SRT schedulability for $wss = 128$ KB.

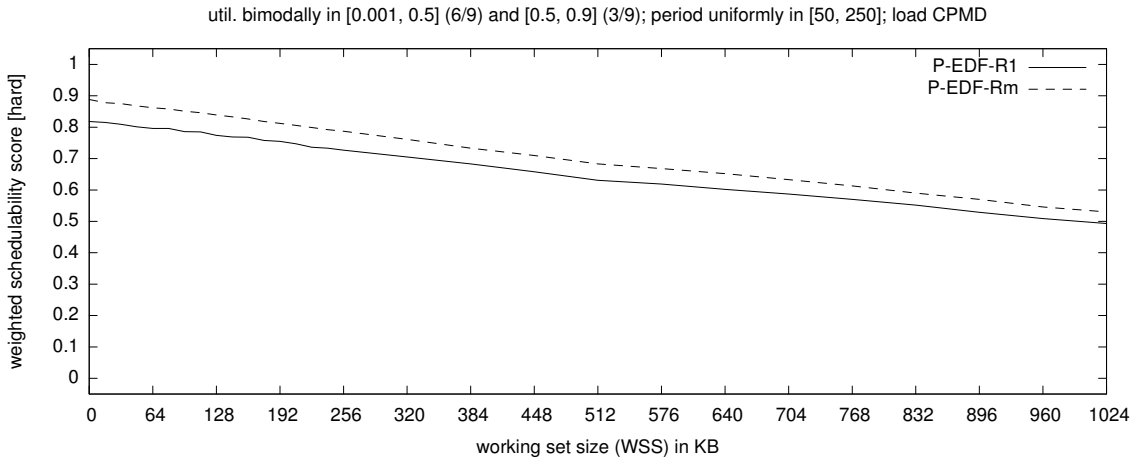


(b) Weighted SRT schedulability score.

Figure 4.20: Graphs showing **(a)** SRT schedulability and **(b)** weighted SRT schedulability score of P-EDF-Rm and P-EDF-R1 for uniform light utilizations and short periods.



(a) HRT schedulability for $wss = 512$ KB.



(b) Weighted HRT schedulability score.

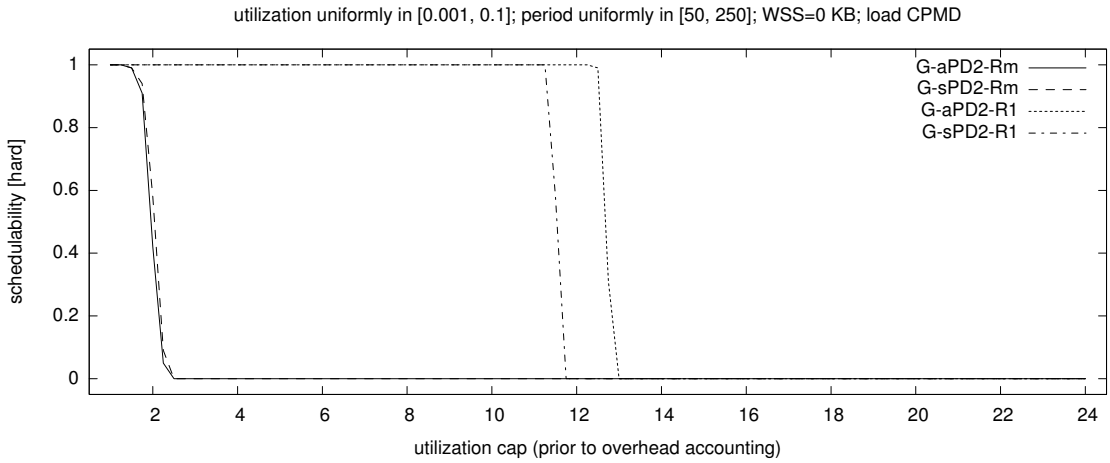
Figure 4.21: Graphs showing **(a)** HRT schedulability and **(b)** weighted HRT schedulability score of P-EDF-R1 and P-EDF-Rm for medium bimodal utilizations and long periods.

an actual implementation, it would not be possible to anticipate for which distributions dedicated interrupt handling is preferable (if any).

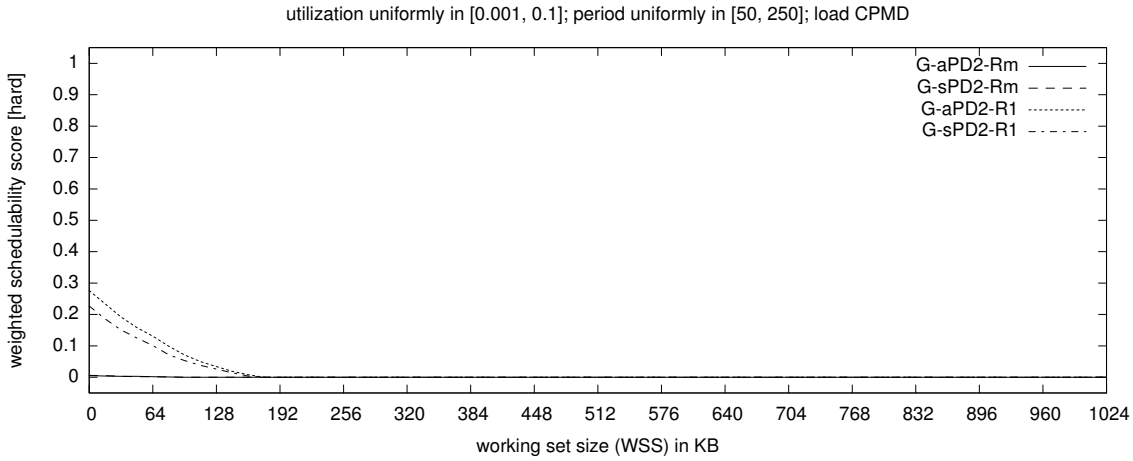
PD^2-based schedulers. In the case of the PD^2 plugin, we considered three cluster sizes in our study ($c = 2$, $c = 6$, and $c = 24$). Each of these cluster sizes can be configured to use either dedicated or global interrupt handling, and either staggered or aligned quanta, for a total of four variants per cluster size. On our platform, the two larger cluster sizes $c = 6$ and $c = 24$ revealed similar trends in the HRT case, whereas small clusters based on L2-sharing ($c = 2$) exhibited different trends. We discuss the two larger cluster sizes first.

Figure 4.22 depicts HRT schedulability under each of the four global PD^2 variants ($c = 24$) for uniform light utilizations, long periods, and assuming a negligible WSS. Two groups of curves are apparent. The two variants using global interrupt handling, G-sPD^2-Rm and G-aPD^2-Rm, achieve very low schedulability due to the pessimistic accounting for release interrupts under quantum-driven scheduling (recall that the effective quantum length is shortened to account for *every* interrupt source). Due to the impact of release interrupts, hardly any difference between staggered and aligned quanta is apparent in this case (the two curves mostly overlap). In contrast, schedulability under dedicated interrupt handling is much higher, and there is a noticeable difference between staggered (G-sPD^2-R1) and aligned (G-aPD^2-R1) quanta. As is apparent in Figure 4.22(a), staggered quanta yield somewhat lower schedulability. This is because the added stagger latency forces periods to be shortened (under HRT analysis), which causes some capacity loss. While one might intuitively assume lower overheads under staggered quanta might compensate for the shortened periods, the observed *worst-case* overheads are actually not lower under staggered quanta than under aligned quanta (in the case of $c = 6$ and $c = 24$). Figure 4.22(b) shows that G-aPD^2-R1 is preferable for larger WSSs, too, until about $wss = 128$ KB, at which point all variants fail to schedule even low-utilization task sets due to high cache-related overhead.

In contrast, in the case of small, L2-based clusters ($c = 2$), we found staggered quanta to be effective at lowering maximum overheads and thus at improving HRT schedulability. This is apparent in Figure 4.23(a), which depicts HRT schedulability under each of the four clustered PD^2 variants with $c = 2$ for exponential light utilizations, long periods, and a negligible WSS. Again, two groups of curves are apparent. As is the case with large clusters, dedicated interrupt handling is preferable to

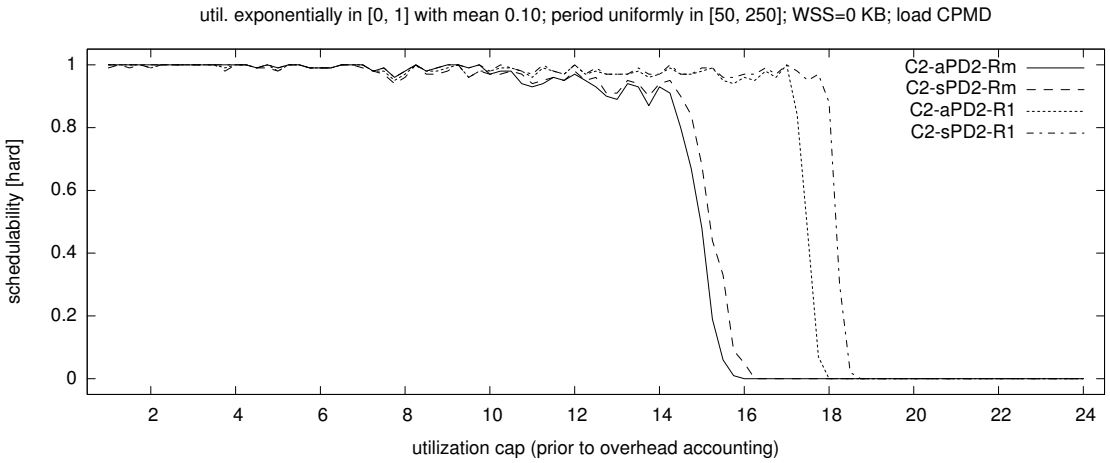


(a) HRT schedulability for $wss = 0$ KB.

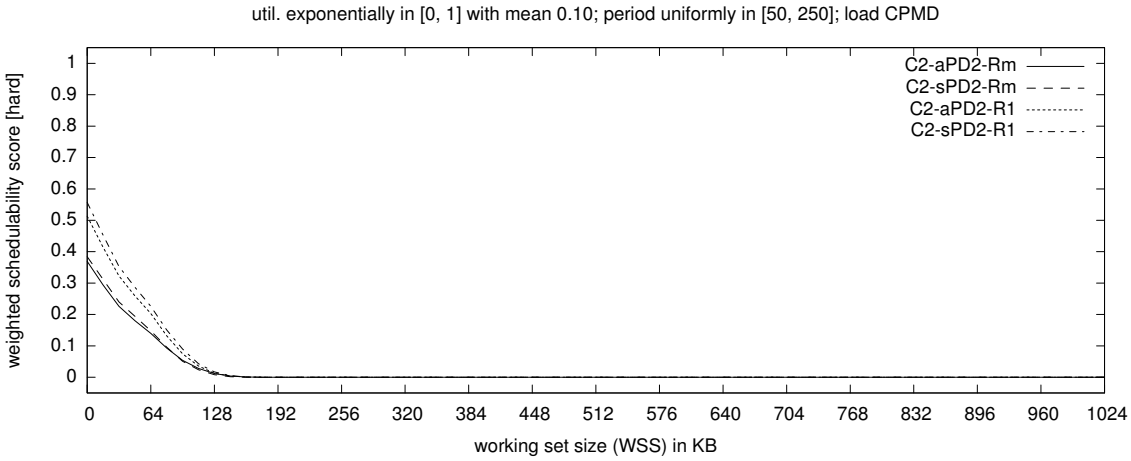


(b) Weighted HRT schedulability score.

Figure 4.22: Graphs showing **(a)** HRT schedulability and **(b)** weighted HRT schedulability score of G-aPD^2-R1, G-aPD^2-Rm, G-sPD^2-Rm and G-sPD^2-R1 for uniform light utilizations and long periods.



(a) HRT schedulability for $wss = 0$ KB.



(b) Weighted HRT schedulability score.

Figure 4.23: Graphs showing **(a)** HRT schedulability and **(b)** weighted HRT schedulability score of C2-aPD^2-R1, C2-aPD^2-Rm, C2-sPD^2-Rm and C2-sPD^2-R1 for exponential light utilizations and long periods.

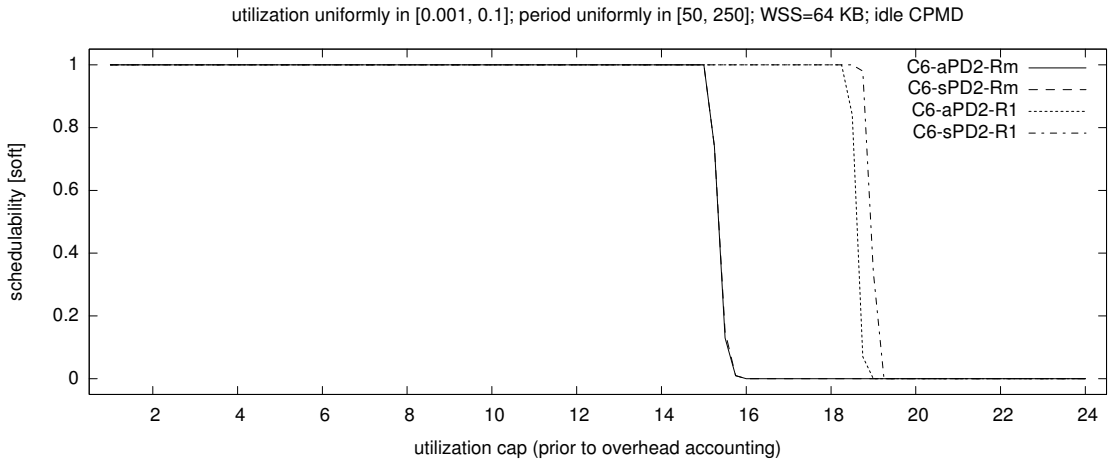
global interrupt handling due to pessimistic release interrupt accounting. However, in contrast to the cases of $c = 6$ and $c = 24$ discussed above, we did observe lower maximum overheads when using staggered quanta if $c = 2$. Consequently, C2-sPD^2-R1 achieves higher schedulability than the other three variants in this scenario. Figure 4.23(b) shows that C2-sPD^2-R1 is also preferable for non-negligible WSSs until $wss = 128$ KB (where, again, all of the variants suffer from high cache-related overhead).

To summarize, in the HRT case, dedicated interrupt handling is generally preferable to global interrupt handling for all three cluster sizes, and aligned quanta are preferable for large clusters ($c = 6$ and $c = 24$), whereas staggered quanta are preferable for small clusters ($c = 2$).

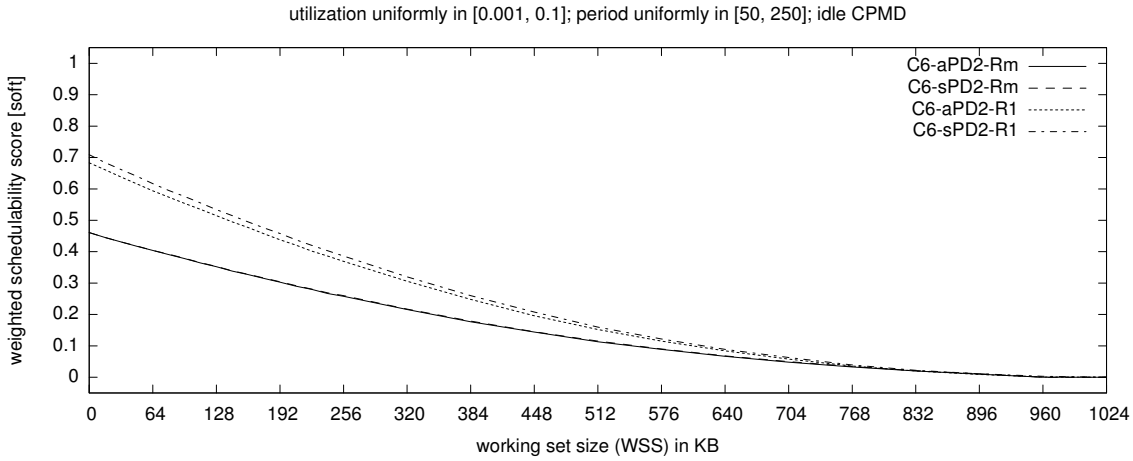
In the SRT case, staggered quanta are in fact preferable to aligned quanta for all tested cluster sizes. One reason is that staggered quanta do not require a shortening of periods to ensure bounded tardiness since stagger latency is simply a source of constant tardiness. Further, *average-case* overheads are much lower when quanta are staggered than when they are aligned since the likelihood of contention is much reduced. Figure 4.24 shows an example of the resulting increased SRT schedulability for uniform light utilizations and long periods. Inset (a) depicts SRT schedulability for a WSS of 64 KB under each of the four clustered PD^2 variants with clusters of size $c = 6$. As previously seen in the HRT case, the curves of the two variants using global interrupt handling, C6-aPD^2-Rm and C6-sPD^2-Rm, overlap and are indistinguishable. In contrast to the HRT case, however, staggered quanta yield a (small) advantage over aligned quanta, as evidenced by the curve of C6-sPD^2-R1, which extends slightly further to the right than that of C6-aPD^2-R1 before all task sets become unschedulable. Inset (b) shows that a small difference persists until the WSS exceeds about 768 KB.

Overall, in the SRT case, C6-sPD^2-R1 and G-sPD^2-R1 are preferable to using global interrupt handling and aligned quanta whenever there are noticeable differences in achieved schedulability. In the case of $c = 2$, bin-packing issues have a bigger impact, such that C2-sPD^2-Rm achieves slightly higher SRT schedulability than C2-sPD^2-R1 if periods are moderate or long.

This concludes our comparison of plugin variants. In the case of P-FP and each of the three PD^2 cluster sizes, one variant was generally preferable in all of the tested scenarios. In the case of EDF-based schedulers, there exist scenarios under which either global or dedicated interrupt handling is preferable. In particular, dedicated interrupt handling is preferable if the numbers of tasks



(a) SRT schedulability for $wss = 64$ KB.



(b) Weighted SRT schedulability score.

Figure 4.24: Graphs showing **(a)** SRT schedulability and **(b)** weighted SRT schedulability score of C6-aPD^2-R1, C6-aPD^2-Rm, C6-sPD^2-Rm and C6-sPD^2-R1 for uniform light utilizations and long periods.

per cluster is large, which results from using large clusters and light utilization distributions, and global interrupt handling is preferable if the number of tasks per cluster is small or bin-packing is difficult, which is typically the case when using small clusters and heavy utilization distributions. In the majority of the tested scenarios, G-EDF-R1 and C6-EDF-R1 are therefore preferable to G-EDF-Rm and C6-EDF-Rm, respectively, whereas C2-EDF-Rm and P-EDF-Rm are typically preferable to C2-EDF-R1 and P-EDF-R1, respectively.

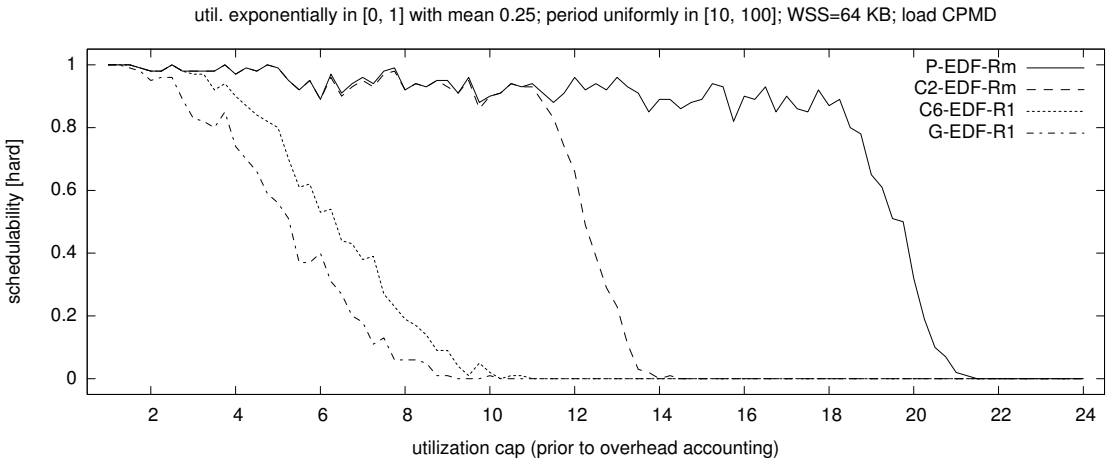
4.5.2 Cluster Size

In this section, we discuss which cluster size is most appropriate for satisfying HRT and SRT constraints on our platform when using either G-EDF or PD^2 within each cluster. For each considered cluster size, we use the plugin configuration that we identified as the best-performing variant as the basis for comparison. We first compare the P-EDF plugin, the C-EDF plugin with $c = 2$ and $c = 6$, and the G-EDF plugin in terms of HRT and SRT schedulability.

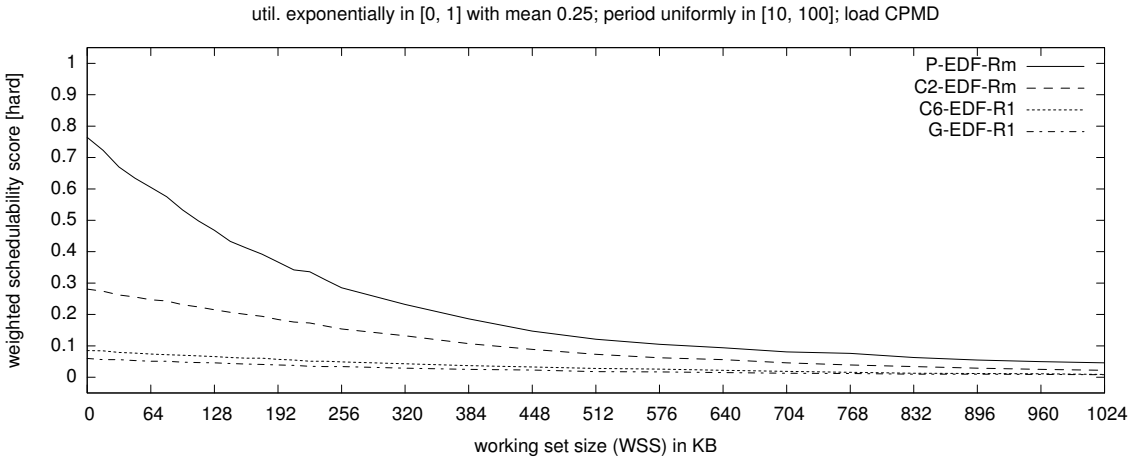
EDF-based schedulers. With regard to HRT schedulability, there is a very clear trend: clusters of size $c = 1$ always yield the highest HRT schedulability, *i.e.*, we found either P-EDF-Rm or P-EDF-R1 to be the best-performing EDF variant in all tested scenarios. A typical example is shown in Figure 4.25. Inset (a) depicts HRT schedulability for medium exponential utilizations, moderate periods, and a WSS of 64 KB under P-EDF-Rm, C2-EDF-Rm, C6-EDF-R1, and G-EDF-R1. The curves of C6-EDF-R1 and G-EDF-R1 quickly start to decrease even for low utilization caps, whereas P-EDF-Rm successfully schedules most task sets with $ucap \leq 18$.

Interestingly, C2-EDF-Rm fails to schedule most task sets starting at $ucap \approx 12$ even though its overheads are not much higher than those of P-EDF-Rm. This reveals algorithmic capacity loss: while the schedulability test applied to each partition under P-EDF-Rm is exact (in the absence of overheads), the G-EDF schedulability tests applied to each partition under C2-EDF-Rm are subject to significant pessimism. Further, EDF is optimal on a uniprocessor (*i.e.*, within each partition), but G-EDF is not optimal on a multiprocessor (*i.e.*, within each cluster).

Figure 4.25(b) shows that P-EDF-Rm achieves a higher weighted schedulability score for all tested WSSs, though the margin decreases as cache-related overheads become predominant. Overall,



(a) HRT schedulability for $wss = 64$ KB.



(b) Weighted HRT schedulability score.

Figure 4.25: Graphs showing **(a)** HRT schedulability and **(b)** weighted HRT schedulability score of four EDF-based schedulers with different cluster sizes for medium exponential utilizations and moderate periods.

our results show that the algorithmic-capacity loss due to bin-packing in P-EDF is less limiting than the overhead-related and algorithmic capacity loss incurred by G-EDF and either C-EDF variant.

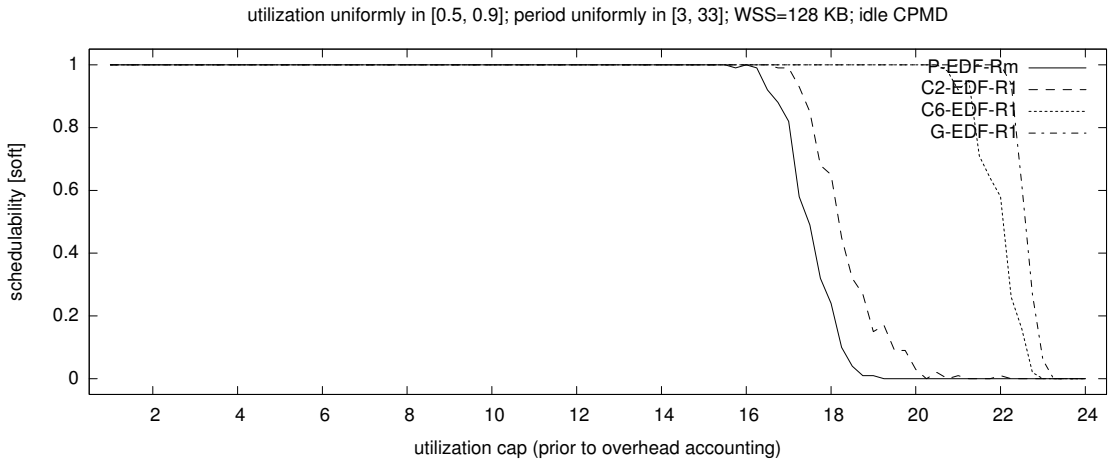
In contrast, in the SRT case, no single cluster size is always preferable. One example illustrating this is shown in Figure 4.26, which depicts SRT schedulability for uniform heavy utilizations and short periods. In the case of $wss = 128$ KB shown in Figure 4.26(a), the order from Figure 4.25 is reversed: G-EDF-R1 guarantees bounded tardiness to the most task sets, followed by the next-largest cluster size C6-EDF-R1, and then followed by C2-EDF-R1 and finally P-EDF-Rm. There are two main contributing factors to this reversal. First, the scenario depicted in Figure 4.26 with uniform heavy utilizations generates task sets that are difficult to partition, which affects smaller cluster sizes negatively. Second, the generated task sets are not very sensitive to overheads since each task set contains only few tasks. Therefore, global scheduling without a separate task assignment phase performs best, despite higher average-case overheads.

However, this is not the case for all WSSs, as is evident in Figure 4.26(b). For small WSSs, G-EDF-R1 is clearly preferable to P-EDF-Rm, C2-EDF-R1, and C6-EDF-R1. However, there exists a cross-over point at $wss = 448$ KB after which P-EDF-Rm is preferable to the other choices. This is an effect of cache affinity: since jobs do not migrate under P-EDF-Rm, cache-related overheads are much lower under it than under cluster sizes that do not maintain L1 cache affinity. Further, note that C2-EDF-R1 exhibits a trend that is similar to that of P-EDF-Rm, but very different from that of G-EDF-R1 and C6-EDF-R1. This highlights the advantage of maintaining L2 cache affinity (C2-EDF-R1) when assuming idle CPMD.

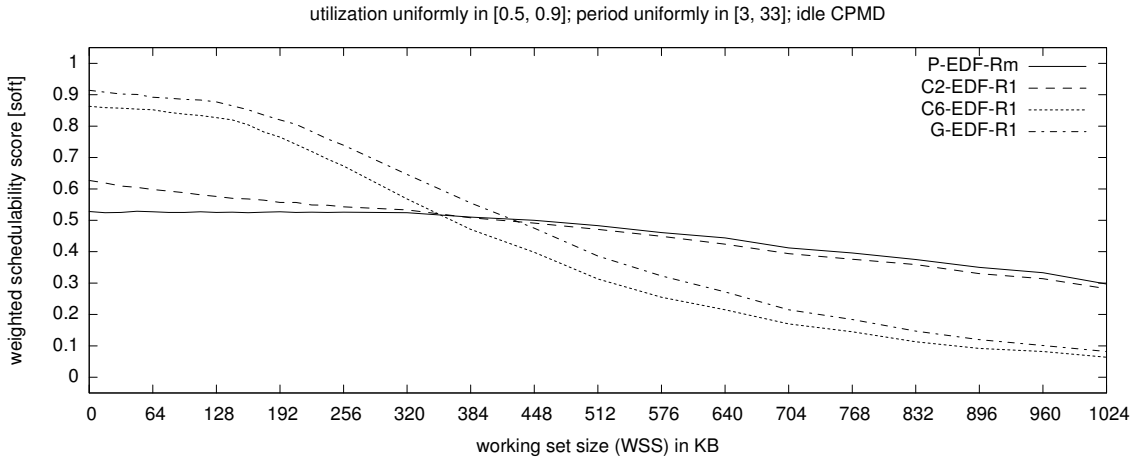
Additional trends are apparent in graphs that are available online (Brandenburg, 2011):

- When assuming load CPMD instead of idle CPMD, then there is little benefit to maintaining cache affinity (recall Section 4.4.2). As a result, no cross-over point exists in the case of uniform heavy utilizations, and G-EDF-R1 is preferable for all WSSs when assuming load CPMD (in the SRT case).

- In scenarios that generate task sets that are less difficult to partition and that are sensitive to scheduling overheads (*i.e.*, medium or light utilization distributions with moderate or short periods), smaller cluster sizes (P-EDF-Rm or C2-EDF-R1) are often preferable to larger cluster sizes (G-EDF-R1 or C6-EDF-R1).



(a) SRT schedulability for $wss = 128$ KB.



(b) Weighted SRT schedulability score.

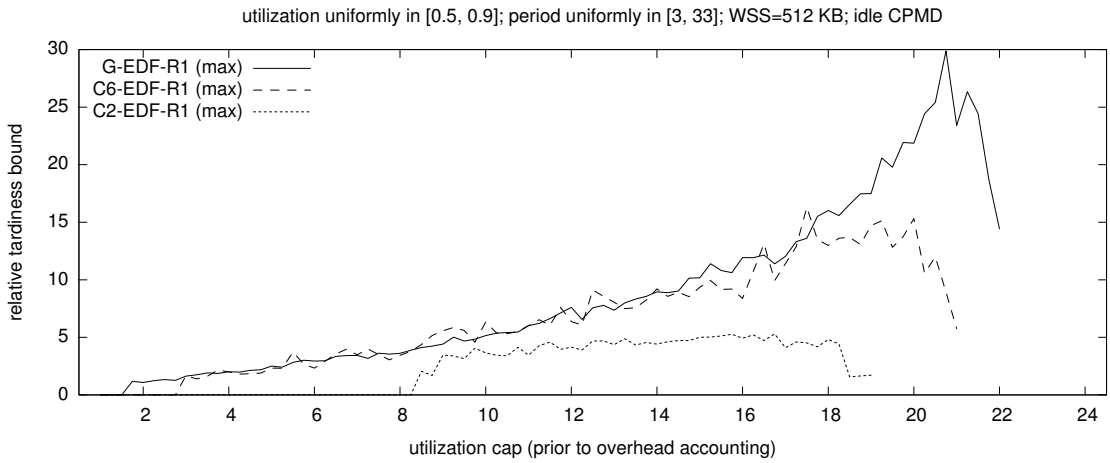
Figure 4.26: Graphs showing **(a)** SRT schedulability and **(b)** weighted SRT schedulability score of four EDF-based schedulers with different cluster sizes for uniform heavy utilizations and short periods.

Overall, in the SRT case, there is no single best cluster size; rather, the best cluster size to use depends on the task parameter distributions, the WSS, and whether load or idle CPMD is assumed. For example, in the case of uniform heavy utilizations, the choice of scheduler is application-dependent: in systems with significant cache and memory bus contention (*i.e.*, in the case of load CPMD), G-EDF-R1 is preferable for all WSSs (supporting graph available online), whereas if jobs can maintain some degree of cache affinity while preempted (*i.e.*, in the case of idle CPMD), then P-EDF-Rm is preferable for large WSSs, as can be seen in Figure 4.26(b). This suggests that cluster sizes should be configurable in RTOSs (as they indeed are when using processor affinity masks).

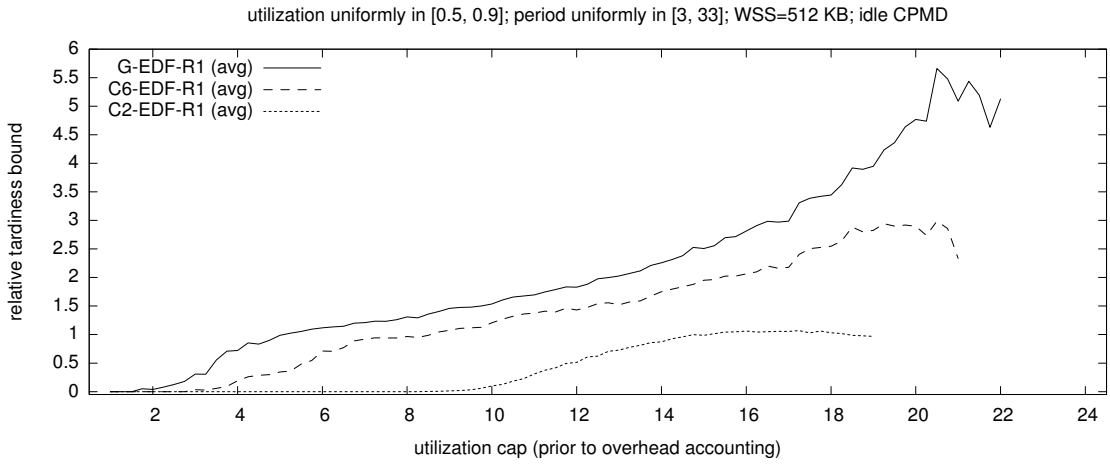
Another consideration in the SRT case is tardiness. In the case of P-EDF, when a task set is SRT schedulable, it has zero tardiness since EDF is HRT optimal on a uniprocessor (aside possible minor tardiness due to the use of average-case overheads). In the case of $c > 1$, however, jobs may incur non-negligible tardiness since G-EDF is not HRT optimal on a multiprocessor.

Two example graphs depicting relative tardiness are shown in Figure 4.27. Recall from Section 4.1.8 that relative tardiness is a normalized notion of tardiness that expresses by how many jobs a task may lag behind in the worst case, and that the reported relative tardiness is based on Devi and Anderson's analytical tardiness bound (Theorem 2.10 on page 85) and *not* based on simulations. That is, Figure 4.27 shows guaranteed upper bounds on maximum relative tardiness which are likely pessimistic in many cases.

Inset (a) shows maximum relative tardiness under C2-EDF-R1, C6-EDF-R1, and G-EDF-R1 for uniform heavy utilizations and short periods, which are the task parameter distributions that resulted in the highest tardiness bounds in our experiments. As can be seen in inset (a), maximum relative tardiness bounds are much larger under global scheduling than when using smaller cluster sizes. Under C2-EDF-R1, the maximum relative tardiness bound is slightly above 5, which implies that buffering data corresponding to six jobs is sufficient to mask worst-case tardiness. In contrast, tasks may lag by up to 30 jobs under G-EDF-R1, and by up to 17 jobs under C6-EDF-R1. Interestingly, the maximum relative tardiness under G-EDF-R1 and C6-EDF-R1 is similar until $ucap \approx 17$. The curves exhibit decreasing trends for large utilization caps and do not fully extend to $ucap = 24$ since only few of the high-utilization task sets are schedulable. (Recall from Section 4.1.8 that task sets with unbounded tardiness are excluded prior to computing maximum and average relative tardiness bounds.)



(a) Maximum relative tardiness for $wss = 512$ KB.



(b) Average relative tardiness for $wss = 64$ KB.

Figure 4.27: Graphs showing **(a)** maximum relative tardiness and **(b)** relative average tardiness of EDF-based schedulers for uniform heavy utilizations and short periods. The reported maximum and average relative tardiness is based on analytical bounds (and not on observed tardiness).

On average, however, relative tardiness bounds are much smaller. This is apparent in Figure 4.27(b). The order of the curves remains the same, with tasks incurring the highest relative tardiness bounds under G-EDF-R1, but the magnitude of the curves is reduced greatly. Under G-EDF-R1, tasks are on average guaranteed to lag at most six jobs behind, at most four jobs under C6-EDF-R1, and only at most one job under C2-EDF-R1. In contrast to inset (a), the curves of G-EDF-R1 and C6-EDF-R1 are clearly distinct.

To summarize, in the HRT case, we found a cluster size of $c = 1$ to be best when using EDF-based scheduling in our system. In the SRT case, a choice of $c = 1$ also works well in many scenarios, unless the generated task sets are difficult to partition, in which case C6-EDF-R1 and G-EDF-R1 result in higher schedulability unless working sets are large. It should be noted, however, that jobs may incur significant tardiness under G-EDF-R1 and C6-EDF-R1, whereas jobs are (mostly) not affected by tardiness under P-EDF.

PD^2-based schedulers. In the case of PD^2, we considered three cluster sizes in our case study: global PD^2 ($c = 24$) and two smaller cluster sizes, one based on shared L2 caches ($c = 2$) and one based on shared L3 caches ($c = 6$). We did not consider partitioned PD^2 as it offers no advantages over P-EDF since EDF is HRT optimal on a uniprocessor. A major difference to EDF-based schedulers is that PD^2 is also HRT optimal on a multiprocessor (with regard to implicit deadlines). Therefore, the choice of cluster size is mainly a tradeoff between overheads, which are lower in smaller clusters due to reduced contention and increased cache affinity, and bin-packing issues, which mostly affect small clusters (*i.e.*, $c = 2$).

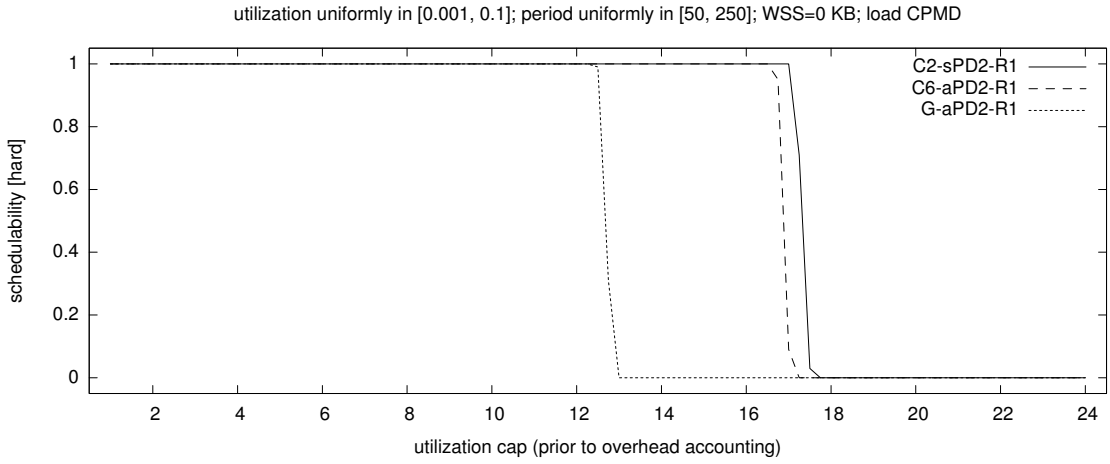
As discussed above in Section 4.5.1, in the HRT case, dedicated interrupt handling is generally preferable for each of the three cluster sizes, and staggered quanta are preferable for $c = 2$, whereas aligned quanta are preferable for $c = 6$ and $c = 24$. We therefore consider only C2-sPD^2-R1, C6-aPD^2-R1, and G-aPD^2-R1 in the comparison of HRT schedulability under PD^2-based schedulers. A clear trend is apparent in our data: in virtually all tested scenarios, C2-sPD^2-R1 yielded (slightly) better schedulability than either of the other two variants. The larger cluster sizes, C6-aPD^2-R1 and G-aPD^2-R1, yield lower HRT schedulability even though they are less affected by bin-packing-related algorithmic capacity loss. This shows that PD^2 is very sensitive to overheads, which are lowest

in the case of $c = 2$. Again, this tradeoff could not have been identified without an overhead-aware evaluation methodology since it depends on the magnitude of overheads.

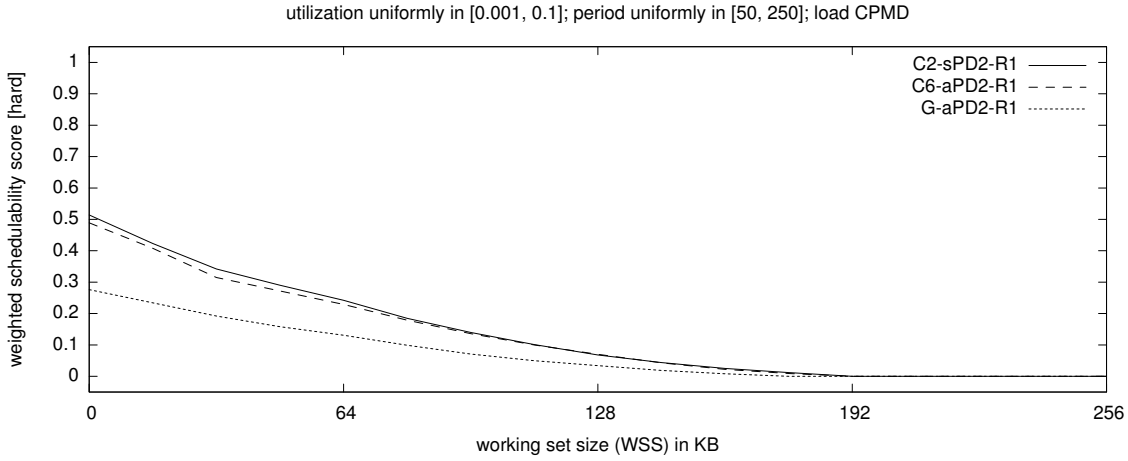
An example graph exhibiting this trend is presented in Figure 4.28(a), which shows HRT schedulability for uniform light utilizations and long periods, and assumes a negligible WSS. While the difference to C6-aPD^2-R1 is not large, it is apparent that C2-sPD^2-R1 can sustain somewhat higher utilizations before task sets become unschedulable. The weighted HRT schedulability score of the three cluster sizes depicted in inset (b) of the same figure shows that performance deteriorates under each variant rapidly as the WSS increases. C2-sPD^2-R1 remains slightly preferable until $wss = 192$ KB. For larger WSSs, none of the generated task sets could be claimed schedulable under any of the PD^2 variants since the effective quantum size is eclipsed by the high CPMD (recall that the depicted HRT graphs assume worst-case CPMD in a system under heavy load).

Recall from Section 4.5.1 that staggered quanta are preferable for all cluster sizes in the SRT case. Dedicated interrupt handling, however, is only preferable for $c = 6$ and $c = 24$. In the case of $c = 2$, global interrupt handling is preferable since the loss of a processor amplifies the bin-packing issues inherent in small cluster sizes. Therefore, we compare C2-sPD^2-Rm, C6-sPD^2-R1, and G-sPD^2-R1 in terms of SRT schedulability. While the schedulability condition for PD^2 is the same under either HRT or SRT scheduling, we assume average-case overheads when testing whether a task set is SRT schedulable. Since average-case overheads are much lower than worst-case overheads, PD^2-based schedulers exhibit much improved schedulability in the SRT case.

An example graph is shown in Figure 4.29(a), which depicts SRT schedulability assuming uniform heavy utilizations, long periods, and a WSS of 80 KB. Notably, the curves of C6-sPD^2-R1 and G-sPD^2-R1 look markedly different from that of C2-sPD^2-Rm. Both C6-sPD^2-R1 and G-sPD^2-R1 exhibit some capacity loss and declining trends even for low utilization caps, whereas C2-sPD^2-Rm achieves a schedulability of 1.0 until $ucap \approx 17$, after which schedulability rapidly drops to 0.0. This difference in trend arises because task sets are claimed unschedulable for different reasons under each scheduler: under C2-sPD^2-Rm, task sets fail to be scheduled when they cannot be partitioned, which is mainly a question of the number of heavy tasks (on average, a utilization cap of 17 generates 24 tasks under this utilization distribution). In contrast, under C6-sPD^2-R1 and G-sPD^2-R1, partitioning is not a primary concern and task sets fail to be schedulable when $e'_i > p'_i$ for some T_i due to an insufficiently small effective quantum size Q'.

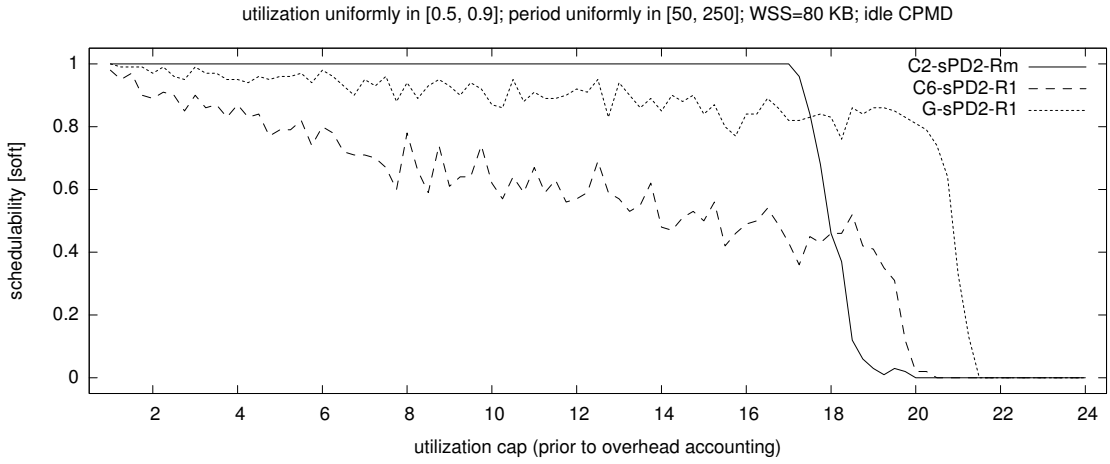


(a) HRT schedulability for $wss = 0$ KB.

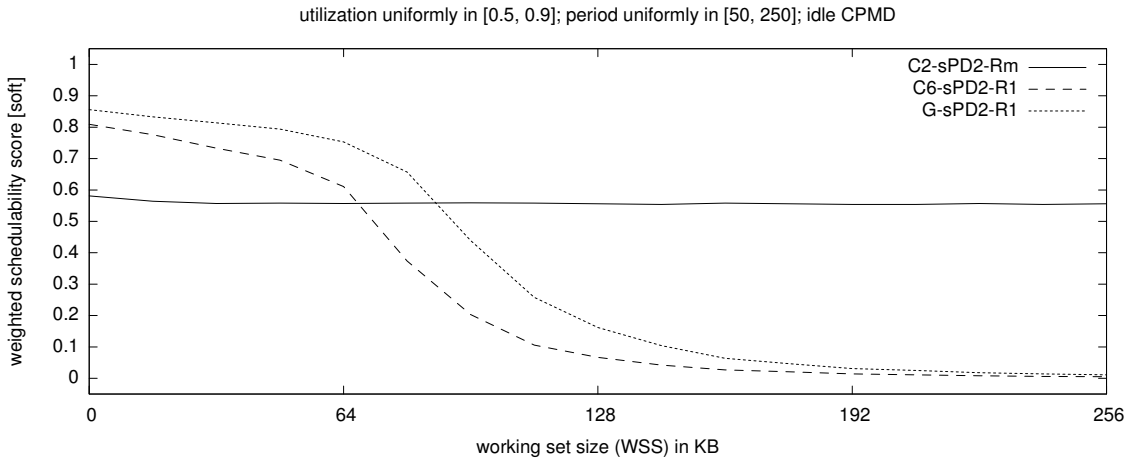


(b) Weighted HRT schedulability score.

Figure 4.28: Graphs showing **(a)** HRT schedulability and **(b)** weighted HRT schedulability score of PD^2-based schedulers for light uniform utilizations and long periods.



(a) SRT schedulability for $wss = 80$ KB.



(b) Weighted SRT schedulability score.

Figure 4.29: Graphs showing **(a)** SRT schedulability and **(b)** weighted SRT schedulability score of PD^2-based schedulers for uniform heavy utilizations and long periods.

Since the effective quantum size is WSS-dependent, one might expect these trends to be different for smaller WSSs. Figure 4.29(b) illustrates that this is indeed the case. Assuming average-case idle CPMD as in Figure 4.29(b), there exists a cross-over point after which C2-sPD^2-Rm achieves a higher weighted SRT schedulability score than either C6-sPD^2-R1 or G-sPD^2-R1, which corresponds to a larger number of task sets being SRT schedulable under C2-sPD^2-Rm than under the other cluster sizes. In the case of smaller WSSs, larger cluster sizes are preferable since bin-packing issues are dominant. This highlights the increasing benefit of maintaining cache affinity as WSSs grow larger. In contrast, different trends are apparent in graphs that are available online (Brandenburg, 2011): when assuming load CPMD instead of idle CPMD, cache affinity issues cease to play a major role and bin-packing issues dominate. As a result, larger clusters become preferable to C2-sPD^2-Rm when assuming load CPMD.

Overall, we found that differences among cluster sizes are typically small in the HRT case, but that if noticeable differences exist, then C2-sPD^2-R1 achieves somewhat higher schedulability. In the SRT case, when assuming CPMD as measured in a system without background load, large cluster sizes are only preferable for small WSSs due to the increased lock contention and the reduced cache affinity inherent in larger cluster sizes. In summary, on our platform, we found C2-sPD^2-R1 to be the best-performing PD^2-based scheduler in most of the tested scenarios. With regard to tardiness, there are no differences among PD^2-based schedulers since the maximum tardiness of SRT schedulable task sets is determined by quantum staggering (aside rare tardiness due to above-average overheads; see Section 4.1.3), which is the same for all cluster sizes.

4.5.3 Scheduler Selection

Having identified the preferable cluster sizes for EDF-based and PD^2-based schedulers, we next compare PD^2-based scheduling, EDF-based scheduling, and P-FP in terms of HRT and SRT schedulability to identify the "best" HRT and SRT schedulers (among those examined) for our platform. We begin with the HRT case.

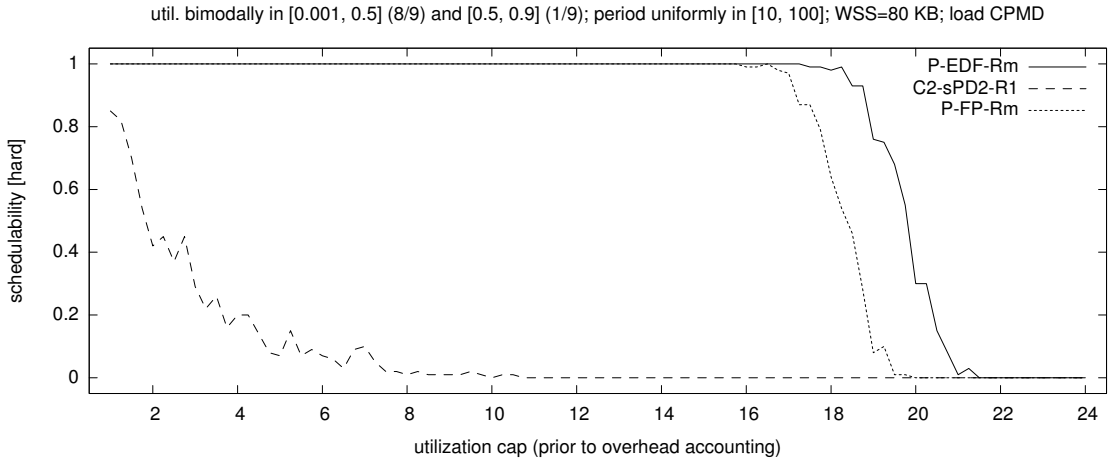
HRT comparison. In the preceding sections, we identified P-FP-Rm, P-EDF (either P-EDF-Rm or P-EDF-R1, depending on task count), and C2-sPD^2-R1 as the preferable HRT schedulers from

each category. We therefore limit our attention to these schedulers in this section; HRT schedulability data and graphs for all 22 evaluated scheduler variants can be found online (Brandenburg, 2011).

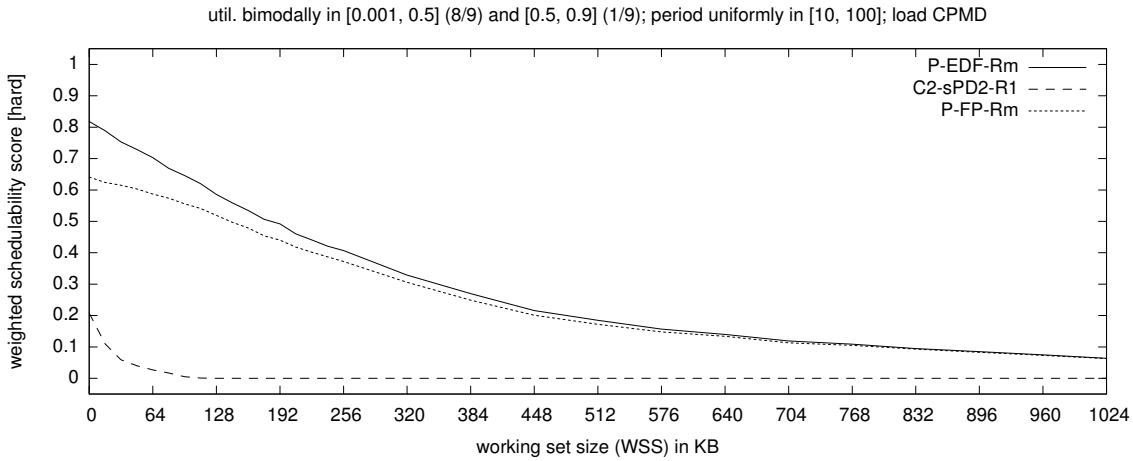
In the HRT case, there is again a very clear trend: in all test scenarios, one of the two P-EDF variants perfoms better than the other choices, or is at least no worse than other well-performing schedulers. A typical example graph is shown in Figure 4.30(a), which depicts HRT schedulability for bimodal light utilizations, moderate periods, and a WSS of 80 KB. Due to high overheads and the large number of preemptions and migrations inherent in PD^2-based scheduling, C2-sPD^2-R1 suffers from high capacity loss even for low-utilization task sets. In contrast, P-EDF-Rm maintains near-optimal schedulability until the total utilization of generated task sets exceeds 18. P-FP-Rm achieves similarly high schedulability until $ucap = 16$. This shows that neither (slightly) lower scheduling overhead nor the less-pessimistic interrupt accounting techniques applicable to P-FP outweigh the optimality of EDF on a uniprocessor in this scenario.

Figure 4.30(b) illustrates that C2-sPD^2-R1 does not become significantly more competitive even for negligible WSSs (though a small improvement in weighted HRT schedulability score is apparent). It further reveals that the gap between P-FP-Rm and P-EDF-Rm narrows as WSSs become larger and finally disappears for $wss > 576$ KB; the largest difference between the two arises when WSSs are negligibly small. This is because the algorithmic differences between the two schedulers have only a small impact when overheads are dominated by preemption costs (large CPMD and high utilizations frequently result in $e_i > p_i$ for some T_i with a short period, which renders the task set infeasible and thus unschedulable under either scheduler).

Overall, one or both of the P-EDF variants perform as well or better than P-FP-Rm in a large majority of the examined scenarios. P-FP-Rm is most competitive in the case of short periods, where it can sustain increases in WSSs somewhat better than P-EDF-Rm, but any difference in schedulability between the best-performing P-EDF variant and the best-performing P-FP variant remain small. In the case of uniform heavy utilizations with short periods, P-FP-Rm does achieve a higher weighted HRT schedulability score if $wss \leq 128$ KB, but the difference in performance is minuscule (less than 0.02). Generally speaking, P-EDF is clearly the scheduler that is suited best to satisfying HRT constraints on our platform (with respect to the 22 evaluated scheduler variants and the tested parameter distributions).



(a) HRT schedulability for $wss = 80$ KB.



(b) Weighted HRT schedulability score.

Figure 4.30: Graphs showing **(a)** HRT schedulability and **(b)** weighted HRT schedulability score of P-FP-Rm, P-EDF-Rm, and C6-aPD^2-R1 for bimodal light utilizations and moderate periods.

SRT constraints. Recall that, in the SRT case, none of the EDF-based schedulers was preferable in all of the tested scenarios. Therefore, we consider P-EDF-Rm, C2-EDF-R1, and G-EDF-R1 in the following SRT schedulability comparison (C6-EDF-R1 performs similarly to G-EDF-R1). Among the PD^2-based schedulers, C2-sPD^2-Rm performs best (unless WSSs are very small). We further omit P-FP-Rm since it did not perform better than P-EDF-Rm in the HRT case. SRT schedulability results for all 22 scheduler variants, including P-FP-Rm, can be found online (Brandenburg, 2011).

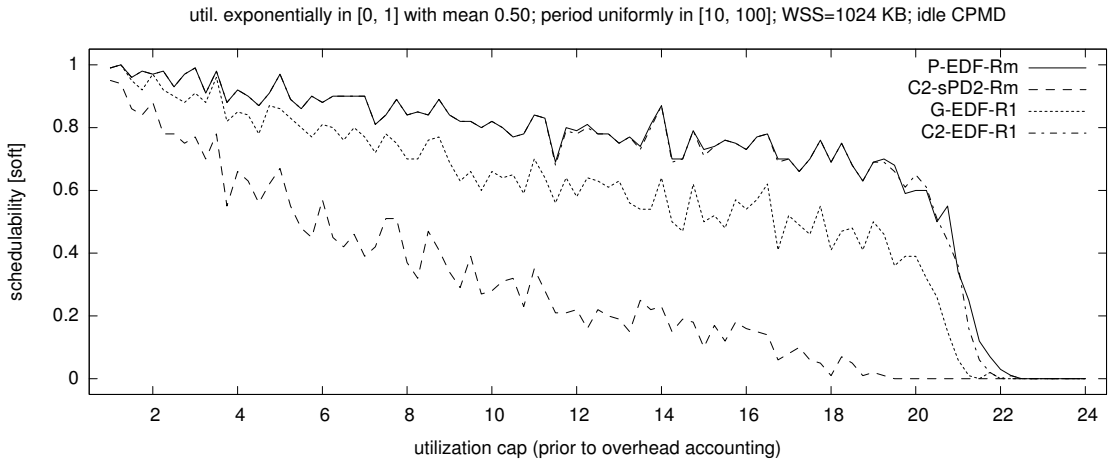
Given the strong performance of both P-EDF variants in the HRT case, it is not surprising that they perform well in many scenarios in the SRT case as well. Most of the distributions generate task sets that, on average, can be partitioned with relative ease using the worst-fit decreasing heuristic. Therefore, bin-packing limitations are typically not the deciding factor. The P-EDF variants thus compare favorably in many cases due to L1 cache affinity and lower scheduling overhead.

An example graph depicting SRT schedulability under each of the considered scheduler variants is shown in Figure 4.31(a). The depicted scenario—exponential heavy utilizations, moderate periods, and a large WSSs of 1024 KB—results in task sets that can be fairly difficult to partition since, on average, each task requires half a processor. Nonetheless, P-EDF-Rm achieves high schedulability.

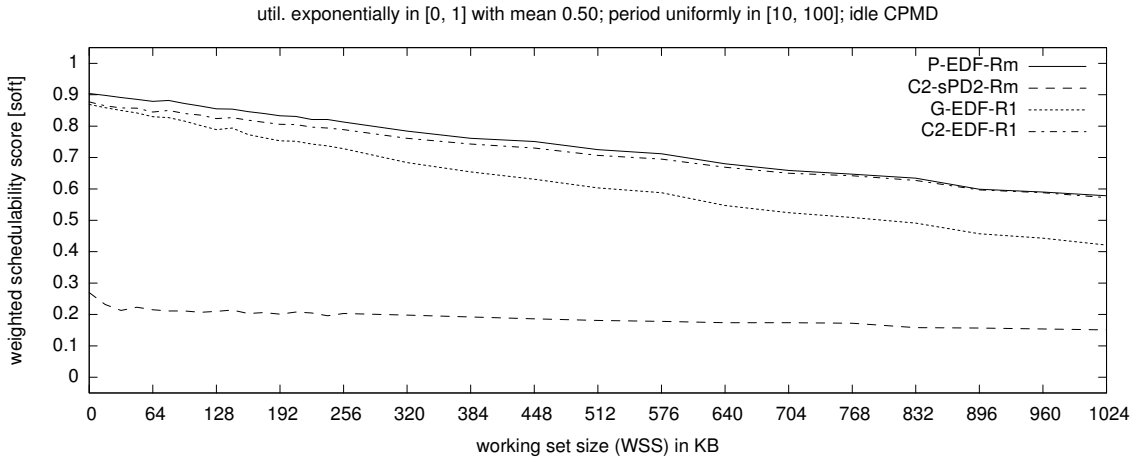
Due to the large WSSs, each scheduler exhibits capacity loss even for low-utilization task sets. However, there is consistently less capacity loss under P-EDF-Rm than under G-EDF-R1 and C2-sPD^2-R1. G-EDF-R1 is subject to larger CPMD than P-EDF-Rm since jobs do not benefit from cache affinity under G-EDF-R1, and C2-sPD^2-R1 is subject to larger overhead-related capacity loss than the other schedulers since it preempts jobs much more frequently. The curve of C2-EDF-R1 coincides in large parts with P-EDF-Rm since C2-EDF-R1 is very similar to P-EDF-Rm in that it incurs only little lock contention and ensures L2 cache affinity. In contrast to the HRT case (*e.g.*, Figure 4.25), C2-EDF-R1 does not suffer capacity loss due to pessimistic schedulability analysis since G-EDF is SRT optimal on multiprocessors.

The weighted SRT schedulability score graph in Figure 4.31(b) shows that these trends also occur for smaller WSSs. Notably, the performance gap between G-EDF-R1 and P-EDF-Rm narrows for small WSSs due to the reduced benefit of cache affinity. The weighted SRT schedulability score of C2-sPD^2-R1, however, does not improve significantly even for very small WSSs.

A notable exception to the discussed trends arises in distributions that predominantly generate high-utilization tasks such as the uniform heavy and the bimodal heavy distributions. The resulting



(a) SRT schedulability for $wss = 1024$ KB.



(b) Weighted SRT schedulability score.

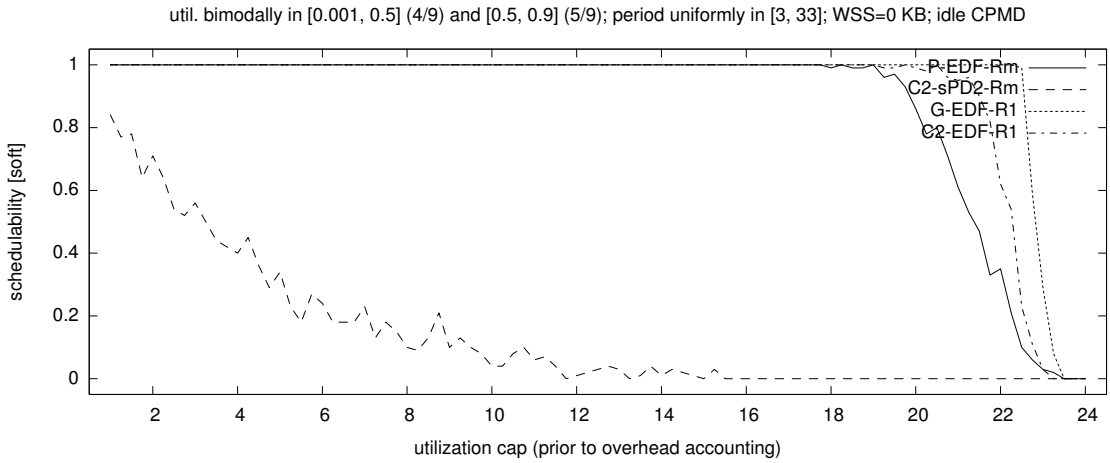
Figure 4.31: Graphs showing **(a)** SRT schedulability and **(b)** weighted SRT schedulability score of Comparison of P-EDF-Rm, C2-EDF-R1, G-EDF-R1, and C2-sPD^2-Rm in terms of SRT schedulability for exponential heavy utilizations and moderate periods.

task sets are very difficult to partition, which causes P-EDF-Rm to incur significant algorithmic capacity loss. Such a scenario is depicted in Figure 4.32(a), which shows SRT schedulability graphs assuming bimodal heavy utilizations, short periods, and a negligible WSS. Bin-packing is the major cause of capacity loss in this scenario and cache affinity is irrelevant since $wss = 0$ KB. Consequently, the best-performing algorithm is G-EDF-R1, which ensures bounded tardiness to all generated task sets until $ucap \approx 23$. In contrast, P-EDF-R1 exhibits capacity loss starting at $ucap \approx 19$. As expected, C2-EDF-R1 is less affected by bin-packing constraints than P-EDF-R1, but still suffers greater capacity loss than G-EDF-R1.

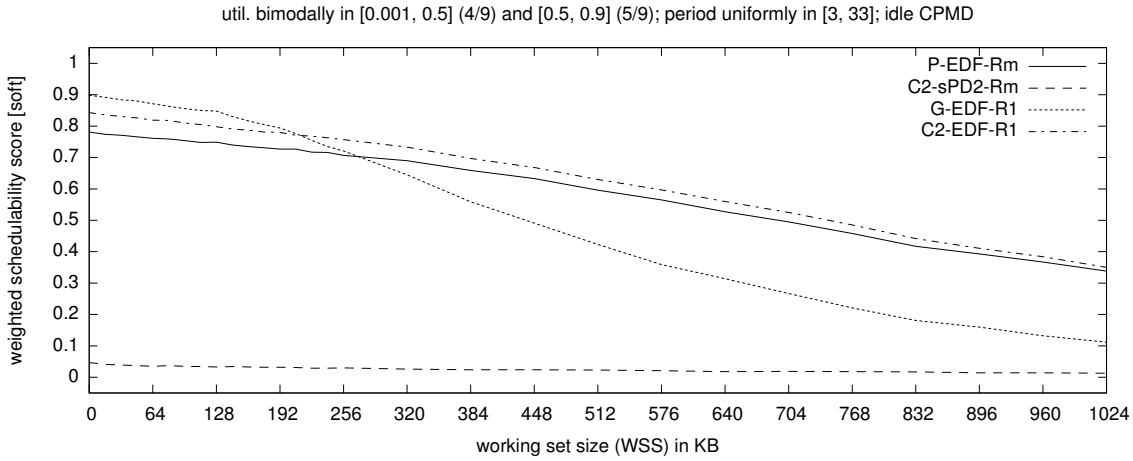
Figure 4.32(b) reveals that the performance of G-EDF-R1 is closely related to the assumed WSS. The weighted SRT schedulability score of G-EDF-R1 decreases more rapidly than that of the other schedulers with increasing WSS. At $wss \approx 192$, the curves of G-EDF-R1 and C2-EDF-R1 cross, which indicates that, on our platform and assuming idle CPMD, C2-EDF-R1 is subject to less capacity loss than G-EDF-R1 for WSSs exceeding 192 KB. In contrast, other trends are apparent in graphs that are available online (Brandenburg, 2011): when assuming load CPMD instead of idle CPMD, the benefit of increased cache affinity under C2-EDF-R1 is small and G-EDF-R1 remains preferable for large WSSs, too.

The example shown in Figure 4.32 illustrates that, in contrast to the HRT case, G-EDF and C-EDF variants can be effective at overcoming bin-packing limitations in the SRT case. There are two reasons for this: first, since average-case overheads are assumed, global and clustered schedulers are less penalized for lock contention than in the HRT case; and second, since G-EDF is SRT optimal on a multiprocessor, no algorithmic capacity loss arises within each cluster.

Depending on the application, a potential disadvantage of G-EDF-R1 and C2-EDF-R1 is increased tardiness. Under P-EDF, if a task set is SRT schedulable, then its tardiness is zero (aside rare tardiness due to above-average overheads). In contrast, tasks can be subject to significant tardiness under G-EDF. This is apparent in Figure 4.33, which shows maximum and average relative tardiness bounds in the scenario depicted in Figure 4.32. As can be seen in inset (a), the maximum relative tardiness bound exceeds 20 for high utilizations, *i.e.*, in some cases, it cannot be ruled out that tasks may fall behind in their execution by up to 21 jobs. On average, as shown in inset (b), tasks are guaranteed a relative tardiness bound in the range $[2, 5]$ if $ucap \geq 18$, *i.e.*, in the range of high total utilization where G-EDF-R1 achieves higher SRT schedulability than P-EDF-Rm in

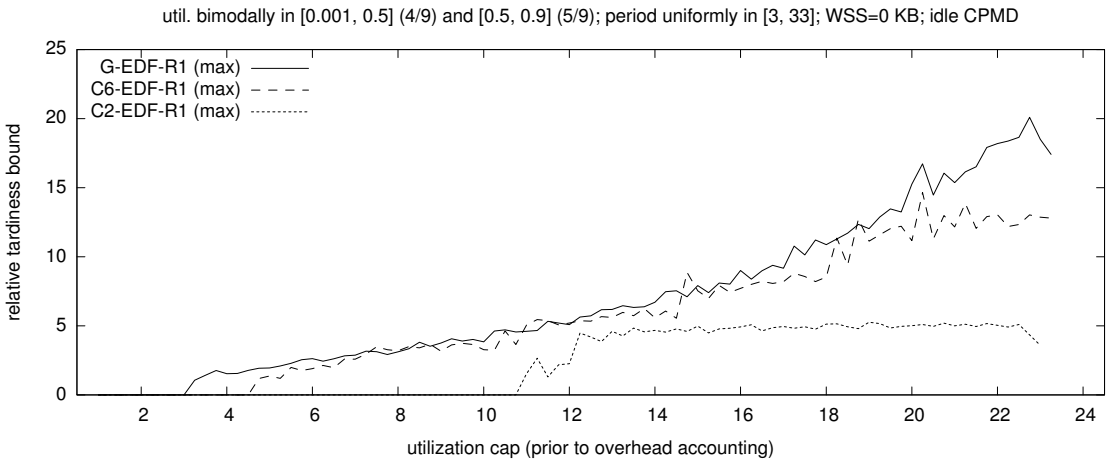


(a) SRT schedulability for $wss = 0$ KB.

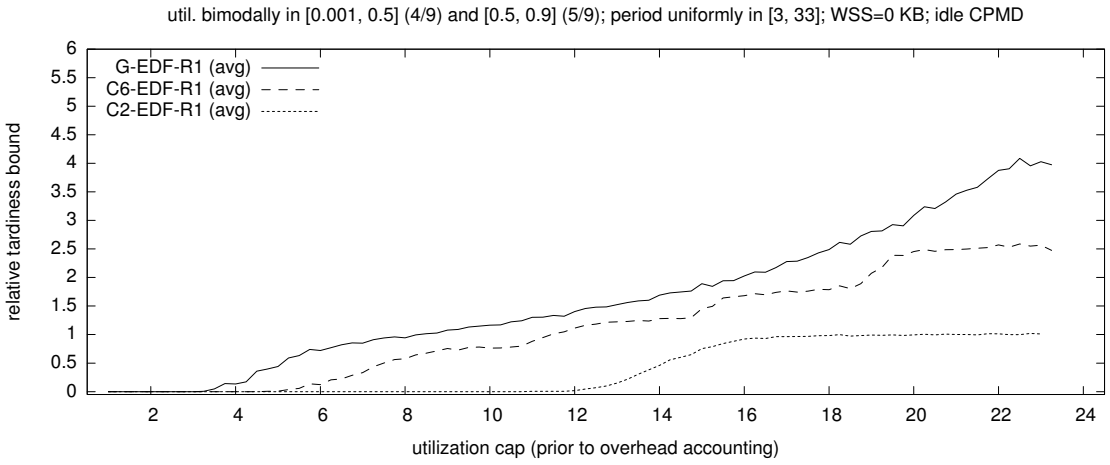


(b) Weighted SRT schedulability score.

Figure 4.32: Graphs showing **(a)** SRT schedulability and **(b)** weighted SRT schedulability score of P-EDF-Rm, C2-EDF-R1, G-EDF-R1, and C2-sPD^2-Rm for bimodal heavy utilizations and short periods.



(a) Maximum relative tardiness for $wss = 1024$ KB.



(b) Average relative tardiness for $wss = 1024$ KB.

Figure 4.33: Graphs showing **(a)** maximum relative tardiness and **(b)** relative average tardiness of EDF-based schedulers for exponential heavy utilizations and moderate periods. The reported maximum and average relative tardiness is based on analytical bounds (and not on observed tardiness).

Figure 4.32(a). Whether G-EDF-R1 is in fact preferable to P-EDF-Rm for high-utilization task sets thus depends on the degree to which a deployed application is sensitive to tardiness (or whether there is sufficient memory available to mask tardiness with input and output buffers).

It is worth reiterating that the reported relative tardiness is based on analytical bounds on worst-case tardiness, and not on observed tardiness. Actual relative tardiness is likely much lower in real systems. In our experience, tasks almost never experience tardiness when provisioned according to their worst-case execution requirements as there is typically some slack in the schedule that counteracts any build-up of tardiness (since it is unlikely, though not impossible, that all jobs exhibit worst-case behavior at the same time).

Summary. This concludes the discussion of our case study. We have demonstrated how to apply the proposed overhead-aware evaluation methodology and reported overheads for 22 scheduler plugin configurations as observed in LITMUS$^{\mathrm{RT}}$ on a real 24-core Intel Xeon system. Based on large-scale schedulability experiments involving 27 different task parameter distributions, we identified P-EDF as the best-performing HRT scheduler on our platform. In the SRT case, P-EDF performs well, too, unless task sets are difficult to partition, in which case G-EDF or one of the C-EDF variants is effective at overcoming bin-packing limitations. Due to comparatively large overhead-related capacity loss, PD^2-based schedulers were only viable in few of the considered scenarios, and only for small WSSs.

4.5.4 Prior Studies in Context

Our overhead-aware methodology and the case study reported herein are the result of many iterative refinements that were accumulated during several years of LITMUS$^{\mathrm{RT}}$-based research. Several prior studies were conducted and published, and our work presented in this dissertation greatly benefited from "lessons learned" in those prior studies. Most influential were four LITMUS$^{\mathrm{RT}}$-based schedulability studies (Calandrino *et al.*, 2006; Brandenburg *et al.*, 2008; Brandenburg and Anderson, 2009a; Bastoni *et al.*, 2010b). In the following, we relate our current methodology and the trends reported in the previous sections to these earlier studies.

Calandrino *et al.* (2006). Not surprisingly, our methodology and results deviate the most from the initial LITMUS$^{\mathrm{RT}}$ study carried out by Calandrino *et al.* (2006), which predates our work. Calandrino

et al. (2006) evaluated G-EDF, P-EDF, and PD^2 with both aligned and staggered quanta. Major differences arise since Calandrino *et al*. used a pre-multicore multiprocessor platform consisting of only four processors (each clocked at 2.7 GHz) with private L2 caches (512 KB). Due to the much smaller number of processors, issues of memory and lock contention were likely less severe in their system. There were also major differences in the implementation of LITMUS$^{\text{RT}}$. The early prototype used by Calandrino *et al*. (2006) used quantum-based scheduling for all plugins and implemented only minimal scheduling functionality (*e.g.*, real-time jobs could not suspend, lest the kernel panic). Such differences in plugin implementation make direct "apples-to-apples" comparisons of observed overheads and obtained schedulability results difficult (if not impossible).

In addition to platform disparities, there are also significant variations in the underlying evaluation methodology. In particular, Calandrino *et al*. (2006) did not employ background workloads to create memory contention (*i.e.*, their setup corresponds to our idle CPMD configuration) and used a much smaller number of overhead samples to derive overheads.[10] Further, their overhead model did not account for release interrupts at all, which we have found to be a significant source of delays and pessimism, and hence to have a major influence on scheduler performance. When partitioning task sets, Calandrino *et al*. (2006) used the first-fit decreasing heuristic, whereas we employ the worst-fit decreasing heuristic. Since the first-fit decreasing heuristic does not spread out load evenly, it is more likely to over-utilize a processor after tasks have been inflated to account for overheads.

Given the number of major differences in both approach and underlying platform, it is not surprising that Calandrino *et al*.'s results differ from ours.

With regard to overheads, Calandrino *et al*. (2006) measured CPMD for WSSs in the range from 4 KB to 256 KB using a write ratio of 1 and observed *maximum* delays that roughly correspond to *average-case* load CPMD measured on our platform as shown in Figure 4.17(b). This difference can likely be attributed to the absence of cache-polluting background processes and to the fact that memory bus contention is much less severe if only four processors compete for memory bandwidth. Another dissimilarity is the way that CPMD was recorded. Calandrino *et al*. (2006) recorded CPMD in kernel threads within kernel space under each of the evaluated plugins at the beginning of a quantum, which is similar to the schedule-sensitive method. In contrast, we used CPMD data

[10]The exact number of samples was not reported by Calandrino *et al*., but it has been confirmed in private communication that at most few hundred samples were collected. In contrast, we recorded billions of valid samples in our study.

obtained with the synthetic method in this dissertation, which cannot reflect differences among schedulers. Calandrino *et al*. found migration costs under PD^2 and G-EDF to be slightly higher than preemption costs under P-EDF, and also found migration costs to be lower under staggered quanta than under aligned quanta (likely due to a reduction in bus contention). In our study, differences in bus contention are reflected by the use of both load and idle CPMD (since high-contention scenarios are also possible when using staggered quanta; they are just less likely to occur).

A striking difference is apparent in the magnitudes of reported scheduling overheads. For P-EDF, Calandrino *et al*. (2006) reported *maximum* scheduling overhead that is comparable in magnitude to *average-case* scheduling overhead measured on our implementation. This can likely be attributed to **(i)** the absence of a background workload in Calandrino *et al*.'s experimental setup, **(ii)** to the benefits of relatively large, private L2 caches and a higher processor frequency, and **(iii)** to the much smaller sample size underlying Calandrino *et al*.'s scheduling overhead estimates.

For G-EDF and PD^2, Calandrino *et al*. reported *maximum* scheduling overhead that is one to two orders of magnitudes lower than *average-case* scheduling overhead on our platform. Reasons (i)–(iii) above likely also explain some of the difference under these plugins. In addition, the number of processors is much larger on our platform, and worst-case lock contention is hence much higher. This is supported by our data that shows that scheduling overheads are much lower in configurations with smaller cluster sizes.

In schedulability experiments assuming fixed WSSs of 4 KB and 128 KB, Calandrino *et al*. found P-EDF to be preferable to G-EDF in the HRT case, which matches our results. However, when testing heavy utilization distributions, Calandrino *et al*. found PD^2 to outperform P-EDF by a large margin, contrary to our results. We believe this discrepancy can be explained by the surprisingly low scheduling overhead on Calandrino *et al*.'s platform, by the use of first-fit decreasing as the partitioning heuristic under P-EDF, and by the fact that release interrupts were not accounted for in Calandrino *et al*.'s study. In the SRT case, Calandrino *et al*. found G-EDF to outperform both P-EDF and PD^2 (with either aligned or staggered quanta) in terms of schedulability in all tested scenarios. This partially matches our results. They further found PD^2 to outperform P-EDF in the presence of high-utilization tasks, again contrary to our results. We attribute this discrepancy with respect to our results to the differences in the underlying platform, overhead accounting, and overhead magnitudes. In addition, the PD^2 plugin implemented in current versions of LITMUS$^{\mathrm{RT}}$ is

much more robust (and likely incurs higher locking overheads because of it) than the version that was implemented in the first LITMUS$^{\mathrm{RT}}$ prototype used in Calandrino *et al.*'s study.

Brandenburg *et al.* (2008). The study reported in Brandenburg *et al.* (2008) is based on LITMUS$^{\mathrm{RT}}$ 2008.1; the evaluated plugins thus correspond much more closely to the current LITMUS$^{\mathrm{RT}}$ implementation. The primary research objective was to investigate the scalability of P-EDF, C-EDF, G-EDF, and PD^2 in terms of HRT and SRT schedulability on a "large" multicore platform, which is similar to setup of the study presented herein. However, a radically different hardware platform was used for the study in (Brandenburg *et al.*, 2008).

Given that commercially-available x86 systems (or embedded platforms) did not feature more than 2–4 cores at the time, a since-discontinued 32-processor Sun UltraSPARC T1 ("Niagara") was chosen as the underlying hardware platform. The Niagara is an 8-core, single-chip multiprocessor with four hardware threads per core (for a total of 32 virtual processors that must be scheduled by the RTOS). In contrast to Intel's x86 architecture, the Niagara implements a RISC-like instruction set. Each core is furthermore very simple in structure and does not implement branch prediction or speculative out-of-order execution. Each core is clocked at 1.2 GHz, which yields an effective speed of 300 MHz for each hardware thread (cycles are distributed among hardware threads in a round robin fashion). In comparison to modern x86 chips, the sequential execution speed of a Niagara core is hence very slow. Additionally, the memory hierarchy is impoverished in comparison to our x86-based platform: on the Niagara, all 32 hardware threads share a single 3 MB L2 cache; each core contains (*i.e.*, four hardware threads share) a 16 KB L1 instruction cache and an 8 KB L1 data cache.

Due to the large gap in hardware capability between the Niagara and the hardware platform used in this dissertation, there is little value in direct overhead comparisons. Further, a simpler, less-accurate outlier removal technique was used in (Brandenburg *et al.*, 2008), where simply the top 1% of the recorded samples was discarded. Nonetheless, it is interesting to note that average-case and worst-case CPMD values, which were determined using a precursor of the scheduler-sensitive method for a fixed WSS of 64 KB and a write ratio of 25%, are roughly comparable in magnitude to idle and load CPMD on our platform. As previously reported by Calandrino *et al.* (2006), observed average-case and worst-case CPMD under staggered quanta were lower than under aligned quanta on the Niagara as well (Brandenburg *et al.*, 2008).

The schedulability experiments were similar to those discussed in this dissertation. HRT and SRT schedulability under each of the schedulers were determined after inflating for worst-case and average-case overheads (respectively). As in this dissertation, task sets were partitioned using the worst-fit decreasing heuristic.[11] Six of 27 parameter distributions considered in this dissertation were used in (Brandenburg *et al.*, 2008), namely the light, medium, and heavy uniform and bimodal utilization distributions with moderate periods. As noted above, a single, fixed WSS of 64 KB was assumed, and only global interrupt handling (and not dedicated interrupt handling) was implemented in the underlying LITMUS$^{\mathrm{RT}}$ version.

The observed trends are in large parts consistent with our study, despite the differences in platform. In the HRT case, P-EDF was the best-performing scheduler in all scenarios except for the case of uniform heavy utilizations (discussed below), G-EDF and PD^2 with aligned quanta were never competitive in the HRT case, and C-EDF (with a cluster size of $c = 4$ based on shared L1 caches) performed better than G-EDF, but not as well as P-EDF. In the SRT case, either C-EDF or P-EDF were the best-performing schedulers for five of the six tested parameter distributions. As an exception, two G-EDF variants (preemptive and non-preemptive) were the best-performing SRT schedulers in the case of uniform heavy utilizations (due to bin-packing issues, just as in our case study—see Figure 4.26).

A notable difference between the schedulability results reported in (Brandenburg *et al.*, 2008) and those reported in this dissertation is that PD^2 with staggered quanta was found to be competitive with event-driven plugins both in the HRT and in the SRT cases in (Brandenburg *et al.*, 2008). In the HRT case, PD^2 with staggered quanta was even found to be preferable to P-EDF in the case of uniform heavy utilizations. In contrast, in the study presented herein, the performance gap between PD^2-variants and event-driven schedulers is significantly larger for most considered WSSs. There are several reasons for this. First, the PD^2 implementation used in (Brandenburg *et al.*, 2008) only supported periodic tasks and not sporadic tasks, and hence was not affected by release interrupts (recall Section 3.5.3). Second, discarding the top 1% of the recorded samples may have caused worst-case scheduling and tick overheads under staggered quanta to be underestimated. Recall that we found worst-case tick overheads not to be improved by quantum staggering, in contrast

[11] Due to an editing mistake, the employed bin-packing heuristic is erroneously identified as the "first-fit decreasing" heuristic in (Brandenburg *et al.*, 2008). In fact, the worst-fit decreasing heuristic was used instead.

to average-case overheads—see Figure 4.9. Since the distribution of overhead samples is skewed under staggered quanta, unconditionally removing the top 1% of samples may have significantly affected the assumed maximum. In the study presented in this dissertation, we did not remove any outliers from timer tick overhead data sets, and removed any outliers in scheduling overhead data sets manually to avoid truncating long-tail distributions. Third, CPMD was recorded without cache-polluting background processes; as a result the experimental setup used in (Brandenburg *et al.*, 2008) effectively employed idle CPMD for staggered quanta and load CPMD for aligned quanta. Fourth, due to an unfortunate scripting error, some task sets were erroneously claimed schedulable under PD^2 even if overhead accounting caused some task(s) to become infeasible, *i.e.*, if $e'_i > p'_i$ for some T_i. Fifth, the fixed WSS of 64 KB assumed in (Brandenburg *et al.*, 2008) is small compared to the range of WSSs assumed in this dissertation. For example, in Figure 4.29, G-sPD^2-R1 performs well for $wss = 64$ KB. That is, similarly to the trends reported in (Brandenburg *et al.*, 2008), there exist scenarios in our case study in which PD^2 is competitive with (but not preferable to) the best-performing event-driven plugin for small WSSs.

Brandenburg and Anderson (2009a). In a follow-up study carried out on the same Niagara machine and using the same evaluation methodology, nine different implementations of G-EDF were evaluated in terms of HRT and SRT schedulability to study implementation tradeoffs in global JLFP schedulers. The schedulability experiments used the same utilization distributions (light, medium, and heavy uniform and bimodal utilizations with moderate periods).

There is little overlap between the study presented in (Brandenburg and Anderson, 2009a) and the one presented in this chapter, as we used the results from (Brandenburg and Anderson, 2009a) to guide the choice of G-EDF and C-EDF implementation considered herein. Based on LITMUS$^{\mathrm{RT}}$ 2008.3, which was the first version of LITMUS$^{\mathrm{RT}}$ to support dedicated interrupt handling, the study presented in (Brandenburg and Anderson, 2009a) contrasted dedicated and global interrupt handling, event-driven and quantum-driven JLFP scheduling, and three appraoches to implementing shared priority queues. The plugin configuration denoted G-EDF-R1 in this dissertation was consistently among the two best-performing tested configurations in (Brandenburg and Anderson, 2009a), where it is denoted CE1. Consistently with the earlier study, we found dedicated interrupt handling to be preferable to global interrupt handling under G-EDF on our platform.

Bastoni *et al*. (2010b). The latest, most-closely related schedulability study was presented in (Bastoni *et al*., 2010b). It is based on the same 24-core Xeon system underlying the case study presented in this chapter, and used a very similar evaluation methodology to contrast four EDF-based schedulers (G-EDF-Rm, C6-EDF-Rm, C2-EDF-Rm, and P-EDF-Rm) in terms of HRT and SRT schedulability, and in terms of HRT and SRT weighted schedulability.

The overheads used in (Bastoni *et al*., 2010b) differ slightly from those used in this dissertation because the implementation of LITMUS$^{\mathrm{RT}}$ was improved in the mean time and because outlier filtering was applied to a lesser extent in the later case study presented herein. Additionally, the version of LITMUS$^{\mathrm{RT}}$ used in (Bastoni *et al*., 2010b) did not support clusters of non-uniform size, which precluded dedicated interrupt handling from being evaluated.

The most significant difference between the study presented in (Bastoni *et al*., 2010b) and the one presented herein is that (Bastoni *et al*., 2010b) is based on a slightly different definition of weighted schedulability score. Whereas the weighted schedulability score is defined as a function of wss in this dissertation, it was defined as a function of Δ^{cpd} in (Bastoni *et al*., 2010b). This makes the resulting graphs in (Bastoni *et al*., 2010b) somewhat more expressive, but less straightforward to interpret. We therefore chose to use the wss-based definition in this dissertation, and plan to continue using the simpler-to-interpret definition in future work.

Given that the results reported in (Bastoni *et al*., 2010b) are based on the same hardware platform, a recent version of LITMUS$^{\mathrm{RT}}$, and a near-identical evaluation methodology, it is hardly surprising that the results reported in (Bastoni *et al*., 2010b) match the results discussed in the preceding sections (with regard to the evaluated EDF-based schedulers). In particular, P-EDF was found to be the best-performing HRT scheduler in both studies. Similarly, clustered scheduling was found to be effective in the SRT case, and particularly so in scenarios in which P-EDF suffers from bin-packing limitations. Since the study presented in (Bastoni *et al*., 2010b) considered neither non-EDF schedulers nor dedicated interrupt handling, the case study presented in this dissertation supersedes the earlier conference version.

4.6 Summary

We have proposed an overhead-aware evaluation methodology for the comparison of real-time schedulers. The approach, which we explain in detail in Section 4.1, consists of two phases. In the OS phase, millions of overhead samples are collected during the execution of synthetic benchmark tasks, which are then distilled into models of average- and worst-case overheads. Thereafter, in the analytical phase, these overhead models are integrated with the overhead accounting methods introduced in the previous chapter to reflect all overhead sources during schedulability analysis. Finally, millions of task sets are randomly generated according to various parameter distributions. The resulting fraction of HRT or SRT schedulable task sets, termed schedulability, is an empirical performance measure that reflects both algorithmic and overhead-related capacity loss.

To demonstrate the effectiveness of the proposed methodology, we conducted a large-scale case study that examined a total of 22 scheduler configuration in LITMUS$^{\mathrm{RT}}$ on a 24-core Intel Xeon platform. On our platform, we found that P-EDF performs best in the HRT case, whereas C-EDF and G-EDF are effective at overcoming bin-packing limitations in the SRT case. These general trends match in large parts those observed in earlier studies.

CHAPTER 5

REAL-TIME SPINLOCK PROTOCOLS*

A limitation of the case study presented in the previous chapter is that all tasks were assumed to be independent, that is, locking concerns were not considered. In a real system, some or all tasks likely share some state and devices (if not explicitly at the application level, then at least at the RTOS level). As discussed in Chapter 2, a common method for the sharing of resources at the application or kernel level is to employ a locking protocol. In a real-time system, the employed locking protocol must enable maximum delays that a job may incur while waiting for resources to be bounded. In this and the following chapter, we propose and analyze several multiprocessor locking protocols that satisfy this requirement. Thereafter, in Chapter 7, we report on a second case study that investigates the impact of locking protocols on schedulability under event-driven scheduling. Since PD^2-based schedulers did not exhibit favorable schedulability even in the absence of synchronization requirements, we focus on JLFP schedulers in the remainder of this dissertation.

Recall from Section 2.4.4 that there are two fundamental mechanisms to implement waiting in a multiprocessor: spinning and suspending. In principle, suspension-based locking protocols are preferable because waiting jobs waste processor cycles under spin-based locking protocols. In practice, spin-based protocols benefit from low overheads (compared to the cost of suspending and resuming tasks), so that spinning is in fact preferable if all critical sections are short (*i.e.*, if tasks use resources for at most a few microseconds—see Chapter 7). In this chapter, we consider non-preemptive mutex and RW spinlocks in detail; in Chapter 6, we investigate semaphore-based mutex, RW, and k-exclusion protocols.

---

* Contents of this chapter previously appeared in preliminary form in the following papers:
Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57; and
Brandenburg, B. and Anderson, J. (2010b). Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87.

Most relevant to our work is prior analysis of non-preemptive FIFO spinlocks under P-EDF and G-EDF scheduling (Gai *et al*., 2003; Devi *et al*., 2006), which is summarized in Section 2.4.4.1. In joint work with Block *et al*. (2007), we developed the *flexible multiprocessor locking protocol* (FMLP) for P-EDF and G-EDF scheduling, which derives its name from the fact that it can be used under either partitioned or global scheduling, and it is agnostic regarding whether waiting is implemented via spinning or suspending. To achieve the latter, the FMLP simply integrates non-preemptive spinlocks to support so-called *short resources*, and incorporates suspension-based protocols to support *long resources*. The terms "short" and "long" arise because (intuitively) spinning is appropriate for short critical sections, whereas long critical sections call for suspension-based protocols. Since the FMLP incorporates non-preemptive spinlocks, it generalizes both Gai *et al*.'s MSRP for P-EDF and Devi *et al*.'s protocol for G-EDF. In subsequent work (Brandenburg and Anderson, 2008b), we extended the FMLP to support P-FP scheduling and developed an improved "holistic blocking analysis" that yields less-pessimistic bounds if jobs request resources more than once (see Section 5.4).

The design and analysis of non-preemptive spinlocks is in large parts scheduler-agnostic (at least with respect to JLFP schedulers). In particular, the way that spinlocks are used in the FMLP is not protocol specific. To avoid confusion, we therefore use the name "FMLP" in the remainder of this dissertation to refer to Block *et al*.'s suspension-based protocols for long resources (discussed in Chapter 6), and consider non-preemptive spinlocks individually in this chapter.

Besides supporting short and long resources, the FMLP was also the first protocol to explicitly support nested accesses to global resources by means of "group locking," which we discuss next in Section 5.1. Thereafter, in Section 5.2, we present the first analysis of RW locks in the context of multiprocessor real-time systems. In particular, we discuss why prior RW locks are ill-suited to worst-case blocking analysis, and present a new type of RW lock based on "phase-fairness," which is a novel progress guarantee designed to overcome the shortcomings of prior RW locks. Generally speaking, spinlocks are quite sensitive to overheads since the associated critical sections are typically short and frequently executed. In Section 5.3, to demonstrate that phase-fairness is practical, we present three "efficient" implementations of phase-fair RW locks, for three notions of "efficiency." Finally, in Section 5.4, we derive detailed bounds on maximum s- and pi-blocking under non-preemptive mutex and RW spinlocks.

5.1 Group Locking

Lock-based synchronization can be either *fine-grained* or *coarse-grained*. Under fine-grained locking, each resource is protected by an individual, resource-specific lock. When a job requires more than one resource at a time, it issues nested resource requests and acquires locks as needed. In contrast, under coarse-grained locking, several resources are protected by the same lock and nested requests are entirely or, at least, mostly avoided. For example, prior to version 2.4, Linux famously used the "big kernel lock," commonly referred to as the BKL, to serialize all kernel activity.

Fine-grained locking is clearly preferable from a parallelism, and hence throughput, perspective: jobs only lock the resources that they actually access and other resources may be used concurrently. However, nested lock acquisitions cause two major problems with regard to temporal correctness: possible deadlock and transitive delays. On a uniprocessor, both can be *optimally* controlled with existing protocols: under the PCP, SRP, and NCP, deadlock is impossible and maximum pi-blocking is limited to the length of one outermost critical section, as discussed in Section 2.4.3.

On multiprocessors, however, it is much more complicated to prevent deadlock (without introducing crippling pessimism) because resources may be requested and used in parallel. Further, even if deadlock can be ruled out because all nesting is well-ordered (*e.g.*, by requiring that resources are only nested in order from lower to higher indexed-resources), it is very difficult to bound transitive delays that arise when resource-holding jobs wait to acquire additional resources. In fact, to the best of our knowledge, no pi-blocking analysis of fine-grained locking on multiprocessors has been published to date. As noted in Section 2.4, all prior multiprocessor locking protocols, including the MPCP, DPCP, MSRP, and Devi *et al.*'s protocol, simply disallow the nesting of global resources.

Group locking is a simple approach that avoids deadlock and transitive blocking while still allowing applications to issue nested resource requests. It works by automatically coalescing fine-grained resource requests into coarse-grained lock acquisitions. Under group locking, the set of all resources is partitioned into disjoint subsets called *resource groups*. Two resources belong to the same resource group if and only if there exists a task that (potentially) requires both resources simultaneously. Resources that are only accessed individually, *i.e.*, that are not involved in nesting, each form their own singleton resource group. Determining appropriate resource groups is equivalent

to identifying connected components in a graph where each vertex is a resource and two vertices are connected by an edge if the corresponding resources are potentially used together.

As implied by its name, resource groups form the fundamental unit of synchronization. Each resource group is protected by a corresponding *group lock*. Prior to accessing any of the resources assigned to a group G, a job J_i must first acquire G's group lock. While J_i holds G's group lock, it may access any of the resources in G, which allows programmers to write applications that contain aribtrarily nested resource requests. By the definition of resource groups, a job holds at most one group lock at any time. Deadlock is thus impossible.

While group locking was originally proposed as part of the FMLP (Block *et al.*, 2007), it is not tied to any particular locking protocol. Each group lock can be seen as a virtual resource that is never nested (with accordingly aggregated parameters $N_{i,q}$ and $L_{i,q}$—see Section 2.4). It is therefore possible to use group locking with any spin- or suspension-based locking protocol, thereby adding some degree of nesting suport to protocols that assume non-nested resource requests.

Group locking could be implemented semi-transparently to programmers in a library or as part of a real-time middleware framework. However, to be effective, resource groups must be determined before tasks (that share resources) commence execution. It is therefore necessary for all possible resource nesting to be declared *a priori*. While this may appear as a limitation on first sight, knowledge of all possible resource nesting is in fact necessary for any non-trivial blocking analysis.

Group locking is admittedly a *very* simple deadlock avoidance mechanism that can be detrimental to throughput. Nonetheless, it is the first, and so far only, approach to multiprocessor real-time locking that allows nesting of global resources at all. Obtaining a *provably* better concurrency control method (in terms of worst-case blocking of outermost requests) remains an interesting open question.

In protocols that differentiate between several types of requests, it may further be necessary to impose additional nesting requirements. For example, in the case of resources protected by RW locks, which we discuss next, read requests may be nested within write requests (since a write requires exclusive access anyway), but write requests may not be nested within read requests (since this would require a "privilege upgrade" that is difficult to support).

To avoid notational complexity, we assume in the remainder of this chapter and Chapter 6 that all nesting has been dealt with by defining appropriate resource groups. That is, the presented blocking analysis should be applied to group locks if nesting is allowed, and not to individual resources.

5.2 Reader-Writer Spinlocks

The need for RW exclusion arises naturally in many situations where information is more frequently used than created or updated. For example, consider a robot such as TU Berlin's autonomous helicopter Marvin (Musial *et al*., 2006): its GPS receiver updates the current position estimate 20 times per second, and the latest position estimate is read at various rates by a flight controller and by image acquisition, camera targeting, and communication modules. While consumers of data must be protected against concurrent state updates, requiring mutual exclusion of consumers would be unnecessarily restrictive.

Another common use case of RW locks is rarely-changing, frequently-used shared data such as a system's current configuration. A typical example was reported by Gore *et al*. (2004), who employed RW locks to reduce latency in a real-time notification service. In their system, every incoming message must be matched against a shared subscription lookup table to determine the set of subscribing clients. Since events occur very frequently and changes to the table occur only very rarely, the use of a regular mutex lock "can unnecessarily reduce [concurrency] in the critical path of event propagation" (Gore *et al*., 2004). In practice, RW locks are in widespread use since they are supported by all POSIX-compliant RTOSs (IEEE, 2008b).

In throughput-oriented computing, RW locks are attractive because they increase average concurrency (compared to mutex locks) if read requests are more frequent than write requests. In a real-time context, RW locks should also lower blocking (either s-blocking or pi-blocking, depending on the protocol type) for readers, that is, the higher degree of concurrency must be reflected in *a priori* worst-case analysis and not just in observed average-case delays.[1] Unfortunately, many RW lock types commonly in use in throughput-oriented systems provide only little analytical benefits because they either allow starvation or needlessly serialize readers.

5.2.1 Request Order

Besides RW locks without strong progress guarantees (such as those implemented in the Linux kernel), work on RW locking has produced four types of RW locks. Courtois *et al*. (1971) were

[1]This requirement applies equally to suspension-based and spin-based locking protocols; the discussion in this chapter focuses on spin-based RW locking protocols, but it also applies to suspension-based RW locking protocols such as the one presented in the next chapter (Section 6.2.3).

the first to investigate RW synchronization and proposed two semaphore-based RW locks: a *writer preference lock*, wherein writers have higher priority than readers, and a *reader preference lock*, wherein readers have higher priority than writers. In later work on scalable synchronization on shared-memory multiprocessors, Mellor-Crummey and Scott (1991b) presented efficient spin-based implementations of reader preference, writer preference, and *task-fair* RW locks, in which readers and writers gain access in strict FIFO order. In recent work, we proposed *phase-fair* RW locks as an alternative that is better suited to worst-case analysis (Brandenburg and Anderson, 2009b, 2010b). Phase-fairness is also based on FIFO ordering, but under it the FIFO constraint is relaxed to apply to groups of jobs (instead of individual jobs) to increase reader parallelism. To highlight the difference between strict FIFO locks and phase-fair RW locks, we use "task-fair" as a synonym for "strict FIFO."

In the following, we illustrate the advantages and disadvantages of each type of lock in the context of multiprocessor real-time systems, assuming that requests are executed non-preemptively. Formal bounds on worst-case blocking are derived subsequently in Section 5.4. We begin with regular (*i.e.*, non-RW) spinlocks as a baseline.

5.2.1.1 Task-Fair Mutex Locks

Recall from Section 2.4.4.1 that in task-fair (or FIFO) mutex locks, competing jobs are served strictly in the order that they issue requests, and that a resource is never held by more than one job at a time. Even though they are intuitively undesirable for RW synchronization, task-fair mutex locks are considered here because they are the only spinlocks for which bounds on worst-case s-blocking were derived prior to our work (Brandenburg and Anderson, 2009b, 2010b). Further, mutex locks may in fact be preferable to RW locks if writes are frequent because mutex spinlocks are simpler in structure and thus incur lower overheads than RW locks (with progress guarantees). That is, if write requests are more frequent than read requests, then lower acquisition costs under mutex locks may outweigh the benefits of potential reader parallelism under RW locks. Hence, we use task-fair mutex locks as a performance baseline in our experiments (see Chapter 7). A major disadvantage of task-fair mutex locks is that strict mutual exclusion among readers is unnecessarily restrictive and can cause deadline misses.

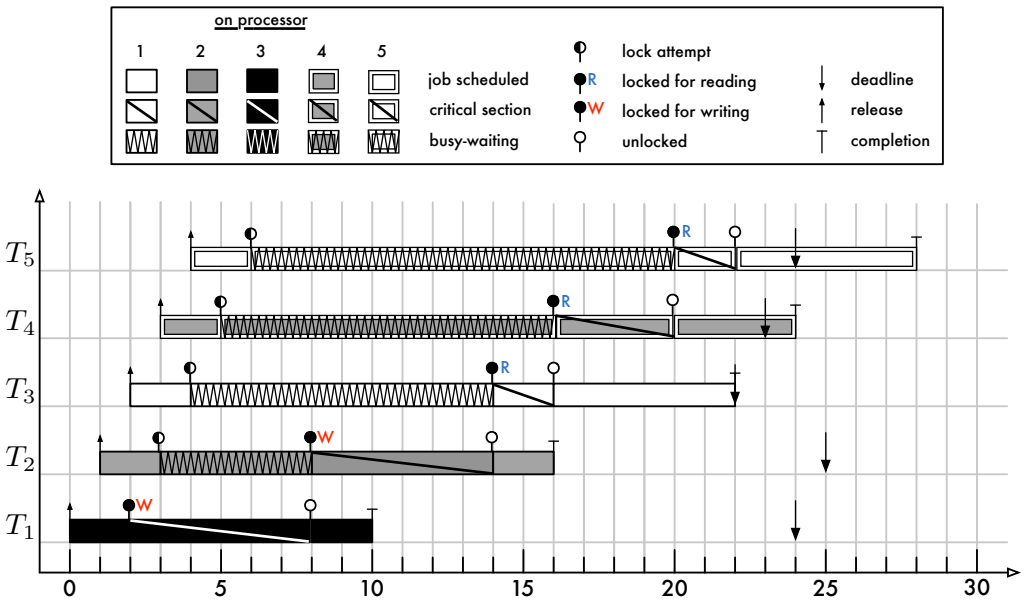


Figure 5.1: Example schedules of two writers (J_1, J_2) and three readers (J_3, J_4, J_5) sharing a resource ℓ_1 that is protected by a task-fair mutex lock. Jobs J_4 and J_5 miss their respective deadlines at times 23 and 24 due to the unnecessary sequencing of readers.

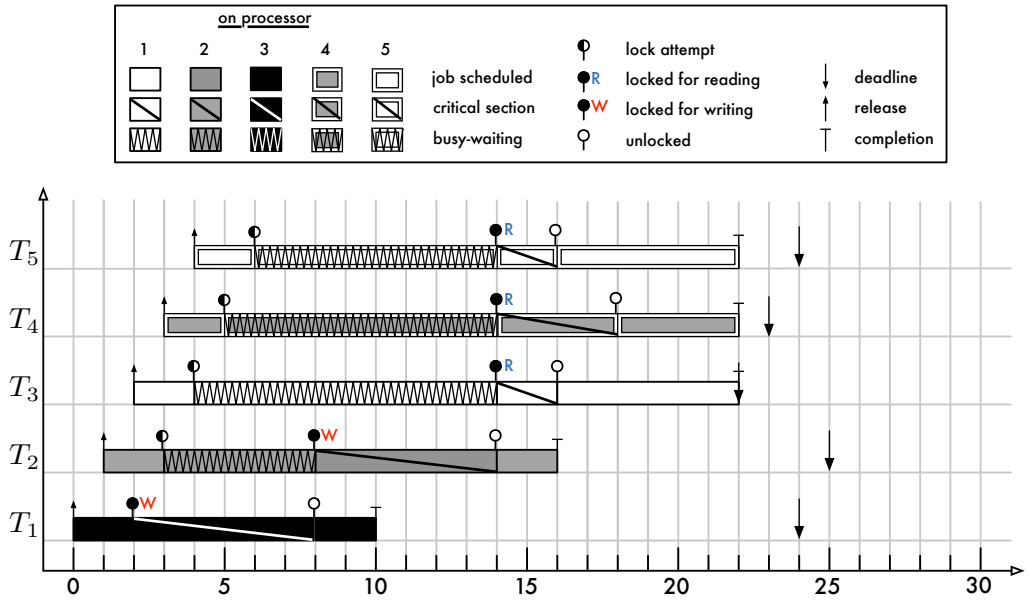
Example 5.1. An example is shown in Figure 5.1, which depicts jobs of five tasks (two writers, three readers) competing for a resource ℓ_1. As ℓ_1 is protected by a task-fair mutex lock, all requests are satisfied sequentially in the order that they are issued. This unnecessarily delays both J_4 and J_5 and causes them to miss their respective deadlines at times 23 and 24. ◊

5.2.1.2 Task-Fair RW Locks

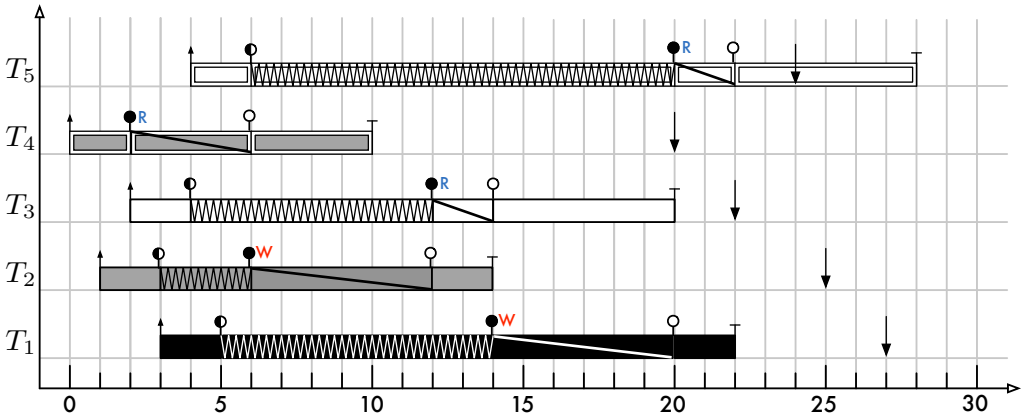
With task-fair RW locks, conflicting requests are still satisfied in strict FIFO order, but the mutual exclusion requirement is relaxed such that resources can be held concurrently by *consecutive* readers (since readers do not conflict with each other). This can reduce delays significantly if read requests are not separated by interspersed write requests.

Example 5.2. The benefit of allowing reader parallelism is apparent in Figure 5.2(a), which depicts the same arrival sequence as Figure 5.1. Note that the read requests of J_3, J_4, and J_5 are satisfied simultaneously at time 14, which in turn allows J_4 and J_5 to meet their deadlines.

Unfortunately, task-fair RW locks may degrade to mutex-like performance when faced with a pathological request sequence, as is shown in Figure 5.2(b). The only difference in the scenarios shown in insets (a) and (b) is that the arrival times of J_1 and J_4 have been switched. This causes ℓ_1 to



(a) Parallel reads.



(b) Sequential reads.

Figure 5.2: Example of reader parallelism (and lack thereof) under task-fair RW locks. **(a)** Given the arrival sequence depicted in Figure 5.1, all readers gain access in parallel and hence meet their respective deadlines. **(b)** If the arrival times of J_1 and J_4 in inset (a) are switched, then all readers are serialized and a deadline is missed. This demonstrates that task-fair RW locks are subject to mutex-like worst-case performance in the presence of multiple writers.

be requested first by a reader (J_4 at time 2), then by a writer (J_2 at time 3), then by a reader again (J_3 at time 4), then by another writer (J_1 at time 5), and finally by the last reader (J_5 at time 6). Reader parallelism is eliminated in this scenario and J_5 misses its deadline at time 24 as a result. ◊

5.2.1.3 Preference RW Locks

In a reader preference lock, readers are statically prioritized over writers, that is, writers are starved as long as there are unsatisfied read requests (Courtois *et al.*, 1971; Mellor-Crummey and Scott, 1991b). The lack of strong progress guarantees for writers makes reader preference locks a problematic choice for real-time systems because deadlines can be missed due to the resulting starvation.

The POSIX standard (IEEE, 2008b) requires writer preference locks with respect to their assigned priorities (*i.e.*, writers always have preference over readers of equal or lower priority) in RTOSs that support the "Thread Execution Scheduling" option, due to the possibility of writer starvation.[2] Since read requests are presumably more frequent than write requests when RW locks are employed, reader starvation under writer preference locks is intuitively less likely to occur than writer starvation under reader preference locks. However, in the worst case, writer preference locks are still subject to starvation and thus an ill choice for predictable real-time systems that require *a priori* analysis.

Example 5.3. This is illustrated in Figure 5.3, which depicts the same arrival sequence as Figure 5.2(b) but assumes that ℓ_1 is protected by a writer preference lock. Again, ℓ_1 is first acquired by a reader (J_4 at time 2) as there are initially no competing write requests. However, all reads are blocked as soon as writers are present (starting at time 3). Hence, J_3 and J_5's read requests remain unsatisfied until both J_1 and J_2's write requests have completed at time 18, which causes each reader to miss its respective deadline. ◊

5.2.1.4 Phase-Fair RW Locks

All previously-proposed RW spinlocks fall within one of the categories discussed so far—task-fair, reader preference, and writer preference locks—or offer no progress guarantees at all. Thus, the discussed examples above demonstrate that *no* such lock reliably reduces worst-case blocking (unless

[2] In the POSIX standard, there are two levels of compliance with regard to suspension-based RW locks. In the base profile, the ordering of conflicting RW lock requests is left unspecified and thus implementation-defined (IEEE, 2008b). Writer preference locks are mandated for RTOSs that support priority-based process scheduling (*i.e.*, SCHED_RR and SCHED_FIFO). For example, QNX implements writer preference locks for this reason (QNX Software Systems, 2011).

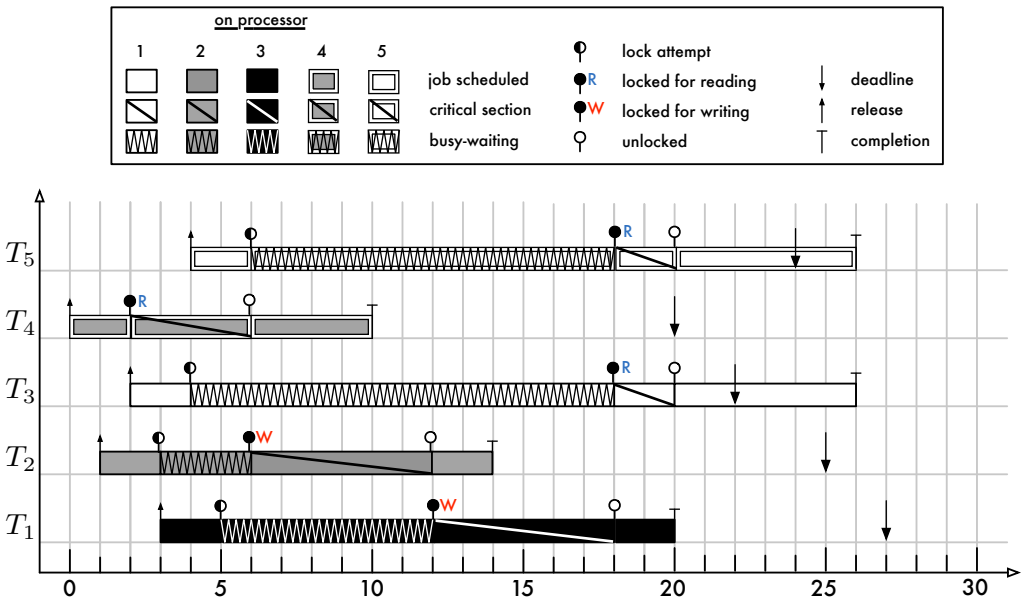


Figure 5.3: Example of reader delay under writer preference locks. Given the arrival sequence from Figure 5.2(b), two readers miss their respective deadlines (instead of only one as in inset (b) of Figure 5.2) because all read requests remain unsatisfied while writers are present, which can significantly delay readers.

writers execute only infrequently, which is a problematic assumption to make in the context of worst-case analysis).

The previous examples reveal two root problems of existing RW locks: first, preference locks cause extended blocking due to intervals in which only requests of one kind are satisfied; and second, task-fair RW locks cause extended blocking due to a lack of parallelism when readers are interleaved with writers. A RW lock better suited for use in predictable real-time systems should avoid these two pitfalls. This discussion motivates the concept of phase-fairness.

Definition 5.1. With respect to a shared resource ℓ_q, a *reader phase* consists of one or more readers (and no writers) acquiring ℓ_q (possibly in parallel). A *writer phase* consists of exactly one writer acquiring ℓ_q exclusively. A RW lock is *phase-fair* if and only if it has the following properties:

PF1 reader phases and writer phases alternate;

PF2 writers are subject to FIFO ordering, but only with regard to other writers;

PF3 at the start of each reader phase, all currently-unsatisfied read requests are satisfied (exactly one write request is satisfied at the start of a writer phase); and

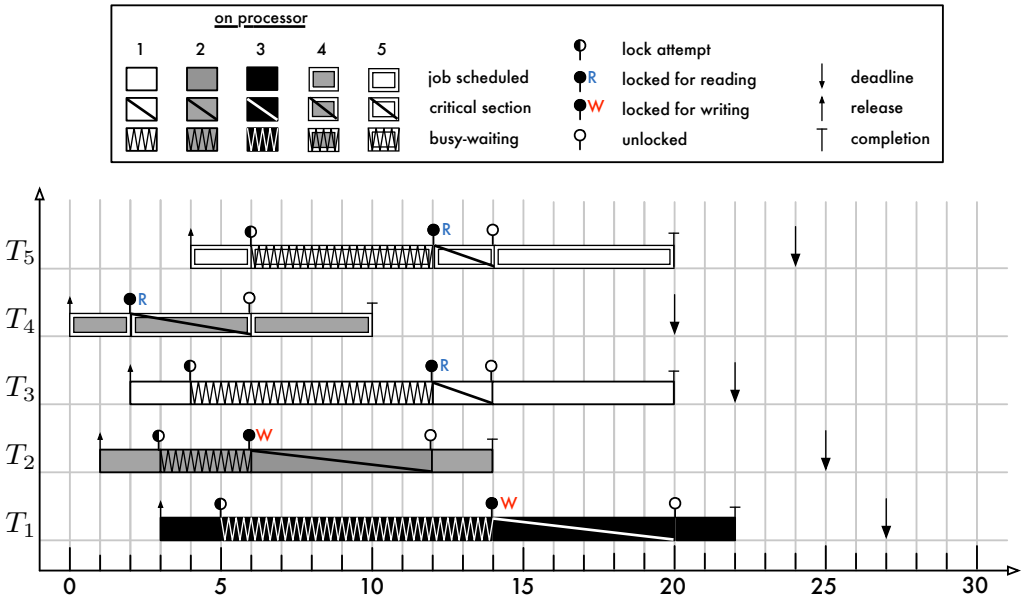


Figure 5.4: Example of reduced blocking under phase-fair locks. All readers meet their respective deadlines because they are not blocked by more than one write request (due to Property PF1). Note how the contending readers (J_3, J_4) yield to a writer (J_2) at time 4, which is followed in turn by a writer (J_1) yielding to the aforementioned readers: this exemplifies the "after-you politeness" implicitly required by the definition of phase-fairness.

PF4 during a reader phase, newly-issued read requests are satisfied only if there are no unsatisfied write requests pending.

Properties PF1 and PF3 ensure that a read request is never blocked by more than one writer phase and one reader phase *irrespective of the number of processors and the length of the write request queue*. Property PF4 ensures that reader phases end. In a non-preemptive phase-fair RW spinlock, Properties PF1 and PF2 ensure that a write request is never blocked by more than $m - 1$ reader and writer phases each (see Section 5.2.2 below). In some sense, phase-fair locks can be understood as being a reader preference lock when held by a writer, and being a writer preference lock when held by readers, that is, readers and writers are "polite" and yield to each other.

Example 5.4. Figure 5.4 depicts a schedule for the pathological arrival sequence from Figures 5.2(b) and 5.3 assuming a phase-fair RW lock. J_4 issues the first read request and thus starts a new reader phase at time 2. J_2 issues a write request that cannot be satisfied immediately at time 3. However, J_2's unsatisfied request prevents the next read request (issued by J_3 at time 4) from being satisfied due to Property PF4. At time 5, T_1 issues a second write request, and at time 6, J_5 issues a final

read request. At the same time, J_4's request completes and the first reader phase ends. The first writer phase lasts from time 6 until time 12 when J_2's write request completes. Due to Property PF1, this starts the next reader phase, and due to Property PF3, *all* unsatisfied read requests are satisfied. Note that, when J_5's read request was issued, two write requests were unsatisfied. However, due to the phase-fair bound on read-request acquisition delay, it was only blocked by one writer phase regardless of the arrival pattern. This allows all jobs to meet their deadlines in this example, and, in fact, for any arrival pattern of the jobs depicted in Figures 5.1–5.4. ◊

5.2.2 Blocking Overview

The preceding examples suggest that phase-fair locks may be a desirable choice for real-time RW synchronization. However, in order to employ RW spinlocks in real-time systems (and to properly compare locking choices), formal bounds on worst-case s-blocking are required. The derivation of reasonably-tight bounds that are sufficiently flexible to allow the analysis of non-trivial task systems incurs some inherent complexity. In Section 5.4, we present such analysis in full technical detail. For the casual reader, we next provide a high-level overview of the blocking properties of the discussed RW locking choices that is sufficient to convey the "big picture." Note, however, that the following overview is not a substitute for the complete analysis presented in Section 5.4 and should not be used for schedulability analysis—it merely highlights simplified instantiations of the derived bounds for particular scenarios to convey an intuitive understanding.

Recall from Section 2.4 that there are two kinds of blocking that must be considered: s-blocking, which a job J_i incurs while it spins non-preemptively, and pi-blocking, which J_i may incur if it is released with a sufficient priority to be scheduled immediately, but cannot be scheduled because another job is executing non-preemptively on J_i's assigned processor. Since jobs are only non-preemptable while spinning or executing a request, a bound on per-request s-blocking implicitly bounds maximum pi-blocking.

Table 5.1 depicts bounds on worst-case blocking under three scenarios, expressed in terms of the number of blocking phases. Let J_i denote the job under consideration.

The *Single Write* scenario assumes that J_i issues exactly one write and no read requests. Since writers require exclusive access, the task-fair mutex, task-fair RW, and writer preference RW bounds reflect the fact that a conflicting write request may be executed on every other processor. Under

Request order	Exclusion	Single write	Single read	Repeated reads
task-fair	mutex	$m-1$	$m-1$	$W+R$
task-fair	RW	$m-1$	$m-1$	$2W$
writer preference	RW	$m-1$	∞	$W+R$
reader preference	RW	∞	1	W
phase-fair	RW	$2 \cdot (m-1)$	2	$2W$

Table 5.1: A comparison of the bounds on worst-case s-blocking in three different scenarios under each of the considered spinlock types. The third column (*Single Write*) lists the maximum number of phases (either read or write) that block a job if it issues exactly one write and no read requests. The fourth column (*Single Read*) lists the maximum number of phases (either read or write) that block a job if it issues exactly one read and no write requests. The fifth column (*Repeated Reads*) lists the maximum number of phases that block a job that repeatedly issues a read requests during some interval $[t_0, t_1)$, where W denotes the number of conflicting write requests and R denotes the number of conflicting read requests issued during $[t_0, t_1)$ (assuming that $2W < R$).

reader preference locks, write requests are subject to starvation and it is thus not possible to bound the number of blocking phases based purely on J_i's requests—J_i will be blocked as long as other jobs keep issuing read requests. Finally, the phase-fair bound reveals a disadvantage of enforcing alternating reader and writer phases: while J_i is blocked by at most $m-1$ writer phases, it can also be *transitively* blocked by $m-1$ interspersed reader phases.

The *Single Read* scenario assumes the complimentary situation, that is, J_i issues one read and no write request. This highlights the main limitation of task-fair locks—under both task-fair mutex and RW locks, read requests may be delayed significantly while progressing through the FIFO queue. Starvation is again an issue under preference locks. However, the situation is reversed: writer preference locks allow reads to be delayed indefinitely, whereas reader preference locks offer the lowest bound of all locks for reader blocking—at most one writer phase can block J_i if said writer phase was active when J_i issued its read request. Phase-fair RW locks offer a close approximation of reader preference locks under this scenario, since satisfying all blocked read requests at the beginning of each reader phase (Property PF3) ensures that at most two phases block a read request. This shows that forcing phases to alternate (Property PF1) benefits readers at the expense of writers. Since reads are expected to occur (much) more frequently than writes when using RW locks in the first place, this is presumably a beneficial tradeoff. We explore this issue in our experiments in Chapter 7.

The third scenario, *Repeated Reads*, looks at cumulative blocking across multiple read requests over some interval $[t_0, t_1)$ in which J_i issues many read requests in quick succession.[3] With regard to this interval, W denotes the total number of potentially-blocking write requests (*i.e.* requests issued by jobs other than J_i) and R denotes the total number of potentially-blocking read requests. Further, it is assumed that $2W < R$, that is, that reads are "much more frequent" than writes. This scenario emphasizes why mutex locks are an undesirable choice for RW synchronization: in the worst case, J_i is delayed by every other request. In contrast, under task-fair RW locks, J_i is only blocked by at most twice the number of potentially-blocking write requests. Intuitively, this is because one of J_i's read requests is only blocked by a reader phase if they are "separated" by a writer phase. Writer preference locks suffer from the same shortcoming as mutex locks—in the worst case, J_i is blocked by all other requests. Under reader preference locks, read requests are never blocked by reader phases—thus, at most W requests can block J_i in this case. Finally, phase-fair RW locks maintain the desirable bound of task-fair RW locks, since, under phase-fairness, reader phases only block read requests if a writer phase is blocking, too.

These comparisons substantiate the observations of Section 5.2.1. To summarize,

- task-fair mutex and task-fair RW locks have similar worst-case blocking behavior for individual requests;
- preference locks offer appealing worst-case behavior only for the prioritized request type and give rise to starvation; and
- phase-fair RW locks strike a compromise between the desirable properties of reader preference and task-fair RW locks: reads are only blocked by a constant number of phases and writes are not starved, albeit at the cost of a factor of two in each bound (compared to the best bound in each scenario).

Mellor-Crummey and Scott presented several efficient implementations of reader preference, writer preference, and task-fair RW and mutex locks for shared-memory multiprocessors (Mellor-Crummey and Scott, 1991a,b). We next show that phase-fair RW locks can be implemented efficiently as well.

[3] J_i is assumed to issue sufficiently many read requests to be blocked by all requests of other tasks that can block J_i, that is, per-request bounds are assumed to not affect the outcome.

5.3 Efficient Phase-Fair Reader-Writer Spinlocks

To be a viable choice, phase-fair RW locks must be efficiently implementable on common hardware platforms. In practice, however, appropriate definitions of "efficient" may vary widely between applications. One commonly-used complexity metric for locking algorithms is to count *remote memory references* (RMR) (Anderson *et al.*, 2003). In a distributed-memory system, remote memory references occur when a processor accesses a non-local memory. On a cache-coherent shared-memory multiprocessor, a remote memory reference is equivalent to a forced cache invalidation. That is, when using RMR complexity as a measure of implementation efficiency, locks are classified by how many cache invalidations occur per request under maximum contention. The assumption underlying RMR complexity is that cache-local operations are (uniformly) cheap and that therefore the number of (uniformly) expensive cache invalidations dominate lock acquisition costs.

Cache consistency traffic can certainly severely limit performance, but, in practice, lock efficiency also strongly depends on the number and cost of the required atomic operations. For example, on a given hypothetical platform, a lock implementation that requires multiple "light-weight" atomic-add instructions may outperform alternate implementations that rely on fewer "heavy-weight" compare-and-swap instructions. Such tradeoffs are strongly hardware dependent and can only be resolved by benchmarking actual lock performance on the platform(s) of interest; we explore this issue in further detail in Chapter 7.

A third efficiency criterion arises in the design of memory-constrained embedded systems, wherein lock size concerns may outrank the desire for low acquisition overheads. In systems with many shared data objects, lock size can especially become a concern when locks are used to protect individual data objects (as opposed to locking code paths), which is often desirable if high throughput is required since it can yield increased parallelism.

In this section, we present three phase-fair RW lock implementations, each addressing one of the aforementioned efficiency requirements:

- a simple-to-implement *ticket-based* phase-fair RW lock, denoted PF-T, that only depends on hardware support for atomic-add, fetch-and-add, and atomic stores, and that requires only two atomic operations[4] each on the reader and writer path;
- a *compact* version of the first algorithm for memory-constrained systems, denoted PF-C, that requires only four instead of 16 bytes per lock, but at the price of requiring additional atomic operations; and
- a *queue-based* implementation, denoted PF-Q, that requires only a constant number of remote memory references.

These algorithms heavily reuse the classic *ticket* (Lamport, 1974) and *list-based queue lock* techniques (Mellor-Crummey and Scott, 1991a,b). We assume familiarity with spinlock algorithms on behalf of the reader—see Anderson *et al.*'s survey for an introduction (Anderson *et al.*, 2003).

5.3.1 A Simple Phase-Fair Reader-Writer Spinlock

The PF-T algorithm, as given in its entirety in Listing 5.1, assumes a 32-bit little-endian architecture. However, it can be easily adapted to other native word sizes and big-endian architectures. We next discuss its structure and the entry and exit procedures, which are both illustrated in Figure 5.5.

Structure. A PF-T lock consists of four counters that keep track of the number of issued (rin, win) and completed ($rout$, $wout$) read and write requests (lines 1–3 of Listing 5.1). Variable rin serves multiple purposes: bits 8–31 are used for counting issued read requests, while bit 1 ($PRES$) is used to signal the presence of unsatisfied write requests and bit 0 ($PHID$) is used to tell consecutive writer phases apart. For efficiency reasons (explained below), bits 2–7 remain unused, as do bits 0–7 of $rout$ for reasons of symmetry. The allocation of bits in rin and $rout$ is illustrated in Figure 5.6.

Readers. The reader entry procedure (lines 10–12, illustrated in Figure 5.5) works as follows. First, a reader atomically increments rin and observes $PRES$ and $PHID$ (line 11). Observing $PRES$ is required to detect the presence of a writer; observing $PHID$ is required to avoid a potential race with multiple writers (discussed below). If no writer is present ($w = 0$), then the reader is admitted immediately (line 12). Otherwise, the reader spins until either of the two writer bits in rin changes:

[4]Not counting atomic loads and stores, which are atomic by default on many current platforms (*e.g.*, Intel x86).

```
1  struct  pft_lock:
2      unsigned integer rin, rout     initially   0
3      unsigned integer win, wout     initially   0

5  let  RINC    =  0x100   // reader increment
6  let  WBITS   =  0x3     // writer bits in rin
7  let  PRES    =  0x2     // writer present bit
8  let  PHID    =  0x1     // phase id bit

10 read_lock(struct  pft_lock* lock):
11     let  w ← fetch_and_add(lock.rin, RINC) & WBITS
12     await  (w = 0) or (w ≠ lock.rin & WBITS)

14 read_unlock(struct  pft_lock* lock):
15     atomic_add(lock.rout, RINC)

17 write_lock (struct  pft_lock* lock):
18     let  ticket ← fetch_and_add(lock.win, 1)
19     await  ticket = lock.wout
20     let  w ← PRES + (ticket & PHID)
21     set  ticket ← fetch_and_add(lock.rin, w)
22     await  ticket = lock.rout

24 write_unlock(struct  pft_lock* lock):
25     let  lsb ← pointer to least-significant byte of lock.rin
26     set  *lsb ← 0
27     set  lock.wout ← lock.wout + 1
```

Listing 5.1: The ticket-based PF-T algorithm. This implementation assumes little-endian 32-bit words. A PF-T lock requires 16 bytes.

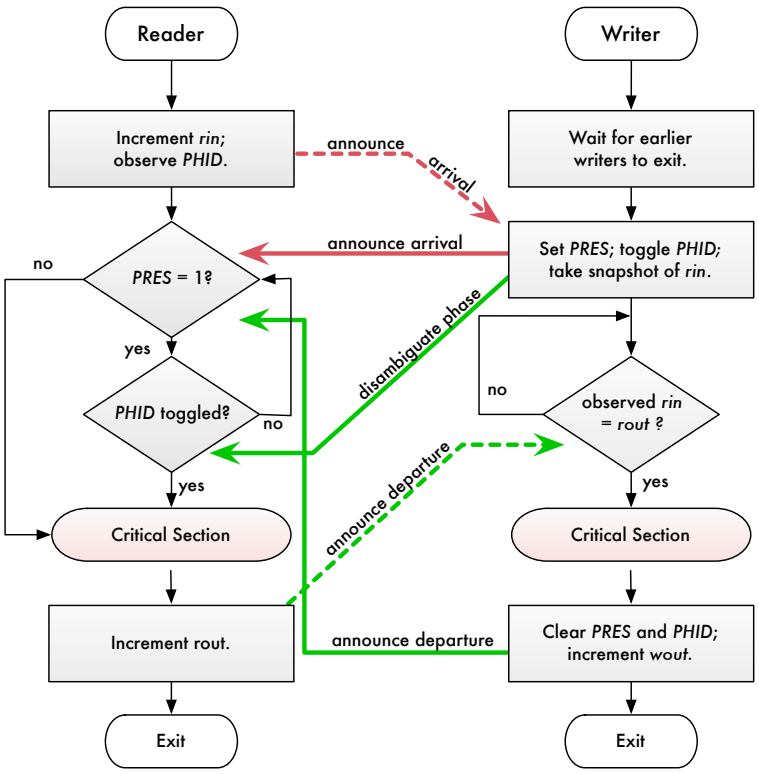


Figure 5.5: An illustration of the reader and writer control flow in the PF-T algorithm. Directed edges indicate control flow; bold arrows indicate communication, where solid arrows indicate information flow from writers to readers and dashed arrows indicate information flow from readers to writers. The linearization point for concurrent reader and writer arrivals is the update of *rin*—if the reader increments *rin* before the writer sets *PRES*, then the reader enters first, otherwise, if the writer first sets *PRES* before *rin* is incremented, then the writer precedes the reader. (Recall from Figure 5.6 that the *PRES* bit is part of the *rin* word.) Deadlock between a slow reader and a quickly-arriving writer is avoided by toggling *PHID* when setting *PRES*.

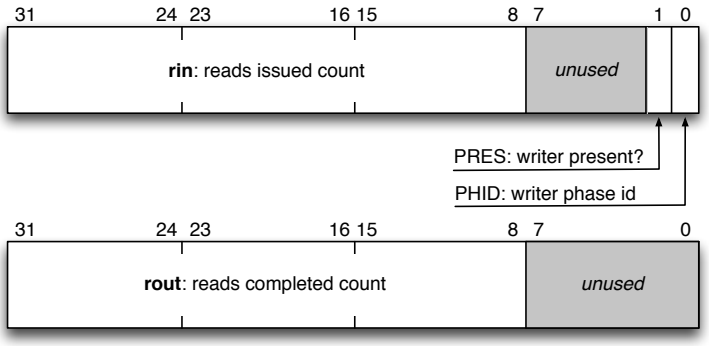


Figure 5.6: The allocation of bits in the reader entry counter and the reader exit counter of the PF-T algorithm (corresponding to line 2 of Listing 5.1).

if both bits are cleared, then no writer is present any longer. Otherwise—if only $PHID$ toggles but $PRES$ remains unchanged—the beginning of a reader phase has been signaled. The reader exit procedure (lines 14–15) only consists of atomically incrementing $rout$, which allows a blocked writer to detect when the lock ceases to be held by readers (as discussed below, line 22).

Writers. Similarly to Mellor-Crummey and Scott's simple task-fair RW lock (Mellor-Crummey and Scott, 1991b), FIFO ordering of writers is realized with a ticket abstraction (Lamport, 1974; Mellor-Crummey and Scott, 1991a). The writer entry procedure (lines 17–22) starts by incrementing win in line 18 and waiting for all prior writers to release the lock (line 19). Once a writer is the head of the writer queue ($ticket = wout$), it atomically sets $PRES$ to one, sets $PHID$ to equal the least-significant bit of its ticket, and observes the number of issued read requests (lines 20–21). Note that the least-significant byte of rin equals zero when observed in line 21 since no other writer can be present. Finally, the writer spins until all readers have released the lock before entering its critical section in line 22.

The writer exit procedure consists of two steps. First, the beginning of a reader phase is signaled by clearing bits 0–7 of rin by atomically writing zero to its least-significant byte (lines 25–26). Clearing the complete least-significant byte instead of just bits 0 and 1 is a performance optimization since writing a byte is usually much faster than atomic read-modify-write instructions on modern hardware architectures. Finally, the writer queue is updated by incrementing $wout$ in line 27.

Phase id bit. The purpose of $PHID$ is to avoid a potential race between a slow reader (J_r) and two writers (J_v, J_w). Assume that J_v holds the lock and that J_r and J_w are blocked. When J_v

releases the lock, $PRES$ is cleared (line 26) and $wout$ is incremented (line 27). Subsequently, J_w re-sets $PRES$ (line 21) and waits for J_r to release the lock. In the absence of $PHID$, J_r could fail to observe the short window during which $PRES$ is cleared and continue to spin in line 12, waiting for the $next$ writer phase to end. This deadlock between J_r and J_w is avoided by checking $PHID$: if J_r misses the window between writers, it can still reliably detect the beginning of a reader phase when $PHID$ is toggled.

Correctness. If there are at most $2^{32} - 1$ concurrent writers and at most $2^{24} - 1$ concurrent readers, then the PF-T lock ensures exclusion of readers and writers and mutual exclusion among writers.[5] Overflowing rin, win, $rout$, and $wout$ is harmless because the counters are only tested for equality: in order for two writers to enter their critical sections concurrently, they must obtain the same ticket value. This is only possible if win wrapped around, that is, if more than $2^{32} - 1$ writers draw a ticket in between times that $wout$ is incremented. Similarly, a writer can only enter its critical section during a reader phase if there are at least 2^{24} reads in progress at the time that the writer sets $PRES$. In order for a reader to enter during a writer phase, either $PHID$ would have to toggle or $PRES$ would have to clear, which is only possible if a writer becomes head of the writer queue while another writer is still executing its critical section—which, as argued above, is impossible.

Liveness is guaranteed because all readers will eventually observe that the $PHID$ bit was toggled, and because a blocked writer will eventually observe that $rout$ increased. PF-T locks are phase-fair because readers cannot enter while a writer is spinning (as $PRES$ is set in this case), and because writers unblock all waiting readers as part of both the writer exit procedure (by clearing $PRES$) and the writer entry procedure (by toggling $PHID$).

5.3.2 A Compact Phase-Fair Reader-Writer Spinlock

In Listing 5.1, rin, win, $rout$, and $wout$ are each defined as four-byte integers. However, requiring 16 bytes per lock may be excessive in the context of memory-constrained applications (*e.g.*, embedded systems). To accomodate such constraints, we next introduce the PF-C algorithm, which only requires 4 bytes and supports up to 127 concurrent readers and writers. The PF-C algorithm, which is given in its entirety in Listing 5.2, closely resembles the PF-T lock given in Listing 5.1; we focus

[5]Note that there can be at most m concurrent readers and writers if requests are executed non-preemptively.

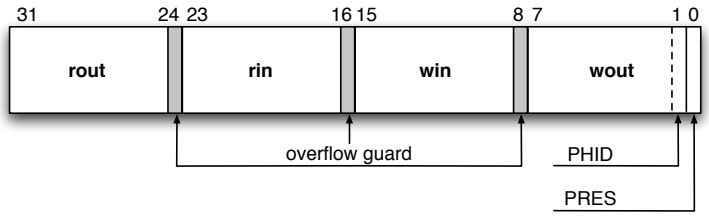


Figure 5.7: The allocation of bits in a PF-C lock (corresponding to lines 1–2 of Listing 5.2). The four ticket counters rin, $rout$, win, and $wout$ consist of only seven bits each and are separated by overflow guards, *i.e.*, bits that are otherwise unused and only employed to detect when each counter wraps. Due to the limited size of the ticket counters, at most 127 readers and 127 writers may issue requests concurrently. Note that $PHID$ overlaps with the least-significant bit of $wout$. ($PRES$ does not overlap with any counter.)

on the notable differences in the following discussion. Except for minor details, the illustration of the PF-T algorithm's control flow (Figure 5.5) applies to the PF-C algorithm as well.

Structure. The lock itself consists of only one 32-bit word (line 1). The four ticket counters rin, $rout$, win, and $wout$ are collapsed into four 7-bit-wide bit fields, as shown in Figure 5.7. The four counters are separated by one bit each that serves as an *overflow guard*. Initialized to zero, an overflow guard prevents an overflow in one counter from corrupting the adjacent counter—of course, this requires the overflow guard to be reset before a second overflow can occur (as discussed below).

The ticket counters are incremented with fetch-and-add, atomic-add, and atomic-sub[6] operations in which the second argument, *i.e.*, the value to be added/subtracted, has been shifted by the offset of the to-be-updated ticket counter. The respective offsets are given in lines 4-7 of Listing 5.2.

Similarly, the ticket counters are observed by reading the lock word and then shifting and masking the observed value. The bit mask $MASK$ (line 8 in Listing 5.2) corresponds to the width of one counter (seven bits).

Compared to the PF-T bit allocation, the positions of the phase id bit, $PHID$, and the writer present bit, $PRES$, are reversed in order to overlay $PHID$ with the least-significant bit of $wout$. Since $wout$ is only incremented at the end of each writer phase, the toggling of the least-significant bit of $wout$ implies the beginning of the next reader phase (if any readers are present). Thus, it is unnecessary to allocate a bit exclusively for $PHID$.

[6] An atomic-sub instruction can be trivially emulated with an atomic-add instruction; the PF-C algorithm thus does not require additional hardware support (compared to the requirements of the PF-T algorithm).

```
 1  struct pfc_lock:
 2      unsigned integer word initially 0

 4  let WOUT_SHIFT  = 1    // least-significant bit of wout
 5  let WIN_SHIFT   = 9    // least-significant bit of win
 6  let RIN_SHIFT   = 17   // least-significant bit of rin
 7  let ROUT_SHIFT  = 25   // least-significant bit of rout
 8  let MASK        = 0x7f // mask for rin/win/rout/wout
 9  let GUARD       = 0x80 // mask for the overflow guard bit
10  let PRES        = 0x1  // mask for PRES
11  let WBITS       = 0x3  // mask for PRES and PHID

13  read_lock(struct pfc_lock* lock):
14      let ticket ← fetch_and_add(lock.word, 1 ≪ RIN_SHIFT)
15      if ((ticket ≫ RIN_SHIFT) & MASK) = MASK:
16          atomic_sub(lock.word, GUARD ≪ RIN_SHIFT)
17      let w ← ticket & WBITS
18      await ((w & PRES) = 0) or (w ≠ (lock.word & WBITS))

20  read_unlock(struct pfc_lock* lock):
21      atomic_add(lock.word, 1 ≪ ROUT_SHIFT)

23  write_lock (struct pfc_lock* lock):
24      let snapshot ← fetch_and_add(lock.word, 1 ≪ WIN_SHIFT)
25      let ticket ← (snapshot ≫ WIN_SHIFT) & MASK
26      if ticket = MASK:
27          atomic_sub(lock.word, GUARD ≪ WIN_SHIFT)
28      await ticket = (lock.word ≫ WOUT_SHIFT) & MASK

30      set snapshot ← fetch_and_add(lock.word, 1)
31      set ticket ← (snapshot ≫ RIN_SHIFT) & MASK
32      await ticket = (lock.word ≫ ROUT_SHIFT) & MASK

34  write_unlock(struct lock* lock):
35      if ((lock.word ≫ WOUT_SHIFT) & MASK) = MASK:
36          atomic_sub(lock.word, (GUARD ≪ WOUT_SHIFT) − 1)
37      else:
38          atomic_add(lock.word, 1)
```

Listing 5.2: The ticket-based PF-C algorithm. This implementation assumes little-endian 32-bit words. A PF-C lock requires only 4 bytes.

Readers. The PF-C reader entry procedure (lines 13–18) implements the same steps as the PF-T reader entry procedure: a reader J_i first atomically increments rin by one (subject to shifting, line 14) and observes both writer bits (line 17), and then—if a writer is present—spins until a writer bit toggles (line 18).

However, an additional step is required to avoid corrupting $rout$ when rin overflows. If all bits of rin were set *before* it was incremented, *i.e.*, if all rin bits are set in J_i's ticket (line 15), then J_i caused rin to overflow. Thus, rin's overflow guard bit is set and must be reset before rin overflows again. Clearing the overflow bit is accomplished by atomically subtracting $GUARD$ (after shifting it to rin's position) from the lock word (line 16). As long as there are at most 127 concurrent readers, rin cannot overflow a second time before J_i has cleared the overflow guard, thus this mechanism ensures that rin never corrupts $rout$.

The reader exit procedure is trivial: $rout$ is simply incremented (lines 20–21). Since $rout$ is the left-most field, it can safely overflow without corrupting any other field. This allocation is intentional—it enables a branch-free and short (and thus fast) reader exit path, which is likely executed more frequently than the writer exit path.

Writers. In the first part of the writer entry procedure, a newly-arriving writer increments win (subject to shifting, line 24), extracts its ticket from the observed lock value (by shifting and masking, line 25), checks for and corrects the overflow (if it occurred, lines 26–27), and then spins until all previous writes have completed (line 28).

Once a writer has become the head of the writer queue, it blocks newly arriving readers from entering their critical sections by setting $PRES$. Since $PRES$ is the first bit in the lock word, this is simply done by atomically adding one to the lock value (line 30). The writer then waits for all readers to exit by waiting for $rout$ to equal rin's observed value at the time of setting $PRES$ (lines 31-32).

Conceptually, the writer exit procedure (lines 34–38) has to accomplish three goals: $PRES$ must be cleared, $wout$ must be incremented, and overflowing $wout$ must be avoided (or corrected). Since $wout$ is allocated directly after $PRES$, it is possible to combine all three into one atomic update. Note that $wout$ is never updated concurrently,[7] hence a possible overflow can be detected before it is updated (line 35). First consider the case that $wout$ does not overflow (*i.e.*, the else-branch in

[7]However, other bitfields in the lock word may still be updated concurrently due to reader and writer arrivals, thus—in contrast to the PF-T algorithm—all updates must be atomic.

lines 37–38). In this case, it is sufficient to clear $PRES$ and increment $wout$. Since $PRES$, which is the least-significant bit of the lock word, is known to be set, both can be accomplished by adding one to the lock word—the addition will carry into $wout$ and $PRES$ will be cleared.

Otherwise, if incrementing $wout$ does cause an overflow, then $wout$'s overflow guard must be cleared by subtracting $GUARD$ (shifted to $wout$'s position) after updating $PRES$ and $wout$. Since $1 - GUARD = -(GUARD - 1)$, both updates can be combined into one atomic subtraction (line 36). The overflow guard between win and $wout$ is thus unused, but cannot be allocated to either win or $wout$ since they must be of equal size.

Correctness. Similar to PF-T locks, since all counters are seven bits wide (and possible overflow is handled in both reader and writer paths), PF-C locks are correct if there are at most $2^8 - 1$ concurrent readers and writers each.[8] PF-C locks implement the same control flow as PF-T locks and hence are also phase-fair.

5.3.3 A Phase-Fair Reader-Writer Spinlock with Constant RMR Complexity

Mellor-Crummey and Scott showed in their seminal work on efficient spin-based synchronization that it is possible to implement both task-fair mutex and RW locks as *list-based queue locks* such that spinning causes only a constant number of cache invalidations—that is, with $O(1)$ RMR complexity—irrespective of the number of processors (Mellor-Crummey and Scott, 1991a,b). In a list-based queue lock, a blocking job enqueues a *queue node* in a linked list prior to spinning on a flag in the queue node. Thus, since each spinning job spins on a private location, it will only incur a cache invalidation when it is unblocked.

In contrast, under the PF-T algorithm, a spinning reader may incur up to m cache invalidations since subsequently arriving readers atomically update rin, which causes the cache line holding rin to be evicted from the caches of all spinning readers. Similarly, a writer may incur up to m cache invalidations once while spinning on $wout$, and again while spinning on $rout$. Further, under the PF-C algorithm, all readers and writers modify the same word, hence a spinning job incurs a cache

[8] Bits could be re-allocated from win and $wout$ to rin and $rout$ to enable higher reader counters, albeit at the expense of reduced writer counts.

invalidation whenever a reader or writer arrives or exits.[9] Thus, both PF-T and PF-C locks have $\Omega(m)$ RMR complexity.

We next show how spin queues can be applied to reduce RMR complexity under phase-fair locks. The resulting PF-Q algorithm is given in Listings 5.3 and 5.4.

Structure. At a high level, the PF-Q algorithm resembles the PF-T algorithm: readers are tracked by the rin and $rout$ counters (line 15 in Listing 5.3), and are blocked from entering their critical sections by setting the $PRES$ bit in rin (line 3). The key difference is that instead of spinning on the lock state itself, blocked jobs enqueue themselves into reader and writer queues.

There are three such queues: a writer queue and two reader queues. The pointer $wtail$ (line 17) holds the address of the last writer's queue node, or has the value NIL (line 7) if no writer is present. The pointer $whead$ (line 18) holds the address of the head of the writer queue's node if said writer is blocked by a reader phase. There are two reader queues, pointed to by *rtail*[0] and *rtail*[1] (line 19), corresponding to the two possible values of the phase id bit in rin (line 2). Each *rtail* pointer can assume two special values: NIL, which signals that arriving readers may proceed, and $WAIT$ (line 8), which signals that arriving readers should spin.

A notable difference to PF-T locks is that the $PRES$ bit is set by writers in both rin and $rout$, with differing semantics. If $PRES$ is set in rin, then readers are not allowed to enter their critical sections and must spin. If $PRES$ is set in $rout$, then a writer is awaiting the end of a reader phase and the last exiting reader must unblock the spinning writer. Since readers may leave their critical sections in an order different from the arrival sequence, the last reader can only be identified when it updates $rout$. For this purpose, the variable $last$ (line 16) holds the value that $rout$ will assume once the reader phase has ended (as discussed below).

Readers. Arriving readers start by making their presence known and observe $PRES$ by incrementing rin (line 22). If $PRES$ is set, then the reader must block (line 23). To do so, it determines the current phase id (either zero or one, line 24), initializes its queue node (line 25), and appends its queue node onto the end of the reader queue corresponding to the current phase by atomically exchanging the tail pointer (line 26). The queue update yields $prev$, which is either the address of the predecessor's

[9] As modern architectures with cache line sizes of less than 16 bytes are rare, a spinning job likely incurs cache invalidations each time a reader or writer arrives or exits under PF-T locks in practice, too (unless each counter is padded with otherwise unused bytes to matche the cache line size).

```
 1  let  RINC   = 0x100      // reader increment
 2  let  PHID   = 0x1        // phase id bit
 3  let  PRES   = 0x2        // writer present bit
 4  let  WMSK   = 0x3        // writer bits in rin/rout
 5  let  TMSK   = ~WMSK      // reader ticket bits in rin/rout

 7  let  NIL    = ...   // NIL and WAIT are two address-like values
 8  let  WAIT   = ...   // that are distinct from any legal address

10  struct pfq_node:
11      boolean              blocked  initially  ⊥
12      struct pfq_node*     next     initially  NIL

14  struct pfq_lock:
15      unsigned integer     rin, rout initially 0
16      unsigned integer     last     initially  0
17      struct pfq_node*     wtail    initially  NIL
18      struct pfq_node*     whead    initially  NIL
19      struct pfq_node*     rtail[2] initially  NIL

21  start_read(struct pfq_lock* lock, struct pfq_node* self):
22      let ticket ← fetch_and_add(lock.rin, RINC)
23      if (ticket & PRES) = PRES:
24          let phase ← ticket & PHID
25          set self.blocked ← ⊤
26          let prev ← fetch_and_store(lock.rtail[phase], self)
27          if prev ≠ NIL:
28              await not self.blocked
29              if prev ≠ WAIT:
30                  set prev.blocked ← ⊥
31          else: // already unlocked
32              set prev ← fetch_and_store(lock.rtail[phase], NIL)
33              set prev.blocked ← ⊥
34              await not self.blocked
35      else: // no writer present

37  end_read(struct pfq_lock* lock):
38      let ticket ← fetch_and_add(lock.rout, RINC)
39      if (ticket & PRES) = PRES and (ticket & TMSK) = lock.last − 1:
40          set lock.whead.blocked ← ⊥
```

Listing 5.3: The queue-based PF-Q algorithm: type declarations and reader entry and exit procedures.

```
41  start_write (struct pfq_lock* lock, struct pfq_node* self):
42      let prev ← fetch_and_store(lock.wtail, self)
43      if prev ≠ NIL:
44          set self.blocked ← ⊤
45          set prev.next ← self
46          await not self.blocked

48      set self.blocked ← ⊤
49      set lock.whead ← self
50      let phase ← lock.rin & PHID
51      set lock.rtail[phase] ← WAIT
52      let ticket ← fetch_and_add(lock.rin, PRES) & TMSK
53      set lock.last ← ticket
54      let exit ← fetch_and_add(lock.rout, PRES) & TMSK
55      if exit ≠ ticket:
56          await not self.blocked

58  end_write(struct pfq_lock* lock, struct pfq_node* self):
59      set lock.rout ← lock.rout & TMSK
60      let phase ← lock.rin & PHID
61      let lsb ← pointer to least-significant byte of lock.rin
62      set *lsb ← (phase + 1) & PHID
63      let prev ← fetch_and_store(lock.rtail[phase], NIL)
64      if prev ≠ WAIT:
65          set prev.blocked ← ⊥

67      if self.next ≠ NIL or
68         not compare_and_swap(lock.wtail, self, NIL):
69          await self.next ≠ NIL
70      if self.next ≠ NIL:
71          set self.next.blocked ← ⊥
```

Listing 5.4: The queue-based PF-Q algorithm: writer entry and exit procedures.

queue node, $WAIT$ if the queue was empty, or NIL if the blocking writer phase ended in the mean time, *i.e.*, after $PRES$ was observed.

In the non-NIL case (lines 27–30), the reader spins on its private $blocked$ flag (line 28)—the reader has made its presence known, and thus can simply wait until it is notified of the end of the blocking writer phase. Once unblocked, it checks whether it is the head of the reader queue (line 29)—if not, it notifies its predecessor by clearing the corresponding $blocked$ flag before entering its critical section. Note that, in contrast to Mellor-Crummey and Scott's task-fair RW queue lock (Mellor-Crummey and Scott, 1991b), readers are unblocked from tail to head.

If $prev$ equals NIL (lines 31–34), then the arriving reader raced with an exiting writer and observed rin before $PRES$ was cleared. In this case, the reader must proceed to enter its critical section since the writer has already exited. However, additional readers may have observed $PRES$, enqueued themselves onto the reader queue, and entered the non-NIL case—and thus may be waiting to be unblocked (line 28). To avoid deadlock in this case, the reader that observed (and replaced) NIL must restore *rtail*[*phase*] (line 32) and unblock the job at the tail of the reader queue (line 33). Note that the waiting readers propagate the update to the head of the queue. Note also that the head of the queue is the job that observed NIL, thus it is already aware of the end of the blocking writer phase. However, the head must still wait for its $blocked$ flag to be toggled before entering its critical section (line 34) because otherwise its successor, if sufficiently delayed, could access $self.blocked$ after it has been deallocated.[10] If there are no successors, then $prev$ = $self$, and thus $self.blocked = \bot$ due to the update in line 33.

As in the previous two phase-fair locks, exiting readers increment the $rout$ counter (line 38). In the PF-T and PF-C locks, the writer spins by comparing $wout$ with its snapshot of rin—this, of course, can cause repeated cache invalidations. Thus, a different approach for detecting the end of a reader phase is required: if the $PRES$ bit is set in $rout$ (line 39), then the head of the writer queue is waiting to be unblocked by the last exiting reader (and the $whead$ pointer is valid). To reliably identify the last reader, the writer records its snapshot of rin in $last$. Thus, each exiting reader can test whether it is the last reader (line 39),[11] and if so, unblock the waiting writer (line 40).

[10] Queue nodes are commonly allocated on the stack, hence delayed writes must be avoided.

[11] Since $ticket$ holds the value of $rout$ before it was incremented, it is compared against $last - 1$.

Writers. Lines 42–46 of the writer entry procedure and lines 67–71 of the writer exit procedure incorporate Mellor-Crummey and Scott's mutex queue lock (Mellor-Crummey and Scott, 1991a) to ensure FIFO queueing of writers: an arriving writer appends its queue node to the writer queue (line 42), updates its predecessor's $next$ pointer (if any, lines 43–45), and then spins until unblocked (line 46). Symmetrically, an exiting writer removes itself from the writer queue (lines 67–69) and unblocks its successor (line 70–71).

Once a writer is at the head of the writer queue (line 48), it stores the address of its queue node in $whead$ (line 49), determines the phase id (line 50), and initializes the corresponding reader queue (line 51). Then it stops newly-arriving readers from entering their critical sections by setting $PRES$ in rin (line 52). The observed value of rin is stored in $last$ to make it available to readers (as discussed above, line 53). Then it sets $PRES$ in $rout$ to make the presence of a blocked writer known to exiting readers (line 54). While setting $PRES$, the writer also observes $rout$—blocking is only necessary if $rout$ is not equal to the previously-observed value of rin, otherwise all earlier-arrived readers have already finished their respective critical sections (lines 55–56).

An exiting writer first clears the $PRES$ bit in $rout$ (line 59), which can be done non-atomically since no concurrent reader exists. Next, it initiates the next reader phase. First, it enables newly-arriving readers to enter their critical sections by clearing the $PRES$ bit in rin (lines 60–62). This is accomplished by writing the next phase id to the least-significant byte of rin to avoid the use of an atomic fetch-and-modify instruction (*i.e.*, $PHID$ is toggled, line 62). Finally, any spinning readers are unblocked by storing NIL in the $rtail$ poiner corresponding to the ending writer phase (line 63) and clearing the $blocked$ flag of the tail node (if any, lines 64–65).

Correctness. Mutual exclusion of writers results from the use of Mellor-Crummey and Scott's task-fair mutex queue lock. Readers cannot enter during a writer phase since $PRES$ is not cleared until the writer exits. Liveness for writers is ensured because they atomically check $rout$ and make their presence known to readers. Liveness for readers is ensured because the use of two special values (NIL and $WAIT$) allows races between exiting writers and entering readers to be detected and corrected. Readers that are still in the process of propagating the start of a new reader phase through the reader queue do not interfere with the next head of the writer queue because the value of

$PHID$ alternates between writers. PF-Q locks have $O(1)$ RMR complexity because neither readers nor writers spin on shared lock state.

In this section, we have shown that phase-fair locks can be implemented efficiently on common hardware platforms. We report on the results of an implementation-based performance study of the considered RW lock choices in Chapter 7.

5.4 Detailed Blocking Analysis

In this section, we first introduce a generic framework for expressing bounds on both pi-blocking and s-blocking, and then apply it to bound both kinds of blocking under three types of non-preemptive spinlocks, namely task-fair mutex, task-fair RW, and phase-fair RW spinlocks. We do not consider preference locks (either reader or writer preference locks) in detail because their analysis is severely pessimistic due to the starvation issues discussed in Section 5.2.2. We refer the interested reader to our previously-published analysis of preference RW locks (Brandenburg and Anderson, 2010b).

The blocking analysis presented in the remainder of this chapter is essential for deriving safe blocking bounds suitable for schedulability analysis. However, such bounds tend to be somewhat technical in nature; the casual reader may safely skip this section and consult the overview presented in Section 5.2.2 instead.

The framework presented in the following is generic in the sense that it is not tied to any particular locking protocol. It serves two purposes. For one, it avoids redundancy in the subsequent analysis of the locking protocol presented in Chapter 6, which has structurally similar blocking terms. Second, the presented analysis takes a *holistic* analysis approach to reduce the pessimism of prior analysis. That is, it is intended to be applied to each job as a whole and bounds blocking across *all* requests that a job issues, instead of bounding delays on a request-by-request basis.

The benefit of a holistic approach is best illustrated with a concrete example. In the following, we assume non-preemptive task-fair spinlocks as used by the MSRP and by Devi *et al.* (2006) to highlight the ideas underlying the generic bound. However, to reiterate, the analysis is not tied to non-preemptive task-fair spinlocks. Since the analysis can be applied to bound either s-blocking or pi-blocking, we use "blocking" as generic term for either delay in this section. We initially assume

mutex constraints in this section and show how to apply the framework to RW locks in Sections 5.4.5 and 5.4.6 below.

The presented holistic analysis approach was first used to analyze the FMLP under P-FP scheduling (Brandenburg and Anderson, 2008b), and subsequently further developed to analyze RW spinlocks (Brandenburg and Anderson, 2010b) and a suspension-based protocol discussed in Chapter 6 (Brandenburg and Anderson, 2010a). The version presented herein has been somewhat simplified compared to the previous variants. We next explain the intuition underlying the approach, which we then formalize in Section 5.4.2 below.

5.4.1 Holistic Blocking Analysis

In the following, let J_i denote an arbitrary job of the task T_i for which a bound on maximum blocking is being derived. The main idea of the holistic approach is to avoid accounting for any individual possibly-blocking request more than once, and to avoid accounting for requests that cannot possibly interfere with J_i's requests.

Recall from Section 2.4 that the analysis of the MSRP and Devi *et al.*'s analysis of non-preemptive FIFO spinlocks analyzes each request individually—see Equations (2.8) and (2.9). A bound on the maximum s-blocking is then obtained by summing the bounds for all requests issued by J_i. If each job issues at most one request for each resource (*i.e.*, if $N_{i,q} \leq 1$ for all T_i and ℓ_q), then the subsequent analysis is equivalent to the simpler bounds derived by Gai *et al*. (2003) and Devi *et al*. (2006). However, if jobs potentially request the same resource more than once, then the holistic approach can avoid substantial pessimism, in particular if the maximum critical section length of other jobs is non-uniform (*i.e.*, if there are large differences among the tasks' $L_{i,q}$ parameters).

While this prior analysis simplifies the expression of the blocking bound, it does introduce considerable pessimism when jobs make repeated requests to the same resource. This analysis can be pessimistic because it may account more than once for a request that interferes only once.

Example 5.5. To illustrate possible pessimism when analyzing requests individually, consider the following scenario. Suppose a task T_i shares a resource ℓ_q with another task T_x under the MSRP. Further, suppose J_i requests ℓ_q up to $N_{i,q} = 20$ times and that jobs of T_x hold ℓ_q for at most $L_{x,q} = 10$ time units. Finally, suppose jobs of T_x require ℓ_q at most once while any J_i is pending.

When analyzing each of J_i's many requests individually, T_x's sole interfering request is effectively considered to block *each* of J_i's requests, which is clearly impossible. Consequently, according to Equation (2.8), J_i's overall bound on s-blocking due to requests for ℓ_q will be $spin(T_i, \ell_q) = N_{i,q} \cdot L_{x,q} = 200$ time units, whereas the actual maximum possible delay is $L_{x,q} = 10$ time units—J_x delays J_i with at most only one blocking request for ℓ_q, and not with up to 20 as predicted by Equation (2.8). $\Diamond$

This example demonstrates that maximum contention should be analyzed as a whole across all of J_i's requests for a particular resource. (Since we assume that requests for resources are not nested, blocking bounds for individual resources are independent from each other and can be derived individually.) The extent to which J_i is blocked due to requests for a resource ℓ_q in the worst case is limited by the following constraints:

1. *Maximum number of requests issued by other jobs.* As discussed above in Example 5.5, if jobs of T_x issue at most k requests while any J_i is pending, then J_i will be blocked by at most k requests of jobs of T_x, regardless of the number of requests issued by J_i.

2. *Maximum number of interfering requests per request issued by J_i.* Suppose J_i requests ℓ_q only once, that $m = 4$, and that ℓ_q is requested by other jobs up to $k = 100$ times while J_i is pending. Under non-preemptive task-fair spinlocks, J_i is delayed by at most $m - 1 = 3$ competing requests, irrespective of the total number of requests k for ℓ_q since the maximum queue length is limited to m non-preemptively executing jobs.

3. *Maximum number of interfering requests per task.* For example, suppose ℓ_q is shared among three tasks T_i, T_x, and T_y. Under non-preemptive task-fair spinlocks, if J_i issues only one request, then it is blocked by at most one request from T_x and one request from T_y, irrespective of the total number of requests issued by these tasks, and irrespective of the number of processors. Due to the FIFO ordering in the spin queue of a task-fair spinlock, each task can precede J_i at most once per request.

4. *Task locality.* For example, suppose T_i shares a resource with tasks T_x and T_y under the MSRP, and that T_i and T_x are assigned to processor 1, whereas T_y is assigned to processor 2. Jobs of T_y can cause J_i to incur s-blocking because they can issue conflicting requests while

J_i is scheduled. In contrast, jobs of T_x cannot cause J_i to incur s-blocking because jobs of T_x are not scheduled while J_i is executing; however, a job J_x can cause J_i to incur pi-blocking if J_x executes non-preemptively when J_i is released. A job J_y cannot directly delay J_i's release since it executes on a different processor, but J_y could transitively delay J_i if J_i incurs pi-blocking due to J_x, and J_x incurs s-blocking due to J_y.

We formalize these four constraints next.

5.4.2 Interference Sets

We begin with Constraint 1 by bounding the maximum resource requirements of competing tasks. In the task model assumed in this dissertation, a task T_i's resource requirements are characterized by the parameters $N_{i,q}$ and $L_{i,q}$. The main advantages of this model are that it is general enough to reflect many possible job behaviors (*e.g.*, no particular request order or minimum separation of requests is assumed) and that the required information can be obtained as part of WCET analysis (or realistically measured if WCET analysis is not available). However, it is possible that more detailed knowledge is available for specific applications.

For example, it could be the case that jobs of a task T_i access a resource ℓ_q twice, and that the second access is always much shorter than the first access. In this case, using single upper bound $L_{i,q}$ for both requests is needlessly pessimistic. A similar concern arises with resources that are not accessed by *every* job of T_i. For, example to reduce overheads, an application T_a could be programmed to record status information in a shared log ℓ_l only once every five jobs. Assuming that each J_a requests access to ℓ_l would needlessly overestimate contention for ℓ_l. However, explicitly incorporating all such considerations yields a task model that is overly complicated for our purposes (which is to study the underlying algorithmic properties of the protocols).

In this dissertation, we use an abstraction called *task interference bound* to achieve a separation of concerns between the modeling of resource requirements and the actual analysis of locking protocols, which is structurally independent from model considerations. A task's interference bound (for non-processor resources) is similar to a demand bound function (for processor time) in that it "upper bounds" a task's worst-case resource requirement during some interval. The actual blocking analysis is expressed in terms of task interference, which can be defined to take advantage of detailed

application-specific resource usage information. The primary benefit of this approach is that derived blocking terms can be reused to derive less pessimistic bounds when additional information in form of a more-detailed task model is available.

In the following, to achieve the desired separation of concerns, we formalize a task's "interference bound" as a set of requests that safely approximates a task's "actual contention" for a resource. Recall from Section 2.4.1 that $\mathcal{R}_{i,q,v}$ denotes the v^{th} request for resource ℓ_q issued by any J_i, and that $\mathcal{L}_{i,q,v}$ denotes the request length of $\mathcal{R}_{i,q,v}$. This allows us to formalize the concept of a task's "contention for a resource."

Definition 5.2. Suppose jobs of a task T_i execute k resource requests for a resource ℓ_q during an interval $[t_0, t_1)$. In a concrete, fixed schedule, the *contention* due to T_i during $[t_0, t_1)$ is the set of requests

$$C_{i,q}(t_0, t_1) \triangleq \left\{\mathcal{R}_{i,q,v}, \mathcal{R}_{i,q,(v+1)}, \ldots, \mathcal{R}_{i,q,(v+k-1)}\right\}$$

such that $\mathcal{R}_{i,q,v}$ is the first request and $\mathcal{R}_{i,q,(v+k-1)}$ is last request issued by any J_i during $[t_0, t_1)$.

In general, v and k are unknown prior to the execution of T_i, as is the length of each request in $C_{i,q}(t_0, t_1)$. To enable *a priori* analysis, a generic notion of worst-case contention is required. The purpose of T_i's request interference bound, given next, is to define a set of *generic requests* (*i.e.*, virtual requests defined for analysis purposes) that upper-bound the worst-case contention during any interval of length $t_1 - t_0$. That is, the interference bound for an interval of length $t_1 - t_0$ contains at least as many requests as T_i issues in any interval of length $t_1 - t_0$ in any actual schedule, and each generic request is at least as long as a corresponding actual one. This can be formalized as follows.

Definition 5.3. The *task interference bound* for an interval of length t, denoted $\mathit{tif}(T_i, \ell_q, t)$, is a set of generic requests that satisfies the following two properties.

1. For any $C_{i,q}(t_0, t_1)$ with regard to some actual schedule, one can choose a set of corresponding generic requests $C'_{i,q} \subseteq \mathit{tif}(T_i, \ell_q, t_1 - t_0)$ that satisfies

$$\left|C'_{i,q}\right| = |C_{i,q}(t_0, t_1)| \qquad \text{and} \qquad \sum_{\mathcal{R}_{i,q,v} \in C_{i,q}(t_0,t_1)} \mathcal{L}_{i,q,v} \quad \leq \sum_{\mathcal{R}_{i,q,w} \in C'_{i,q}} \mathcal{L}_{i,q,w}.$$

2. Interference bounds are inclusive:

$$t \leq t' \Rightarrow \mathit{tif}(T_i, \ell_q, t) \subseteq \mathit{tif}(T_i, \ell_q, t').$$

Property 1 ensures that the task interference bound does not underestimate the number and length of requests in any actual execution of T_i, and Property 2 ensures that a derived bound remains valid when analyzing a larger-than-necessary interval (*i.e.*, when over-estimating a job's response time).

In the case of RW constraints, we analogously define task T_i's *read interference bound*, denoted as $\mathit{rif}(T_i, \ell_q, t)$, with respect to read requests for ℓ_q, and T_i's *write interference bound*, denoted as $\mathit{wif}(T_i, \ell_q, t)$ with respect to write requests for ℓ_q. We further assume in this case that T_i's regular task interference $\mathit{tif}(T_i, \ell_q, t)$ applies to requests of either kind. That is, with respect to any interval of length t, $\mathit{rif}(T_i, \ell_q, t)$ bounds the worst-case frequency and length of read requests, $\mathit{wif}(T_i, \ell_q, t)$ bounds the worst-case frequency and length of write requests, and $\mathit{tif}(T_i, \ell_q, t)$ bounds the worst-case frequency and length of requests of either kind.

These definitions serve as an interface that allows the analysis of specific lock types presented in the following sections to be seamlessly integrated with more-refined task and resource models. Next, we provide suitable definitions of $\mathit{tif}(T_i, \ell_q, t)$, $\mathit{rif}(T_i, \ell_q, t)$, and $\mathit{wif}(T_i, \ell_q, t)$, for the model assumed in this dissertation. To this end, we establish the following bound on the maximum number of jobs that can execute requests in a given interval. Recall from Section 2.2 that p_i denotes T_i's period and r_i denotes T_i's maximum response time (and thus reflects possible tardiness, if any).

Lemma 5.1. *At most* $\left\lceil \dfrac{t + r_i}{p_i} \right\rceil$ *distinct jobs of a task* T_i *can execute in any interval of length* t.

Proof. By contradiction (see Figure 5.8 for an illustration). Suppose that there exists an interval $[t_0, t_0 + t)$ of length $t \geq 0$ in which $k \in \mathbb{N}$ jobs of T_i execute such that

$$k \geq \left\lceil \frac{t + r_i}{p_i} \right\rceil + 1. \tag{5.1}$$

Let $J_{i,x}$ denote the first and $J_{i,z}$ the last job of T_i to execute in $[t_0, t_0 + t)$, where $z = x + k - 1$ (note that $J_{i,x} \neq J_{i,z}$ since $r_i \geq e_i > 0$ and hence $k \geq 2$). In order for a job to execute in $[t_0, t_0 + t)$,

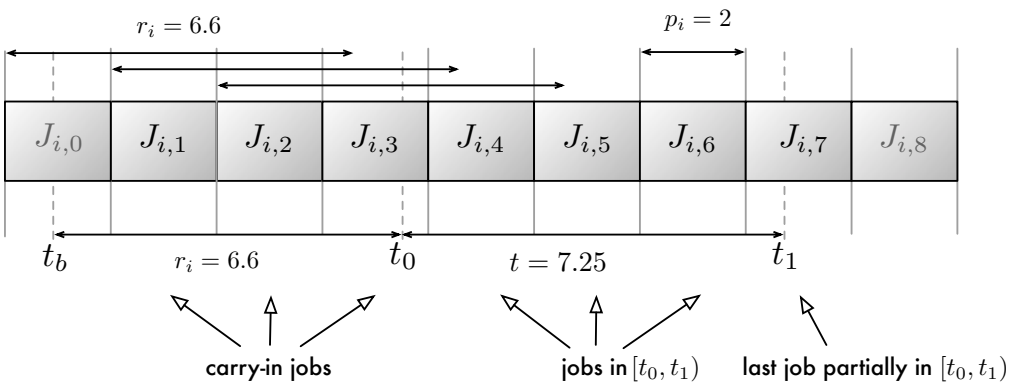


Figure 5.8: An illustration of Lemma 5.1. At most seven jobs of T_i with $p_i = 2$ and $r_i = 6.6$ can be pending in any interval $[t_0, t_1)$ of length $t_1 - t_0 = t = 7.25$ (illustration drawn to scale). Only jobs released at or after $t_b = t_0 - r_i$ and before t_1 can be pending in $[t_0, t_1)$. In the worst case (*i.e.*, with periodic arrivals and jobs completing as late as possible), jobs $J_{i,1}$, $J_{i,2}$, and $J_{i,3}$ carry execution into $[t_0, t_1)$. Jobs $J_{i,4}$, $J_{i,5}$, and $J_{i,6}$ are released and complete in $[t_0, t_1)$, and $J_{i,7}$ is released before t_1. Note that moving the start of the interval t_0 to an earlier point such that $a_{i,0} \geq t_b$ causes the last counted job $J_{i,7}$ to no longer be pending within $[t_0, t_1)$, *i.e.*, $t_1 < a_{i,7}$. Hence, $maxjobs(T_i, 7.25) = \lceil \frac{13.85}{2.0} \rceil = 7$.

it must be pending some time in $[t_0, t_0 + t)$, that is,

$$t_0 - r_i \leq a_{i,x} \quad \text{and} \tag{5.2}$$

$$a_{i,z} < t_0 + t. \tag{5.3}$$

Further, since job releases are separated by at least one period, it follows that

$$a_{i,x} + (k-1) \cdot p_i \leq a_{i,z}. \tag{5.4}$$

By substituting Inequalities (5.2) and (5.3) into Inequality (5.4), we obtain

$$t_0 - r_i + (k-1) \cdot p_i \leq t_0 + t,$$

which, by re-arranging, yields

$$k < \frac{t + r_i}{p_i} + 1 \leq \left\lceil \frac{t + r_i}{p_i} \right\rceil + 1.$$

This contradicts Inequality (5.1) above. □

It follows from Lemma 5.1 and the definition of $N_{i,k}$ that jobs of T_i issue at most $\lceil (t + r_i)/p_i \rceil \cdot N_{i,k}$ requests for ℓ_q over any interval of length t. In the worst case, each request for ℓ_q is of length $L_{i,q}$. This yields the following interference bound for the task model assumed herein.

Definition 5.4. The *request interference bound* for task T_i with respect to resource ℓ_q over any interval of length t is the set of requests

$$tif(T_i, \ell_q, t) \triangleq \left\{ \mathcal{R}_{i,q,v} \mid 1 \leq v \leq N_{i,q} \cdot \left\lceil \frac{t + r_i}{p_i} \right\rceil \right\},$$

where $\mathcal{L}_{i,q,v} = L_{i,q}$ for each $\mathcal{R}_{i,q,v}$. If T_i does not access a given resource ℓ_q, then $tif(T_i, \ell_q, t) = \emptyset$ for all t. Task T_i's read and write interference bounds, respectively, are analogously defined as

$$rif(T_i, \ell_q, t) \triangleq \left\{ \mathcal{R}^R_{i,q,v} \mid 1 \leq v \leq N^R_{i,q} \cdot \left\lceil \frac{t + r_i}{p_i} \right\rceil \right\} \text{ and}$$
$$wif(T_i, \ell_q, t) \triangleq \left\{ \mathcal{R}^W_{i,q,v} \mid 1 \leq v \leq N^W_{i,q} \cdot \left\lceil \frac{t + r_i}{p_i} \right\rceil \right\},$$

where $\mathcal{L}^R_{i,q,v} = L^R_{i,q}$ for each $\mathcal{R}^R_{i,q,v}$ and $\mathcal{L}^W_{i,q,v} = L^W_{i,q}$ for each $\mathcal{R}^W_{i,q,v}$.

Based on per-task interference bounds, we next introduce a generic, parametrized "aggregate interference bound" for use in the subsequent analysis of specific locking protocols. We first define three convenience functions over sets of requests, which serve to simplify the expression of "aggregate interference" and protocol-specific bounds on blocking.

Definition 5.5. Given a set of requests S, we let S_k denote the k^{th} longest request in S, where $1 \leq k \leq |S|$ (with ties broken arbitrarily but consistently). Formally, if $1 \leq k \leq l \leq |S|$ and $S_k = \mathcal{R}_{a,b,c}$ and $S_l = \mathcal{R}_{x,y,z}$, then $\mathcal{L}_{a,b,c} \geq \mathcal{L}_{x,y,z}$.

Definition 5.6. Given a set of requests S, we denote the set of the l longest requests in S as

$$top(l, S) \triangleq \{ S_k \mid 1 \leq k \leq \min(l, |S|) \}$$

and their total duration as

$$total(l, S) \triangleq \sum_{\mathcal{R}_{i,q,v} \in\, top(l,S)} \mathcal{L}_{i,q,v}.$$

If $l = 0$ or $S = \emptyset$, then $total(l, S) = 0$.

A task interference bound limits the maximum contention from jobs of a single task. Using the above definitions, we can formalize the notion of contention from a set of tasks. Recall Constraint 3 from Section 5.4.1 above, namely that the number of requests per task that can possibly cause J_i to incur acquisition delay is limited if jobs wait in FIFO order. If a task T_x can delay J_i with at most l requests, then it is sufficient to consider only the l longest requests in T_x's interference bound. We therefore define the aggregate interference bound with a per-task "interference limit" parameter.

Definition 5.7. The *aggregate interference bound* of a set of tasks τ with respect to a resource ℓ_q over any interval of length t and subject to an *interference limit* l is given by

$$tifs(\tau, \ell_q, t, l) \triangleq \bigcup_{T_x \in \tau} top\left(l,\ tif(T_x, \ell_q, t)\right).$$

That is, given an interference limit l, $tifs(\tau, \ell_q, t, l)$ contains the l longest requests in each task's interference bound for ℓ_q and t.[12] *Aggregate read interference* and *aggregate write interference* are defined analogously as

$$rifs(\tau, \ell_q, t, l) \triangleq \bigcup_{T_x \in \tau} top\left(l,\ rif(T_x, \ell_q, t)\right) \text{ and}$$
$$wifs(\tau, \ell_q, t, l) \triangleq \bigcup_{T_x \in \tau} top\left(l,\ wif(T_x, \ell_q, t)\right).$$

We have incorporated Constraints 1 and 3 from Section 5.4.1 in a generic fashion. The remaining Constraints 2 and 4 are easier to incorporate on a protocol-by-protocol basis. We next demonstrate how to apply holisitic blocking analysis to a concrete locking protocol.

5.4.3 Task-Fair Mutex Spinlocks

In the following, we derive bounds on maximum s-blocking and pi-blocking under non-preemptive task-fair mutex spinlocks under clustered event-driven scheduling with $1 \leq c \leq m$. We make no further assumptions about the specific scheduling policy in use; the following analysis therefore applies equally to P-FP, EDF, C-EDF, and G-EDF scheduling.

[12] In the task model assumed in this dissertation, each request in $tif(T_x, \ell_q, t)$ is in fact of the same length $\mathcal{L}_{x,q,v} = L_{x,q}$ (see Definition 5.4). We define $tifs(\tau, \ell_q, t, l)$ with additional generality to accomodate more-expressive task models for which $tif(T_x, \ell_q, t)$ may contain non-uniform request lengths.

Since task locality can be exploited to limit pessimism, we apply Definition 5.7 on a per-cluster basis. Recall that we let τ_j denote the set of tasks assigned to the j^{th} cluster, and that P_i denotes the cluster to which task T_i has been assigned. In total, there are $\frac{m}{c}$ of c processors each.[13]

We first establish a bound on maximum s-blocking due to requests for resource ℓ_q.

Lemma 5.2. *Under non-preemptive task-fair mutex locks, a job J_i incurs at most*

$$spin(T_i, j, \ell_q) = \begin{cases} total(N_{i,q} \cdot c,\ tifs(\tau_j, \ell_q, r_i, N_{i,q})) & \text{if } j \neq P_i \\ total(N_{i,q} \cdot (c-1),\ tifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, N_{i,q})) & \text{if } j = P_i \end{cases}$$

s-blocking due to requests for resource ℓ_q issued by jobs of tasks assigned to the j^{th} cluster.

Proof. First consider the case of $j \neq P_i$. Since requests are satisfied in FIFO order, a job of each task in τ_j can precede J_i at most once each time that J_i issues a request. As J_i issues at most $N_{i,q}$ requests for ℓ_q, each remote task is subject to an interference limit of $N_{i,q}$. Further, at most c jobs of tasks in the j^{th} cluster execute requests or busy-wait at any time since requests are executed non-preemptively. Hence J_i is preceded by at most $N_{i,q} \cdot c$ requests issued by jobs of tasks in τ_j in total. In the case of $j = P_i$, one of the c processors is taken up by J_i itself. Hence, at most $N_{i,q} \cdot (c-1)$ request block J_i in total. Since jobs and tasks are sequential, J_i is not delayed by requests of (other) jobs of T_i. □

Lemma 5.2 highlights the holistic nature of the analysis: instead of bounding the busy-waiting of each request issued by J_i individually, Definition 5.7 yields a single approximation of worst-case contention across all $N_{i,q}$ requests issued by J_i. By aggregating across all clusters and all resources, this yields the following bound on maximum s-blocking.

Theorem 5.1. *Under non-preemptive task-fair mutex locks, a job J_i incurs at most*

$$s_i = \sum_{q=1}^{n_r} \sum_{j=1}^{m/c} spin(T_i, j, \ell_q)$$

s-blocking due to requests for shared resources.

[13] Non-uniform cluster sizes (*e.g.*, due to dedicated interrupt handling) can be trivially integrated. We assume uniform clusters sizes in the following for the sake of simplified notation.

Proof. Since resource requests are not nested, and since tasks do not migrate across clusters boundaries, the sum of the per-cluster, per-resource bounds (Lemma 5.2) bounds the total delay. □

In addition to s-blocking, jobs may also incur pi-blocking upon release if a lower-priority job is non-preemptively accessing a shared resource. The maximum duration of pi-blocking is limited to one request span (*i.e.*, one request including non-preemptable busy-waiting).[14]

Since pi-blocking is only caused by lower-priority jobs that are non-preemptable at the time of J_i's release, we only need to consider the set of tasks that could have released a lower-priority job prior to J_i's arrival. This set of tasks necessarily depends on the specific scheduling policy.

Definition 5.8. We let $lower(T_i)$ denote the set of local tasks that could potentially cause J_i to incur pi-blocking upon release.

$$lower(T_i) \triangleq \begin{cases} \{T_x \mid T_x \in \tau_{P_i} \wedge x > i\} & \text{under FP-based schedulers} \\ \{T_x \mid T_x \in \tau_{P_i} \wedge d_x > d_i\} & \text{under EDF-based schedulers (Lemma 2.4)} \end{cases}$$

The maximum duration of pi-blocking incurred by any J_i is bounded by the maximum request span of any job of tasks in $lower(T_i)$. We next bound the maximum acquisition delay incurred by any lower-priority job.

Lemma 5.3. *Let J_x denote a lower-priority job that causes J_i to incur pi-blocking upon release by non-preemptively executing a request for resource ℓ_q. J_x's request is delayed for at most*

$$spin'_i(T_x, j, \ell_q) = \begin{cases} total(c,\ tifs(\tau_j, \ell_q, r_x, 1)) & \text{if } j \neq P_i \\ total(c-1,\ tifs(\tau_j \setminus \{T_i, T_x\}, \ell_q, r_x, 1)) & \text{if } j = P_i \end{cases}$$

time units due to requests for ℓ_q issued by jobs of tasks assigned to the j^{th} cluster.

Proof. Analogous to Lemma 5.2. Since we consider only a single request, the per-task interference limit is one (due to the FIFO ordering of requests), and, if $j \neq P_i$, at most c earlier-issued requests cause J_x to incur acquisition delay. In the case of $j = P_i$, one processor is taken up by J_x itself.

[14] Assuming link-based scheduling is employed if $c > 1$ (see Section 3.3.3).

Further, since J_i has just been released and not yet scheduled, J_x is not delayed by requests of jobs of T_i while it causes J_i to incur pi-blocking. □

Lemma 5.3 allows us to express the bound on maximum pi-blocking.

Theorem 5.2. *Under non-preemptive task-fair mutex locks, a job J_i incurs at most*

$$b_i = \max_{1 \leq q \leq n_r} \quad \max_{\substack{T_x \in lower(T_i) \\ N_{x,q} > 0}} \left(L_{x,q} + \sum_{j=1}^{m/c} spin'_i(T_x, j, \ell_q) \right)$$

pi-blocking upon release due to the non-preemptive execution of resource requests.

Proof. Follows from Definition 5.8, Lemma 5.3, and the fact that J_i incurs pi-blocking at most once upon release in the absence of self-suspensions. □

In this dissertation, we assume that jobs do not suspend unless suspended by a semaphore-based locking protocol (Chapter 6). If J_i self-suspends for any reason when non-preemptive spinlocks are employed, then it may incur additional pi-blocking due to lower-priority, non-preemptive jobs each time that it resumes. Such pi-blocking can be bounded analogously to Theorem 5.2.

It should be noted that, in the case of EDF-based schedulers, Definition 5.8 is not sufficient in characterizing pi-blocking after a self-suspension since tasks with shorter relative deadlines may release lower-priority jobs while J_i is pending. That is, in the general case $lower(T_i) = \tau_{P_i} \setminus \{T_i\}$ when bounding maximum pi-blocking after a self-suspension under EDF-based schedulers. The definition of $lower(T_i)$ under FP-based schedulers remains unchanged since a job's priority is unaffected by its release time under FP scheduling.

This concludes our analysis of non-preemptive task-fair mutex locks. In the following, we apply our holistic analysis approach to task-fair and phase-fair spinlocks. We start by bounding the maximum number of blocking reader phases.

5.4.4 Minimum Reader Parallelism

The defining property of an RW lock is that readers do not *directly* block other readers. That is, in the absence of any writers, a reader is not delayed under task-fair or phase-fair RW locks regardless of the number of concurrent read requests. Similarly, a writer that is not delayed by other writers incurs

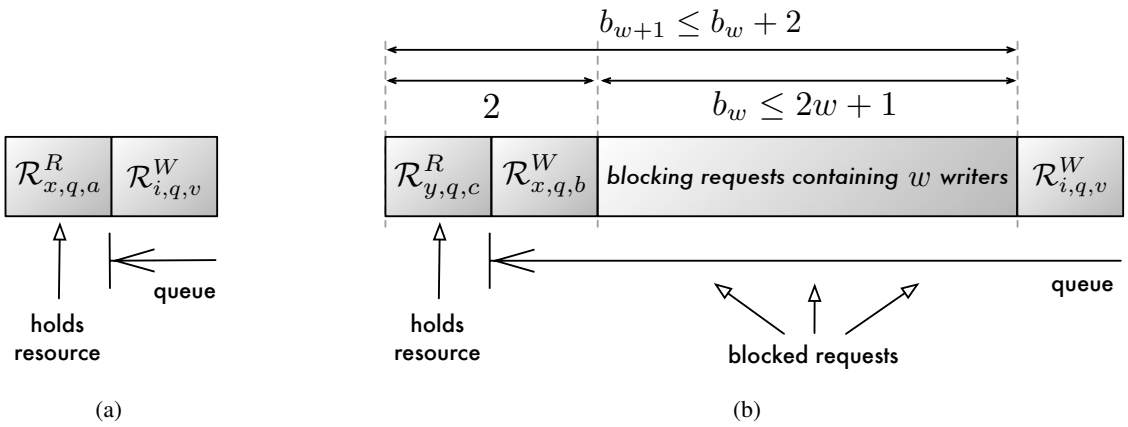


Figure 5.9: Illustration of Lemma 5.4 for the case $N^W_{i,q} = 1$ wherein a job J_i issues only one write request $\mathcal{R}^W_{i,q,v}$. **(a)** Base case $w = 0$: at most one reader phase can block $\mathcal{R}^W_{i,q,v}$. **(b)** Induction step: allowing one additional write request $\mathcal{R}^W_{x,q,b}$ can cause an additional reader phase to block J_i and hence increases the maximum number of blocking phases of either kind by at most two.

acquisition delay for the duration of at most one read request regardless of the number of blocking readers. For example, if $m - 1$ readers hold a resource ℓ_q when J_i issues a write request for ℓ_q, then all $m - 1$ readers proceed in parallel and J_i incurs s-blocking only for the duration of the longest earlier-issued read request.

Intuitively, a reader phase can only *transitively* block a read request if said phase is "assisted" by an also-blocking, interspersed writer phase. In this section, we formalize this intuition by establishing two lemmas that characterize reader parallelism under fair RW locks (both task-fair and phase-fair).

Lemma 5.4. *Let J_i denote a job that issues at most $N^W_{i,q}$ write requests for a resource ℓ_q, and let*

- *w denote the number of writer phases that directly delay J_i's write requests for ℓ_q,*
- *r denote the number of reader phases that directly delay J_i's write requests for ℓ_q,*
- *b_w denote the total number of phases (either read or write) that directly delay J_i's write requests for ℓ_q.*

If ℓ_q is protected by a task-fair or phase-fair RW lock, then $b_w \leq 2w + N^W_{i,q}$ and $r \leq w + N^W_{i,q}$.

Proof. By induction over w. Note that $b_w = w + r$.

Base case: $w = 0$. If there are no blocking writer phases, then J_i's write requests (if any) can only be blocked by one reader phase each. In task-fair locks, this is due to FIFO ordering; in

phase-fair locks this follows from Properties PF1 and PF4. Hence, $r \leq N_{i,q}^W$, and thus $r = r + w = b_w \leq 2w + N_{i,q}^W$. This is illustrated in inset (a) of Figure 5.9.

Induction step $w \to w + 1$*:* Adding one blocking write request $\mathcal{R}_{x,q,b}^W$ (*i.e.*, one writer phase) increases the number of phases blocking J_i by at most two: first, $\mathcal{R}_{x,q,b}^W$ directly blocks J_i once, and, second, $\mathcal{R}_{x,q,b}^W$ can itself be blocked by an additional read request $\mathcal{R}_{x,q,b}^R$ (and thus one reader phase), which then transitively also blocks J_i. Thus, $b_{w+1} \leq b_w + 2$. This is illustrated in inset (b) of Figure 5.9. By the induction hypothesis, $b_w \leq 2w + N_{i,q}^W$, thus

$$b_{w+1} \leq b_w + 2 \leq (2w + N_{i,q}^W) + 2 = 2(w + 1) + N_{i,q}^W.$$

Hence, $b_w \leq 2w + N_{i,q}^W$ for all $w \in \mathbb{N}$. Since $b_w = w + r$, this implies $r \leq w + N_{i,q}^W$. □

Lemma 5.4 bounds b_w for a known w. Next, we derive a bound on r given a bound on b_w.

Lemma 5.5. *Let J_i denote a job that issues at most $N_{i,q}^W$ write requests for a resource ℓ_q, and define w, r, and b_w as in Lemma 5.4 above. If there exists a bound $a \in \mathbb{N}$ such that $b_w \leq a$, then*

$$r \leq \left\lfloor \frac{a + N_{i,q}^W}{2} \right\rfloor.$$

Proof. We first show $r \leq \frac{a+N_{i,q}^W}{2}$ by contradiction. Suppose

$$\frac{a + N_{i,q}^W}{2} < r.$$

Since $b_w = w + r$ and thus $r \leq a - w$, the stated inequality implies

$$\frac{a + N_{i,q}^W}{2} < a - w$$

and thus

$$w < a - \frac{a + N_{i,q}^W}{2} = \frac{a - N_{i,q}^W}{2}.$$

By solving for a we obtain

$$a > 2w + N_{i,q}^W \geq b_w,$$

where the last inequality follows from Lemma 5.4. This contradicts the assumption $b_w \leq a$, and thus, $r \leq \frac{a+N_{i,q}^W}{2}$ holds. Since $r \in \mathbb{N}$, the stated inequality follows. □

As we show next, Lemma 5.5 allows us to bound s-blocking under phase-fair and task-fair RW locks more accurately if the number of writers is low.

5.4.5 Task-Fair RW Spinlocks

Recall that under task-fair RW locks, writers gain exclusive access to resource groups, whereas *consecutive* readers may access resource groups concurrently. Consequently, a writer that is directly blocked by a reader phase may cause the reader phase to transitively block later-arriving readers.

Since task-fair RW locks degrade to mutex-like serialization in the worst case, the bounds for task-fair mutex locks derived in Section 5.4.3 also apply to task-fair RW locks. However, if writes are infrequent, then Lemma 5.5 can be used to derive much less pessimistic bounds. To this end, we first determine the set of write requests that delay a job J_i in the worst case.

Lemma 5.6. *Under non-preemptive task-fair RW locks, a job J_i is directly blocked by no more than $|W(T_i, j, \ell_q)|$ write requests for resource ℓ_q issued by jobs of tasks assigned to the j^{th} cluster, where*

$$W(T_i, j, \ell_q) = \begin{cases} top((N_{i,q}^W + N_{i,q}^R) \cdot c,\ wifs(\tau_j, \ell_q, r_i, (N_{i,q}^W + N_{i,q}^R))) & \text{if } j \neq P_i \\ top((N_{i,q}^W + N_{i,q}^R) \cdot (c-1),\ wifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, (N_{i,q}^W + N_{i,q}^R))) & \text{if } j = P_i. \end{cases}$$

Proof. Analogously to Lemma 5.2. Due to the FIFO ordering of requests, each task can precede J_i at most once with a write request each time that J_i issues a read or write request, which J_i does at most $N_{i,q}^W + N_{i,q}^R$ times. Since requests are executed non-preemptively, J_i is preceded by at most $(N_{i,q}^W + N_{i,q}^R) \cdot c$ other jobs in ℓ_q's spin queue across all requests if $j \neq P_i$, and by at most $(N_{i,q}^W + N_{i,q}^R) \cdot (c-1)$ otherwise since J_i occupies one of the processors if $P_i = j$. □

Together with Lemma 5.4, Lemma 5.6 implies a bound on the maximum number of blocking reader phases. In the next step, we determine the set of longest read requests issued by jobs in each cluster that must be considered when bounding maximum s-blocking.

Definition 5.9. In Lemma 5.7 and Theorem 5.3 below, define W_q, w_q, and r_q as follows.

$$W_q = \bigcup_{j=1}^{m/c} W(T_i, j, \ell_q) \qquad w_q = |W_q| \qquad r_q = w_q + N_{i,q}^W$$

Lemma 5.7. *Under non-preemptive task-fair RW locks, a job J_i incurs s-blocking due to reader phases while waiting to acquire resource ℓ_q for at most $total(r_q,\ R_q)$ time units, where*

$$R_q = \bigcup_{j=1}^{m/c} R(T_i, j, \ell_q)$$

$$R(T_i, j, \ell_q) = \begin{cases} top((N_{i,q}^W + N_{i,q}^R) \cdot c,\ rifs(\tau_j, \ell_q, r_i, l)) & \text{if } j \neq P_i \\ top((N_{i,q}^W + N_{i,q}^R) \cdot (c-1),\ rifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, l)) & \text{if } j = P_i \end{cases}$$

and $l = \min(r_q,\ N_{i,q}^W + N_{i,q}^R)$.

Proof. $R(T_i, j, \ell_q)$ is derived analogously to Lemma 5.2. The set of all read requests R_q (regardless of cluster) is the union of all per-cluster interference sets since tasks do not migrate across cluster boundaries. By Lemma 5.4, at most r_q reader phases block J_i since it is blocked by at most w_q writer phases across all clusters (Lemma 5.6). The per-task interference limit $l = \min(r_q,\ N_{i,q}^W + N_{i,q}^R)$ is due to the fact that each task can precede J_i with a read request at most once per request, but no more than r_q times (the maximum number of reader phases). J_i's total delay due to reader phases is bounded by the duration of the r_q longest read requests, which are upper-bounded in duration by the r_q longest requests in R_q. □

Theorem 5.3. *Under non-preemptive task-fair RW locks, a job J_i incurs at most*

$$s_i = \sum_{q=1}^{n_r} \min \left(\sum_{j=1}^{m/c} spin(T_i, j, \ell_q),\ total(w_q,\ W_q) + total(r_q,\ R_q) \right)$$

s-blocking due to requests for shared resources, where $spin(T_i, j, \ell_q)$ is defined as in Lemma 5.2.

Proof. Since task-fair RW locks cause no more blocking than task-fair mutex locks, the bound for task-fair mutex locks from Lemma 5.2 also bounds maximum blocking under task-fair RW locks (with regard to each resource and each cluster). It follows from Lemma 5.6, Lemma 5.7, and from the fact that tasks do not migrate across clusters boundaries that $total(w_q,\ W_q) + total(r_q,\ R_q)$ also bounds maximum s-blocking with regard to each resource ℓ_q. As resource requests are not nested, the sum of the less-pessimistic per-resource bounds total s-blocking. □

As with s-blocking, the bound on maximum pi-blocking under non-preemptive task-fair mutex spinlocks given in Theorem 5.2 also applies to non-preemptive task-fair RW spinlocks. Further, Lemma 5.4 could be applied as above to yield a second, possibly lower bound on maximum pi-blocking. The required derivation is omitted since it is nearly identical to Lemmas 5.6 and 5.7 assuming a per-task interference limit of one (similar to Lemma 5.3).

5.4.6 Phase-Fair RW Spinlocks

In the following, we derive bounds on maximum s-blocking and maximum pi-blocking under phase-fair RW spinlocks based on the defining properties of phase-fairness. Worst-case s-blocking under phase-fair RW spinlocks is not equivalent to task-fair mutex lock. The analysis in Section 5.4.3 hence does not apply to phase-fair RW spinlocks.

Since write requests are satisfied in FIFO order with respect to other write requests (Rule PF2), maximum s-blocking *incurred by writers* due to earlier-issued write requests is the same under task-fair and phase-fair RW locks. However, since reader and writer phases alternate (Rule PF1), maximum s-blocking *incurred by readers* due to earlier-issued write requests is limited to one phase. That is, whereas up to $(N_{i,q}^W + N_{i,q}^R) \cdot c$ write requests issued by jobs of a remote cluster may block a job J_i under non-preemptive task-fair RW spinlocks, only $N_{i,q}^W \cdot c + N_{i,q}^R$ such requests can block J_i under phase-fair RW spinlocks (since each of the $N_{i,q}^R$ read requests is blocked by at most one writer phase). In the case of J_i's local cluster, if $c > 1$, then the same reasoning applies and no more than $N_{i,q}^W \cdot (c-1) + N_{i,q}^R$ write requests block J_i. As a special case, if $c = 1$, local jobs cannot directly block J_i. This yields the following bound.

Lemma 5.8. *Under non-preemptive phase-fair RW locks, a job J_i is directly blocked by no more than $|W(T_i, j, \ell_q)|$ write requests for resource ℓ_q issued by jobs of tasks assigned to the j^{th} cluster,*

where

$$W(T_i, j, \ell_q) = \begin{cases} top(x^{rem},\ wifs(\tau_j, \ell_q, r_i, (N_{i,q}^W + N_{i,q}^R))) & \text{if } j \neq P_i \\ top(x^{loc},\ wifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, (N_{i,q}^W + N_{i,q}^R))) & \text{if } j = P_i \text{ and } c > 1 \\ \emptyset & \text{if } j = P_i \text{ and } c = 1 \end{cases}$$

and $x^{rem} = N_{i,q}^W \cdot c + N_{i,q}^R$ *and* $x^{loc} = N_{i,q}^W \cdot (c-1) + N_{i,q}^R$.

Proof. Follows from the preceding discussion analogously to Lemma 5.6. □

As in the case of task-fair RW locks, Lemma 5.8 and Lemma 5.4 together imply an upper bound on the maximum number of blocking reader phases.

Definition 5.10. In Lemma 5.9 and Theorem 5.4 below, define $W(T_i, j, \ell_q)$ as in Lemma 5.8 above and let $w_q = |W_q|$, where

$$W_q = \bigcup_{j=1}^{m/c} W(T_i, j, \ell_q).$$

Lemma 5.9. *Under non-preemptive phase-fair RW locks, a job J_i is blocked by at most*

$$r_q = \min(w_q + N_{i,q}^W,\ N_{i,q}^R + (m-1) \cdot N_{i,q}^W)$$

reader phases.

Proof. Lemma 5.8 implies that J_i is blocked by at most w_q writer phases with respect to ℓ_q and all tasks (in all clusters). By Lemma 5.4, J_i is hence blocked by at most $w_q + N_{i,q}^W$ reader phases (with respect to ℓ_q and all tasks). This yields the first bound.

The second bound follows from the maximum queue length. Each of J_i's read requests is preceded by at most one writer phase. Since writer and reader phases alternate (Rule PF1), J_i's (up to) $N_{i,q}^R$ read requests are transitively delayed by at most $N_{i,q}^R$ earlier reader phases. Further, also due to Rule PF1, any two writer phases are separated by at most one reader phase. Since J_i's (up to) $N_{i,q}^W$ write requests are delayed by at most $N_{i,q}^W(m-1)$ write requests ($c-1$ per request in J_i's local

cluster, c in each remote cluster), J_i's write request are transitively delayed by at most $N_{i,q}^W(m-1)$ reader phases, for a total of at most $N_{i,q}^R + (m-1) \cdot N_{i,q}^W$ blocking reader phases. □

This yields the following bound on maximum delay due to blocking reader phases.

Lemma 5.10. *Under non-preemptive phase-fair RW locks, a job J_i incurs s-blocking due to reader phases while waiting to acquire resource ℓ_q for at most $total(r_q,\ R_q)$ time units, where*

$$R_q = \bigcup_{j=1}^{m/c} R(T_i, j, \ell_q)$$

$$R(T_i, j, \ell_q) = \begin{cases} top(r_q,\ rifs(\tau_j, \ell_q, r_i, r_q)) & \text{if } j \neq P_i \\ top(r_q,\ rifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, r_q)) & \text{if } j = P_i \text{ and } c > 1 \\ \emptyset & \text{if } j = P_i \text{ and } c = 1 \end{cases}$$

and r_q is defined as in Lemma 5.9.

Proof. By Lemma 5.9, at most r_q reader phases block J_i during requests for ℓ_q. It is thus sufficient to consider the r_q longest read requests from each task and each cluster to determine the maximum cumulative duration of r_q reader phases. (Unless $c = 1$, in which case local tasks cannot block J_i.) Further, each task can block J_i with at most individual r_q read requests (*i.e.*, the task interference limit is r_q) since the fact that J_i is waiting implies that read requests are satisfied only once per reader phase, namely at the beginning of each reader phase (Rules PF3 and PF4). Based on these considerations, the stated bound follows analogously to Lemma 5.7. □

Finally, overall s-blocking under non-preemptive phase-fair RW spinlocks is bounded as follows.

Theorem 5.4. *Under non-preemptive phase-fair RW locks, a job J_i incurs at most*

$$s_i = \sum_{q=1}^{n_r} total(w_q,\ W_q) + total(r_q,\ R_q)$$

s-blocking due to requests for shared resources.

Proof. Follows from Lemma 5.8 and Lemma 5.10. □

A bound on maximum pi-blocking under non-preemptive phase-fair RW locks can be easily derived analogously to Theorem 5.2 by assuming an interference limit of one for writers.

This concludes our analysis of spin-based locking protocols. In Chapter 7, we present an empirical comparison of non-preemptive task-fair mutex, task-fair RW, and phase-fair RW spinlocks.

5.5 Summary

We have proposed and analyzed several non-preemptive spinlock protocols. The holistic analysis approach employed in Section 5.4 improves upon prior analysis of non-preemptive task-fair mutex spinlocks if jobs request resources repeatedly. If a significant fraction of requests for a resource only observe state without changing it, then s-blocking can be further reduced by employing RW locks instead of mutex locks. We have discussed the advantages and disadvantages of several types of RW locks with regard to worst-case analysis. In particular, we have proposed phase-fairness, a progress guarantee that is well-suited to worst-case analysis, and presented three efficient implementations of phase-fair spinlocks. Additionally, we have discussed how resource nesting can be accommodated to some degree by means of group locking. Group locking can be combined with virtually any locking protocol and thus also applies to semaphore-based synchronization, which we consider next.

CHAPTER 6

REAL-TIME SEMAPHORE PROTOCOLS*

In the preceding chapter, we have considered spin-based protocols, which, as we later show in Chapter 7, are efficient in practice if critical sections are short and contention is low. Nonetheless, suspension-based protocols are still needed to support shared resources that inherently cause critical sections to be long (*e.g.*, accesses to stable storage), as spinning would result in substantial wastage in such cases. Additionally, suspension-based protocols may be the only option in OSs that do not allow non-preemptable execution or that do not implement link-based scheduling (*e.g.*, Linux supports neither), which are both necessary preconditions for the analysis presented in Chapter 5 (link-based scheduling is only required if $c > 1$).

When suspension-based locking protocols are used in real-time systems, bounds on pi-blocking are required during schedulability analysis. While pi-blocking is well-understood in the context of uniprocessor real-time systems, the same is not true in the multiprocessor case. Because the definition of pi-blocking is rooted in the notion of a "priority inversion," a formal definition of the former requires a formal definition of the latter. While the notion of a priority inversion is straightforward to define in the uniprocessor case (see Definition 2.8), we argue in Section 6.1.1 that an appropriate definition in the multiprocessor case hinges on whether schedulability analysis is *suspension-oblivious* (*s-oblivious)* or *suspension-aware* (*s-aware)*.

The former type, s-oblivious schedulability analysis, does not allow for self-suspension times to be explicitly accounted for. This lack of expressivity in the task model necessitates such times to be

---

* Contents of this chapter previously appeared in preliminary form in the following papers:
Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57;
Brandenburg, B. and Anderson, J. (2010a). Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st Real-Time Systems Symposium*, pages 49–60; and
Brandenburg, B. and Anderson, J. (2011). Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the International Conference on Embedded Software*.

modeled as computation instead. Consequently, s-oblivious analysis over-estimates the processor demand of resource-sharing tasks and thereby yields pessimistic but sound results. In contrast, s-aware schedulability analysis considers suspensions explicitly and thus uses less-pessimistic estimates of processor demand.

To classify locking protocols, we introduce *maximum pi-blocking* as a blocking complexity measure (Section 6.1.2). Our complexity bounds apply to a system of n abitrary-deadline sporadic tasks scheduled by a clustered JLFP scheduler on m processors, where the number of critical sections per job and the length of each critical section are taken to be constant, and resource requests are not nested (or, equivalently, resource nesting is dealt with by group locking—recall Section 5.1).

For s-oblivious analysis, we establish a lower bound of $\Omega(m)$ on maximum pi-blocking (per resource request) for any m-processor locking protocol (under any JLFP scheduler). We further introduce a family of locking protocols for clustered JLFP scheduling that ensures $O(m)$ maximum s-oblivious pi-blocking and is thus asymptotically optimal (Section 6.2).

Perhaps surprisingly, the improvement in analysis accuracy in s-aware analysis comes at the cost of an *increased* lower bound for mutex protocols: we establish a lower bound of $\Omega(n)$ on maximum pi-blocking (assuming $n \geq m$), and present a variant of the FMLP that achieves $O(n)$ maximum s-aware pi-blocking under partitioned JLFP scheduling (Section 6.3). In the case of $c > 1$, no protocol with provably $O(n)$ maximum s-aware pi-blocking under arbitrary JLFP schedulers is currently known, however, the global FMLP is shown to ensure $O(n)$ s-aware pi-blocking in the case of G-EDF with constrained deadlines and no tardiness (Section 6.3.2).

The difference in lower bounds under s-aware and s-oblivious schedulability analysis arises because the nature of what constitutes a "priority inversion" is changed by the assumption underlying s-oblivious analysis. Intuitively, the analytical trick is to "reuse" some of the pessimism inherent in treating suspensions as execution time to derive less pessimistic bounds on priority inversion length. This idea is formalized in the next section.

6.1 Blocking Optimality

The main goal in the design of suspension-based real-time locking protocols is to minimize the worst-case duration of priority inversions, which (intuitively) occur when a high-priority job must wait for a lower-priority one.

In the uniprocessor case, locking protocols that ensure a *provably* optimal upper bound on pi-blocking have long been known. Indeed, under the NCP, PCP (Sha *et al.*, 1990; Rajkumar, 1991), and SRP (Baker, 1991), jobs block for the duration of at most one (outermost) critical section, which is obviously asymptotically optimal.

In the multiprocessor case, however, the situation is much more murky, despite the considerable body of prior work on multiprocessor real-time locking protocols (reviewed in Section 2.4.4). In fact, to the best of our knowledge, general, *precise* definitions of what actually constitutes "blocking" in this case had not been formalized prior to our work. Rather, existing protocols have been analyzed using informally defined notions of blocking; to the effect that different locking protocols were analyzed using different assumptions.[1] Without a precise definition of blocking, we clearly have no understanding of what constitutes *optimal* pi-blocking on multiprocessors.

Motivated by these considerations, we next formalize two notions of pi-blocking and establish lower bounds on maximum pi-blocking under each definition to provide a clear foundation for the study of suspension-based locking protocols.

6.1.1 Priority Inversions in Multiprocessor Systems

We need to clarify the concept of a "priority inversion on a multiprocessor," which is complicated by the fact that multiprocessor schedulability analysis has not yet matured to the point that suspensions can be analyzed under all schedulers. In particular, none of the major G-EDF schedulability tests reviewed in Chapter 2 inherently accounts for self-suspensions. Such analysis is *suspension-oblivious* (*s-oblivious*): jobs may suspend, but each e_i must be inflated by b_i prior to applying the test to account for all additional delays. This approach is safe—converting execution time to idle time does not increase response times—but pessimistic, as even suspended jobs are (implicitly)

[1] Easwaran and Andersson (2009) provide a definition of "job blocking" that conceptually resembles our notion of s-aware pi-blocking (Definition 6.2). However, their definition specifically applies to G-FP scheduling and does not encompass all of the effects that we consider to be "blocking" (*e.g.*, such as priority boosting).

considered to prevent lower-priority jobs from being scheduled. In contrast, *suspension-aware* (*s-aware*) schedulability analysis that explicitly accounts for b_i is available for FP, P-FP, and, to some extent, for G-FP scheduling (Rajkumar, 1991; Audsley *et al*., 1993; Easwaran and Andersson, 2009; Lakshmanan *et al*., 2009). Notably, suspended jobs are *not* considered to occupy a processor under s-aware analysis.

Consequently, priority inversion is defined differently under s-aware and s-oblivious analysis: since suspended jobs are counted as demand under s-oblivious analysis—the maximum suspension time of each such job is included in its execution requirement e_i—the mere *existence* of m pending higher-priority jobs rules out a priority inversion. In contrast, under s-aware schedulability analysis only *ready* higher-priority jobs can nullify a priority inversion (since suspension times are not included in e_i).

The difference in what constitutes a priority inversion leads to two notions of pi-blocking. Since schedulability tests are applied on a cluster-by-cluster basis, pi-blocking is defined in both cases with respect to the tasks in each cluster. Recall from Section 2.4.1 that P_i denotes the cluster that T_i has been assigned to, and that τ_{P_i} denotes the set of tasks assigned to cluster P_i.

Definition 6.1. Under **s-oblivious** schedulability analysis, a job J_i incurs *s-oblivious pi-blocking* at time t if J_i is pending but not scheduled and fewer than c higher-priority jobs of tasks in τ_{P_i} are **pending**.

Definition 6.2. Under **s-aware** schedulability analysis, a job J_i incurs *s-aware pi-blocking* at time t if J_i is pending but not scheduled and fewer than c higher-priority ready jobs of tasks in τ_{P_i} are **scheduled**.

In both cases, "higher-priority" is interpreted with respect to base priorities. In the case of $c = 1$, Definition 6.2 reduces to the uniprocessor definition of pi-blocking (recall Definition 2.8 from Section 2.4.2). Notice that Definition 6.1 is weaker than Definition 6.2. Thus, lower bounds on s-oblivious pi-blocking apply to s-aware pi-blocking as well, and the converse is true for upper bounds.

Example 6.1. The difference between s-oblivious and s-aware pi-blocking is illustrated in Figure 6.1, which shows a G-EDF schedule of three jobs sharing one resource. Job J_1 suffers acquisition delay during $[1, 3)$, and since no higher-priority jobs exist it is pi-blocked under either definition. Job J_3 is

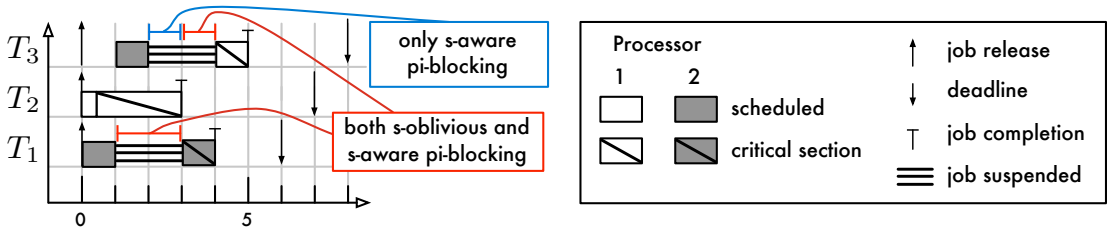


Figure 6.1: Example of s-oblivious and s-aware pi-blocking in a G-EDF schedule of three jobs sharing one resource on $m = c = 2$ processors.

suspended during $[2, 4)$. It suffers pi-blocking under either definition during $[3, 4)$ since it is among the $c = m = 2$ highest-priority pending jobs. However, J_3 suffers only s-aware pi-blocking during $[2, 3)$ since J_1 is pending but not ready then. ◊

6.1.2 A Blocking Complexity Measure

As mentioned above, the principal goal in designing a real-time locking protocol is to minimize pi-blocking. Some degree of pi-blocking is inherently unavoidable if (some) resource accesses require mutual exclusion. A locking protocol, however, must strike a balance between favoring resource requests of some jobs over those of others. For example, in the extreme, a protocol could guarantee a task to never incur pi-blocking if resources are never granted to other tasks. Clearly, such a protocol is not useful, but it highlights that just considering the pi-blocking bound of high-priority (or privileged) tasks is not representative of a locking protocol's blocking behavior.

To compare locking protocols, we thus consider *maximum pi-blocking*, formally $\max_{1 \leq i \leq n}\{b_i\}$, to characterize a protocol's overall blocking behavior. Maximum pi-blocking reflects the per-task bound required for schedulability analysis of the task that incurs the most pi-blocking. It is worth emphasizing that it does *not* necessarily reflect the maximum acquisition delay, which is irrelevant from a schedulability analysis point of view (recall Section 2.4.2).

Concrete bounds on pi-blocking must necessarily depend on each $L_{i,k}$—long requests will cause long priority inversions under any protocol. Similarly, bounds for any reasonable protocol grow linearly with the total number of requests per job. Thus, when deriving asymptotic bounds, we consider, for each T_i, $\sum_{1 \leq q \leq n_r} N_{i,q}$ and each $L_{i,q}$ to be constants and assume $n \geq m$. All other parameters are considered variable (or dependent on m and n). In particular, we do not impose

constraints on the ratio $\max\{p_i\}/\min\{p_i\}$, the number of resources n_r, or the number of tasks sharing each ℓ_q.

To simplify our notation, we use the following shorthand when deriving asymptotic bounds.

Definition 6.3. The maximum critical section length is denoted as L^{max}, where

$$L^{max} \triangleq \max_{1 \leq i \leq n} \max_{1 \leq q \leq n_r} \{L_{i,q}\}.$$

To reiterate, we assume $L^{max} = O(1)$.

In accordance with the goal of minimal pi-blocking, we seek to design protocols under which the amount of time lost to pi-blocking (by *any* task set) is bounded within a constant factor of the loss shown to be unavoidable in the worst case (for some task sets). To this end, we next establish lower bounds on maximum pi-blocking under s-oblivious and s-aware schedulability analysis.

6.1.3 Lower Bound on Maximum S-Oblivious Pi-Blocking

In the case of s-oblivious schedulability analysis, $\Omega(m)$ maximum pi-blocking is unavoidable in some cases. Consider the following pathological high-contention task set.

Definition 6.4. Let $\tau^{seq}(n)$ denote a task set of n identical tasks that share one resource ℓ_1 such that $e_i = 1$, $p_i = 2n$, $N_{i,1} = 1$, and $L_{i,1} = 1$ for each T_i, where $n \geq m \geq 2$.

Lemma 6.1. *There exists an arrival sequence for $\tau^{seq}(n)$ such that, under s-oblivious analysis, $\max_{1 \leq i \leq n}\{b_i\} = \Omega(m)$ under any locking protocol and JLFP scheduler.*

Proof. Without loss of generality, assume that n is an integer multiple of m. Consider the schedule resulting from the following periodic arrival sequence: each $J_{i,j}$ is released at time $a_{i,j} = (\lceil i/m \rceil - 1) \cdot m + (j-1) \cdot p_i$, and issues one request $\mathcal{R}_{i,1,j}$, where $\mathcal{L}_{i,1,j} = 1$.[2] That is, releases occur in groups of m jobs and each job requires ℓ_1 for its entire computation. The resulting G-EDF schedule is illustrated in Figure 6.2.

There are n/m groups of m tasks each that release jobs simultaneously. Each group of jobs of $T_{g \cdot m+1}, \ldots, T_{g \cdot m+m}$, where $g \in \{0, \ldots, n/m - 1\}$, issues m concurrent requests for ℓ_1. Since

[2]Recall from Section 5.4.2 that $\mathcal{R}_{i,q,v}$ denotes the v^{th} request of T_i for resource ℓ_q, and that $\mathcal{L}_{i,q,v}$ denotes the request length of $\mathcal{R}_{i,q,v}$.

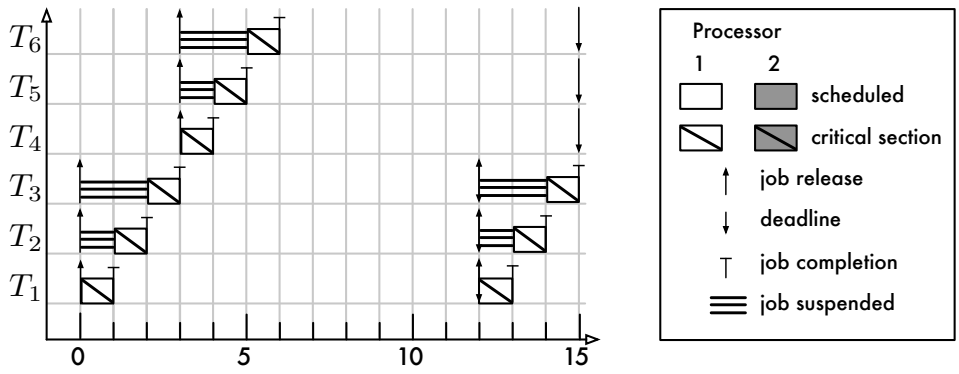


Figure 6.2: Illustration of Lemma 6.1. The depicted example shows a G-EDF schedule of $\tau^{seq}(n)$ for $n = 6$ and $m = 3$, and thus $g \in \{0, 1\}$. The first group of jobs $(J_{1,1}, J_{2,1}, J_{3,1})$ is released at time 0; the second group $(J_{4,1}, J_{5,1}, J_{6,1})$ is released at time 3. Each group incurs $0 + 1 + 2 = \sum_{i=0}^{m-1} i$ total s-oblivious pi-blocking.

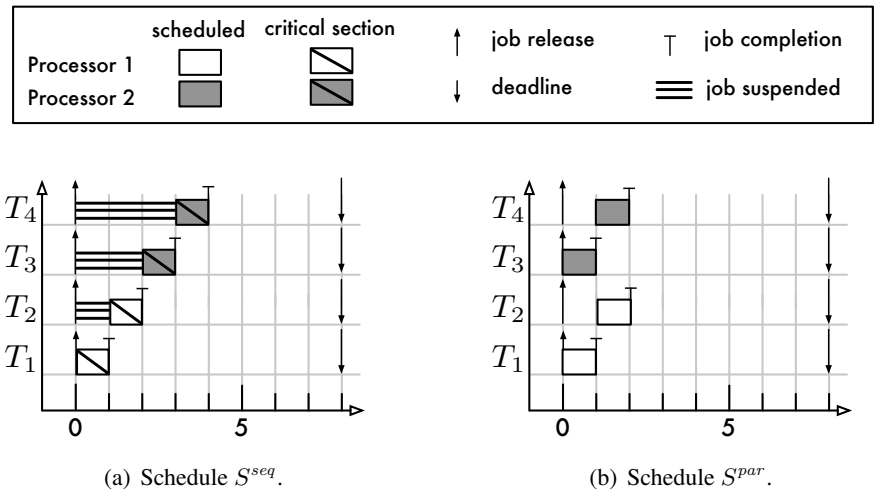
ℓ_1 cannot be shared, any locking protocol must impart some order, and thus there exists a job in each group that incurs d time units of pi-blocking for each $d \in \{0, \ldots, m-1\}$. Hence, for each g, $\sum_{i=g \cdot m+1}^{g \cdot m+m} b_i \geq \sum_{i=0}^{m-1} i = \Omega(m^2)$, and thus, across all groups,

$$\begin{aligned}\sum_{i=1}^{n} b_i &= \sum_{g=0}^{(n/m-1)} \sum_{i=g \cdot m+1}^{g \cdot m+m} b_i \\ &= n/m \cdot \Omega(m^2) \\ &= \Omega(nm),\end{aligned}$$

which implies $\max_{1 \leq i \leq n}\{b_i\} = \Omega(m)$.

By construction, the schedule does not depend on G-EDF scheduling since no more than m jobs are pending at any time, and thus applies to other global JLFP schedulers as well. The lower bound applies equally to clustered JLFP schedulers with $c < m$ since $\tau^{seq}(n)$ can be trivially partitioned such that each processor serves at least $\lfloor n/c \rfloor$ and no more than $\lceil n/c \rceil$ tasks. □

This bound is known to be tight for spin-based protocols: as discussed in Chapter 5, if jobs busy-wait non-preemptively in FIFO order, then they must wait for at most $m - 1$ earlier requests. However, prior work has not yielded an $O(m)$ suspension-based protocol. We present two protocols that are asymptotically optimal under s-oblivious analysis in Sections 6.2.2 and 6.2.5.



(a) Schedule S^{seq}. (b) Schedule S^{par}.

Figure 6.3: Illustration of Lemma 6.2 for $n = 4$ and $m = 2$ under P-EDF. Note that $b_i \geq r_i^{seq} - r_i^{par}$. For example, J_4 incurs pi-blocking during $[0, 2)$ in S^{seq}; consequently $b_4 \geq r_4^{seq} - r_4^{par} = 4 - 2 = 2$. Similarly, $b_3 \geq r_3^{seq} - r_3^{par} = 3 - 1 = 2$.

6.1.4 Lower Bound on Maximum S-Aware Pi-Blocking

Under s-aware schedulability analysis, $O(m)$ maximum pi-blocking is impossible. That is, asymptotically speaking, suspension-based locking protocols are at a disadvantage under s-aware schedulability analysis compared to non-preemptive FIFO spinlocks, which ensure $O(m)$ s-blocking and hence also $O(m)$ pi-blocking. To show this, we next establish that maximum s-aware pi-blocking of $\Omega(n)$ is fundamental under s-aware schedulability analysis.

Lemma 6.2. *There exists an arrival sequence for $\tau^{seq}(n)$ (see Definition 6.4) such that, under s-aware analysis, $\max_{1 \leq i \leq n}\{b_i\} = \Omega(n)$ under any locking protocol and JLFP scheduler.*

Proof. Without loss of generality, assume that n is an integer multiple of m. We first consider the partitioned case ($c = 1$) and assume that $P_i = \lceil i/m \rceil$, *i.e.*, n/m tasks are assigned to each processor.

Consider the schedule S^{seq} resulting from a synchronous, periodic arrival sequence: each $J_{i,j}$ is released at $a_{i,j} = (j-1) \cdot p_i$, and issues one request $\mathcal{R}_{i,1,j}$, where $\mathcal{L}_{i,1,j} = 1$. S^{seq} is illustrated in Figure 6.3(a) assuming P-EDF scheduling. Note that linear suspension times are immediately apparent. However, to bound b_i under *any* JLFP scheduler, we need to take into account the times in which a suspended job is not pi-blocked because a higher-priority job executes.

To this end, consider the schedule S^{par} resulting from the same arrival sequence if jobs are independent, *i.e.*, each J_i executes for e_i without requesting ℓ_1. S^{par} is illustrated in Figure 6.3(b).

Under S^{seq}, as jobs are serialized by ℓ_1, only one job completes every time unit until no jobs are pending; thus, $\sum_{i=1}^{n} r_i^{seq} = \sum_{i=1}^{n} i$ irrespective of how requests are ordered or jobs prioritized.

Under S^{par}, as jobs are independent and the scheduler is, by assumption, work-conserving, m jobs complete concurrently every time unit until no jobs are pending; thus, under any job prioritization, $\sum_{i=1}^{n} r_i^{par} = \sum_{i=1}^{n} \lceil i/m \rceil$.

By construction, no job is pi-blocked in S^{par}. In contrast, jobs incur pi-blocking in S^{seq} under the *same* JLFP *scheduler*, *i.e.*, jobs are prioritized consistently in S^{par} and S^{seq}. Thus, the observed response time increase of every job reflects the amount of pi-blocking incurred in S^{seq}. Therefore, for each T_i, $b_i \geq r_{i,1}^{seq} - r_{i,1}^{par}$, and thus

$$\begin{aligned}
\sum_{i=1}^{n} b_i &\geq \sum_{i=1}^{n} r_{i,1}^{seq} - \sum_{i=1}^{n} r_{i,1}^{par} \\
&= \sum_{i=1}^{n} i - \sum_{i=1}^{n} \left\lceil \frac{i}{m} \right\rceil \\
&\geq \sum_{i=1}^{n} i - \frac{1}{m} \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 \\
&= \left(1 - \frac{1}{m}\right)(n+1)\frac{n}{2} - n \\
&= \Omega(n^2).
\end{aligned}$$

This implies $\max_{1 \leq i \leq n}\{b_i\} = \Omega(n)$. Since at most one job is scheduled in S^{seq} at any time, pi-blocking does not decrease under global or clustered scheduling with $c > 1$. □

In Section 6.3, we discuss a variant of the FMLP in the context of s-aware schedulability analysis and show it to be asymptotically optimal.

6.2 Locking under S-Oblivious Schedulability Analysis

In this section, we present four locking protocols designed specifically for s-oblivious analysis that are collectively called the *family of $O(m)$ locking protocols* (the OMLP family). It includes a mutex protocol, an RW protocol, and a k-exclusion protocol for clustered scheduling with arbitrary cluster

sizes ($1 \leq c \leq m$), and a mutex protocol for the special case of global scheduling (Brandenburg and Anderson, 2010a, 2011).[3] The OMLP family's main features are the following.

- Both mutex protocols ensure maximum pi-blocking that is optimal within a factor that approaches two under s-oblivious analysis. All previously proposed suspension-based locking protocols are suboptimal with respect to maximum pi-blocking under s-oblivious analysis.
- The OMLP's RW and k-exclusion variants are the first suspension-based multiprocessor locking protocols of their kind (prior work on suspension-based multiprocessor locking protocols was focused on mutex constraints).
- The RW protocols ensure maximum pi-blocking for writers that is optimal within a factor that approaches four for large m under s-oblivious analysis (the lower bounds on maximum pi-blocking do not apply to readers since they assume mutual exclusion—see Section 6.2.6 below).
- The k-exclusion protocol ensures $O(\frac{m}{\min\{k_q\}})$ maximum pi-blocking. In the worst case of $\min\{k_q\} = 1$, the ensured bound on maximum pi-blocking is optimal within a factor that approaches two for large m under s-oblivious analysis.
- The OMLP is the first published suspension-based locking protocol that has been designed and analyzed for the case of $1 < c < m$ (prior work addressed only global or partitioned scheduling).

The last point, support for truly clustered scheduling, poses significant challenges from a locking perspective because clusters with $1 < c < m$ exhibit aspects of both partitioned and global scheduling, which seem to necessitate fundamentally different means for bounding priority inversions. We begin by describing a novel method for controlling priority inversions that is key to the OMLP's optimality.

6.2.1 Resource-Holder Progress

To prevent maximum pi-blocking from becoming unbounded or unsuitably large (*i.e.*, bounds should not include job execution costs in addition to request lengths), a locking protocol must ensure

[3] In the initial description of the OMLP (Brandenburg and Anderson, 2010a), we also proposed a variant of the OMLP for partitioned scheduling ($c = 1$). This special case is not considered herein because it has since been superseded by the OMLP mutex protocol for clustered scheduling (Brandenburg and Anderson, 2011).

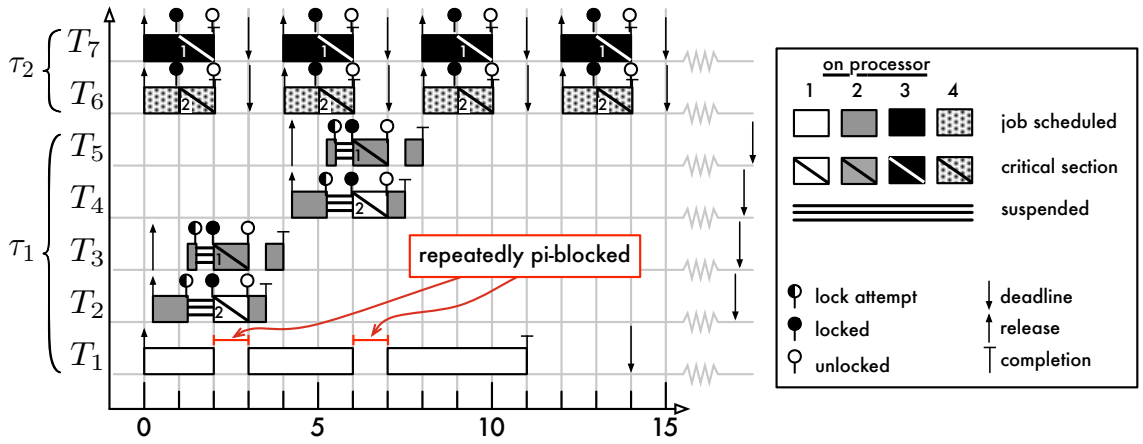


Figure 6.4: Example schedule of seven tasks sharing two resources (ℓ_1, ℓ_2) across two two-processor clusters under C-EDF scheduling. The digit within each critical section indicates which resource was requested. The example shows that priority boosting may cause jobs to incur pi-blocking repeatedly if $c > 1$. If $c = 1$, then lower-priority jobs cannot issue requests while higher-priority jobs execute and repeated pi-blocking due to priority boosting is not an issue.

that resource-holding jobs progress in their execution when high-priority jobs are waiting. That is, low-priority jobs must be scheduled in spite of their low base priority when they cause other higher-priority jobs to incur pi-blocking. A real-time locking protocol thus requires a mechanism to raise the effective priority of resource holders, either on demand (when a waiting job incurs pi-blocking) or unconditionally. As discussed in Section 2.4.4, all prior protocols employ priority inheritance and priority boosting to this end—unfortunately, neither generalizes to clustered scheduling.

Since priority inheritance is ineffective with regard to bounding priority inversion length when applied across partition or cluster boundaries (recall Figure 2.37 on page 125), all prior protocols for partitioned scheduling instead rely on priority boosting to ensure resource-holder progress. Priority boosting prevents preempted jobs from transitively delaying waiting higher-priority jobs by unconditionally raising the effective priority of resource-holding jobs above that of non-resource-holding jobs. While conceptually simple, the unconditional nature of priority boosting may itself cause pi-blocking. Under partitioning ($c = 1$), this effect can be controlled such that jobs incur at most $O(m)$ s-oblivious pi-blocking (Brandenburg and Anderson, 2010a), but this approach does not extend to $c > 1$. This is best illustrated with an example.

Example 6.2. For the sake of simplicity, suppose that requests are satisfied in FIFO order, and that a resource holder's priority is boosted. A possible result is shown in Figure 6.4: jobs of tasks in τ_2

repeatedly request ℓ_1 and ℓ_2 in a pattern that causes low-priority jobs of tasks $T_2, \ldots, T_5$ in τ_1 to be priority-boosted simultaneously. Whenever there are $c = 2$ jobs priority-boosted at the same time, J_1 is necessarily preempted, which causes it to be pi-blocked repeatedly. In general, as c jobs must be priority-boosted to force a preemption, priority boosting may cause $\Omega(\frac{n}{c})$ pi-blocking, which makes it unsuitable for constructing a protocol with $O(m)$ maximum pi-blocking. $\Diamond$

Priority donation. To overcome this limitation, we devised a novel mechanism for ensuring resource-holder progress named *priority donation* that ensures the following two properties.

P1 A resource-holding job is always scheduled.

P2 The duration of s-oblivious pi-blocking caused by the progress mechanism (*i.e.*, the rules that maintain P1) is bounded by the maximum request span (with regard to any job).

Priority boosting unconditionally forces resource holders to be scheduled (Property P1), but it does not specify which job will be preempted as a result. As Figure 6.4 shows, if $c > 1$, this is problematic since an "unlucky" job (like J_1) can repeatedly be a preemption "victim," thereby invalidating P2. Priority donation is a form of priority boosting in which the "victim" is predetermined such that each job is preempted at most once. This is achieved by establishing a *donor relationship* when a potentially harmful job release occurs (*i.e.*, one that could invalidate P1). In contrast to priority boosting, priority donation only takes effect when needed.

Request rule. In the following, let J_i denote a job that requires a resource ℓ_q at time t_1, as illustrated in Figure 6.5. In the examples and the discussion below, we assume mutex locks for the sake of simplicity; however, the proposed protocol applies equally to RW and k-exclusion locks. Priority donation achieves P1 and P2 for $1 \leq c \leq m$ in two steps: it first requires that J_i has a high base priority, and then ensures that J_i's effective priority remains high until J_i releases ℓ_q.

D1 J_i may issue a request only if it is among the c highest-priority pending jobs in its cluster (with regard to base priorities). If necessary, J_i suspends until it may issue a request.

Rule D1 ensures that a job has sufficient priority to be scheduled without delay at the time of request. That is, Property P1 holds at time t_2 in Figure 6.5. However, some—but not all—later job releases during $[t_2, t_4]$ could preempt J_i.

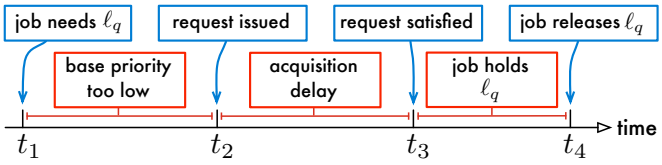


Figure 6.5: Illustration of the request phases under priority donation. A job J_i requires a resource ℓ_q at time t_1. J_i suspends until time t_2, when it becomes one of the c highest-priority pending jobs in its assigned cluster (Rule D1). J_i remains suspended while it suffers acquisition delay from t_2 until its request is satisfied at t_3. Priority donation ensures that J_i is continuously scheduled in $[t_3, t_4)$.

Consider a list of all pending jobs in J_i's cluster sorted by decreasing base priority, and let x denote J_i's position in this list at time t_2. In other words, J_i is the x^{th} highest-priority pending job at time t_2. By Rule D1, $x \leq c$. If there are at most $c - x$ higher-priority jobs released during $[t_2, t_4]$, then J_i remains among the c highest-priority pending jobs and no protocol intervention is required. However, when J_i is the c^{th} highest-priority pending job in its cluster, a higher-priority job release may cause J_i to be preempted or to have insufficient priority to be scheduled when it resumes, thereby violating P1. Priority donation intercepts such releases.

Donor rules. A *priority donor* is a job that suspends to allow a lower-priority job to complete its request. Each job has at most one priority donor at any time. We define how jobs become donors and when they suspend next and illustrate the rules with an example thereafter. Let J_d denote J_i's priority donor (if any), and let t_a denote J_d's release time.

D2 J_d becomes J_i's priority donor at time t_a if **(a)** J_i was the c^{th} highest-priority pending job prior to J_d's release (with regard to its cluster), **(b)** J_d has one of the c highest base priorities, and **(c)** J_i has issued a request that is incomplete at time t_a (that is, $t_a \in [t_2, t_4]$ with regard to J_i's request).

D3 J_i inherits the priority of J_d (if any) during $[t_2, t_4)$.

The purpose of Rule D3 is to ensure that J_i will be scheduled if ready. However, J_d's relative priority could decline due to subsequent releases. In this case, the donor role is passed on.

D4 If J_d is displaced from the set of the c highest-priority jobs by the release of J_h, then J_h becomes J_i's priority donor and J_d ceases to be a priority donor. (By Rule D3, J_i thus inherits J_h's priority.)

Rule D4 ensures that J_i remains among the c highest-priority *pending* jobs (with regard to its cluster). The following two rules ensure that J_i and J_d are never ready at the same time, thereby freeing a processor for J_i to be scheduled on.

D5 If J_i is ready when J_d becomes J_i's priority donor (by either Rule D2 or D4), then J_d suspends immediately.

D6 If J_d is J_i's priority donor when J_i resumes at time t_3, then J_d suspends (if ready).

Further, a priority donor may not execute a request itself and may not prematurely exit.

D7 A priority donor may not issue requests. J_d suspends if it requires a resource while being a priority donor.

D8 If J_d finishes execution while being a priority donor, then its completion is postponed, *i.e.*, J_d suspends and remains pending until it is no longer a priority donor.

J_d may continue once its donation is no longer required, or when a higher-priority job takes over.

D9 J_d ceases to be a priority donor as soon as either **(a)** J_i completes its request (*i.e.*, at time t_4), **(b)** J_i's base priority becomes one of the c highest (with regard to pending jobs in J_i's cluster), or **(c)** J_d is relieved by Rule D4. If J_d suspended due to Rules D5–D7, then it resumes.

Under a JLFP scheduler, Rule D9b can only be triggered when higher-priority jobs complete.

Example 6.3. Figure 6.6 shows a resulting schedule assuming jobs wait in FIFO order. Priority donation occurs first at time 3, when the release of J_1 displaces J_3 from the set of the $c = 2$ highest-priority pending jobs of tasks in τ_1. Since J_3 holds ℓ_1, J_1 becomes J_3's priority donor (Rule D2) and suspends immediately since J_3 is ready (Rule D5). J_1 resumes when its duties cease at time 6 (Rule 9a). If J_1 would not have donated its priority to J_3, then it would have preempted J_3, thereby violating P1.

At time 3, J_6 also requests ℓ_1 and suspends as ℓ_1 is unavailable. It becomes a priority recipient when J_4 is released at time 4 (Rule D2). Since J_6 is already suspended, Rule D5 does not apply and J_4 remains ready. However, at time 5, J_4 requires ℓ_2, but since it is still a priority donor, it may not issue a request and must suspend instead (Rule D7). J_4 may resume and issue its request at time 7 since J_5 finishes, which causes J_6 to become one of the two highest-priority pending jobs of tasks in

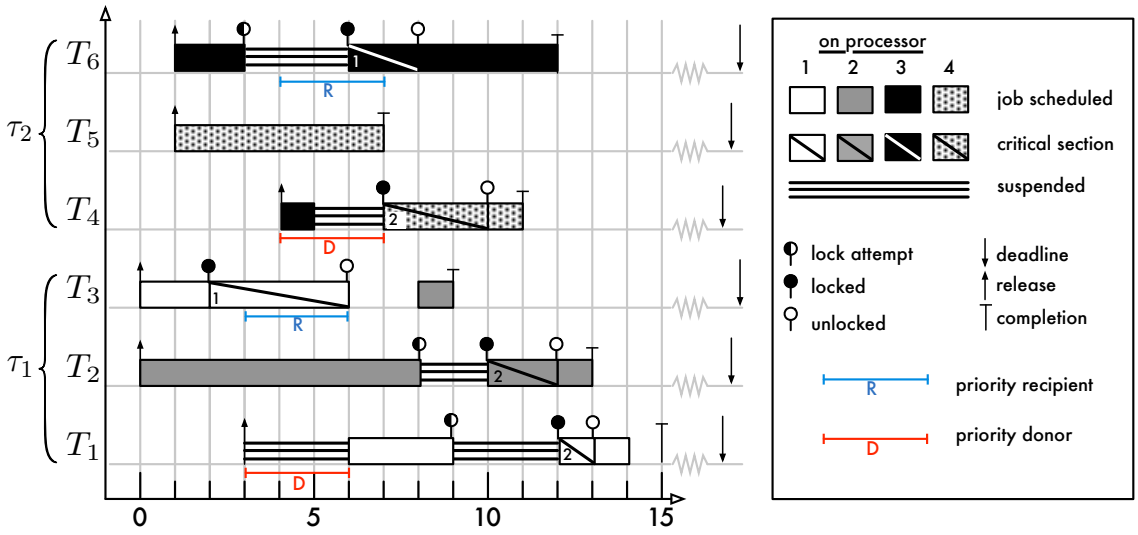


Figure 6.6: Schedule of six tasks sharing two serially-reusable resources across two two-processor clusters under C-EDF scheduling. The digit within each critical section indicates which resource was requested. Under the OMLP for clustered scheduling, progress is ensured with priority donation (Section 6.2.1) and jobs wait in FIFO order (Section 6.2.2).

τ_2 (Rule 9b). If priority donors were allowed to issue requests, then J_4 would have been suspended while holding ℓ_2 when J_6 resumed at time 6, thereby violating P1. ◊

Taken together, Rules D1–D9 ensure resource-holder progress under clustered scheduling with arbitrary cluster sizes ($1 \leq c \leq m$).

Lemma 6.3. *Priority donation ensures Property P1.*

Proof. Rule D7 prevents Rules D5 and D6 from suspending a resource-holding job. Rule D1 establishes Property P1 at time t_2. If J_i's base priority becomes insufficient to guarantee P1, its effective priority is raised by Rules D2 and D3. Rules D4 and D8 ensure that the donated priority is always among the c highest (with regard to pending jobs in J_i's cluster), which, together with Rules D5 and D6, effectively reserves a processor for J_i to run on when ready. □

By establishing the donor relationship at release time, priority donation ensures that a job is a "preemption victim" at most once, even if $c > 1$.

Lemma 6.4. *Priority donation ensures Property P2.*

Proof. A job incurs s-oblivious pi-blocking if it is among the c highest-priority pending jobs in its cluster and either **(i)** suspended or **(ii)** ready and not scheduled (*i.e.*, preempted). We show that (i) is bounded and that (ii) is impossible.

Case (*i*). Only Rules D1 and D5–D8 cause a job to suspend. Rule D1 does not cause s-oblivious pi-blocking: the interval $[t_1, t_2)$ ends as soon as J_i becomes one of the c highest-priority pending jobs. Rules D5–D8 apply to priority donors. J_d becomes a priority donor only immediately upon release or not at all (Rules D2 and D4), *i.e.*, each J_d donates its priority to some J_i only once. By Rule D2, the donor relationship starts no earlier than t_2, and, by Rule D9, ends at the latest at time t_4. By Rules D8 and D9, J_d either resumes or completes when it ceases to be a priority donor. J_d suspends thus for the duration of at most one entire request span.

Case (*ii*). Let J_x denote a job that is ready and among the c highest-priority pending jobs (with regard to base priorities) of tasks in cluster τ_j, but not scheduled. Let A denote the set of ready jobs of tasks in τ_j with higher base priorities than J_x, and let B denote the set of ready jobs of tasks of τ_j with higher effective priorities than J_x that are not in A. Only jobs in A and B can preempt J_x. Let D denote the set of priority donors of jobs in B.

By Rule D3, every job in B has a priority donor that is, by construction, unique: $|B| = |D|$. By assumption, $|A| + |B| \geq c$ (otherwise J_x would be scheduled), and thus also $|A| + |D| \geq c$. By the definition of B, every job in D has a base priority that exceeds J_x's base priority. Rules D5 and D6 imply that no job in D is ready (since every job in B is ready): $A \cap D = \emptyset$. Every job in D is pending (Rule D8), and every job in A is ready and hence also pending. Thus, there exist at least c pending jobs of tasks in τ_j with higher base priority than J_x. Contradiction. □

Priority donation further limits maximum concurrency, which is key to the analysis of the protocols presented next in Sections 6.2.2–6.2.4.

Lemma 6.5. *Let $R_j(t)$ denote the number of requests issued by jobs of tasks in cluster τ_j that are incomplete at time t. Under priority donation, $R_j(t) \leq c$ at all times.*

Proof. Similar to Case (ii) above. Suppose $R_j(t) > c$ at time t. Let H denote the set of the c highest-priority pending jobs of tasks in τ_j (at time t with regard to base priorities), and let I denote the set of jobs of tasks in τ_j that have issued a request that is incomplete at time t.

Let A denote the set of high-priority jobs with incomplete requests, *i.e.*, $A = H \cap I$, and let B denote the set of low-priority jobs with incomplete requests, *i.e.*, $B = I \setminus A$.

Let D denote the set of priority donors of jobs in B. Together, Rules D2, D4, D8, and D9 ensure that every job in B has a unique priority donor. Therefore $|B| = |D|$.

By definition, $|A| + |B| = |I| = R_j(t)$. By our initial assumption, this implies $|A| + |B| > c$ and thus $|A| + |D| > c$. By Rules D2 and D4, $D \subseteq H$ (only high-priority jobs are donors). By Rule D7, $A \cap D = \emptyset$ (donors may not issue requests). Since, by definition, $A \subseteq H$, this implies $|H| \geq |A| + |D| > c$. Contradiction. □

In the following, we show that Lemmas 6.3–6.5 provide a strong foundation that enables the design of simple, yet asymptotically optimal, locking protocols.

6.2.2 Mutual Exclusion under Clustered Scheduling

We begin with ensuring mutex constraints, which is the most straightforward case. An asymptotically optimal mutex protocol can be layered on top of priority donation by using simple FIFO queues just as they are used in non-preemptive spinlocks. The following protocol's simplicity demonstrates that priority donation is a powerful aid for worst-case analysis.

Structure. For each serially-reusable resource ℓ_q, there is a FIFO queue FQ_q that is used to serialize conflicting accesses. The job at the head of FQ_q holds ℓ_q.

Rules. Access to each resource is granted according to the following rules. Let J_i denote a job that issues a request $\mathcal{R}_{i,q,v}$ for ℓ_q.

X1 J_i is enqueued in FQ_q when it issues $\mathcal{R}_{i,q,v}$. If FQ_q was non-empty, then J_i suspends until $\mathcal{R}_{i,q,v}$ is satisfied.

X2 $\mathcal{R}_{i,q,v}$ is satisfied when J_i becomes the head of FQ_q.

X3 J_i is dequeued from FQ_q when $\mathcal{R}_{i,q,v}$ is complete. The new head of FQ_q (if any) is resumed.

Rules X1–X3 correspond to times t_2–t_4 in Figure 6.5.

Example 6.4. Figure 6.6 depicts an example of the clustered OMLP for serially-reusable resources. (Figure 6.6 was previously discussed in the context of priority donation.) At time 2, J_3 requests

ℓ_1 and is enqueued in FQ_1 (Rule X1). Since FQ_1 was empty, J_3's request is satisfied immediately (Rule X2). When J_6 requests the same resource at time 3, it is appended to FQ_1 and suspends. When J_3 releases ℓ_1 at time 6, J_6 becomes the new head of FQ_1 and resumes (Rule X3).

At time 7, J_4 acquires ℓ_2 and enqueues in FQ_2, which causes J_2 and J_1 to suspend when they, too, request ℓ_2 at times 8 and 9. Importantly, priorities are ignored in each FQ_q: when J_4 releases ℓ_2 at time 10, J_2 becomes the resource holder and is resumed, even though J_1 has a higher base priority. While using FIFO queues instead of priority queues in real-time systems may seem counterintuitive, priority queues are in fact problematic in a multiprocessor context since they allow starvation, which renders them unsuitable for constructing protocols with $O(m)$ maximum pi-blocking (see Section 6.3.3). ◊

Priority donation is crucial in two ways: requests complete without delay and maximum contention is limited.

Lemma 6.6. *At most m jobs are enqueued in any FQ_q.*

Proof. By Lemma 6.5, at most c requests are incomplete at any point in time in each cluster. Since there are $\frac{m}{c}$ clusters, no more than $\frac{m}{c} \cdot c = m$ jobs are enqueued in some FQ_q. □

Lemma 6.7. *A job J_i that requests a resource ℓ_q incurs acquisition delay for the duration of at most $m - 1$ requests.*

Proof. By Lemma 6.6, at most $m - 1$ other jobs precede J_i in FQ_q. By Lemma 6.3, the job at the head of FQ_q is always scheduled. Therefore, J_i becomes the head of FQ_q after the combined length of $m - 1$ requests. □

This property suffices to prove asymptotic optimality.

Theorem 6.1. *The clustered OMLP for serially-reusable resources causes a job J_i to incur at most $b_i = m \cdot L^{max} + \sum_{q=1}^{n_r} N_{i,q} \cdot (m-1) \cdot L^{max} = O(m)$ s-oblivious pi-blocking.*

Proof. By Lemma 6.4, the duration of s-oblivious pi-blocking caused by priority donation is bounded by the maximum request span. By Lemma 6.7, maximum acquisition delay per request is bounded by $(m-1) \cdot L^{max}$. The maximum request span is thus bounded by $m \cdot L^{max}$. Recall from Section 6.1.2 that $\sum_{q=1}^{n_r} N_{i,q}$ and L^{max} are presumed constant. The bound follows. □

The protocol for serially-reusable resources is thus asymptotically optimal with regard to maximum s-oblivious pi-blocking. A practical, non-asymptotic bound on maximum pi-blocking that takes individual request lengths and frequencies into account is presented in Section 6.4.

6.2.3 Reader-Writer Exclusion under Clustered Scheduling

Priority donation also lends itself to constructing simple RW locks. In this section, we transfer the concept phase-fair RW locks to suspension-based locks. Other RW request orders such as task-fairness or preference locks could similarly be implemented.

Recall from Section 5.2.1 the following key properties of phase-fairness.

- Reader phases and writer phases alternate (unless there are only requests of one kind).
- At the beginning of a reader phase, all incomplete read requests are satisfied.
- One write request is satisfied at the beginning of a writer phase.
- Read requests are allowed to join a reader phase that is already in progress only if there are no incomplete write requests.

This results in $O(1)$ acquisition delay for read requests without starving write requests. Note that this does not contradict the lower bound on s-oblivious pi-blocking (Lemma 6.1) because the bound depends on mutual exclusion. It thus only applies to write requests (which must be exclusive), but not to read requests (which may be satisfied concurrently with other read requests).

The following rules define a suspension-based phase-fair RW lock. Due to the explicit use of queues, it is structurally similar to the PF-Q lock presented in Section 5.3.3.

Structure. For each RW resource ℓ_q, there are three queues: a FIFO queue for writers, denoted WQ_q, and two reader queues RQ_q^1 and RQ_q^2. Initially, RQ_q^1 is the *collecting* and RQ_q^2 is the *draining* reader queue. The roles, denoted as CQ_q and DQ_q, switch as reader and writer phases alternate; that is, the designations "collecting" and "draining" are not static.

Reader rules. Let J_i denote a job that issues a read request $\mathcal{R}_{i,q,v}^R$ for ℓ_q. The distinction between CQ_q and DQ_q serves to separate reader phases. Readers always enqueue in the (at the time of request) collecting queue. If queue roles change, then a writer phase starts when the last reader releases ℓ_q.

R1 J_i is enqueued in CQ_q when it issues $\mathcal{R}^R_{i,q,v}$. If WQ_q is non-empty, then J_i suspends.

R2 $\mathcal{R}^R_{i,q,v}$ is satisfied either immediately if WQ_q is empty when $\mathcal{R}^R_{i,q,v}$ is issued, or when J_i is subsequently resumed.

R3 Let RQ^y_q denote the reader queue in which J_i was enqueued due to Rule R1. J_i is dequeued from RQ^y_q when $\mathcal{R}^R_{i,q,v}$ is complete. If RQ^y_q is DQ_q and J_i is the last job to be dequeued from RQ^y_q, then the current reader phase ends and the head of WQ_q is resumed (WQ_q is non-empty).

Writer rules. Let J_w denote a job that issues a write request $\mathcal{R}^W_{i,q,v}$ for ℓ_q. Conflicting writers wait in FIFO order. The writer at the head of WQ_q is further responsible for starting and ending reader phases by switching the reader queues.

W1 J_w is enqueued in WQ_q when it issues $\mathcal{R}^W_{i,q,v}$. J_w suspends until $\mathcal{R}^W_{i,q,v}$ is satisfied, unless $\mathcal{R}^W_{i,q,v}$ is satisfied immediately. If WQ_q is empty and CQ_q is not, then the roles of CQ_q and DQ_q are switched.

W2 $\mathcal{R}^W_{i,q,v}$ is satisfied either immediately if WQ_q and CQ_q are both empty when $\mathcal{R}^W_{i,q,v}$ is issued,[4] or when J_w is subsequently resumed.

W3 J_w is dequeued from WQ_q when $\mathcal{R}^W_{i,q,v}$ is complete. If CQ_q is empty, then the new head of WQ_q (if any) is resumed. Otherwise, each job in CQ_q is resumed and, if WQ_q remains non-empty, the roles of CQ_q and DQ_q are switched.

Rules R1–R3 and W1–W3 correspond to times t_2–t_4 in Figure 6.5 (respectively), and are illustrated in Figure 6.7.

Example 6.5. Figure 6.7 depicts six tasks in two clusters sharing one resource. The resource ℓ_1 is first read by J_5, which is enqueued in RQ^1_q, the initial collecting queue, at time 1 (Rule R1). When J_2 issues a read request at time 1, it is also enqueued and its request is satisfied immediately since WQ_1 is still empty (Rule R2). J_1 issues a write request at time 4. Since CQ_1 is non-empty, the roles of CQ_1 and DQ_1 are switched, *i.e.*, RQ^1_q becomes the draining reader queue, and J_1 suspends (Rule W1).

[4] If WQ_q and CQ_q are both empty, then DQ_q is necessarily empty, too, as any readers in the draining queue would have had to enqueue when it was still the collecting queue (Rule R1) and the roles of CQ_q and DQ_q are only switched when a writer is waiting (Rules W1 and W3).

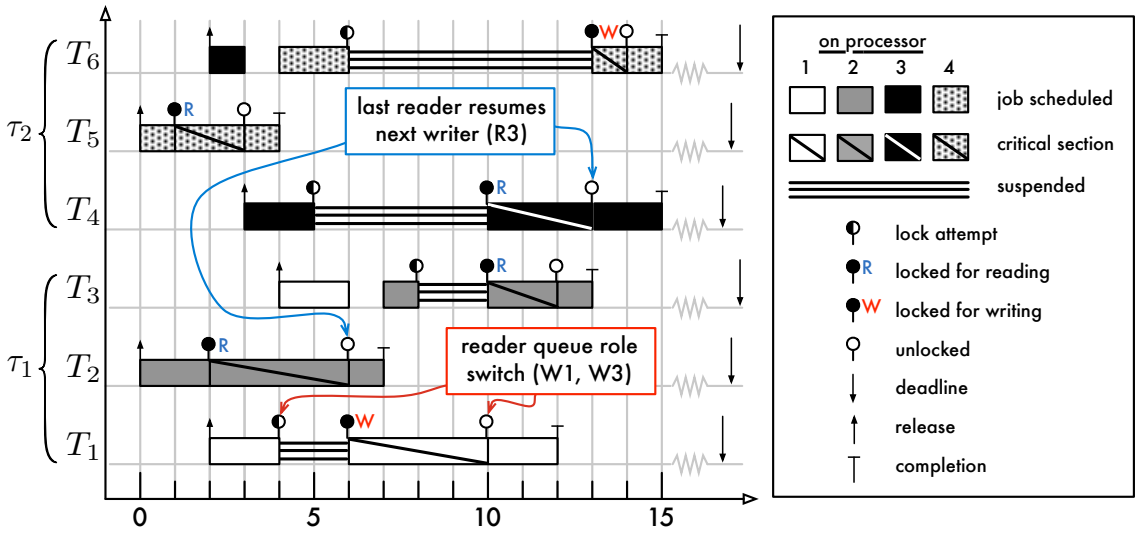


Figure 6.7: Schedule of six tasks sharing one RW resource across two two-processor clusters under C-EDF scheduling. (Priority donation does not occur in this example schedule.)

J_4 issues a read request soon thereafter and is enqueued in RQ_q^2 (Rule R1), which is the collecting queue after the role switch. J_4 suspends since WQ_1 is not empty (Rule R2), even though J_2 is still executing a read request. This is required to ensure that write requests are not starved. The reader phase ends when J_2 releases ℓ_1 at time 6, and the next writer, J_1, is resumed (Rules R3 and W2). J_1 releases ℓ_1 and resumes all readers that have accumulated in RQ_q^2 (J_3 and J_4). Since WQ_1 is non-empty (J_6 was enqueued at time 6), RQ_q^2 becomes the draining reader queue (Rule W3). Under task-fair RW locks, J_3 would have remained suspended since it requested ℓ_1 after J_6. In contrast, J_6 must wait until the next writer phase at time 13 and *all* waiting readers are resumed at the beginning of the next reader phase at time 10 (Rule W3). ◊

Together with priority donation, the reader and writer rules above realize a phase-fair RW lock. Due to the intertwined nature of reader and writer phases, we first consider the head of WQ_q (a writer phase), then CQ_q (a reader phase), and finally the rest of WQ_q.

Lemma 6.8. *Let J_w denote the head of WQ_q. J_w incurs acquisition delay for the duration of at most one read request length before its request is satisfied.*

Proof. J_w became head of WQ_q in one of two ways: by Rule W1 (if WQ_q was empty prior to J_w's request) or by Rule W3 (if J_w had a predecessor in WQ_q). In either case, there was a reader queue role switch when J_w became head of WQ_q (unless there were no unsatisfied read requests, in which

case the claim is trivially true). By Rule R3, if a reader phase delayed J_w, then J_w is resumed as soon as the last reader in DQ_q releases ℓ_q. By Rule R1, no new readers enter DQ_q. Due to priority donation, there are at most $m-1$ jobs in DQ_q (Lemma 6.5), and each job holding ℓ_q is scheduled (Lemma 6.3). The claim follows. □

Lemma 6.9. *Let J_i denote a job that issues a read request for ℓ_q. J_i incurs acquisition delay for the combined duration of at most one read and one write request.*

Proof. If WQ_q is empty, then J_i's request is satisfied immediately (Rule R2). Otherwise, it suspends and is enqueued in CQ_q (Rule R1). This prevents consecutive write phases (Rule W3). J_i's request is thus satisfied as soon as the current head of WQ_q releases ℓ_q (Rule W3). By Lemma 6.8, the head of WQ_q incurs acquisition delay for no more than the length of one read request (which transitively impacts J_i). Due to priority donation, the head of WQ_q is scheduled when its request is satisfied (Lemma 6.3). Therefore, J_i waits for the duration of at most one read and one write request. □

Lemma 6.9 shows that readers incur $O(1)$ acquisition delay. Next, we show that writers incur $O(m)$ acquisition delay.

Lemma 6.10. *Let J_w denote a job that issues a write request for ℓ_q. J_w incurs acquisition delay for the duration of at most $m-1$ write and m read requests before its request is satisfied.*

Proof. It follows from Lemma 6.5 that at most $m-1$ other jobs precede J_w in WQ_q (analogously to Lemma 6.6). By Lemma 6.3, J_w's predecessors together hold ℓ_q for the duration of at most $m-1$ write requests. By Lemma 6.8, each predecessor incurs acquisition delay for the duration of at most one read request once it has become the head of WQ_q. Thus, J_w incurs transitive acquisition delay for the duration of at most $m-1$ read requests before it becomes head of WQ_q, for a total of at most $m-1+1=m$ read requests. □

These properties suffice to prove asymptotic optimality with regard to maximum s-oblivious pi-blocking.

Theorem 6.2. *The clustered OMLP for RW resources causes a job J_i to incur at most*

$$b_i = 2 \cdot m \cdot L^{max} + \sum_{q=1}^{n_r} N_{i,q} \cdot (2 \cdot m - 1) \cdot L^{max} = O(m)$$

s-oblivious pi-blocking.

Proof. By Lemma 6.4, the duration of s-oblivious pi-blocking caused by priority donation is bounded by the maximum request span. By Lemma 6.10, maximum acquisition delay per write request is bounded by $(2m-1) \cdot L^{max}$; by Lemma 6.9, maximum acquisition delay per read request is bounded by $2 \cdot L^{max}$. The maximum request span is thus bounded by $2 \cdot m \cdot L^{max}$. Recall from Section 6.1.2 that $\sum_{q=1}^{n_r} N_{i,q}$ and L^{max} are constant. The bound follows. □

A detailed, non-asymptotic bound on maximum pi-blocking that takes individual request lengths and frequencies into account is given in Section 6.4.

6.2.4 k-Exclusion under Clustered Scheduling

For some resource types, one option to reduce contention is to *replicate* them. For example, if potential overload of a DSP co-processor is found to pose a risk in the design phase, the system designer could introduce additional instances to improve response times.

As with multiprocessors, there are two fundamental ways to allocate replicated resources: either each task may only request a specific replica, or every task may request any replica. The former approach, which corresponds to partitioned scheduling, has the advantage that a mutex protocol suffices, but it also implies that some resource replicas may idle while jobs wait to acquire their designated replica. The latter approach, equivalent to global scheduling, avoids such bottlenecks, but needs a k-exclusion protocol to do so. Priority donation yields such a protocol for clustered scheduling.

Recall that k_q is the number of replicas of resource ℓ_q. In the following, we assume $1 \leq k_q \leq m$. The case of $k_q > m$ is discussed in Section 6.2.6 below.

Structure. Jobs waiting for a replicated resource ℓ_q are kept in a FIFO queue denoted as KQ_q. The replica set RS_q contains all idle instances of ℓ_q. If $\text{RS}_q \neq \emptyset$, then KQ_q is empty.

Rules. Let J_i denote a job that issues a request $\mathcal{R}_{i,q,v}$ for ℓ_q.

K1 If $\text{RS}_q \neq \emptyset$, then J_i acquires an idle replica from RS_q. Otherwise, J_i is enqueued in KQ_q and suspends.

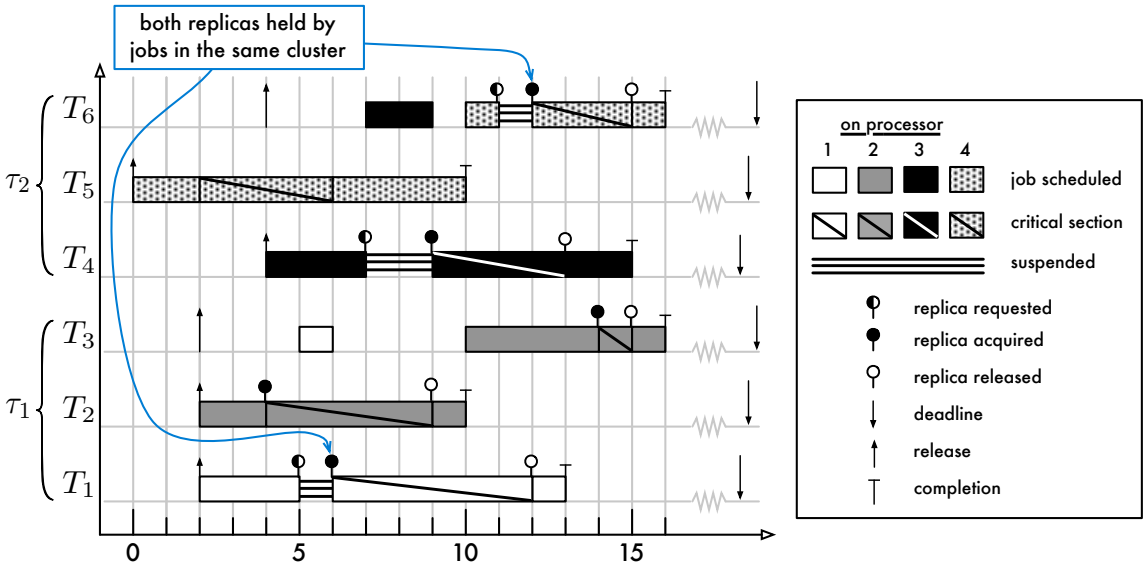


Figure 6.8: Six tasks sharing two instances of one resource across two two-processor clusters under C-EDF scheduling. (Priority donation does not occur in this particular example.)

K2 $\mathcal{R}_{i,q,v}$ is satisfied either immediately (if $\text{RS}_q \neq \emptyset$ at the time of request) or when J_i is removed from KQ_q.

K3 If KQ_q is non-empty when $\mathcal{R}_{i,q,v}$ completes, the head of KQ_q is dequeued, resumed, and acquires J_i's replica. Otherwise, J_i's replica is released into RS_q.

As it was the case with the definition of the previous protocols, Rules K1–K3 correspond to times t_2–t_4 in Figure 6.5.

Example 6.6. Figure 6.8 depicts an example schedule for one resource (ℓ_1) with $k_1 = 2$. J_5 obtains a replica from RS_1 at time 2 (Rule K1). The second replica of ℓ_1 is acquired by J_2 at time 4. As RS_1 is now empty, J_1 is enqueued in KQ_1 and suspends when it requests ℓ_1 at time 5. However, it is soon resumed when J_5 releases its replica at time 6 (Rule K3). This illustrates one advantage of using k-exclusion locks: if instead one replica would have been statically assigned to each cluster (which reduces the resource-sharing problem to a mutex constraint), then J_1 would have continued to wait while τ_2's replica would have idled. This happens again at time 12: since no job of tasks in τ_1 requires ℓ_1 at the time, both instances are used by jobs of tasks in τ_2. ◊

As with the previous protocols, priority donation is essential to ensure progress and to limit contention.

Lemma 6.11. *At most $m - k_q$ jobs are enqueued in KQ_q.*

Proof. Lemma 6.5 implies that there are at most m incomplete requests. Since only jobs waiting for ℓ_q are enqueued in KQ_q, at most $m - k_q$ jobs are enqueued in KQ_q. □

Lemma 6.12. *Let J_i denote a job that issues a request $\mathcal{R}_{i,q,v}$ for ℓ_q. J_i incurs acquisition delay for the duration of at most $\lceil (m - k_q)/k_q) \rceil$ maximum request lengths.*

Proof. By Lemma 6.11, at most $m - k_q$ requests must complete before J_i's request is satisfied ($m - k_q - 1$ for J_i to become the head of KQ_q, and one more for J_i to be dequeued). Rules K1 and K3 ensure that all replicas are in use whenever jobs wait in KQ_q. Since resource holders are always scheduled due to priority donation (Lemma 6.3), requests are satisfied at a rate of at least k_q requests per maximum request length until $\mathcal{R}_{i,q,v}$ is satisfied. The stated bound follows. □

Lemma 6.12 shows that J_i incurs at most $O(\frac{m}{k_q})$ pi-blocking per request (and none if $k_q = m$), which implies asymptotic optimality with regard to maximum s-oblivious pi-blocking.

Definition 6.5. Let $k^{min} \triangleq \min_{1 \leq q \leq r}\{k_q\}$ denote the minimum degree of replication.

Theorem 6.3. *The clustered OMLP for replicated resources causes a job J_i to incur at most*

$$\begin{aligned} b_i &= \left(1 + \left\lceil \frac{m - k^{min}}{k^{min}} \right\rceil\right) \cdot L^{max} + \sum_{q=1}^{n_r} \left(N_{i,q} \cdot \left\lceil \frac{m - k_q}{k_q} \right\rceil\right) \cdot L^{max} \\ &\leq \left(1 + \left\lceil \frac{m - k^{min}}{k^{min}} \right\rceil\right) \cdot L^{max} + \sum_{q=1}^{n_r} \left(N_{i,q} \cdot \left\lceil \frac{m - k^{min}}{k^{min}} \right\rceil\right) \cdot L^{max} \\ &= O(m/k^{min}) \end{aligned}$$

s-oblivious pi-blocking.

Proof. By Lemma 6.12, maximum acquisition delay per request for ℓ_q is bounded by $\lceil (m - k_q)/k_q) \rceil \cdot L^{max}$. The maximum request span is thus bounded by $(\lceil (m - k^{min})/k^{min}) \rceil + 1) \cdot L^{max}$. Lemma 6.4 limits the duration of s-oblivious pi-blocking due to priority donation to the maximum request span. The bound follows since $\sum_{q=1}^{n_r} N_{i,q}$ and L^{max} are constant (Section 6.1.2). □

A detailed, non-asymptotic bound on maximum pi-blocking that takes individual request lengths and frequencies into account is provided in Section 6.4.

Theorem 6.3 implies asymptotic optimality for any $k^{min} \leq m$. While Lemma 6.1 applies only to mutual exclusion (*i.e.*, $k^{min} = 1$), it is trivial to extend the argument to $1 \leq k^{min} \leq m$.

Lemma 6.13. *There exists an arrival sequence for $\tau^{seq}(n)$ such that, under s-oblivious analysis, $\max_{1 \leq i \leq n}\{b_i\} = \Omega(m/k^{min})$ under any k-exclusion locking protocol and JLFP scheduler, where $1 \leq k^{min} < m$.*

Proof. Analogously to Lemma 6.1. Recall from Definition 6.4 that $\tau^{seq}(n)$ consists of n tasks, and that each job of each task requires a shared resource ℓ_1 for the entirety of its computation (*i.e.*, $e_i = L_{i,1} = L^{max} = 1$). If there are $k_1 = k^{min}$ replicas of ℓ_1, then at most k^{min} jobs are scheduled at any time. As in the proof of Lemma 6.1, consider the arrival sequence shown in Figure 6.2: if m jobs request ℓ_1 simultaneously, then any k-exclusion protocol must impart an order among the requests such that only k^{min} requests are satisfied concurrently. To complete each of the m concurrent requests, the k^{min} replicas must be used for $m \cdot L^{max}$ time units in total. This implies that the last request to be satisfied completes no earlier than $m \cdot L^{max}/k^{min}$ time units after it was issued. Therefore, it incurred at least

$$\frac{m \cdot L^{max}}{k^{min}} - L^{max} = \left(\frac{m}{k^{min}} - 1\right) \cdot L^{max}$$

acquisition delay. Further, as each request is sequential and since all requests are of uniform length $L^{max} = 1$, requests are only satisfied at times that are integer multiples of L^{max} (*i.e.*, requests are satisfied only $x \cdot L^{max}$ time units after they are issued, where $0 \leq x \leq \lceil m/k^{min} \rceil - 1$). Therefore, the last of the m concurrent requests to complete was not satisfied until

$$\left\lceil \frac{m}{k^{min}} - 1 \right\rceil \cdot L^{max} = \Omega\left(\frac{m}{k^{min}}\right)$$

time units after the requests were issued. Since at most m jobs are pending at any time in the periodic arrival sequence shown in Figure 6.2, this implies that $\Omega(m/k^{min})$ s-oblivious pi-blocking is unavoidable in the general case. □

The clustered OMLP for replicated resources is hence asymptotically optimal with regard to maximum s-oblivious pi-blocking.

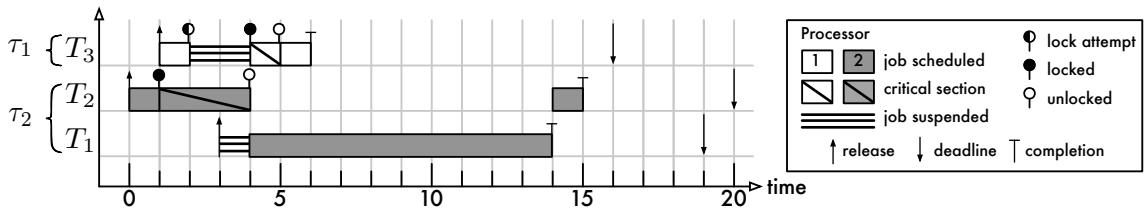


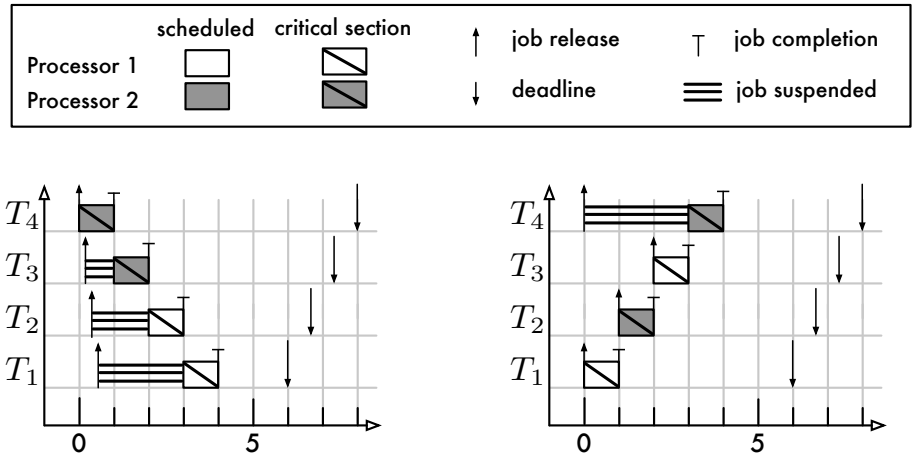
Figure 6.9: Example showing the OMLP family's mutex protocol for clustered scheduling under P-EDF on $m = 2$ processors. Even though job J_1 is independent, it incurs pi-blocking when it serves as J_2's priority donor during $[3, 4)$. This example demonstrates that it is in general unavoidable for independent jobs to be subject to pi-blocking: if J_1 were allowed to preempt J_2 (to avoid being pi-blocked), then J_3 would incur pi-blocking for the entire duration of J_1's execution (*i.e.*, J_3 would incur an "unbounded" priority inversion, just as in Figure 2.37 on page 125).

6.2.5 Mutual Exclusion under Global Scheduling

As demonstrated in the preceding sections, the primary advantage of priority donation is that it enables simple, asymptotically optimal locking protocols. An undesirable property of priority donation is that *every* task is subject to potential pi-blocking—even those that are independent—because any job may be required to serve as a priority donor upon release. While undesirable, this is fundamental to lock-based real-time synchronization if $c < m$, that is, if priority inversions must be bounded, there is more than one cluster, and tasks may not migrate across cluster boundaries. This is illustrated in Figure 6.9. However, under global scheduling (*i.e.*, if $c = m$), it is possible to design locking protocols based on priority inheritance under which independent jobs never incur s-oblivious pi-blocking. In this section, we present such a protocol, namely the *global OMLP* (which realizes mutex constraints).

The OMLP variant for clustered scheduling relies on simple FIFO queues to serialize conflicting resource requests. Unfortunately, when FIFO queues are combined with priority inheritance (which, unlike priority donation, does not limit the maximum number of incomplete requests), jobs can incur $\Omega(n)$ s-oblivious pi-blocking, as demonstrated in Figure 6.10(a).

As priority inheritance is used together with priority queues in the uniprocessor case, (*e.g.*, in the PIP and PCP—see Section 2.4.3), it is perhaps not surprising that FIFO ordering by itself is ill-suited to ensuring $O(m)$ maximum pi-blocking. However, ordering requests by job priority, as for instance done in (Easwaran and Andersson, 2009), does not improve the bound: since a low-priority job can be starved by later-issued higher-priority requests, it is easy to construct an arrival sequence in which



(a) Requests satisfied in FIFO order.

(b) Requests satisfied in order of job priority.

Figure 6.10: G-EDF schedules of $n = 4$ tasks sharing one resource ℓ_1 on $m = 2$ processors. Each job requires ℓ_1 for the entirety of its computation. **(a)** If conflicting requests are satisfied in FIFO order, then the job with the earliest deadline (J_1) may incur $\Omega(n)$ pi-blocking if its request is issued just after all other requests. **(b)** If conflicting requests are satisfied in order of job priority, then a job's request may be deferred repeatedly even though it is among the m highest-priority jobs.

a job incurs $\Omega(n)$ s-oblivious pi-blocking, as seen in Figure 6.10 (b). Thus, ordering *all* requests by job priority is, at least asymptotically speaking, not preferable to the simpler FIFO queuing.

Fortunately, it is possible to use priority inheritance to realize $O(m)$ maximum s-oblivious pi-blocking by combining FIFO and priority ordering. In the global OMLP, each resource is protected by two locks: a priority-based m-exclusion lock that limits access to a regular FIFO mutex lock, which in turn serializes access to the resource. This idea is formalized by the following rules.

Structure. For each resource ℓ_q, there are two job queues: FQ_q, a FIFO queue of length at most m, and PQ_q, a priority queue (ordered by job priority) that is only used if more than m jobs are contending for ℓ_q. The job at the head of FQ_q (if any) holds ℓ_q.

Rules. Let $queued_q(t)$ denote the number of jobs queued in both FQ_q and PQ_q at time t. Requests are ordered according to the following rules.

G1 A job J_i that issues a request $\mathcal{R}_{i,q,v}$ for ℓ_q at time t is appended to FQ_q if $queued_q(t) < m$; otherwise, if $queued_q(t) \geq m$, it is added to PQ_q. $\mathcal{R}_{i,q,v}$ is satisfied when J_i becomes the head of FQ_q.

G2 All queued jobs are suspended, with the exception of the job at the head of FQ_q, which is ready and inherits the priority of the highest-priority job in FQ_q and PQ_q.

G3 When J_i releases ℓ_q, it is dequeued from FQ_q and the new head of FQ_q (if any) is resumed. Also, if PQ_q is non-empty, then the highest-priority job in PQ_q is moved to FQ_q.

The key insight is the use of an m-exclusion lock to safely defer requests of lower-priority jobs without allowing a pi-blocked job to starve. This can be observed in the example shown in Figure 6.11.

Example 6.7. Figure 6.11 depicts a G-EDF schedule of six jobs sharing one resource ℓ_1 on $m = 2$ processors under the global OMLP mutex protocol. At time 1, J_6 requests ℓ_1 and enters FQ_1 immediately (Rule G1). At time 2, ℓ_1 is requested by J_5, which is also enqueued in FQ_1 and suspended since it was non-empty.

At time 4, $m = 2$ jobs hold the m-exclusion lock (*i.e.*, have entered FQ_1) and thus J_4 must enter PQ_1 instead (Rule G1). Hence it is safely deferred when ℓ_1 is later requested by higher-priority jobs (J_3, J_2, J_1). At the same time, J_5, which incurs pi-blocking until J_3's arrival at time 5, precedes the later-issued requests since it already held the m-exclusion lock—this avoids starvation in scenarios such as the one depicted in Figure 6.10 (b). Note that J_5 incurs pi-blocking until time 5 (and not only until time 4) because the release of J_4 at time 4 does not displace J_5 from the set of the $c = m = 2$ highest-priority pending jobs (J_6 is also pending at time 4, but has a later deadline than J_5). $\lozenge$

Next, we bound maximum s-oblivious pi-blocking under the global OMLP mutex protcol. In the following analysis, let t_0 denote the time at which J_i issues $\mathcal{R}_{i,q,v}$, t_1 denote the time at which J_i enters FQ_q, and t_2 denote the time at which $\mathcal{R}_{i,q,v}$ is satisfied (this is illustrated in Figure 6.11 for J_4)

Further, let $entered(t)$, $t_0 \leq t < t_1$, denote the number of jobs that have been moved from PQ_q to FQ_q during $[t_0, t]$ due to Rule G3. That is, $entered(t)$ counts the jobs that preceded J_i in entering FQ_q. For example, for J_4 in Figure 6.11, $entered(5) = 0$, $entered(10) = 1$, and $entered(11) = 2$.

Lemma 6.14. *For each point in time $t \in [t_0, t_1)$, if J_i incurs s-oblivious pi-blocking at time t, then $entered(t) < m$.*

Proof. By Rule G3, because J_i has not yet entered FQ_q at time t, there must be m pending jobs queued in FQ_q. Due to FIFO ordering, if $entered(t) \geq m$, then each job queued in FQ_q at time t must have been enqueued in FQ_q during $[t_0, t]$. By Rule G3, this implies that each job in FQ_q must

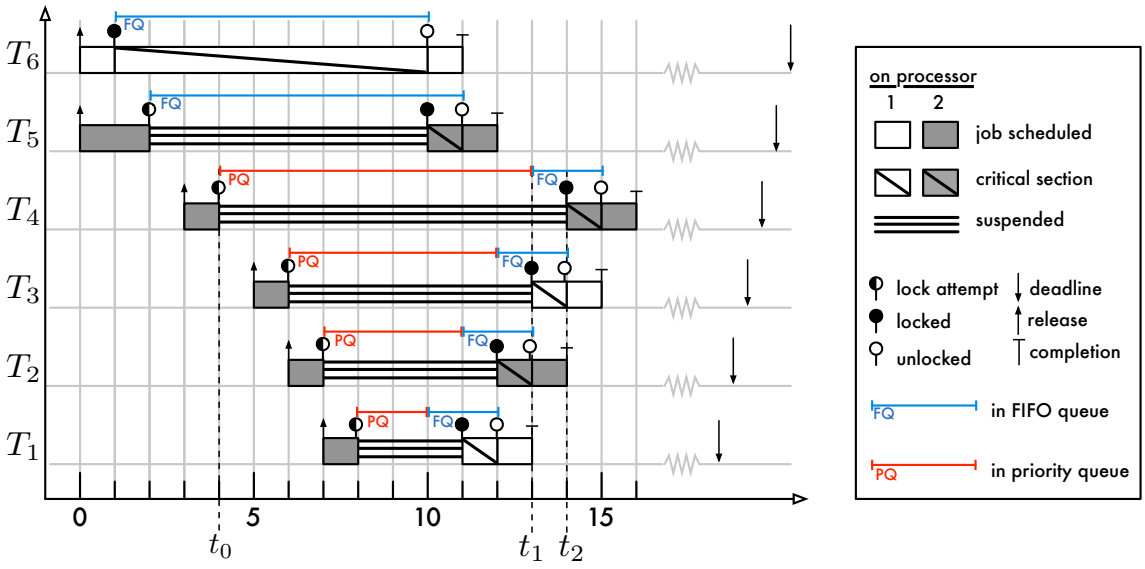


Figure 6.11: Example showing the OMLP mutex protocol for global scheduling under G-EDF for six tasks sharing one resource on $m = 2$ processors. J_4 issues a request at $t_0 = 4$, enters FQ_1 at $t_1 = 11$, and holds ℓ_1 at $t_2 = 14$. Note that J_6 and J_5 enter FQ_1 immediately for lack of contention, and thus J_5's request precedes J_1's request in spite of J_1 having an earlier deadline. In contrast, J_1 and J_2 arrive and enqueue after J_4, but enter FQ_1 before J_4 due to their earlier deadlines and Rule G3. Similarly, J_3 acquires ℓ_1 before J_4, despite J_4's earlier request.

have a priority that exceeds J_i's priority. By the definition of s-oblivious pi-blocking (Definition 6.1), the presence of m higher-priority pending jobs implies that J_i is not pi-blocked. □

Lemma 6.15. *During $[t_0, t_2)$, J_i incurs s-oblivious pi-blocking for the combined duration of at most $2 \cdot m - 1$ requests.*

Proof. Due to the bounded length of FQ_q, at most $m - 1$ requests complete in $[t_1, t_2)$ before a given request is satisfied. By Lemma 6.14 and Rule G3, at most m requests complete before J_1 is no longer pi-blocked in $[t_0, t_1)$. □

Combining Lemma 6.15 with the maximum request length for each ℓ_q yields the following bound.

Lemma 6.16. *J_i is pi-blocked for at most*

$$b_i \triangleq \sum_{k=1}^{n_r} N_{i,q} \cdot (2 \cdot m - 1) \cdot L^{max} = O(m).$$

Scheduling	Constraint	Progress Mechanism	Bound	Analysis
global	mutex	priority inheritance	$N_i \cdot (2m-1)$	Section 6.2.5
clustered	mutex	priority donation	$m + N_i \cdot (m-1)$	Section 6.2.2
clustered	k-exclusion	priority donation	$m + N_i \cdot \left\lceil \frac{m-\min\{k_q\}}{\min\{k_q\}} \right\rceil$	Section 6.2.4
clustered	RW—writers	priority donation	$2m + N_i \cdot (2m-1)$	Section 6.2.3
clustered	RW—readers	priority donation	$2m + N_i \cdot 2$	Section 6.2.3

Table 6.1: Summary of the OMLP per-job s-oblivious pi-blocking bounds, given in terms of the maximum number of blocking requests, where m denotes the number of processors and $N_i = \sum_q N_{i,q}$ denotes the maximum number of requests issued by the job.

Proof. By Lemma 6.15, J_i is pi-blocked for the duration of at most $2 \cdot (m-1)$ requests each time it requests a resource ℓ_q. Due to priority inheritance, the resource-holding job has an effective priority among the m highest priorities whenever J_i is pi-blocked; requests are thus guaranteed to progress towards completion when J_i is pi-blocked. As J_i requests ℓ_q at most $N_{i,q}$ times, it suffices to consider the longest request $N_{i,k} \cdot (2 \cdot m - 1)$ times. The sum of the per-resource bounds yields b_i. By assumption (Section 6.1.2), $L^{max} = O(1)$ and $\sum_q N_{i,1} = O(1)$, and hence $b_i = O(m)$. □

A detailed, non-asymptotic bound on maximum pi-blocking that takes individual request lengths and frequencies into account is presented in Section 6.4.

6.2.6 Optimality, Combinations, and Limitations

In this section, we conclude our discussion of the OMLP family by examining various optimality properties and limitations in more detail and discuss how each of the protocol variants can be integrated with each other and OMLP-unrelated self-suspensions (such as suspensions due to I/O with dedicated devices).

Besides being asymptotically optimal, the four protocols of the OMLP family also have constant factors, summarized in Table 6.1, that are small enough for the protocols to be practical. Let $N_i = \sum_q N_{i,q}$ denote the maximum number of requests issued by any J_i. In the following, we assume $N_i > 0$, that is, the following discussion does not apply to independent tasks. From the example shown in Figure 6.2, it is apparent that a lower bound *per request* is $m-1$ blocking requests. Therefore, a lower bound on the maximum number of blocking requests (under s-oblivious analysis) is $N_i \cdot (m-1)$. This allows to characterize how far the OMLP's bounds are from being optimal.

Since L^{max} is presumed constant, we focus on the number of blocking requests in the following discussion.

We begin with the global OMLP (for mutex constraints), which ensures that a job is pi-blocked by at most $N_i \cdot (2m-1)$ requests (see Table 6.1). As discussed in the preceding section, the principal advantage of the global OMLP over the clustered OMLP is that independent jobs do not incur pi-blocking (*i.e.*, if $N_i = 0$, then $b_i = 0$). The guaranteed upper bound is optimal within at most[5] a factor of

$$\frac{N_i \cdot (2m-1)}{N_i \cdot (m-1)} = \frac{N_i \cdot (2(m-1)+1)}{N_i \cdot (m-1)} = 2 + \frac{1}{m-1}.$$

That is, for large m, the global OMLP ensures bounds on maximum pi-blocking that is (almost) within a factor of two of the lower bound.

As summarized in Table 6.1, the clustered OMLP for mutex constraints ensures that a job J_i is pi-blocked by at most $m + N_i \cdot (m-1)$ conflicting requests. The mutex protocol is hence optimal within at most a factor of

$$\frac{m + N_i \cdot (m-1)}{N_i \cdot (m-1)} = 1 + \frac{m}{N_i \cdot (m-1)} \leq 1 + \frac{m}{(m-1)} = 2 + \frac{1}{m-1}.$$

The ratio is maximized for $N_i = 1$, in which case it approaches two for large m, just as the global OMLP. If $N_i > 1$, then the clustered OMLP ensures a smaller bound than the global OMLP, albeit at the cost of potentially delaying otherwise independent jobs.

In the case of the OMLP's k-exclusion protocol, if $k^{min} < m$, then Theorem 6.3 and Lemma 6.13 imply that the upper bound on s-oblivious pi-blocking is within at most a factor of

$$\begin{aligned}
\frac{\left(1 + \left\lceil \frac{m-k^{min}}{k^{min}} \right\rceil\right) + \left(N_i \cdot \left\lceil \frac{m-k^{min}}{k^{min}} \right\rceil\right)}{N_i \cdot \left\lceil \frac{m}{k^{min}} - 1 \right\rceil} &= \frac{\left(1 + \left\lceil \frac{m}{k^{min}} - 1 \right\rceil\right) + \left(N_i \cdot \left\lceil \frac{m}{k^{min}} - 1 \right\rceil\right)}{N_i \cdot \left\lceil \frac{m}{k^{min}} - 1 \right\rceil} \\
&= \frac{1}{N_i \cdot \left\lceil \frac{m}{k^{min}} - 1 \right\rceil} + \frac{1}{N_i} + 1 \\
&\leq 2 + \frac{1}{\left\lceil \frac{m}{k^{min}} - 1 \right\rceil}
\end{aligned}$$

[5]It is unknown whether $N_i \cdot (m-1)$ is a tight lower bound in absolute terms (*i.e.*, non-asymptotically).

of the lower bound that is unavoidable in the general case (for tasks that share resources). In the worst case, $k^{min} = 1$, the ratio reduces to $2 + \frac{1}{m-1}$. The k-exclusion protocol is thus no worse (in terms of the maximum number of blocking requests) than the clustered OMLP's mutex protocol.

In the degenerate case of $k^{min} = m$, maximum blocking under the clustered OMLP reduces to 1 (akin to a non-preemptive section), but the above ratio is undefined since the lower bound reduces to 0 in this case. This is because Lemma 6.13 does not take preemptions from higher-priority, later-arriving jobs into account. However, it is trivial to construct an example in which m lower-priority jobs request all k^{min} replicas such that a later-arriving, higher-priority job incurs s-oblivious pi-blocking for the duration of one critical section.

In the case of the OMLP's phase-fair RW lock, writers are delayed by additional requests because a reader phase may separate any two writer phases. This has the effect of virtually doubling the factor. That is, if job J_i issues N_i write requests (and no read requests), then the ensured bound is within at most

$$\begin{aligned}\frac{2m + N_i \cdot (2m-1)}{N_i \cdot (m-1)} &= \frac{2m + N_i \cdot (2(m-1)+1)}{N_i \cdot (m-1)} \\ &= 2 \cdot \left(1 + \frac{m}{N_i \cdot (m-1)}\right) + \frac{1}{m-1} \\ &\leq 2 \cdot \left(2 + \frac{1}{(m-1)}\right) + \frac{1}{m-1} \\ &= 4 + \frac{3}{m-1}\end{aligned}$$

of the optimal bound for mutex constraints. That is, for large m, the bound on maximum pi-blocking for (pure) writers approaches four and is hence approximately twice as large as the bounds of the mutex protocols. This suggests that RW locks should only be employed if the write ratio is small.

Optimality of relaxed-exclusion protocols. Under phase-fair RW locks, read requests incur at most $O(1)$ acquisition delay. Similarly, requests for ℓ_q incur only $O(\frac{m}{k_q})$ acquisition delay under the k-exclusion protocol. Yet, we only prove $O(m)$ and $O(m/k^{min})$ *maximum* s-oblivious pi-blocking bounds, respectively—since both relaxed-exclusion constraints generalize mutual exclusion, this is unavoidable in the general case (*i.e.*, *some* jobs may incur $\Omega(m)$ pi-blocking if $k^{min} = 1$ or if some resource is shared among m writers).

However, as discussed in Section 6.2.5, *any* job may become a priority donor and thus suspend (at most once) for the duration of the maximum request span. This seems undesirable for tasks that do not partake in mutual exclusion. For example, why should "pure readers" (*i.e.*, tasks that never issue write requests) not have an $O(1)$ bound on pi-blocking? It is currently unknown if this is even possible in general, as lower bounds for specific task types (*e.g.*, "pure readers," "DSP tasks") are an to-date unexplored topic that warrants further attention.

Since priority inheritance is sufficient for the global OMLP mutex protocol, one might wonder if it is possible to apply the same design using priority inheritance instead of priority donation to obtain an RW protocol under global scheduling with $O(m)$ maximum pi-blocking for writers and $O(1)$ maximum pi-blocking for readers. Unfortunately, this is not the case. The reason is that the analytical benefits of priority inheritance under s-oblivious analysis do not extend to RW exclusion. When using priority inheritance with mutual exclusion, there is always a one-to-one relationship: a priority is inherited by at most one ready job at any time. In contrast, a single high-priority writer may have to "push" multiple low-priority readers. In this case, the high priority is "duplicated" and used by multiple jobs on different processors at the same time. This significantly complicates the analysis. In fact, simply instantiating Rules R1–R3 and W1–W3 from Section 6.2.3 with priority inheritance may cause $\Omega(\frac{n}{c})$ s-oblivious pi-blocking since it is possible to construct schedules that are conceptually similar to the one shown in Figure 6.4. A naive application of priority inheritance to the k-exclusion problem would lead to the same result. This demonstrates the power of priority donation, and also highlights the value of the clustered OMLP even for the special cases $c = m$ and $c = 1$: the clustered OMLP RW and k-exclusion protocols are the first multiprocessor real-time locking protocols of their kind for the special cases of global and partitioned scheduling as well.

In very recent work, Elliott and Anderson (2011) presented a k-exclusion protocol for global JLFP schedulers that guarantees asymptotically optimal maximum s-oblivious pi-blocking while ensuring that independent jobs do not incur pi-blocking. Similar to the global OMLP, Elliott and Anderson's protocol uses priority inheritance in combination with a hybrid FIFO/priority queue. Due to the challenges of k-exclusion, their hybrid queue is of a more complicated structure than the one used in the global OMLP. Interestingly, Elliott and Anderson's protocol uses a technique akin to priority donation to ensure progress within each hybrid queue.

To the best of our knowledge, no suspension-based RW protocol with $O(1)$ maximum pi-blocking for pure readers has been proposed to date.

Highly replicated resources. Our k-exclusion protocol assumes $1 \leq k_q \leq m$ since additional replicas would remain unused as priority donation allows at most m incomplete requests. (The same assumption is made in Elliott and Anderson's k-exclusion protocol for global scheduling.) This has little impact on resources that require jobs to be scheduled (*e.g.*, shared data structures), but it may be a severe limitation for resources that do not require a processor (*e.g.*, there could be more than m DSP co-processors).

However, would a priority donation replacement that allows more than c jobs in a cluster to hold a replica be a solution? Surprisingly, the answer is no. This is because s-oblivious schedulability analysis (implicitly) assumes the number of processors as the maximum degree of parallelism (since all *pending* jobs cause processor demand under s-oblivious analysis). In other words, s-aware schedulability analysis is required to derive analytical benefits from highly replicated resources. However, as we discuss in Section 6.3.4, s-aware schedulability analysis poses additional challenges and no k-exclusion protocol for s-aware analysis, optimal or otherwise, has been proposed to date.

Unrelated self-suspensions. An issue that arises in practice with real-time locking protocols is how blocking bounds are affected by suspensions that are unrelated to resource sharing. For example, a job may self-suspend when it performs I/O using a private device (*i.e.*, one that is not under control of a locking protocol). In uniprocessor locking protocols such as the SRP or the PCP, a job resuming from a self-suspension may incur additional pi-blocking just as if it were newly released. That is, a job J_i that self-suspends η_i times can incur pi-blocking for the duration of up to $1 + \eta_i$ outermost critical sections under SRP and PCP.

This effect also applies to multiprocessor real-time locking protocols. In the case of non-preemptive spinlocks, a job may similarly be subject to additional pi-blocking when it resumes from a self-suspension if it cannot be scheduled immediately due to a non-preemptable lower-priority job. Under the MPCP and DPCP, a self-suspending job allows lower-priority jobs to issue requests for global resources and thus may also incur additional pi-blocking after it resumes when lower-priority jobs are subsequently priority boosted.

Remarkably, the OMLP's blocking bounds are not affected by self-suspensions. In the case of the global OMLP, a job incurs pi-blocking only when it issues a request itself, which is not affected by self-suspensions. Further, while it may appear on first sight that priority donation is affected by self-suspensions, this is not the case: a resuming job is never required to serve as a priority donor. This is because priority donation is defined in terms of *pending* jobs, and not in terms of *ready* jobs. A job that self-suspends or resumes does not alter the set of pending jobs. Further, any job serving as a priority donor upon release may self-suspend (since a priority donor's purpose is to suspend anyway). A priority donor that resumes from a self-suspension while the priority recipient executes is effectively not resumed until its donor services are no longer required.

The OMLP is hence not affected by self-suspensions and the presented analysis can be used in environments where jobs self-suspend. However, if jobs may self-suspend while holding a resource, then the maximum self-suspension time must be reflected in each $L_{i,q}$.

Protocol combinations. The clustered mutex protocol (Section 6.2.2) generalizes the partitioned OMLP proposed in (Brandenburg and Anderson, 2010a) in terms of blocking behavior; there is thus little reason to use both in the same system or to prefer the partitioned OMLP over the more general clustered OMLP.

The global OMLP cannot be used with any of the clustered OMLP variants since priority inheritance is incompatible with priority donation (from an analytical point of view). As discussed above, both the clustered and global mutex protocols have an $O(m)$ s-oblivious pi-blocking bound, but differ in constant factors and with regard to which jobs incur pi-blocking. Specifically, only jobs that request resources risk s-oblivious pi-blocking under the global OMLP, while even otherwise independent jobs may incur s-oblivious pi-blocking if they serve as a priority donor. The global OMLP may hence be preferable for $c = m$ if only few tasks share resources.

The clustered protocol variants can be freely combined since they all rely on priority donation and because their protocol rules do not conflict. However, care must be taken to correctly identify the maximum request span, which determines the maximum pi-blocking caused by priority donation.

This concludes our discussion of locking protocols for s-oblivious schedulability analysis. Next, we consider the case of s-aware schedulability analysis.

6.3 Locking under S-Aware Schedulability Analysis

Avoiding extended pi-blocking is much more difficult under s-aware analysis than under s-oblivious analysis. In the s-oblivious case, only jobs among the c highest-priority pending jobs (with respect to a cluster) can incur pi-blocking; lower-priority jobs are automatically exempt from pi-blocking since any delays are already accounted for by the execution time inflation of higher-priority jobs. In contrast, under s-aware schedulability analysis, *all* jobs that are not scheduled may be pi-blocked simultaneously since acquisition delay is not modeled as execution time. As we show in Section 6.3.2, this poses significant challenges since it allows long-running jobs of "intermediate priority" to be pi-blocked repeatedly.

Recall from Section 6.1.4 that $\Omega(n)$ maximum s-aware pi-blocking is unavoidable in the general case. We previously considered locking protocols for s-aware schedulability analysis for global and partitioned scheduling (Brandenburg and Anderson, 2010a). In the case of partitioned scheduling, we showed that the $\Omega(n)$ bound is asymptotically tight by devising a simple, but impractical locking protocol based on a single, global FIFO queue. We improve upon this simple protocol in Section 6.3.1 by adjusting the priority boosting rules of the FMLP for partitioned scheduling (Block *et al.*, 2007; Brandenburg and Anderson, 2008b), which is also based on FIFO queues, such that it ensures $O(n)$ maximum s-aware pi-blocking in the case of $c = 1$.

In the case of global scheduling, we previously argued, but did not formally prove, that Block *et al.*'s FMLP for global scheduling (Block *et al.*, 2007) ensures $O(n)$ s-aware pi-blocking under global JLFP scheduling. In the specific example discussed in (Brandenburg and Anderson, 2010a), namely G-EDF with implicit deadlines and HRT constraints, maximum s-aware pi-blocking is indeed bounded as claimed (Section 6.3.2). Unfortunately, in the general case of global JLFP scheduling, and also in the case of tasks with arbitrary deadlines under G-EDF, this is not necessarily the case. In Section 6.3.2, we demonstrate that there exist corner cases in which priority inheritance does not ensure $O(n)$ maximum s-aware pi-blocking under G-EDF scheduling (no matter which queue order is assumed), and hence also not under JLFP scheduling in general.

We conclude this chapter with a discussion of open questions and challenges regarding locking protocols for s-aware analysis in Section 6.3.4. In particular, we show that priority queues give rise to

non-optimal maximum s-aware pi-blocking, and further discuss the challenges involved in supporting RW and k-exclusion constraints under s-aware analysis.

6.3.1 Mutual Exclusion under Partitioned Scheduling

The partitioned FMLP was first designed for P-EDF (Block *et al.*, 2007) and later adapted to P-FP scheduling (Brandenburg and Anderson, 2008b). Its main feature is simplicity: conflicting resource accesses are serialized using per-resource FIFO queues, and resource-holder progress is ensured using straightforward priority boosting. However, both prior versions of the partitioned FMLP use a sub-optimal rule for ordering priority-boosted jobs: if two or more jobs are priority-boosted simultaneously on the same processor, then the partitioned FMLP requires them to be scheduled in FIFO order with respect to the time that their priority was raised (*i.e.*, the times at which their resource requests were satisfied). This choice of tie-breaking rule greatly complicates the analysis of the FMLP and yields only an $O(\min(n, n_r) \cdot n)$ bound on maximum s-aware pi-blocking (recall that n_r denotes the number of shared resources—see Section 2.4.1).

In this section, we provide an updated definition of the partitioned FMLP that ensures $O(n)$ maximum s-aware pi-blocking under JLFP scheduling (*i.e.*, it subsumes both earlier versions for P-EDF and P-FP). To avoid ambiguities, we refer to the following protocol as the *FIFO mutex locking protocol*, denoted as FMLP$^+$.

Structure. For each serially-reusable resource ℓ_q, there is a FIFO queue FQ_q that is used to serialize conflicting accesses. The job at the head of FQ_q holds ℓ_q.

Rules. Let J_i denote a job that issues a request $\mathcal{R}_{i,q,v}$ for ℓ_q. Resource access is granted according to the following rules.

F1 When J_i issues $\mathcal{R}_{i,q,v}$, it is appended to FQ_q. If FQ_q was non-empty, then J_i suspends until $\mathcal{R}_{i,q,v}$ is satisfied.

F2 $\mathcal{R}_{i,q,v}$ is satisfied when J_i becomes the head of FQ_q.

F3 J_i is dequeued from FQ_q when $\mathcal{R}_{i,q,v}$ is complete. The new head of FQ_q (if any) is resumed.

Priority boosting. The FMLP$^+$ uses priority boosting to ensure resource-holder progress. Jobs that hold resources have higher priority than those that do not, and resource-holding jobs are ordered by

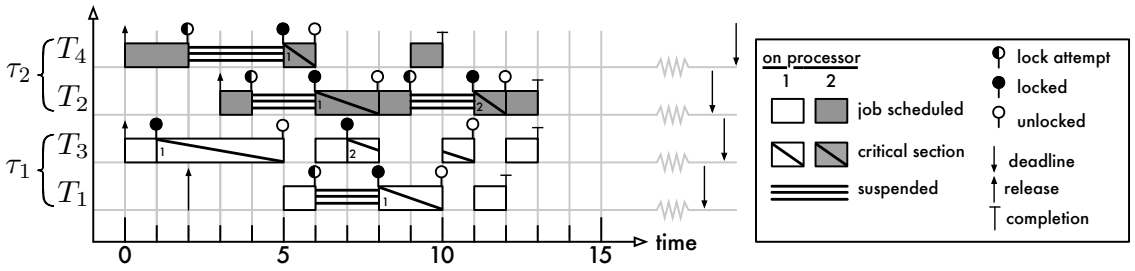


Figure 6.12: P-FP example schedule for $m = 2$ illustrating the FMLP$^+$ with preemptable critical sections. In this example, J_3 is preempted in the middle of its critical section by J_1 at time 8 when J_1's earlier-issued request is satisfied.

the time that they requested their respective resource. Formally, let J_i denote a job that is pending at time t. If J_i issued a request $\mathcal{R}_{i,q,v}$ that is incomplete at time t, then let $t_{i,q,v}$ denote the time at which J_i issued $\mathcal{R}_{i,q,v}$, where $t_{i,q,v} \leq t$. The effective priority of a job J_i at time t is defined as follows:

$$\mathsf{y}(J_i, t) \triangleq \begin{cases} (0, \quad t_{i,q,v}) & \text{if } \mathcal{R}_{i,q,v} \text{ exists and is incomplete at time } t, \\ (1, \quad \mathsf{Y}(J_i, t)) & \text{otherwise,} \end{cases} \tag{6.1}$$

with the interpretation that $(a_x, b_x) < (a_y, b_y)$ if and only if $a_x < a_y \vee (a_x = a_y \wedge b_x < b_y)$. Since at most one job is scheduled at any time in each partition, the time at which a job requested a resource is necessarily unique (with respect to one partition).

Preemptive critical sections. The tie-breaking rule used by the prior variants of the partitioned FMLP essentially resulted in the non-preemptive execution of critical sections. Under the FMLP$^+$, critical sections can be executed either preemptively or non-preemptively. Allowing jobs to be preempted during a critical section lowers bounds on maximum pi-blocking (although only by a constant factor), but risks higher overheads, as discussed in Chapter 7. We consider the case of preemptive critical sections first.

Example 6.8. An example of the FMLP$^+$ under P-FP scheduling for $m = 2$ is shown in Figure 6.12, which depicts a schedule of four tasks sharing two resources. The defining property of the FMLP$^+$, namely FIFO ordering, is apparent at time 5: since J_4 requests ℓ_1 already at time 2 and prior to J_2 (which requests ℓ_1 at time 4), J_4's request is satisfied when J_3 releases the resources despite J_2 having a higher priority.

When J_1 suspends at time 6 because ℓ_1 is unavailable, the local lower-priority job J_3 is given a chance to execute. As a result, J_3 requests and acquires ℓ_2 at time 7. While holding ℓ_2, J_3 is priority-boosted. However, when J_1's earlier request is satisfied at time 8, its priority is also boosted. Consequently, J_1 preempts J_3 since J_1's effective priority—the time at which it requested ℓ_1 (time 6)—is higher than J_3's effective priority (time 8). In contrast, under the prior versions of the partitioned FMLP, J_3 would have had higher priority because it was priority-boosted at an earlier point in time, which would have caused J_1 to incur additional pi-blocking. $\Diamond$

We next bound maximum s-aware pi-blocking under the FMLP$^+$ with preemptive critical sections. There are three principle sources of s-aware pi-blocking under the FMLP$^+$: **(i)** a job J_i is directly delayed by a remote job J_x if J_x holds a resource requested by J_1; **(ii)** J_i is transitively delayed by a remote job J_y if J_y has a higher priority than J_x while J_x directly delays J_i; and **(iii)** J_i incurs pi-blocking whenever a local priority-boosted job with lower base priority executes a request. Causes (i) and (ii) occur only while J_i incurs acquisition delay (*i.e.*, they require J_i to issue a request); cause (iii) occurs after local lower-priority jobs were scheduled and issued requests for global resources (this source of pi-blocking also affects independent jobs).

We first bound causes (i) and (ii) by bounding acquisition delay due to remote jobs as a whole.

Lemma 6.17. *If critical sections are executed preemptively, then each remote task delays a request of J_i for a resource ℓ_q with at most one critical section (either directly or indirectly, but not both).*

Proof. Let $\mathcal{R}_{i,q,v}$ denote J_i's request, and let $t_{i,q,v}$ denote the time at which J_i issued $\mathcal{R}_{i,q,v}$. Consider a remote job J_x that executes a request $\mathcal{R}_{x,y,z}$ and thereby causes J_i to incur acquisition delay.

If J_x delays J_i directly (*i.e.*, if $\mathcal{R}_{x,y,z}$ precedes $\mathcal{R}_{i,q,v}$ in FQ$_q$), then J_x necessarily issued $\mathcal{R}_{x,y,z}$ no later than $t_{i,q,v}$ (since FQ$_q$ is a FIFO queue).

If J_x delays J_i indirectly, then it prevents a remote job J_k holding ℓ_q from being scheduled, and J_k's request $\mathcal{R}_{k,q,w}$ precedes $\mathcal{R}_{i,q,v}$ in FQ$_q$. By the definition of priority boosting under the FMLP$^+$, resource-holding jobs are only preempted by jobs that execute earlier-issued requests. Therefore, J_x necessarily issued $\mathcal{R}_{x,y,z}$ prior to $t_{i,q,v}$.

Consequently, J_i incurs acquisition delay, either directly or indirectly, only due to earlier-issued requests. Clearly, J_i is not blocked by requests that were already complete prior to $t_{i,q,v}$. Therefore,

J_i incurs acquisition delay only due to requests that were incomplete at time $t_{i,q,v}$. Since tasks and jobs are sequential, at most one request per task is incomplete at time $t_{i,q,v}$. □

When a local lower-priority job J_l executes a request, then J_i incurs s-aware pi-blocking. In particular, it is irrelevant which resource J_l accesses. That is, J_i incurs pi-blocking regardless of whether J_i requires the resource that J_l holds. Therefore, pi-blocking due to local tasks depends only on the number of times that J_i suspends, and not on the identity of the resources that it requests.

Lemma 6.18. *If J_i suspends η_i times, then lower-priority jobs of each local task pi-block J_i for the duration of at most $\eta_i + 1$ critical sections.*

Proof. In order to pi-block J_i, a lower-priority job J_l must hold a resource. To subsequently hold a resource, J_l must issue a request first. To issue a request, J_l must be scheduled (while not holding a resource, *i.e.*, without being priority-boosted). Since J_l is local to J_i, J_l is only scheduled when J_i is not ready (*i.e.*, prior to J_i's release and while J_i is suspended). Due to the sequentiality of tasks and jobs, there is at most one incomplete request for each local lower-priority job J_l each time that J_i becomes ready (*i.e.*, when J_i is released and each time that it resumes). □

This leads to the following bound on maximum pi-blocking.

Theorem 6.4. *Let $n_c = |\tau_{P_i}|$ denote the number of tasks assigned to J_i's partition. If critical sections are executed* ***preemptively****, then the FMLP$^+$ causes a job J_i to incur at most*

$$\begin{aligned} b_i &= (n_c - 1) \cdot \left(1 + \sum_{q=1}^{n_r} N_{i,q}\right) \cdot L^{max} + \left(\sum_{q=1}^{n_r} N_{i,q} \cdot (n - n_c)\right) \cdot L^{max} \\ &= (n_c - 1) \cdot L^{max} + \sum_{q=1}^{n_r} N_{i,q} \cdot (n - 1) \cdot L^{max} \\ &= O(n) \end{aligned}$$

s-aware pi-blocking.

Proof. J_i may suspend each time that it issues a request. Therefore, by Lemma 6.18, each of the $n_c - 1$ local tasks (other than T_i) pi-blocks J_i for the duration of at most $1 + \sum_{q=1}^{n_r} N_{i,q}$ requests (once due to a request issued prior to J_i's release, and once each time J_i suspends). By Lemma 6.17,

each of the $n - n_c$ remote tasks pi-block J_i with at most one request each time J_i issues a request, which J_i does at most $\sum_{q=1}^{n_r} N_{i,q}$ times. The stated bound follows. □

A detailed, non-asymptotic bound on maximum pi-blocking that takes individual request lengths and frequencies into account is derived in Section 6.4.

Non-preemptive critical sections. In the case that critical sections are executed non-preemptively, additional pi-blocking due to remote tasks is possible if a lower-priority (with regard to effective priority), non-preemptable job prevents a higher-priority job from being scheduled that holds a resource that J_i is waiting for. Pi-blocking due to local tasks, however, is unaffected by whether local jobs are preemptable—a local higher-priority job never causes J_i to incur pi-blocking while it executes, and a local lower-priority job J_l only preempts J_i if J_l is priority-boosted, which always delays J_i eventually. That is, with regard to local jobs, there is no benefit in critical sections being preemptive and Lemma 6.18 thus also applies to the case of non-preemptive critical sections. We next bound the additional delay due to remote tasks that is not reflected by Lemma 6.17.

Lemma 6.19. *Let x_j denote the number of times that J_i is directly delayed by a job of a task in τ_j, where $T_i \notin \tau_j$. If critical sections are executed non-preemptively, then J_i is indirectly delayed by a non-preemptive job of a task in τ_j for the total duration of at most x_j critical sections.*

Proof. A remote non-preemptive job J_l that prevents a higher-priority job J_h from being scheduled delays J_i only if J_h holds a resource that J_i requested. Since J_h has a higher-priority, J_l must be executing a request that it issued later than J_h. J_h will be scheduled as soon as J_l becomes preemptable again, and J_l becomes preemptable as soon as its critical section ends. Therefore, J_l delays J_i indirectly for the duration of at most one critical section. Since J_i is directly delayed by jobs of tasks in τ_j at most x_j times in total, the non-preemptive execution of jobs of tasks in τ_j delays J_i for the combined duration of at most x_j critical sections. □

Since Lemma 6.17 bounds the number of times that J_i is directly blocked (at most once per remote task), Lemma 6.19 yields the following bound.

Theorem 6.5. *Let $n_c = |\tau_{P_i}|$ denote the number of tasks assigned to J_i's partition. If critical sections are executed* ***non-preemptively****, then the* FMLP$^+$ *causes a job J_i to incur at most*

$$b_i = (n_c - 1) \cdot \left(1 + \sum_{q=1}^{n_r} N_{i,q}\right) \cdot L^{max} + 2 \cdot \left(\sum_{q=1}^{n_r} N_{i,q} \cdot (n - n_c)\right) \cdot L^{max}$$
$$= O(n)$$

s-aware pi-blocking.

Proof. Analogously to Theorem 6.4, Lemma 6.18 implies that J_i is pi-blocked by local lower-priority critical sections at most $(n_c - 1) \cdot \left(1 + \sum_{q=1}^{n_r} N_{i,q}\right)$ times, and Lemma 6.17 implies that J_i is pi-blocked, either directly or indirectly, due to earlier-issued requests of remote tasks at most $\sum_{q=1}^{n_r} N_{i,q} \cdot (n - n_c)$ times. The latter also implies an upper bound on the number of times that J_i is blocked by a remote job. By Lemma 6.19, J_i incurs at most L^{max} additional indirect delay each time that it is directly blocked by a remote job. The stated bound follows. □

A detailed, non-asymptotic bound on maximum pi-blocking based on interference sets is given in Section 6.4. Having designed and analyzed a simple protocol with $O(n)$ maximum s-aware pi-blocking for partitioned scheduling, we next consider the case of global scheduling. Surprisingly, bounding maximum s-aware pi-blocking is much more complicated if $c > 1$ because lower-priority jobs can issue requests while the job under analysis J_i is scheduled.

6.3.2 Priority Inheritance and Boosting under Global Scheduling

Under the global FMLP (Block *et al.*, 2007), each resource is protected by a FIFO queue and the job at the head of the FIFO queue holds the resource. Progress is ensured by letting the job holding a resource ℓ_q inherit the priority of any job waiting for ℓ_q.

Consequently, each time that a job J_i requests a resource ℓ_q, it is delayed by at most $n - 1$ prior requests for ℓ_q. In total, a job J_i is pi-blocked *while incurring acquisition delay* for at most $L^{max} \cdot (n - 1) \cdot \sum_q N_{i,q} = O(n)$ time units. However, since priority inheritance may cause lower-priority resource holders to preempt J_i, it is possible for J_i to incur s-aware pi-blocking even while

Task	e_i	p_i	d_i	$N_{i,1}$	$L_{i,1}$
T_1	3	5	5	0	0
T_2	2	5	5	1	1
T_3	$1.5 + 2 \cdot (\phi - 1)$	$1 + 5 \cdot \phi$	$1 + 5 \cdot \phi$	0	0
T_4	1.5	5	$5 + 5 \cdot \phi$	1	1.5

Table 6.2: Parameters of the task set τ^ϕ, where $\phi \in \mathbb{N}$.

not incurring acquisition delay. That is, under s-aware analysis, even independent jobs may incur pi-blocking due to priority inheritance (in contrast to the s-oblivious case discussed in Section 6.2.5).

In the case of G-EDF with arbitrary deadlines, or in the general case of arbitrary JLFP policies, this side effect of priority inheritance can give rise to what appears to be non-optimal maximum s-oblivious pi-blocking. To demonstrate this, we construct a task set τ^ϕ for which there exists an arrival sequence such that an independent job incurs ϕ time units of s-aware pi-blocking under G-EDF scheduling on $m = 2$ processors, where ϕ can be chosen to be arbitrarily large (*i.e.*, maximum s-aware pi-blocking cannot be bounded in terms of m or n).

Definition 6.6. In the following, consider the set of tasks $\tau^\phi = \{T_1, \ldots, T_4\}$ that share $n_r = 1$ resource. Tasks $T_1, \ldots, T_4$ are defined by the parameters listed in Table 6.2.

Note that τ^ϕ is designed such that $\frac{\max\{p_i\}}{\min\{p_i\}} \approx \phi$ (for large ϕ). This allows us to construct an arrival sequence such that a job of T_3 incurs s-aware pi-blocking each time that T_1, T_2, and T_4 release jobs in a certain pattern, which, by design, can occur up to ϕ times while a job of T_3 is pending.

Lemma 6.20. *There exists an invocation sequence of the tasks in τ^ϕ such that a job incurs ϕ time units s-aware pi-blocking if scheduled by G-EDF with the global FMLP on $m = 2$ processors.*

Proof. In the following, let $1 \leq j \leq \phi$. Suppose jobs of T_4 require ℓ_1 for the entirety of their execution, and that jobs of T_2 require ℓ_1 for the latter half of their execution. Consider the schedule that arises when $J_{3,1}$ and the first ϕ jobs of T_1, T_2, and T_4 arrive at the following times.

$$a_{1,j} = 1 + (j-1) \cdot p_1 \qquad a_{2,j} = 1 + (j-1) \cdot p_2$$

$$a_{3,1} = 0 \qquad a_{4,j} = 0.5 + (j-1) \cdot p_4$$

The resulting schedule for $\phi = 5$ is shown in Figure 6.13.

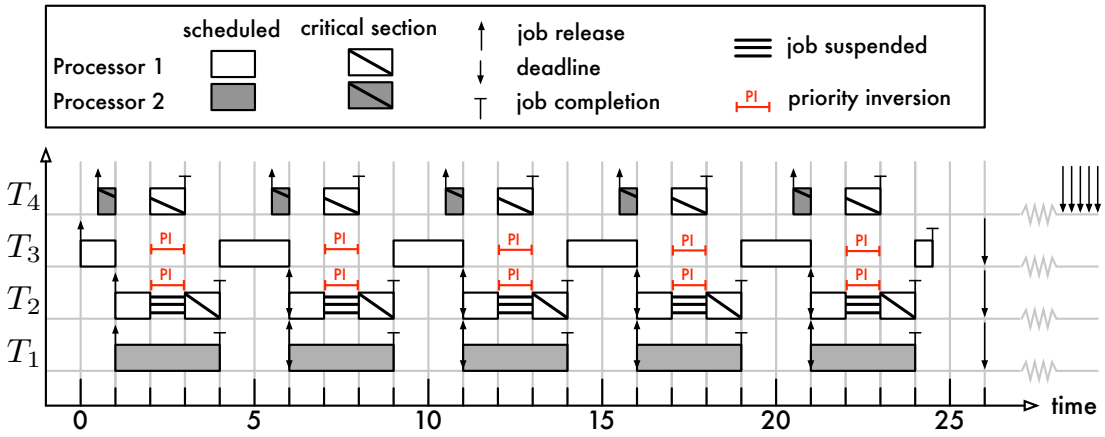


Figure 6.13: G-EDF example schedule for $m = 2$ illustrating τ^{ϕ} for $\phi = 5$. The absolute deadlines for jobs of T_4 are indicated as being in the future since $d_4 = 5 + 5 \cdot \phi$. Because lower-priority jobs of T_4 repeatedly inherit the priority of higher-priority jobs of T_2, job J_3 incurs s-aware pi-blocking for ϕ time units. Note that it is irrelevant in this example whether jobs wait in FIFO or priority order since the resource that causes priority inheritance to take effect is shared by only two tasks.

By construction, all $J_{4,j}$ have lower priority than $J_{3,1}$, and all $J_{1,j}$ and $J_{2,j}$ have higher priority than $J_{3,1}$. (The absolute deadlines of jobs $J_{1,\phi}$ and $J_{2,\phi}$ are tied with $J_{3,1}$'s absolute deadline at time $1+5\cdot\phi$; $J_{3,1}$ has a lower priority because we break deadline ties in favor of lower-indexed tasks.) Therefore, when $J_{1,1}$ and $J_{2,1}$ are released at time 1, they preempt $J_{3,1}$ and $J_{4,1}$. The preempted jobs do not incur s-aware pi-blocking because $m = c = 2$ higher-priority jobs are scheduled.

At time 2, $J_{2,1}$ requests ℓ_1. Since the resource is currently unavailable, the resource-holder $J_{4,1}$ inherits $J_{2,1}$'s priority and is scheduled immediately. At this point, $J_{3,1}$ incurs s-aware pi-blocking because a lower-priority job (with respect to base priorities) is scheduled while $J_{3,1}$ is not. $J_{3,1}$ ceases to be pi-blocked when $J_{4,1}$ completes its critical section at time 3.

At time 4, $J_{3,1}$ is scheduled again when $J_{1,1}$ and $J_{2,1}$ complete. At time 5.5, the sequence of events starts over for $j = 2$ when $J_{4,2}$ is released. As is apparent in Figure 6.13, the pattern repeats ϕ times while $J_{3,1}$ is pending. Job $J_{3,1}$ hence incurs ϕ time units of s-aware pi-blocking in total. □

There are several important things to note. First, since ϕ can be chosen arbitrarily, this example demonstrates that the suspension-based global FMLP does not ensure $O(n)$ maximum s-aware pi-blocking in all cases. Second, since each resource is only shared between two tasks, the order in which resource requests are satisfied is irrelevant (*i.e.*, the schedule shown Figure 6.13 does not depend on FIFO ordering of requests). Therefore, this example demonstrates that priority inheritance

is not suited to designing locking protocols that ensure $O(n)$ maximum s-aware pi-blocking under all global JLFP schedulers. Third, the task set τ_ϕ can be trivially generalized to arbitrary n and m by adding independent tasks with appropriately chosen periods and relative deadlines. And, finally, since the relative priorities of the jobs in this example match their respective task's indices, Figure 6.13 demonstrates that the global FMLP does not ensure $O(n)$ maximum s-aware pi-blocking for G-FP scheduling for arbitrary priority assignments.

Nonetheless, there exist JLFP schedulers and types of task sets for which priority inheritance *does* yield an $O(n)$ protocol. In particular, this is the case for the G-EDF scheduling of implicit-deadline task sets when deadline tardiness is not allowed (and only if jobs are not scheduled prior to their respective release times, *i.e.*, in the absence of early releasing), which is the specific example in which the global FMLP was previously considered under s-aware analysis (Brandenburg and Anderson, 2010a). That is, if all tasks meet implicit (or, in fact, constrained) deadlines, then repeated pi-blocking such as in Figure 6.13 is impossible, which we show next. Recall from Section 2.2 that $a_{i,j}$ denotes job $J_{i,j}$'s release time and that $f_{i,j}$ denotes $J_{i,j}$'s completion time.

Lemma 6.21. *Let τ denote a set of n constrained-deadline tasks, and let $J_{i,j}$ denote an arbitrary job of a task in τ. If τ is HRT schedulable under G-EDF, then at most $n-1$ jobs of lower priority than $J_{i,j}$ are scheduled during $[a_{i,j}, f_{i,j})$, that is, while $J_{i,j}$ is pending.*

Proof. Let $J_{l,k}$ denote a lower-priority job of an arbitrary task $T_l \in \tau$ (and $i \neq l$) that is scheduled during $[a_{i,j}, f_{i,j})$. $J_{l,k}$ having a lower priority than $J_{i,j}$ implies $d_{l,k} \geq d_{i,j}$. Further, $a_{l,k} < f_{i,j}$ since $J_{l,k}$ is scheduled during $[a_{i,j}, f_{i,j})$. Since $d_k \leq p_k$, $J_{l,k+1}$'s earliest-possible release time is $d_{l,k}$, and hence $d_{i,j} \leq d_{l,k} \leq a_{l,k+1}$; since τ is HRT schedulable, we have $f_{i,j} \leq d_{i,j} \leq a_{l,k+1}$. $J_{l,k+1}$ is thus not scheduled during $[a_{i,j}, f_{i,j})$, no matter its priority. Similarly, due to T_l's constrained deadline, the latest-possible absolute deadline of $J_{l,k-1}$ (if $k > 1$) is $a_{l,k}$. Because $a_{l,k} < f_{i,j}$ ($J_{l,k}$ executed during $[a_{i,j}, f_{i,j})$) and $f_{i,j} \leq d_{i,j}$ (τ is HRT schedulable), it follows that $d_{l,k-1} < d_{i,j}$. That is, $J_{l,k-1}$ necessarily has an earlier deadline than $J_{i,j}$. Therefore, at most one lower-priority job of T_l executes during $[a_{i,j}, f_{i,j})$, and thus at most $n-1$ in total. □

This allows us to bound maximum s-aware pi-blocking as follows.

Theorem 6.6. *Let τ denote a set of n constrained-deadline tasks. If τ is HRT schedulable under G-EDF and the global FMLP on m processors, then jobs incur at most $O(n)$ maximum s-aware pi-blocking.*

Proof. Under the global FMLP, a job J_i incurs pi-blocking only if a lower-priority, resource-holding job J_l inherits the priority of a job J_h with a priority higher than that of J_i. For this to occur, J_l must execute while J_i is pending. By Lemma 6.21, at most $O(n)$ such jobs execute while J_i is pending. Each lower-priority job issues at most $\sum_{q=1}^{n_r} N_{l,q} = O(1)$ requests, each of which requires at most $L^{max} = O(1)$ time units to complete (recall Section 6.1.2). Therefore, lower-priority jobs inherit higher priorities for at most $O(n)$ time units while J_i is pending, which bounds the maximum duration of s-aware pi-blocking. □

An argument analogous to Lemma 6.21 can be used to show that at most $2 \cdot (n-1)$ lower-priority jobs overlap with any J_i under G-FP scheduling with RM priorities, constrained deadlines, and HRT correctness, which also yields $O(n)$ maximum s-aware pi-blocking under the global FMLP in this case. Further, in recent work, Leontyev *et al*. (2009) and Erickson and Anderson (2011) studied a class of *G-EDF-like schedulers* that prioritize each job by a fixed point in time after its release. Theorem 6.6 can be transferred to such schedulers if each task's *relative priority point* does not exceed its period (analogously to constrained deadlines under G-EDF). Notably, this class of "constrained, fixed priority-point schedulers" includes global FIFO scheduling.

With regard to τ^{ϕ} and the example shown in Figure 6.13, Theorem 6.6 shows that it is fundamental that T_4 has an arbitrary deadline with $d_i > p_i$. This shows that s-aware pi-blocking is in large parts determined by the frequent arrival of both lower- and higher-priority jobs of tasks with short periods that, together, cause a long-running job to repeatedly incur pi-blocking. Priority inheritance does not adequately deal with such repeated arrivals. However, this problem is not isolated to priority inheritance.

Priority boosting. Other options besides priority inheritance for ensuring resource-holder progress are priority boosting and priority donation. Of these, priority donation is unsuitable since Rule D1 (page 432) is incompatible with s-aware schedulability analysis (a job may incur s-aware pi-blocking while waiting to become one of the c highest-priority pending jobs). However, priority boosting is not suitable for designing $O(n)$ protocols either. This is illustrated in Figure 6.14, which shows a

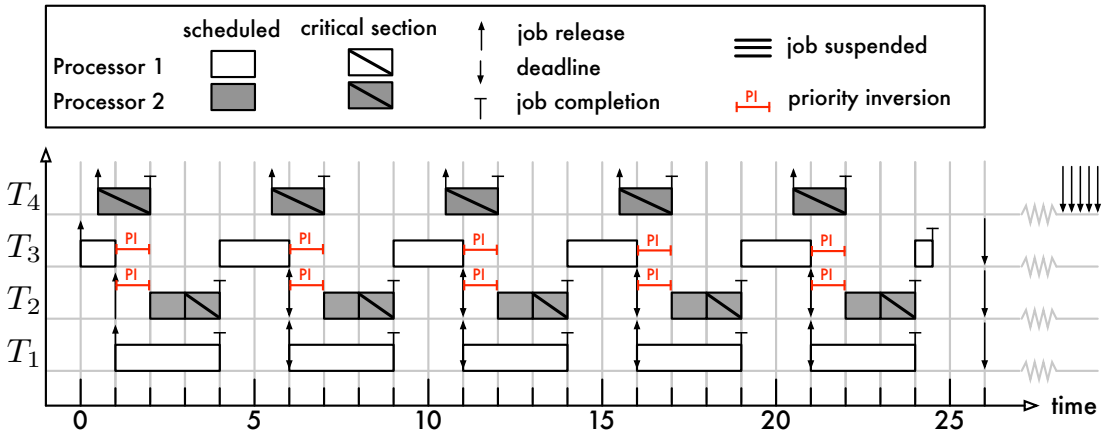


Figure 6.14: G-EDF example schedule of the task set τ^{ϕ} as given in Table 6.2 for $m = 2$ and $\phi = 5$ assuming unrestricted priority boosting instead of priority inheritance. $J_{3,1}$ still incurs $\Omega(\phi)$ time units of s-aware pi-blocking. Unrestricted priority boosting is thus unsuitable for constructing locking protocols with $O(n)$ maximum s-aware pi-blocking.

schedule of τ^{ϕ} assuming priority boosting instead of priority inheritance. The depicted schedule demonstrates that $\Omega(\phi)$ s-aware pi-blocking is possible if resource-holder progress is enforced with unrestricted priority boosting.

The fundamental problem with the arrival sequence shown in Figures 6.13 and 6.14 is that $J_{4,j}$ must be scheduled to release the shared resource to avoid blocking $J_{2,j}$. (If progress of $J_{4,j}$ is not enforced, then one can construct task sets that give rise to unbounded priority inversions.) Regardless of whether this is achieved with priority inheritance or priority boosting, if $J_{4,j}$ is scheduled and $J_{3,1}$ is not, then $J_{3,1}$ necessarily incurs a priority inversion. Therefore, there are only two options to avoid $\Omega(\phi)$ s-aware pi-blocking in this scenario: either the resource request of each $J_{4,j}$ must not be satisfied immediately (despite being uncontested), or some job other than $J_{3,1}$ must be selected to incur s-aware pi-blocking instead of $J_{3,1}$. The former is difficult to formalize for *sporadic* tasks since future releases are unknown;[6] the latter seems more promising. In particular, in this specific example, each $J_{1,j}$ could serve as a priority donor for each $J_{4,j}$, thereby preventing $J_{3,1}$ from incurring any s-aware pi-blocking. However, in the general case, this approach fails if there are more than $m - 1$ jobs like $J_{3,1}$ that must be protected from repeated preemptions (*i.e.*, if there are more than $m - 1$ jobs at risk of $\Omega(\phi)$ s-aware pi-blocking that must be scheduled).

[6]Note that this is different from the PCP's system ceiling rule: under the PCP, *some* resource is being held if the system ceiling prevents a request for an available resource from being satisfied immediately (see Section 2.4.3), whereas the request of each $J_{4,j}$ would have to be delayed while no job holds any resources.

To summarize, straightforward priority inheritance or priority boosting as the sole progress mechanism does not yield locking protocols with $O(n)$ maximum s-aware pi-blocking for all JLFP policies. Nonetheless, the global FMLP does ensure $O(n)$ maximum s-aware pi-blocking for constrained-deadline task sets that are HRT schedulable under G-EDF scheduling or G-FP scheduling with RM priorities. This raises a number of questions, which we discuss in Section 6.3.4 after first establishing in the next section that locking protocols based on priority queues (instead of FIFO queues) are similarly vulnerable to repeated higher-priority job arrivals.

6.3.3 Maximum Pi-Blocking in Priority Queues

Intuitively, one might reasonably expect queuing disciplines that order lock requests on an FP or EDF basis to cause no more blocking than simple FIFO queuing. However, asymptotically speaking, this is not the case. Consider the following implicit-deadline task set.

Definition 6.7. Let $\tau^{prio}(n)$, where $n \geq 2m$, denote a set of n tasks sharing one resource ℓ_1 such that, for each T_i, $e_i = 1$, $N_{i,1} = 1$, $L_{i,1} = 1$, and $p_i = m$ if $i < m$, $p_i = mn/2$ if $m \leq i \leq 2m - 2$, and $p_i = mn$ otherwise.

Lemma 6.22. *There exists an arrival sequence for $\tau^{prio}(n)$ such that $\max_{1 \leq i \leq n}\{b_i\} = \Omega(mn)$ under s-aware analysis when ordering requests by either non-decreasing job deadline or fixed priority under any JLFP scheduler.*

Proof. Without loss of generality, assume that n is an integer multiple of $2m$. We first consider partitioned scheduling ($c = 1$) and assume that $\tau^{prio}(n)$ is partitioned such that $P_i = i$ if $i < m$ and $P_i = m$ otherwise.

Consider the synchronous, periodic arrival sequence, that is, each $J_{i,j}$ is released at $a_{i,j} = (j - 1) \cdot p_i$, and issues one request $\mathcal{R}_{i,q,v}$, where $\mathcal{L}_{i,q,v} = 1$. The resulting schedule for $m = 3$ and $n = 6$ is illustrated in Figure 6.15.

Since the task set is serialized by ℓ_1, the order of job completions is fully determined by the queuing discipline under any work-conserving scheduler. Recall from Chapter 2 that tasks are indexed in order of decreasing priority, and that deadline ties are broken in favor of jobs of higher-indexed tasks. Thus, if requests are either EDF- or FP-ordered, then, by construction, $J_{n,1}$'s request is the last one to be satisfied at time $n \cdot m - 1$. By Definition 6.2, $J_{n,1}$ incurs pi-blocking

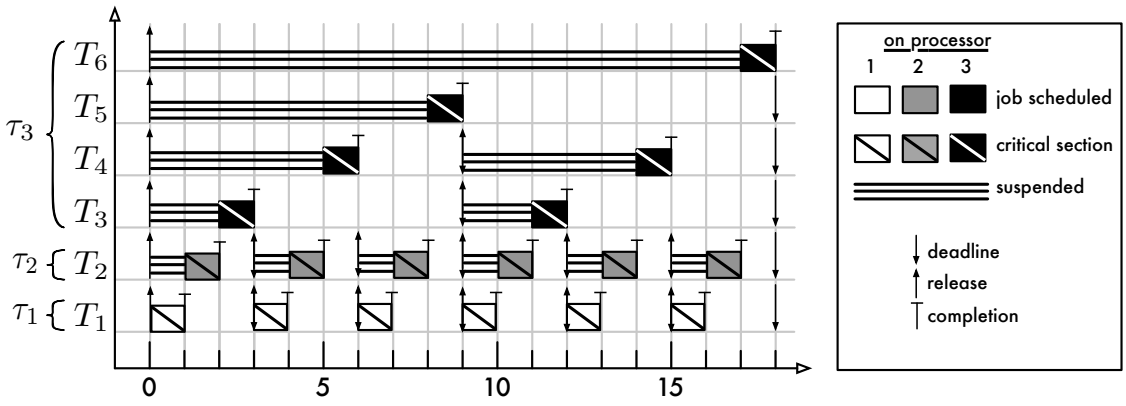


Figure 6.15: Illustration of $\tau^{prio}(n)$ for $n = 6$ and $m = 3$. The depicted schedule arises under any partitioned JLFP scheduler if requests are ordered either by deadline or by fixed priority (with task indexed in order of decreasing priority). An equivalent schedule arises under global scheduling since only one job is scheduled at any time. J_6 incurs pi-blocking for $(m-1) \cdot n = 12$ time units under partitioning and for $mn - 1 = 17$ time units under global scheduling (throughout $[0, 17)$ only one job is scheduled).

whenever no higher-priority job is scheduled on P_n during $[0, n \cdot m - 1)$. By construction, P_n is used for only $n - 1$ time units during $[0, n \cdot m - 1)$. Thus, $J_{n,1}$ is pi-blocked for at least $b_n \geq n \cdot m - 1 - (n-1) = (m-1) \cdot n = \Omega(mn)$ time units. Since at most one job is scheduled at any time, s-aware pi-blocking does not decrease if $c > 1$. □

Note that example shown in Figure 6.15 does not depend on resource-holder progress being enforced. This shows that priority queuing gives rise to non-optimal maximum s-aware pi-blocking regardless of whether priority boosting, priority inheritance, or some other, yet to be developed progress mechanism is employed, and even if implicit deadlines are assumed.

6.3.4 Open Questions

In the case of s-aware schedulability analysis, we have established an $\Omega(n)$ lower bound on maximum pi-blocking under any clustered JLFP scheduler (Section 6.1.4). In the case of $c = 1$, the existence of the FMLP$^+$ shows this bound to be asymptotically tight. However, in the case of $c > 1$, to the best of our knowledge, no locking protocol is known to ensure $O(n)$ maximum s-aware pi-blocking under *any* JLFP scheduler.

The global FMLP does achieve $O(n)$ maximum pi-blocking in special cases in which there are only $O(n)$ concurrent, lower-priority jobs, but in the general case it is subject to $\Omega(\phi)$ maximum

pi-blocking, where ϕ corresponds to the ratio of the maximum and minimum period lengths and hence is not bounded by m or n (recall Section 6.1.2). This lower bound, however, is not dependent on the FMLP's use of FIFO queues; instead, it arises for any queue choice when resource-holder progress is ensured by means of priority inheritance or priority boosting (Section 6.3.2).

Conversely, the use of priority queues gives rise to $\Omega(mn)$ maximum s-aware pi-blocking, regardless of the choice of progress mechanism. This seems to imply that—from a purely asymptotical point of view—priority queues are ill-suited to multiprocessor real-time locking since they are prone to starvation if high-priority tasks with short periods issue many requests.

These observations highlight a number of open questions, which we hope to investigate in greater detail in future work.

- *Is $\Omega(n)$ maximum s-aware pi-blocking an asymptotically tight lower bound for $c > 1$?* Since no locking protocol for the case of $c > 1$, or even the (presumably simpler) case of $c = m$, with $O(n)$ maximum s-aware pi-blocking is currently known (with respect to arbitrary JLFP scheduling policies or arbitrary deadlines), it cannot be ruled out that a larger lower bound exists. Given that such protocols exist for partitioned scheduling ($c = 1$), it would be surprising to find such a fundamental difference between partitioned and global scheduling. However, it is entirely possible that the case of $c = 1$, which lacks intra-cluster concurrency, poses an asymptotically simpler synchronization problem. One reason this might indeed be the case is that the main source of analytical complications in case of $c > 1$ is resource requests by concurrently-scheduled, lower-priority jobs that subsequently require intervention by the progress mechanism.

- *Are there other progress mechanisms suitable for s-aware analysis besides priority inheritance and priority boosting?* Assuming that $\Omega(n)$ is indeed an asymptotically tight lower bound for $c > 1$ as well, task sets such as τ^{ϕ} highlight that a new progress mechanism will be required to guarantee $O(n)$ maximum s-aware pi-blocking. In particular, new ways of dealing with frequently-arriving lower-priority jobs will have to be developed. This is much more difficult than in the s-oblivious case since scheduled lower-priority jobs that issue potentially problematic requests cannot be delayed without incurring s-aware pi-blocking themselves (*i.e.*, priority donation Rule D1 does not transfer to s-aware analysis).

- *Can hybrid queues be applied to limit s-aware pi-blocking?* With respect to the previous question, it is worth considering whether a combination of FIFO and priority queues can be applied to the s-aware case. However, since jobs are typically suspended while enqueued, the queue structure seems to have little impact on s-aware pi-blocking (if starvation is avoided).

Given that fundamental questions concerning mutex constraints are still open, it is hardly surprising that RW exclusion and k-exclusion are even less understood. In fact, to the best of our knowledge, the OMLP variants for both relaxed-exclusion constraints are the first of their kind with regard to suspension-based multiprocessor locking protocols. Since the OMLP is targeted at s-oblivious analysis, the RW exclusion and k-exclusion problems remain unsolved in the case of s-aware schedulability analysis (aside from simply using a mutex protocol such as the FMLP$^+$).

It would certainly be possible to extend the FMLP$^+$ by mimicking the OMLP's rules for RW and k-exclusion. However, the resulting protocols, while amenable to worst-case analysis, would still cause a suspending job to be subject to $\Omega(n)$ s-aware pi-blocking after it resumes due to the use of priority boosting in the FMLP$^+$ (*i.e.*, it is possible to construct schedules in which Lemma 6.18 is tight, even in the case of relaxed-exclusion constraints). Similarly, it is not clear that worst-case acquisition delay could be improved beyond $O(n)$ maximum s-aware pi-blocking since resource-holding jobs could be preceded by $\Omega(n)$ priority-boosted jobs executing earlier-issued requests. In other words, priority boosting causes such significant delays that the benefit of RW or k-exclusion protocols may be marginal. At the very least, a novel approach to priority boosting will be required to harness the potential for increased parallelism under relaxed-exclusion constraints.

In light of this chapter's focus on blocking optimality, it is important to note that asymptotic optimality as a function of m or n does *not* imply that a locking protocol is the best to use in all circumstances. Obviously, asymptotic claims ignore constant factors. Additionally, a non-optimal algorithm could yield lower pi-blocking delays for some task systems. Empirical evaluations, ideally under consideration of overheads, are required to investigate such considerations.

6.4 Detailed Blocking Analysis

In this section, we apply the holistic blocking analysis framework from Section 5.4 to derive fine-grained (*i.e.*, non-asymptotic) bounds on maximum s-oblivious and s-aware pi-blocking for the five

locking protocols presented in this chapter, namely the four protocols in the OMLP family and the partitioned FMLP$^+$. We do not consider the global FMLP under G-EDF in detail because no s-aware HRT schedulability analysis for G-EDF has been published to date. As is the case with Section 5.4, the following analysis is quite technical in nature and may be safely skipped by the casual reader.

Recall that we use a per-task interference bound to characterize its maximum resource requirements (with regard to the frequency and duration of resource requests) during some interval. A key feature of this abstraction is its independence from analysis assumptions and how waiting is implemented—that is, whether or not a job requires a given resource does not depend on whether it will spin or suspend while waiting, and also not on whether suspensions are accounted for explicitly or modeled as execution time. As a result, the following bounds on maximum pi-blocking structurally resemble those given in Sections 5.4.3 and 5.4.6.

6.4.1 Global OMLP

We begin with the global OMLP for mutex constraints under s-oblivious schedulability analysis. Since the global OMLP uses a hybrid queue that consists of a FIFO queue FQ_q (which holds at most m waiting jobs) and of a priority queue PQ_q (which is only used if at least m jobs are waiting), maximum s-oblivious pi-blocking under the global OMLP depends on how many tasks share a given resource.

Definition 6.8. In the following, let $A_q \triangleq |\{T_i \mid T_i \in \tau \wedge N_{i,q} > 0\}|$ denote the number of tasks that access resource ℓ_q.

If $A_q \leq m + 1$, then at most m jobs are waiting to acquire ℓ_q at any time, which implies that at most one job is queued in PQ_q. In this case, the global OMLP reduces to a simple FIFO protocol.

Lemma 6.23. *Under the global OMLP, if $A_q \leq m + 1$, then a job J_i incurs at most*

$$b_{i,q} = total((A_q - 1) \cdot N_{i,q},\ tifs(\tau \setminus \{T_i\}, r_i, N_{i,q}))$$

s-oblivious pi-blocking due to requests for resource ℓ_q.

Proof. J_i's response time r_i upper-bounds the duration of the interval during which other jobs can issue conflicting requests; that is, the aggregate task interference bound $tifs(\tau \setminus \{T_i\}, r_i, N_{i,q})$ for

any interval of length r_i is a sufficient approximation of the resource demands of competing tasks. If J_i is never enqueued in PQ_q, then the lemma follows trivially.

Otherwise, if J_i is enqueued in PQ_q, then m jobs are already enqueued in FQ_q at the time of J_i's request. Since $A_q \leq m+1$, this implies that no other job is enqueued in PQ_q. As soon as the head of FQ_q releases ℓ_q, J_i is moved to FQ_q. Hence there is at most one job in PQ_q at any time, and the ordering of PQ_q is irrelevant.

The FIFO ordering of FQ_a implies that each of J_i's requests is preceded by at most one request from each other task that accesses ℓ_q. The per-task interference limit is hence $N_{i,q}$. Since ℓ_q is shared among only $A_q \leq m+1$ tasks, one of which is T_i, no more than $(A_q - 1) \cdot N_{i,q}$ requests pi-block J_i in total. Priority inheritance ensures that the resource-holding job is scheduled whenever J_i incurs s-oblivious pi-blocking; the cumulative duration of the $(A_q - 1) \cdot N_{i,q}$ longest requests for ℓ_q by tasks other than J_i thus bounds maximum s-oblivious pi-blocking. □

In the case of $A_q > m+1$, higher-priority jobs of some other task T_x may "skip ahead" of J_i repeatedly while J_i waits in PQ_q. However, the per-task interference limit is still limited to $2 \cdot N_{i,q}$, that is, the per-task interference limit is only doubled even if jobs of T_x "skip ahead" an arbitrary number of times.

Lemma 6.24. *Let T_x denote some task other than T_i that accesses ℓ_q (i.e., $T_i \neq T_x$ and $N_{x,q} > 1$). Under the global OMLP, jobs of T_x cause J_i to incur s-oblivious pi-blocking for at most the duration of two requests each time that J_i requests ℓ_q.*

Proof. In order to pi-block J_i, a request issued by some J_x must precede J_i's request in FQ_q (*i.e.*, J_x enters FQ_q before J_i does). If $A_q \leq m$, the bound follows immediately since FQ_q is FIFO-ordered.

Hence assume $A_q > m$. In this case, jobs of T_x may enter FQ_q repeatedly while J_i waits in PQ_q. Let t_a denote the first time that a job of T_x, denoted $J_{x,a}$, enters FQ_q, and let t_b denote the second time that a job of T_x, denoted $J_{x,b}$, enters FQ_q while J_i is continuously waiting in PQ_q. Further, let t_1 denote the time that J_i enters FQ_q (as indicated in Figure 6.11). If t_1 does not exist (*i.e.*, if J_i never enters FQ_q), then either FQ_q is continuously populated with higher-priority jobs and J_i does not incur s-oblivious pi-blocking, or some requests fails to complete (which is not possible since each $L_{i,q}$ is presumed finite). Therefore assume t_1 exists.

Since tasks are sequential, $J_{x,b}$ necessarily issued its request after J_i issued its request (this is not necessarily the case with $J_{x,a}$).

J_i does not incur s-oblivious pi-blocking during $[t_b, t_1)$. Since J_i is waiting in PQ_q at time t_a, $J_{x,a}$ is necessarily preceded by $m-1$ other jobs in FQ_q, which must complete before $J_{x,a}$'s request is satisfied. Since tasks are sequential, $J_{x,a}$ has completed its request before $J_{x,b}$ enters FQ_q at time t_b. Therefore, at least m higher-priority jobs must have entered FQ_q during $[t_a, t_b)$; otherwise, J_i would no longer be waiting in PQ_q at time t_b. The presence of m higher-priority pending jobs rules out s-oblivious pi-blocking after t_b (until J_i enters FQ_q at time t_1).

Therefore, at most one of the requests issued by jobs of T_x *after* J_i issued its request pi-blocks J_i. Since sporadic tasks are sequential, at most one request of T_x that was issued *prior* to J_i's request is incomplete when J_i issues its request. Hence, at most two requests of T_x cause J_i to incur pi-blocking. □

As a result, the per-task interference limit in the case of $A_q > m$ is $2 \cdot N_{i,q}$. This yields the following bound.

Lemma 6.25. *Under the global OMLP, if $A_q > m$, then a job J_i incurs at most*

$$b_{i,q} = total((2 \cdot m - 1) \cdot N_{i,q},\ tifs(\tau \setminus \{T_i\}, r_i, 2 \cdot N_{i,q}))$$

s-oblivious pi-blocking due to requests for resource ℓ_q.

Proof. By Lemma 6.15, J_i incurs s-oblivious pi-blocking for the combined duration of at most $2 \cdot m - 1$ requests each time that it requests ℓ_q, which implies that J_i is delayed by at most $(2 \cdot m - 1) \cdot N_{i,q}$ requests in total. Lemma 6.24 implies an interference limit of $2 \cdot N_{i,q}$. Priority inheritance ensures that the resource-holding job is scheduled whenever J_i incurs pi-blocking. The bound follows. □

This yields the following overall bound on maximum s-oblivious pi-blocking.

Theorem 6.7. *Under the global OMLP, a job J_i incurs s-oblivious pi-blocking for at most*

$$b_i = \sum_{q=1}^{n_r} total((x_q - 1) \cdot N_{i,q},\ tifs(\tau \setminus \{T_i\}, r_i, l_q \cdot N_{i,q}))$$

time units, where $x_q = A_q$ *and* $l_q = 1$ *if* $A_q \leq m$*, and* $x_q = 2 \cdot m$ *and* $l_q = 2$ *if* $A_q > m$*.*

Proof. Follows from Lemmas 6.23 and 6.25, since resource requests are not nested, and since J_i does not incur s-oblivious pi-blocking under the global OMLP while not requesting resources. □

This concludes the analysis of the global OMLP. Next, we consider the clustered OMLP from Sections 6.2.2–6.2.4, which uses priority donation instead of priority inheritance.

6.4.2 Clustered OMLP

A job J_i is subject to two sources of s-oblivious pi-blocking under the clustered OMLP. J_i can be delayed each time it issues requests for shared resources, and additionally once upon release if it serves as a priority donor. These two sources of delays are analogous to s-blocking and pi-blocking, respectively, in non-preemptive spinlock protocols. In fact, the bounds for the clustered OMLP's mutex and phase-fair RW protocols are structurally identical to the bounds for non-preemptive task-fair mutex and phase-fair RW spinlock protocols.

6.4.2.1 Mutual Exclusion

We begin with the mutex variant of the clustered OMLP, which is the simplest of the three protocols based on priority donation. Recall from Section 6.2.2 that each resource ℓ_q is protected by a simple FIFO queue FQ_q.

Lemma 6.26. *Under the clustered OMLP's mutex protocol, a job* J_i *incurs at most*

$$b_{i,q,j} = \begin{cases} total(N_{i,q} \cdot c,\ tifs(\tau_j, \ell_q, r_i, N_{i,q})) & \text{if } j \neq P_i \\ total(N_{i,q} \cdot (c-1),\ tifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, N_{i,q})) & \text{if } j = P_i \end{cases}$$

pi-blocking due to requests for resource ℓ_q *issued by jobs of tasks assigned to the* j^{th} *cluster.*

Proof. Analogously to Lemma 5.2. By Lemma 6.5, priority donation ensures that at most c requests are incomplete at any time in each cluster; therefore, at most c requests precede J_i in FQ_q each time that it issues a request. The strict FIFO ordering in FQ_q ensures a per-task interference limit of $N_{i,q}$. Due to priority donation, resource-holding jobs are always scheduled (Lemma 6.3). In the case of

J_i's local cluster, only $c-1$ requests can interfere since J_i's own request counts towards the limit of c concurrent requests imposed by priority donation. □

Pi-blocking that J_i incurs while serving as a priority donor is similar to pi-blocking that a job may incur due to non-preemptive request execution under non-preemptive spinlock protocols. Recall from Definition 5.8 that we let $lower(T_i)$ denote the set of tasks local to T_i that could potentially cause J_i to incur pi-blocking upon release.

Lemma 6.27. *Under the clustered OMLP's mutex protocol, a job J_i incurs at most b_i^D s-oblivious pi-blocking upon release while serving as a priority donor, where*

$$b_i^D = \max_{1 \leq q \leq n_r} \max_{\substack{T_x \in lower(T_i) \\ N_{x,q} > 0}} \left(L_{x,q} + \sum_{j=1}^{m/c} b'_{x,q,j} \right), \quad \textit{and}$$

$$b'_{x,q,j} = \begin{cases} total(c,\ tifs(\tau_j, \ell_q, r_x, 1)) & \textit{if } j \neq P_x, \\ total(c-1,\ tifs(\tau_j \setminus \{T_i, T_x\}, \ell_q, r_x, 1)) & \textit{if } j = P_x. \end{cases}$$

Proof. By Lemma 6.4, maximum s-oblivious pi-blocking due to priority donation is limited to one request span. Analogously to Lemma 5.3 and Theorem 5.2, b_i^D bounds the maximum request span of any local, potentially lower-priority job J_x by considering the c longest requests in each remote cluster that could cause J_x to incur acquisition delay, and the $c-1$ longest requests in J_x's local cluster. □

Theorem 6.8. *Under the clustered OMLP's mutex protocol, a job J_i incurs at most*

$$b_i = b_i^D + \sum_{q=1}^{n_r} \sum_{j=1}^{m/c} b_{i,q,j}$$

s-oblivious pi-blocking due to requests for shared resources, where $b_{i,q,j}$ and b_i^D are defined as in Lemmas 6.26 and 6.27, respectively.

Proof. Follows from Lemmas 6.26 and 6.27, and the assumptions that resource requests are not nested and that tasks do not migrate across cluster boundaries. □

6.4.2.2 Reader-Writer Exclusion

The bounds on maximum pi-blocking under the OMLP's RW protocol are structurally equivalent to the bounds on maximum s-blocking and pi-blocking under non-preemptive phase-fair RW spinlocks presented in Section 5.4.6. This is because the OMLP implements phase-fairness, and because priority donation allows at most c concurrent requests in each cluster, which has an effect that is equivalent to non-preemptive execution. For the sake of convenience, the bound is restated here in terms of s-oblivious pi-blocking for the clustered OMLP's RW protocol.

Definition 6.9. In the following, let $x^{rem} = N_{i,q}^W \cdot c + N_{i,q}^R$ and $x^{loc} = N_{i,q}^W \cdot (c-1) + N_{i,q}^R$, and define the sets of possibly-interfering write requests from jobs in the j^{th} cluster, denoted as $W(T_i, j, \ell_q)$, as follows.

$$W(T_i, j, \ell_q) = \begin{cases} top(x^{rem},\ wifs(\tau_j, \ell_q, r_i, (N_{i,q}^W + N_{i,q}^R))) & \text{if } j \neq P_i \\ top(x^{loc},\ wifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, (N_{i,q}^W + N_{i,q}^R))) & \text{if } j = P_i \text{ and } c > 1 \\ \emptyset & \text{if } j = P_i \text{ and } c = 1 \end{cases}$$

Further, let $W_{i,q}$ denote the union of all possibly-interfering write requests across all clusters, and let $w_{i,q}$ denote the maximum number of blocking write requests.

$$W_{i,q} = \bigcup_{j=1}^{m/c} W(T_i, j, \ell_q) \qquad\qquad w_{i,q} = |W_{i,q}|$$

Definition 6.10. Let $r_{i,q} = \min(w_{i,q} + N_{i,q}^W,\ N_{i,q}^R + (m-1) \cdot N_{i,q}^W)$, and define the sets of possibly-interfering read requests from jobs in the j^{th} cluster, denoted as $R(T_i, j, \ell_q)$, as follows.

$$R(T_i, j, \ell_q) = \begin{cases} top(r_{i,q},\ rifs(\tau_j, \ell_q, r_i, r_{i,q})) & \text{if } j \neq P_i \\ top(r_{i,q},\ rifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, r_{i,q})) & \text{if } j = P_i \text{ and } c > 1 \\ \emptyset & \text{if } j = P_i \text{ and } c = 1 \end{cases}$$

Analogously to $W_{i,q}$, let $R_{i,q}$ denote the set of all possibly interfering read requests across all clusters.

$$R_{i,q} = \bigcup_{j=1}^{m/c} R(T_i, j, \ell_q)$$

This yields the following bound on maximum delay due to requests for a given resource.

Lemma 6.28. *Under the clustered OMLP's RW protocol, a job J_i incurs pi-blocking due its read and write requests for resource ℓ_q for at most $b_{i,q} = total(w_{i,q},\ W_{i,q}) + total(r_{i,q},\ R_{i,q})$ time units.*

Proof. Analogously to the discussion in Section 5.4.6. Each time that J_i issues a write request, it can be preceded by up to c other write requests in each cluster since the writer queue WQ_q is FIFO ordered, and because priority donation allows at most c concurrent requests per cluster. Also due to the FIFO order, each other task can block each of J_i's write requests with at most one request. Each time that J_i issues a read request, it is blocked by at most one write request since the OMLP implements phase-fairness. Therefore, the per-task interference with regard to write requests is $N_{i,q}^W + N_{i,q}^R$, and in total J_i's $N_{i,q}^R$ read requests and $N_{i,q}^W$ write requests are blocked by at most $N_{i,q}^R + N_{i,q}^W \cdot c$ write requests in the case of a remote cluster, and by at most $N_{i,q}^R + N_{i,q}^W \cdot (c-1)$ requests in the case of J_i's local cluster. The definitions of $W(T_i, j, \ell_{i,q})$ and $W_{i,q}$ follow.

By Lemma 5.4, the upper bound on the total number of blocking writes $w_{i,q}$ implies an upper bound of $w_{i,q} + N_{i,q}^W$ on the number of blocking reader phases. The total number of blocking reader phases is also limited to $N_{i,q}^R + (m-1) \cdot N_{i,q}^W$: due to priority donation and because reader and writer phases alternate in a phase-fair RW lock, each of J_i's read requests is transitively blocked by at most one reader phase, and each of J_i's write requests is blocked by at most $m-1$ interspersed reader phases (since at most $m-1$ write requests block each of J_i's write requests). The lesser of the two bounds limits the total number of blocking reader phases $r_{i,q}$. The definitions of $R(T_i, j, \ell_q)$ and $R_{i,q}$ follow.

Since J_i is blocked by at most $w_{i,q}$ writer phases and $r_{i,q}$ reader phases, total s-oblivious pi-blocking is bounded by the $w_{i,q}$ longest requests in $W_{i,q}$ and the $r_{i,q}$ longest request in $R_{i,q}$. □

Since the clustered OMLP uses priority donation, a job may also incur s-oblivious pi-blocking when serving as a priority donor. The duration of priority donation depends on the request span of the priority recipient's request, which may be either a write or a read. The maximum acquisition delay of

a single write request for resource ℓ_q issued by job J_i can be bounded by instantiating Definitions 6.9 and 6.10 assuming $N_{i,q}^R = 0$ and $N_{i,q}^W = 1$. Similarly, the maximum acquisition delay of a read request for ℓ_q can be bounded by instantiating said definitions assuming $N_{i,q}^R = 1$ and $N_{i,q}^W = 0$. To avoid needless repetition, we use the following definitions to denote these two special cases.

Definition 6.11. Let $W'_{i,q}$ and $w'_{i,q}$ denote the values of $W_{i,q}$ and $w_{i,q}$, respectively, that result when assuming $N_{i,q}^R = 0$ and $N_{i,q}^W = 1$ in Definition 6.9 above. Similarly, let $W''_{i,q}$ and $w''_{i,q}$ denote the values of $W_{i,q}$ and $w_{i,q}$, respectively, that result when assuming $N_{i,q}^R = 1$ and $N_{i,q}^W = 0$ in Definition 6.9 above.

Definition 6.12. Let $R'_{i,q}$ and $r'_{i,q}$ denote the values of $R_{i,q}$ and $r_{i,q}$, respectively, that result when assuming $N_{i,q}^R = 0$ and $N_{i,q}^W = 1$ in Definition 6.10 above. Similarly, let $R''_{i,q}$ and $R''_{i,q}$ denote the values of $R_{i,q}$ and $R_{i,q}$, respectively, that result when assuming $N_{i,q}^R = 1$ and $N_{i,q}^W = 0$ in Definition 6.10 above.

With these definitions in place, we can express the maximum duration that J_i may have to serve as a priority donor. Recall from Definition 5.8 that we let $lower(T_i)$ denote the set of tasks local to T_i that could potentially cause J_i to incur pi-blocking upon release.

Lemma 6.29. *Under the clustered OMLP's RW protocol, a job J_i incurs at most $b_i^D = \max(b'_i, b''_i)$ s-oblivious pi-blocking upon release while serving as a priority donor, where*

$$b'_i = \max_{1 \le q \le n_r} \max_{\substack{T_x \in lower(T_i) \\ N_{x,q}^W > 0}} \left\{ L_{x,q}^W + b'_{x,q} \right\}, \text{ and}$$

$$b'_{x,q} = total(w'_{x,q},\ W'_{x,q}) + total(r'_{x,q},\ R'_{x,q}),$$

bounds the case of a writing priority recipient, and where

$$b''_i = \max_{1 \le q \le n_r} \max_{\substack{T_x \in lower(T_i) \\ N_{x,q}^R > 0}} \left\{ L_{x,q}^R + b''_{x,q} \right\}, \text{ and}$$

$$b''_{x,q} = total(w''_{x,q},\ W''_{x,q}) + total(r''_{x,q},\ R''_{x,q}).$$

bounds the case of a reading priority recipient.

Proof. Follows analogously to Lemma 6.27 since J_i serves as a priority donor at most once and at most for the duration of one request span. The maximum request span of a lower-priority write request is bounded by b'_i; the maximum request span of a lower-priority read reqeust is bounded by b''_i. The maximum of either scenario bounds maximum s-oblivious pi-blocking due to priority donation under the clustered OMLP for RW exclusion. □

This yields the following bound on s-oblivious pi-blocking.

Theorem 6.9. *Under the clustered OMLP's mutex protocol, a job J_i incurs at most*

$$b_i = b_i^D + \sum_{q=1}^{n_r} b_{i,q}$$

s-oblivious pi-blocking due to read and write requests for shared resources, where $b_{i,q}$ and b_i^D are defined as in Lemmas 6.28 and 6.29, respectively.

Proof. Follows from Lemmas 6.28 and 6.29, and the assumptions that resource requests are not nested and that tasks do not migrate across cluster boundaries. □

6.4.2.3 k-Exclusion

In this section, we establish a bound on s-oblivious pi-blocking under the clustered OMLP for k-exclusion, which is presented in Section 6.2.4. As we did not consider spin-based k-exclusion protocols (since we are not aware of any real-world applications that make use of k-exclusion spinlocks), there is no equivalent to the following analysis in Chapter 5. Nonetheless, a non-preemptive k-exclusion spinlock could be analyzed analogously to the following discussion.

The presented analysis is reasonably tight if blocking requests are relatively uniform in duration. However, if request lengths are heavily skewed (*i.e.*, if there are some infrequent, long-running requests, but most requests are short), then a more accurate bound could likely be obtained by applying multiprocessor response-time analysis for non-preemptive global FIFO scheduling to each resource. In the following simpler analysis, which suffices for our purposes, some pessimism arises because Lemma 6.12, which implicitly lower-bounds the request completion rate, does not take non-uniform request lengths into account.

Lemma 6.30. *Under the clustered OMLP's k-exclusion protocol, a job J_i incurs at most $b_{i,q}$ s-oblivious pi-blocking due to requests for resource ℓ_q, where*

$$b_{i,q} = total\left(N_{i,q} \cdot \left\lceil \frac{m - k_q}{k_q} \right\rceil, \bigcup_{j=1}^{m/c} b_{i,q,j}\right), \quad \text{and}$$

$$b_{i,q,j} = \begin{cases} top(N_{i,q} \cdot c,\ tifs(\tau_j, \ell_q, r_i, N_{i,q})) & \text{if } j \neq P_i \\ top(N_{i,q} \cdot (c-1),\ tifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, N_{i,q})) & \text{if } j = P_i. \end{cases}$$

Proof. By Lemma 6.5, priority donation ensures that at most c requests are incomplete at any time in each cluster; therefore, at most c requests in each cluster precede J_i in KQ_q or hold a replica of ℓ_q at the time that J_i issues a request. The FIFO ordering of jobs in KQ_q ensures a per-task interference limit of $N_{i,q}$. Therefore, the set of the $N_{i,q} \cdot c$ longest requests issued by jobs in the j^{th} cluster, denoted $b_{i,q,j}$, bounds the worst-case interference from jobs in that cluster. In the case of J_i's local cluster, only $c - 1$ requests can interfere since J_i's own request counts towards the limit of c concurrent requests imposed by priority donation.

Lemma 6.12 implies that J_i holds a replica of ℓ_q after at most $\lceil (m - k_q)/k_q \rceil$ prior requests for ℓ_q complete. Therefore, across all $N_{i,q}$ requests, J_i is pi-blocked at most for the cumulative duration of the $N_{i,q} \cdot \lceil (m - k_q)/k_q \rceil$ longest requests issued by jobs in any cluster. □

To bound maximum s-oblivious pi-blocking due to priority donation, we again require a bound for a single request. Such a bound can be obtained by applying Lemma 6.30 above to a single request.

Definition 6.13. Let $b'_{i,q}$ denote the value of $b_{i,q}$ computed assuming $N_{i,q} = 1$ in Lemma 6.30 above.

Recall from Definition 5.8 that we let $lower(T_i)$ denote the set of tasks local to T_i that could potentially cause J_i to incur pi-blocking upon release.

Lemma 6.31. *Under the clustered OMLP's k-exclusion protocol, a job J_i incurs at most*

$$b_i^D = \max_{1 \leq q \leq n_r} \quad \max_{\substack{T_x \in lower(T_i) \\ N_{x,q} > 0}} \{L_{x,q} + b'_{x,q}\}$$

s-oblivious pi-blocking upon release while serving as a priority donor.

Proof. Follows analogously to Lemma 6.27 and Lemma 6.29. □

Theorem 6.10. *Under the clustered OMLP's k-exclusion protocol, a job J_i incurs at most*

$$b_i = b_i^D + \sum_{q=1}^{n_r} b_{i,q}$$

s-oblivious pi-blocking due to requests for shared resources, where $b_{i,q}$ and b_i^D are defined as in Lemmas 6.30 and 6.31, respectively.

Proof. Follows from Lemmas 6.30 and 6.31, and since resource requests are not nested. □

This completes our analysis of s-oblivious pi-blocking under clustered JLFP scheduling with arbitrary cluster sizes. Next, we consider s-aware pi-blocking under the FMLP$^+$ in the case of partitioned JLFP scheduling.

6.4.3 Partitioned FMLP$^+$

While s-aware analysis differs substantially from s-oblivious analysis, the structure of the underlying holistic analysis remains largely unchanged since each task's request interference bound is not affected by the way that suspension times are accounted for. In particular, since the partitioned FMLP$^+$ uses FIFO queues (like the clustered OMLP does), the per-task interference limit remains unchanged (at most one directly blocking request per request). What changes, however, is the upper bound on the number of requests that cause pi-blocking, which is $n-1$ per request under the FMLP$^+$.

Since s-aware schedulability analysis for P-FP scheduling requires separate bounds on local and remote pi-blocking (recall Theorem 2.14), we derive two corresponding bounds for the FMLP$^+$. The bound on s-aware pi-blocking due to remote tasks depends on whether requests are executed preemptively or non-preemptively. We begin with the preemptive case, under which there are fewer sources of pi-blocking.

6.4.3.1 Preemptive Request Execution

Recall from Section 6.3 that resource-holding jobs are priority-boosted under the FMLP$^+$, and that multiple resource-holding jobs are scheduled in order of the time at which they issued their respective requests. This tie-breaking rule, together with the preemptive execution of requests, ensures that a

request of J_i is only blocked by remote requests that were issued prior to its request. Therefore, each remote task can block J_i at most once per request, that is, the per-task interference limit is one.

Additionally, in order for a task T_x on a remote processor P_x (the processor to which T_x has been assigned) to block J_i (either directly or indirectly), J_i must be directly blocked by some job on processor P_x. That is, if the set of resources requested by J_i is disjoint from the set of resources accessed by tasks on processor P_x, then J_i is independent from those tasks. In general, if J_i accesses resources shared with tasks on processor P_x at most y times, than each task in τ_{P_x} can directly or indirectly block J_i at most y times. To express this constraint, we require the following two definitions.

Definition 6.14. Let db_j denote the maximum number of times that J_i is *directly blocked* by tasks in τ_j (*i.e.*, that a job of a task in τ_j precedes J_i in some queue FQ_q); formally

$$db_j \triangleq \left| \bigcup_{q=1}^{n_r} tifs\left(\tau_j, \ell_q, r_i, N_{i,q}\right) \right|.$$

Definition 6.15. Let D_j denote the maximum number of times that J_i requests resources that are also accessed by tasks in τ_j; formally,

$$D_j \triangleq \sum_{\ell_q \in A(j)} N_{i,q},$$

where $A(j) = \{\ell_q \mid \exists T_x \in \tau_j \colon N_{x,q} > 0\}$ denotes the set of resources accessed by tasks in τ_j.

With the above definitions in place, we can state the following bound.

Lemma 6.32. *Under the FMLP with preemptive request execution, a job J_i incurs at most*

$$b_i^r = \sum_{\substack{j=1 \\ j \neq P_i}}^{m/c} \sum_{T_x \in \tau_j} total\left(\min(db_j, D_j),\ \bigcup_{q=1}^{n_r} tif(T_x, \ell_q, r_i)\right)$$

s-aware pi-blocking due to requests for any resource issued by remote jobs.

Proof. By Lemma 6.17, each remote task T_x blocks J_i with at most one request, either directly or indirectly, each time that J_i requests a resource that is also accessed by tasks in τ_j. J_i issues such

requests at most D_j times. Further, each remote task T_x can indirectly block J_i only if J_i is directly blocked, which happens at most db_j times. Therefore, the maximum s-aware pi-blocking incurred by J_i due to requests of T_x is bounded by the duration of the $\min(db_j, D_j)$ longest requests issued by jobs of T_x for any resource. Summing across all remote processors and all tasks on those processors yields the aggregate bound on s-aware pi-blocking due to remote tasks. □

A similar argumen yields a bound on maximum s-aware pi-blocking due to local lower-priority tasks. Since the FMLP$^+$ is tied to partitioned scheduling, local higher-priority tasks never cause J_i to incur pi-blocking.

Definition 6.16. We let $lower'(T_i)$ denote the set of tasks local to T_i that may release lower-priority jobs that execute while J_i is pending. Under P-FP scheduling, $lower'(T_i) = \{T_l \mid T_l \in \tau_{P_i} \wedge l > i\}$ since we assume that tasks are indexed in order of decreasing priority. Under P-EDF scheduling, $lower'(T_i) = \{T_l \mid T_l \in \tau_{P_i} \setminus \{T_i\} \wedge d_l \geq d_i - r_i\}$ since such a task T_l may release a job with an absolute deadline that is no earlier than J_i's absolute deadline while J_i is still pending.

Under partitioned scheduling, lower-priority jobs can only issue requests for global resources when J_i suspends due to being blocked by a remote task (or if J_i self-suspends for locking-unrelated reasons). A straightforward bound on the number of locking-related suspensions is given by $\sum_{q=1}^{n_r} N_{i,q}$ (*i.e.*, in the worst case, J_i is blocked on each request). However, in corner cases where the resources requested by J_i are only infrequently contested, $\sum_{q=1}^{n_r} N_{i,q}$ may overestimate the number of times that J_i will suspend (due to being directly blocked). In such cases, it is worthwhile to consider a second bound that considers the number of times that J_i is directly blocked.

Definition 6.17. Let $z_{i,q}$ denote the maximum number of times that J_i suspends because one of its requests for resource ℓ_q is directly blocked; formally

$$z_{i,q} = \min(N_{i,q},\ |\mathit{tifs}\,(\tau \setminus \tau_{P_i}, \ell_q, r_i, N_{i,q})|)\,.$$

Lemma 6.33. *Under the FMLP, a job J_i incurs at most*

$$b_i^l = \sum_{T_x \in lower'(T_i)} total\left(1 + \sum_{q=1}^{n_r} z_{i,q},\ \bigcup_{q=1}^{n_r} tif(T_x, \ell_q, r_i)\right)$$

s-aware pi-blocking due to requests for any resource issued by local lower-priority jobs.

Proof. By Lemma 6.18, each local task T_l blocks J_i with at most one request (for any resource) after J_i is released and each time that J_i resumes from a self-suspension (*i.e.*, from being directly blocked by a remote task). By assumption J_i does not self-suspend for locking-unrelated reasons; J_i therefore resumes at most $\sum_{q=1}^{n_r} z_{i,q}$ times in total. Therefore, the maximum s-aware pi-blocking incurred by J_i due to requests of each T_l is bounded by the duration of the $1 + \sum_{q=1}^{n_r} z_{i,q}$ longest requests issued by jobs of each T_l for any resource. □

6.4.3.2 Non-Preemptive Request Execution

If requests are executed non-preemptively, then J_i is subject to additional s-aware pi-blocking from remote tasks (pi-blocking due to local tasks remains unchanged, as previously discussed on page 462). By Lemma 6.19, J_i can be delayed by the duration of one additional request each time that J_i is directly delayed by a remote job. That is, if J_i is directly blocked db_j times by tasks on the j^{th} processor, then J_i can incur additional transitive pi-blocking for the cumulative duration of at most db_j non-preemptive requests.

Lemma 6.34. *Under the FMLP with non-preemptive request execution, a job J_i incurs at most $b_i^r = b_i^{pre} + b_i^{np}$ s-aware pi-blocking due to requests for any resource issued by remote jobs, where*

$$b_i^{pre} = \sum_{\substack{j=1 \\ j \neq P_i}}^{m/c} \sum_{T_x \in \tau_j} total\left(\min(db_j, D_j),\ \bigcup_{q=1}^{n_r} tif(T_x, \ell_q, r_i)\right),$$

$$b_i^{np} = \sum_{\substack{j=1 \\ j \neq P_i}}^{m/c} total\left(db_j,\ \bigcup_{q=1}^{n_r} tifs(\tau_j, \ell_q, r_i, db_j)\right), \text{ and}$$

where db_j and D_j are defined as in Definition 6.14 and Definition 6.15, respectively.

Proof. The first term, b_i^{pre}, is equal to b_i^r in Lemma 6.32 and follows analogously. The second term, b_i^{np}, bounds additional delays due to non-preemptive request execution: each time that J_i is directly blocked by a remote jobs of tasks in τ_j, which occurs at most db_j times, J_i could be delayed for the duration of one critical section. The sum of the duration of the db_j's longest requests (for any resource) by tasks in τ_j thus upper-bounds the additional delay due to non-preemptive execution. □

In summary, Lemmas 6.32, 6.33, and 6.34 yield the following bound on s-aware pi-blocking under the FMLP$^+$.

Theorem 6.11. *Under the FMLP, a job J_i incurs at most $b_i = b_i^l + b_i^r$ s-aware pi-blocking, where b_i^l is defined as in Lemma 6.33, and b_i^r is defined as in Lemma 6.32 if requests are executed preemptively or as in Lemma 6.34 if requests are executed non-preemptively.*

This concludes our analysis of the suspension-based locking protocols proposed in this dissertation. An empirical evaluation is presented in Chapter 7.

6.5 Summary

We have discussed pi-blocking in suspension-based locking protocols and proposed maximum pi-blocking as a natural measure of a locking protocol's blocking behavior. We identified two classes of commonly-used schedulability analysis, namely s-oblivious and s-aware analysis, and showed lower bounds on maximum pi-blocking of $\Omega(m)$ and $\Omega(n)$, respectively. The OMLP family of protocols for mutual, RW, and k-exclusion is asymptotically optimal under s-oblivious analysis under any JLFP scheduler with arbitrary cluster sizes. In the special case of mutex constraints under global scheduling, this can be achieved with priority inheritance. To achieve optimality in the other cases (*i.e.*, RW and k-exclusion, or if $1 < c < m$), a new form of restricted priority boosting, named priority donation, is required. In the case of s-aware analysis, the FMLP$^+$, which is based on priority boosting, is asymptotically optimal under any partitioned JLFP scheduler. In the case of $c > 1$, no protocol is currently known to be asymptotically optimal for arbitrary JLFP schedulers. In particular, priority inheritance and priority boosting may cause $\Omega(\phi)$ s-aware pi-blocking if $c > 1$, where ϕ can be chosen to be arbitrarily large. Nonetheless, the global FMLP ensures $O(n)$ s-aware pi-blocking in the case of zero tardiness and G-EDF scheduling with constrained deadlines, G-EDF-like scheduling with constrained, fixed relative priority points, and G-FP scheduling with RM priorities. In addition to asymptotic bounds, we have also presented detailed blocking analysis of each of the protocols in the OMLP family and the partitioned FMLP$^+$.

CHAPTER 7

OVERHEAD-AWARE EVALUATION OF REAL-TIME LOCKING PROTOCOLS*

In the preceding two chapters, we have introduced several real-time locking protocols for spin-based and semaphore-based synchronization. For the OMLP family of protocols and for the partitioned FMLP$^+$, we have established asymptotic optimality with regard to s-oblivious and s-aware maximum pi-blocking, respectively. However, as noted in Section 6.3.4, claims of optimality do not necessarily translate to meaningful real-world performance since they do not reflect constant factors and differences in implementation overhead. For example, we discussed a similar case in Chapter 4: even though pfair scheduling is optimal (with respect to implicit-deadline tasks) on a multiprocessor, in practice, we found the non-optimal P-EDF to be preferable in terms of achieved schedulability due to P-EDF's lower implementation overheads. An overhead-aware evaluation of the proposed locking protocols is thus required to assess their practicality. In this chapter, we present such an evaluation (in terms of schedulability) of the proposed locking protocols on the same experimental platform previously described in Section 2.1 and Chapter 4. To reiterate, the experiments discussed in this chapter were carried out on a 24-core, UMA Intel Xeon L7455 system with three levels of shared caches (with a cache-line size of 64 bytes) and 64 GB of main memory.

In this chapter, we first explain how we accounted for locking overheads during schedulability analysis. Thereafter, in Section 7.2, we report on micro-benchmarks that we carried out to determine whether ticket-based or queue-based spinlocks are more efficient on our hardware platform. Finally, we compare and contrast mutex and RW spinlocks, semaphore protocols for s-oblivious analysis, and semaphore protocols for s-aware analysis in terms of HRT and SRT schedulability in Sections 7.3–7.6.

* Contents of this chapter previously appeared in preliminary form in the following paper:
Brandenburg, B. and Anderson, J. (2010b). Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87.

Event-driven scheduling is assumed throughout this chapter since we do not consider locking under PD^2, as discussed in Chapter 5.

7.1 Accounting for Locking Overheads

Locking protocols cause jobs to incur additional overheads. Conceptually, these overheads can be accounted for similarly to scheduling overheads, that is, by inflating task and resource sharing parameters to obtain a safe approximation of a task set prior to applying overhead-unaware blocking analysis. However, while the approach is the same, deriving safe approximations of locking-related overheads is in fact much more complicated than is the case when taking just scheduling overheads into account. For one, locking protocols expose jobs to several additional overhead sources besides regular scheduling overheads. These include direct overheads (such as system calls) and indirect overheads that arise when resource-holding jobs are delayed. Additional complexity arises because lock requests on one processor may race with unlock operations on another processor, which allows for corner cases that are difficult to analyze. Similar races are possible with other asynchronous events such as job releases. Generally speaking, the concurrency inherent in multiprocessor locking protocols makes identifying the scenarios that expose jobs to worst-case overheads a significant challenge.

To the best of our knowledge, safe approximations of locking-related overheads have not been studied in detail in prior work. In this section, we present an initial analysis of overheads as they arise in spinlock and semaphore protocols. However, due to the inherent complexity and the number of considered locking protocols, our analysis remains non-exhaustive. That is, although we derive safe approximations that consider all major overheads in the common case, it is not clear that the considered scenarios necessarily represent worst-case scenarios (see Section 7.1.6). The presented analysis should thus be understood as an initial step towards complete locking overhead accounting that will have to be augmented in future work (as discussed in further detail in Chapter 8). We begin with a discussion of interrupts, which greatly complicate locking overhead accounting.

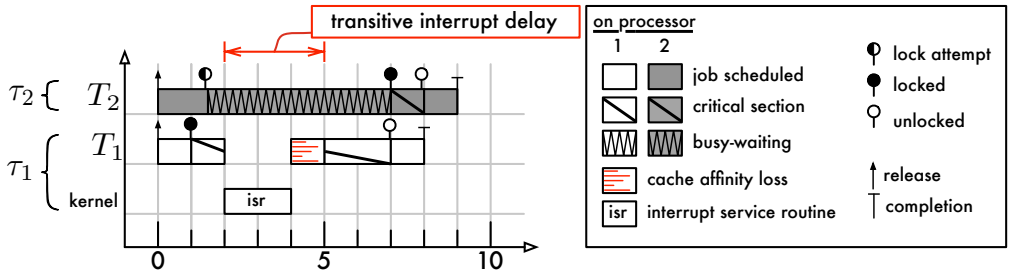


Figure 7.1: Schedule illustrating locking-related transitive interrupt delays. When a resource-holding job is stopped by an interrupt service routine (ISR), any jobs waiting for the resource to be released are transitively delayed as well. In the depicted example, job J_1 on processor 1 is stopped by an ISR during [2–4]. This increases J_1's effective request length, which causes job J_2 on processor 2 to incur additional s-blocking.

7.1.1 Transitive Interrupt Delays

An ill-timed interrupt can stop a resource-holding job if interrupt delivery is not disabled entirely while jobs access shared resources. This extends the duration during which the resource is unavailable, thereby transitively extending the acquisition delay (and hence pi-blocking or s-blocking) of any job waiting for the resource to be released.

Example 7.1. An example of this effect is shown in Figure 7.1, which depicts an MSRP schedule of two jobs sharing one resource under P-EDF. Scheduling and locking overheads have been omitted for the sake of clarity. Job J_1 acquires the shared resource ℓ_1 first at time 1. Shortly thereafter, job J_2 also requests ℓ_1 and busy-waits. At time 2, J_1 is stopped due to an interrupt, which transitively delays J_2. After the ISR completes at time 4, J_1 is further delayed by a partial cache affinity loss due to the execution of the ISR,[1] which also transitively affects J_2. As a result, J_2 incurs s-blocking for more than five time units even though J_1's request is only three time units long. ◊

In a process-based system such as LITMUS$^{\mathrm{RT}}$ where non-preemptive sections are not implemented by disabling interrupts, all interrupt sources can cause transitive delays in the presence of locks. However, even in embedded RTOSs that allow jobs to disable interrupts, transitive interrupt

[1] Realistically, the impact of cache-affinity loss inside a critical section should be small since critical sections are typically short in well-designed systems, which implies a small working set. Nonetheless, such delays will have to be taken into account when developing WCET analysis techniques (which is not within the scope of this dissertation). If the effect is known, then it can be accounted for as part of the costs of interrupts as discussed in Section 3.4.4.

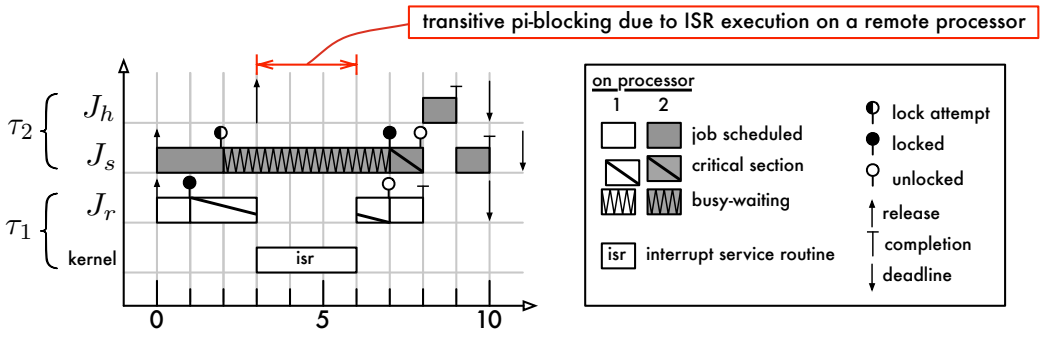


Figure 7.2: P-EDF schedule illustrating pi-blocking due to remote ISR execution.

delays may arise in the case of semaphore protocols, where jobs are typically preemptable (priority boosting has no effect on ISRs, which are not subject to scheduling).

Accounting for transitive delays due to remote interrupts that stopped a resource-holding job is challenging due to the indirect nature of the delay, and because interrupt accounting is already pessimistic even when jobs do not acquire locks. To illustrate the possibly long dependency chain, consider the example shown in Figure 7.2: a job J_h incurs pi-blocking upon release due to a non-preemptably spinning local job J_s, which in turn is delayed by a remote interrupt that stopped the resource-holding job J_r. There are two approaches that could be followed to resolve this situation.

One possible method is to inflate the maximum critical section length to account for each ISR that could stop the resource-holding job. Since request lengths are typically much shorter than task periods and the quantum size, this effectively amounts to inflating *each* request length $L_{i,q}$ by $\Omega(n)$ ISRs, which results in tremendous pessimism. For example, a typical critical section length is $10\mu s$; a typical release interrupt under P-EDF or P-FP is not much shorter. Inflating such a critical section to account for each possible release interrupt increases its length by several orders of magnitude.

The second approach exploits the fact that ISRs execute with statically higher priority and thus are always considered to delay real-time jobs. For example, suppose J_h, J_s, and J_r are scheduled under G-EDF. Then the release interrupt that stopped J_r might just as well have stopped J_s, or J_h had it been scheduled on another processor. That is, under G-EDF, each job already accounts for all possible interrupt sources anyway (with the exception of timer ticks, see below), such that it does not matter whether the delay was incurred directly or indirectly.

Under partitioned and clustered scheduling, however, the parameters are not inflated to account for interrupts in *other* clusters or partitions. If locking protocols are used, then this separation unfortunately has to be removed to account for transitive delays. That is, when creating ISR tasks under FP scheduling or when applying preemption-centric interrupt accounting under EDF-based schedulers, *all* interrupt sources must be modeled: each release interrupt source, regardless whether the corresponding task has been assigned to a remote cluster, and each timer tick on *every* processor (since a job may incur transitive delays from multiple timer ticks). This effectively models interrupts as executing on all processors at the same time. While this is also pessimistic, it is likely less so than inflating each critical section by several orders of magnitudes.

Unfortunately, no less-pessimistic accounting approach has been proposed in the published literature to date.[2] In some sense, delays due to remote interrupts are the price that is paid for introducing cross-processor (or cross-cluster) dependencies. The risk of such delays also suggests that it may be preferable to allow jobs to turn off interrupts completely if temporal constraints are stringent to avoid remote interrupt delays, which is somewhat counterintuitive. Such considerations and tradeoffs, however, remain the subject of future work at this point. In the following, we avoid the complex interaction between locking and interrupt handling altogether by assuming dedicated interrupt handling.

7.1.2 Spinlock Protocols

In the absence of interrupts, non-preemptive spinlock protocols are the easiest to analyze, due to two reasons. First, waiting jobs do not yield their assigned processors and hence do not lose cache affinity, and, second, they do not require system calls for the lock and unlock procedures (assuming non-preemptive sections are implemented as described in Section 3.3.2).

Example 7.2. Figure 7.3 depicts an example MSRP schedule showing two jobs on two processors sharing a resource. The choice of locking protocol, however, is irrelevant as the depicted overheads arise under any locking protocol. Locking-related overheads have been exaggerated for illustration purposes; other scheduling overheads have been omitted for clarity.

[2] To the best of our knowledge, prior work has not considered locking-induced transitive interrupt delays at all.

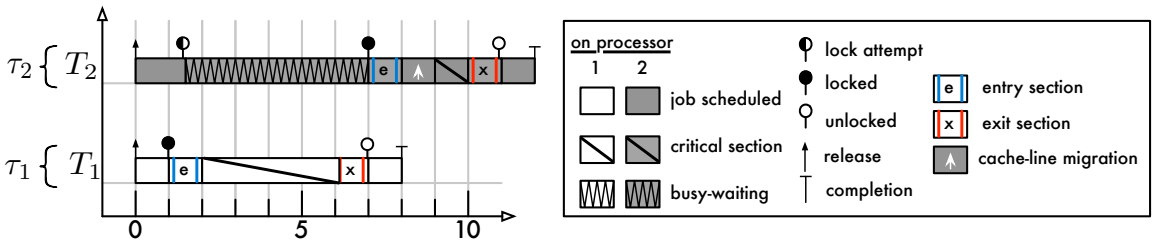


Figure 7.3: Schedule illustrating locking overheads in non-preemptive spinlock protocols. The effective maximum request length $L'_{i,q}$ depends on the length of the protocol entry section (during which the lock is acquired), the protocol exit section (during which the lock is released), and the cost of migrating the shared resource's cache lines (if any). Since cache-line migrations are generally unavoidable for shared data structures, we assume in this dissertation that the cost of cache-line migrations is already included in each maximum request length $L_{i,q}$.

Job J_1 requests the shared resource at time 1 on processor 1 and starts to execute the *entry section* of the protocol, which includes executing LITMUS$^{\mathrm{RT}}$'s non-preemptive section protocol (as described in Section 3.3.2). J_1 does not actually start to use the resource until time 2 when it has completed the entry section. Meanwhile job J_2 is busy-waiting on processor 2 and transitively delayed by J_1's entry section overhead.

At time 6, J_1 has completed its critical section. However, it must first execute the *exit section* to pass ownership of the lock to J_2, and to check whether a delayed preemption is required as part of LITMUS$^{\mathrm{RT}}$'s non-preemptive section protocol. J_2 holds the lock at time 7, but must finish executing the entry section before it starts executing its request. While busy-waiting is implemented as part of the entry section, we do not consider it part of the entry section overhead since it is due to resource unavailability, and not due to implementation overheads. The entry section overhead encompasses all costs associated with acquiring a lock that are not related to busy-waiting (such as initializing data structures, becoming non-preemptable, and enqueueing in wait queues).

A special case of cache-related delay is incurred by J_2 during $[8, 9)$, namely *cache-line migrations*. Since J_1 just finished using the shared resource on processor 1, the memory-resident shared state is virtually guaranteed to still reside in (at least) the lower-level caches of processor 1. When J_2 updates these cache lines, they must be invalidated on processor 1 and loaded into the cache of processor 2. This illustrates that if a global resource is contested, then critical sections typically execute in a cache-cold manner. ◊

Notation	Overhead
Δ^{in}	protocol entry section overhead
Δ^{out}	protocol exit section overhead
Δ^{sci}	system call entry path overhead
Δ^{sco}	system call return path overhead

Table 7.1: Summary of the overheads that arise in locking protocols and our notation. See Table 3.2 for a summary of scheduling-related overheads.

Entry and exit section. Accounting for the time spent executing entry and exit sections is straightforward. A job J_i acquires a resource at some point in time during the execution of the entry section, and the resource becomes available again to other jobs at some point during the execution of the exit section. This means that execution of the entry and exit section comprise part of J_i's critical section and can thus be accounted for by inflating the maximum request length.

Let Δ^{in} denote the maximum duration of the locking protocol's entry section (not counting any acquisition delay), and let Δ^{out} denote the maximum duration of the locking protocol's exit section. A summary of our locking-related notation is given in Table 7.1. A safe approximation with regard to entry and exit section overhead can be obtained by inflating each $L_{i,q}$ by $\Delta^{in} + \Delta^{out}$ time units.

The processor time spent entering and exiting critical sections must also be reflected in T_i's inflated execution time requirement. Recall that $N_{i,q}$ denotes the maximum number of times that any J_i accesses ℓ_q. The amount of time spent entering and exiting critical sections is thus bounded by $(\Delta^{in} + \Delta^{out}) \cdot \sum_{1 \leq q \leq n_r} N_{i,q}$.

Delayed preemption. While not shown in Figure 7.3, a job may have to invoke a system call, namely sched_yield(), as part of the non-preemptive section protocol when a delayed preemption is required. Invoking a system call requires a transition to and from kernel mode, which causes the preempted job to incur additional overheads. Notably, this cost is incurred after the lock has been released and hence does not affect the effective maximum request length. Nonetheless, the additional system call does potentially increase the worst-case execution requirement of the job and hence must be reflected by an increase in the inflated execution time requirement e'_i.

Let Δ^{sci} ("system call in") denote the maximum cost of entering the kernel as part of a system call, and let Δ^{sco} ("system call out") denote the maximum cost of the system call return path. The

amount of time spent entering and exiting the kernel due to delayed preemptions is then bounded by $(\Delta^{sci} + \Delta^{sco}) \cdot \sum_{1 \leq q \leq n_r} N_{i,q}$.

Taken together, entry and exit sections and delayed preemptions require the following increases in each task's maximum request lengths and execution requirement to account for spinlock overheads.

$$L'_{i,q} \geq L_{i,q} + \Delta^{in} + \Delta^{out} \quad \text{if } N_{i,q} > 0 \tag{7.1}$$

$$e'_i \geq e_i + (\Delta^{in} + \Delta^{out} + \Delta^{sci} + \Delta^{sco}) \cdot \sum_{1 \leq q \leq n_r} N_{i,q} \tag{7.2}$$

This transformation allows direct spinlock-related overheads—in the absence of interrupts—to be accounted for when applying the overhead-unaware blocking analysis presented in Chapter 5 (and when applying overhead-unaware schedulability tests). In the case of RW spinlocks, overheads can be accounted for using the same approach, although read and write requests typically have different entry and exit section costs.

Cache effects. An additional cache-related delay that is not reflected in Equations (7.1) and (7.2) occurs during $[8, 9)$ when J_2's critical section causes cache-lines to be migrated from processor 1 to processor 2. In general, this is unavoidable since a shared resource is very likely cached in remote processors whenever it is contended. Therefore, we assume that each bound $L_{i,q}$ already accounts for this. If such a bound is determined using WCET analysis, a cold cache must be assumed at the beginning of the critical section; if critical section lengths are approximated empirically, then the cache should be flushed prior to measurements (or, better yet, measured immediately after migrating the benchmark task).

Next, we discuss overheads as they arise in semaphore-based locking protocols.

7.1.3 Non-Preemptive Semaphore Protocols

Overheads are generally much higher under semaphore protocols than under spinlock protocols as kernel support is required to enact priority boosting, priority inheritance, or priority donation. Therefore, system call overhead is incurred each time that a job executes an entry or exit section. Further, a requesting job suspends if a resource is unavailable, which causes it to lose cache affinity, and to incur CPMD when it resumes, just as if it were preempted.

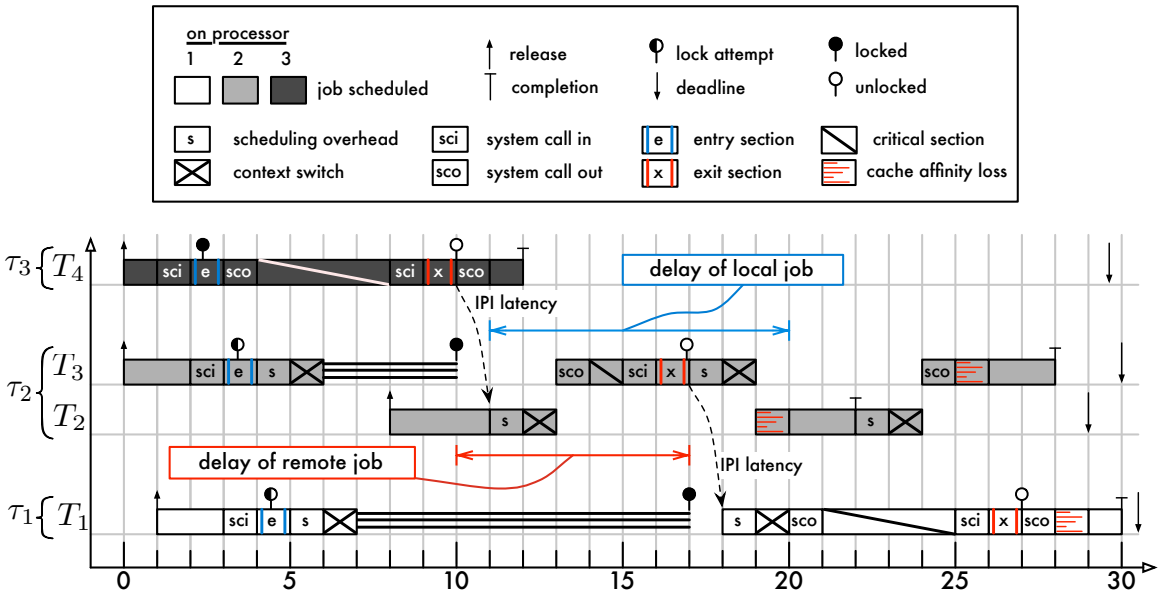


Figure 7.4: Schedule illustrating locking overheads in non-preemptive semaphore protocols. Since suspending requires kernel support, the entry and exit sections of semaphore protocols are implemented as system calls. Jobs issuing resource requests thus incur the cost of entering and exiting the kernel (denoted in the figure as "sci" and "sco", respectively). Additional overheads arise due to the invocation of the scheduler, required context switches, and the loss of cache affinity. Finally, as remote scheduler invocations require IPIs to be sent, jobs are also delayed by IPI latency.

The semaphore protocols considered in this dissertation can be organized into two groups: those that allow critical sections to be preempted (such as the MPCP, the DPCP, the global OMLP, and the preemptive variant of the FMLP$^+$), and those that do not (such as the clustered OMLP based on priority donation and the non-preemptive variant of the FMLP$^+$). We first consider the latter kind.

Example 7.3. Figure 7.4 depicts a non-preemptive FMLP$^+$ schedule depicting four tasks sharing one resource ℓ_1 on three processors (under P-FP scheduling). Only locking-related overheads are shown, all other overheads have been omitted to avoid clutter.

The execution of job J_4 on processor 3 demonstrates the overheads involved in an uncontested semaphore acquisition. J_4 requires ℓ_1 at time 1. To issue a request, it invokes a system call and hence incurs system call entry overhead during [1, 2). In kernel mode, it executes the entry section of the FMLP$^+$ during [2, 3), during which it acquires ℓ_1 and becomes priority-boosted. However, before being able to execute its request, J_4 must first execute the system call return path. After finishing its request at time 8, J_4 unlocks the resource by issuing a second system call to execute the exit section,

which causes the next waiting job to be resumed and to restore J_4's base priority. Processor 2 is notified of the need to schedule the next resource holder, job J_3, by means of an IPI.

J_3 on processor 2 demonstrates the overheads involved in a contested lock acquisition that is aided by priority boosting. J_3 requires ℓ_1 at time 2. It invokes a system call to lock the resource, but since it is already being held by J_4, J_3 suspends instead. When a job suspends, the scheduler (called at time 4) is required to select another process to execute (possibly the idle process) and a context switch must be carried out (at time 5). This highlights another cost of suspension-based locking protocols: jobs that must wait indirectly trigger the acquisition of scheduler locks. In global and clustered schedulers with heavy lock contention, this can result in high lock acquisition costs.

While J_3 is suspended, the higher-priority job J_2 is released on processor 2. It first commences execution normally, but at time 10 resource ℓ_1 becomes available, which is the resource that J_3 has been waiting for. J_3 hence benefits from priority boosting and preempts J_2 when the scheduler is triggered by the reception of an IPI at time 11. After the context switch at time 12, J_3 resumes, leaves kernel mode, and then executes its short critical section. As part of unlocking ℓ_1, J_3 loses the benefit of priority boosting, which lowers its effective priority below that of J_2. Consequently, the scheduler is invoked at time 17 and a context switch to J_2 is carried out at time 18. This allows J_2 to finish its normal execution, however, due to the preemption by J_3, it has (partially) lost cache affinity. Therefore, it is slowed down (to some extent) by CPMD during [19,20).

Finally, job J_1 (which requested ℓ_1 shortly after J_3 at time 4) is resumed at time 18. As is the case with J_3 during [10, 11), J_1 is not immediately resumed when it becomes the resource holder. Rather, its processor must first be notified with an IPI to trigger the scheduler. Therefore, J_1 is delayed by IPI latency during [17, 18). After resuming, J_1 executes its request similarly to J_3. However, since there is no higher-priority job pending on processor 1, J_1 continues execution at time 28 after unlocking ℓ_1. Since J_1 was suspended during [7, 17), it has likely lost cache affinity (for example, non-real-time background jobs may have polluted the cache in the meantime). Therefore, it suffers CPMD when it resumes execution, just as if it had been preempted. CPMD does not increase the duration of the request itself since we assume that each maximum request length $L_{i,q}$ already incorporates all cache effects. $\Diamond$

Using the safe approximation approach, the overheads that arise in this scenario can be integrated into overhead-unaware blocking and schedulability analysis by inflating each e'_i and $L'_{i,q}$. However, special attention must be paid to the location of blocked jobs.

Local vs. remote request length. The preceding example illustrates that the effective length of J_3's critical section differs depending on the point of view of local and remote jobs. From the point of view of the remote job J_1, J_3's request lasts from time 10, when J_4 releases ℓ_1, until time 17, when J_1 becomes the resource holder. The delays that affect J_1 include the IPI latency that affects when a previously-waiting resource holder is resumed (in Figure 7.4 at time 10), a scheduling decision, a context switch, a system call, and the cost of executing the locking protocol's exit section.

A local job is further affected by all local scheduler invocations, including those triggered when priority boosting (or priority donation) ceases. In the depicted example, the local job J_2 is delayed by the scheduler invocation *after* J_3's request is complete (in Figure 7.4 at time 17).

Due to this difference in delay, we differentiate in the following between the *local* maximum effective request length, denoted as $L'^{loc}_{i,q}$, and the *remote* maximum effective request length, denoted as $L'^{rem}_{i,q}$. Before bounding both effective request lengths, we must take into account the effect of priority boosting on cache affinity.

Critical sections and CPMD. In semaphore-based locking protocols, a job executing a request can cause other jobs to incur additional CPMD. One case in which this happens is when a job with higher *base* priority is preempted by a resource-holding job with higher *effective* priority (either due to priority boosting or due to priority inheritance). For example, in Figure 7.4, J_3 is priority-boosted and preempts J_2 at time 13. Consequently, when J_2 continues to execute at time 19, it has partially lost cache affinity and incurs CPMD. However, since the cache footprint of a critical section is likely small (compared to job's entire WSS), the extent of CPMD is likely much smaller than that after a "regular" preemption by a job with higher base priority (and a large cache footprint). It would thus be unfairly pessimistic to inflate each critical section to account for "full" CPMD (as measured after a complete loss of cache affinity); instead, the cache footprint of each critical section should be considered when bounding cache-related delays.

Conceptually, CPMD due to priority-boosted request execution could be determined using the methodology that we used for regular CPMD (see Section 4.4). However, the effect of reasonably-

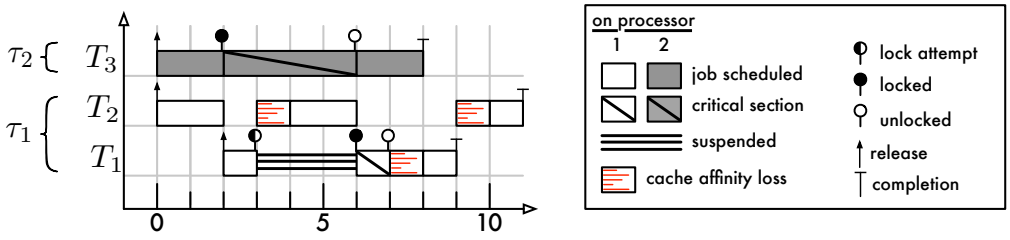


Figure 7.5: P-FP schedule illustrating that a lower-priority job can incur additional CPMD due to the self-suspension of a higher-priority job. In the example, J_1 preempts J_2 once when J_1 is released and once when J_1 resumes from waiting. Consequently, J_2 incurs CPMD twice—once at time 4, and again at time 9. Of these, only the CPMD at time 9 is accounted for by non-locking-related CPMD accounting as described in Section 3.4.3.

sized critical sections is likely so small as to be indistinguishable from noise. We therefore make the simplifying assumption that preemptions caused by priority-boosted request execution do not cause CPMD. This matches our earlier assumption that ISR execution does not cause CPMD, either. Note, however, that this may potentially introduce a slight bias in favor of semaphore protocols and against spinlock protocols (which do not cause any additional CPMD).

Another case in which resource sharing causes additional CPMD is when a lower-priority job suffers additional preemptions. This is illustrated in Figure 7.5.

Example 7.4. The P-FP schedule depicted in Figure 7.5 shows three jobs on two processors sharing one resource under the MPCP. Overheads other than CPMD have been omitted.

Job J_3 on processor 2 acquires resource ℓ_1 first. On processor 1, the lower-priority job J_2 is scheduled first and develops cache affinity. At time 2, it is preempted by the higher-priority job J_1, which proceeds to request ℓ_1. Since J_3 is already holding ℓ_1 at the time, J_1 suspends. This allows J_2 to continue execution; however, it has lost cache affinity due to J_1's execution and hence suffers CPMD during [3,4).

When J_1 resumes at time 6, it again preempts J_2. Since J_1 has a higher base priority than J_2, J_1 remains scheduled after releasing ℓ_1 at time 7. After J_1 completes at time 9, J_2 is scheduled again and suffers CPMD a second time during [9,10). Without J_1's self-suspension, J_2 would have only been preempted by J_1's arrival, and hence incurred CPMD only once. ◊

The example demonstrates that a job that resumes can affect lower-priority jobs as if it were released again. Similar to how "regular" CPMD is charged to the preempting higher-priority job (recall Section 3.4.3), this additional delay must be reflected in the parameters of tasks that suspend.

Safe approximation. To account for the overheads depicted in Figures 7.4 and 7.5, a task's effective execution requirement, each effective maximum request length, and its maximum self-suspension time must be inflated. We consider the execution requirement first.

For each request issued by a job J_i, the effective execution requirement e'_i must reflect the following sources of overhead.

- Two system calls: one to lock and one to unlock the requested resource.
- The entry and exit sections to acquire and release the lock (executed in kernel mode).
- Three scheduler invocations and context switches: one to suspend while waiting, one to resume, and one when priority boosting or priority donation ceases. For example, in Figure 7.4, J_3 is delayed by these costs at times 4, 11, and 17, respectively.
- Two instances of CPMD: one due to the loss of cache affinity while waiting to acquire the lock, and one to account for the loss of cache affinity of a preempted lower-priority job (if any). For example, in Figure 7.5, J_1 incurs the former kind of CPMD at time 7, and causes the latter kind of CPMD at time 3.

Formally, e'_i is inflated as follows.

$$e'_i \geq e_i + \left(\sum_{1 \leq q \leq n_r} N_{i,q} \right) \cdot (2\Delta^{sci} + 2\Delta^{sco} + \Delta^{in} + \Delta^{out} + 3\Delta^{sch} + 3\Delta^{cxs} + 2\Delta^{cpd}) \quad (7.3)$$

Note that if real-time tasks are not shielded from interrupts, then the inflation of the execution requirement is required *prior* to interrupt accounting since locking overheads increase J_i's exposure to interrupts. Further, recall from Section 3.4.5 that preemption-centric interrupt accounting is based on the assumption that each "subjob" executes consecutively (between preemptions). If jobs suspend, then this is no longer the case. Therefore, when using preemption-centric accounting, each possible suspension requires an additional inflation of e'_i by c^{pre} time units after all other overheads have been accounted for to reflect the potential creation of an additional "subjob." However, since we assume

in this chapter that real-time tasks are shielded from interrupts, this concern is not relevant to our experiments.

Next, we bound the effective maximum request length. We first consider the remote case $L_{i,q}^{'rem}$. From the point of view of a waiting remote job (such as J_1 in Figure 7.4), the remote effective request length must reflect the following overhead sources.

- IPI latency, which transitively delays waiting jobs while the resource holder is not yet scheduled. In Figure 7.4, this delay affects J_1 during [10,11) while processor 2 awaits the incoming IPI.
- One scheduler invocation and a context switch, which occur when the priority-boosted job resumes. For example, in Figure 7.4, J_3 delays J_1 with these overheads during [11,13).
- The system call return path, which is executed before the resuming job can execute its request. For example, J_3 leaves the kernel at time 13 in Figure 7.4.
- The system call entry path, with is executed prior to unlocking the resource. In Figure 7.4, J_3 invokes the unlock system call at time 15.
- The cost of executing the exit section, which prolongs the duration during which J_i is priority boosted (*e.g.*, J_3 remains priority-boosted until time 17).

Formally, $L_{i,q}^{'rem}$ is inflated as follows (if $N_{i,q} > 0$).

$$L_{i,q}^{'rem} \geq L_{i,q} + \Delta^{sch} + \Delta^{cxs} + \Delta^{sci} + \Delta^{sco} + \Delta^{out} + \Delta^{ipi} \tag{7.4}$$

As discussed above, from the point of view of a local higher-priority job that is preempted (such as J_2 in Figure 7.4), the effective maximum request length must also take the effect of the final scheduler invocation and context switch into account. Formally, $L_{i,q}^{'loc}$ is inflated as follows (if $N_{i,q} > 0$).

$$L_{i,q}^{'loc} \geq L_{i,q} + 2\Delta^{sch} + 2\Delta^{cxs} + \Delta^{sci} + \Delta^{sco} + \Delta^{out} + \Delta^{ipi} \tag{7.5}$$

Note that, under the clustered OMLP, there is at most one local blocking request since a job serves as priority donor at most once.

Finally, the effect of IPI latency on the maximum duration of self-suspension must be considered. Since a job is not immediately resumed when it becomes the resource holder, any IPI latency increases the duration of self-suspension. Therefore, the total maximum duration of self-suspension (after blocking terms have been determined) must be increased as follows.

$$susp'_i \geq susp_i + \left(\sum_{1 \leq q \leq n_r} N_{i,q} \right) \cdot \Delta^{ipi} \tag{7.6}$$

This is in addition to the increase in effective remote request length reflected by Equations (7.4) and (7.5) because they account only for IPI latency while *other* jobs hold ℓ_q, whereas Equation (7.6) accounts for IPI latency while J_i holds ℓ_q. Under s-oblivious analysis, $susp'_i$ is added to the execution cost (*i.e.*, $susp'_i$ is part of e'_i).

7.1.4 Preemptive Semaphore Protocols

The preceding analysis is appropriate of the non-preemptive variant of the FMLP$^+$ and the clustered OMLP based on priority donation (since priority donation prevents that resource-holding jobs are preempted). When resource-holding jobs can be preempted by other resource-holding jobs, as is the case under the preemptive FMLP$^+$, under the MPCP, and under the global OMLP, then remote jobs may transitively incur additional scheduling costs. This is illustrated in Figure 7.6.

Example 7.5. The depicted schedule shows four jobs on three processors sharing two resources ℓ_1 and ℓ_2. Jobs J_1 and J_2 both request resource ℓ_1; jobs J_3 and J_4 request resource ℓ_2. P-FP scheduling with the MPCP is assumed in this example. Recall that under the MPCP jobs are priority-boosted to the highest remote priority, as specified in Equation (2.10) on page 129. Consequently, when both J_2 and J_3 are priority boosted, J_2 has a higher effective priority since it shares a resource with J_1 on processor 1.

Consider the events on processor 2. Both J_2 and J_3 are released at time 0. Since J_2 has higher priority than J_3, J_2 is scheduled first. It requests ℓ_1 at time 2, which, however, has already been locked by J_1 at time 1 on processor 1. The resulting suspension of J_2 gives J_3 a chance to execute and to request and acquire ℓ_2. While J_3 is still executing its request, J_1 releases ℓ_1 and J_2 resumes

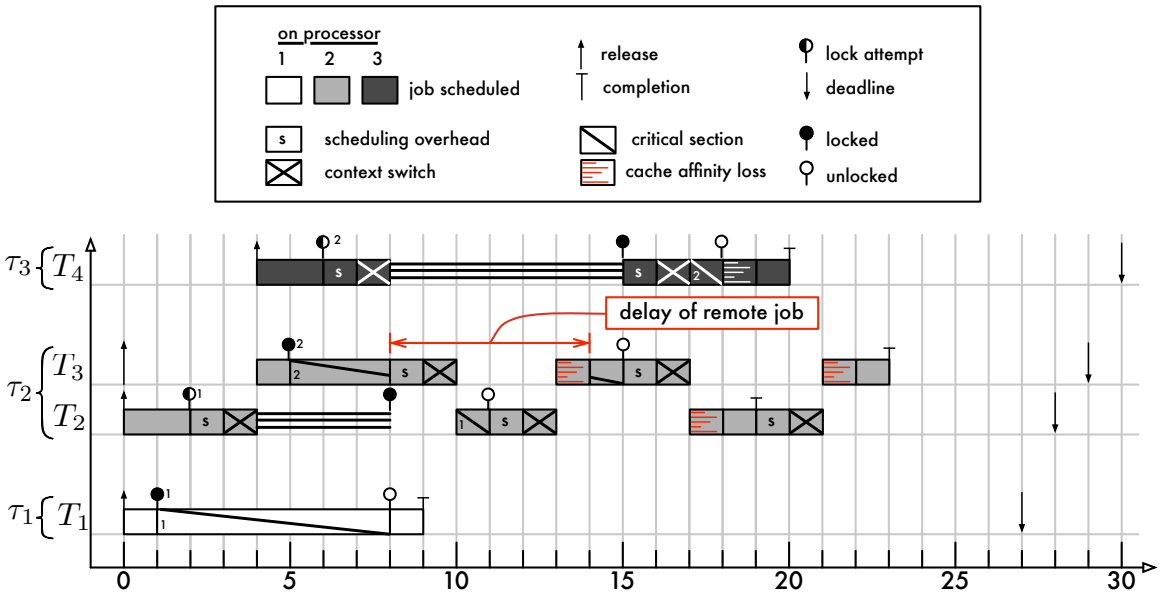


Figure 7.6: Schedule illustrating additional preemption overheads in preemptive semaphore protocols. System call, entry section, and exit section overheads have been omitted to avoid clutter. In the depicted MPCP example, job J_2 preempts the resource-holding job J_3 at time 8 since J_2 has a higher priority than J_2 and both jobs are priority-boosted at the time. This increases J_3's effective request length by the cost of two scheduler invocations and cache affinity loss. Because J_1 is waiting for J_3 to release ℓ_2, J_2's preemption of J_3 transitively increases the duration of pi-blocking incurred by J_1.

as a result. As both J_2 and J_3 hold resources at the time, J_2 preempts J_3 according to the MPCP's priority boosting rule. This preemption creates two additional delays.

First, when J_3 continues executing its critical section at time 13, it has potentially lost some cache affinity (however small it may be inside a critical section). Since we have currently no way of measuring such small disturbances, we consider this delay to be negligible for a lack of alternatives.

Second, the additional scheduler invocations and context switches transitively delay J_4, which is waiting on processor 3 for J_3 to release ℓ_2. In particular, note that J_4's additional delay includes both the scheduler invocation and context switch that occur when J_2 preempts J_3 as well as those that occur when J_3 resumes execution. ◊

Generally speaking, this example shows that, if resource requests are executed preemptively, then remote jobs can be transitively delayed by the scheduling and context switch cost that arise *after* a resource has been released. The effective maximum request length—from the point of view of remote jobs—must hence be inflated to account for two scheduling decisions, and not just one as in

Equation (7.4), if requests are executed preemptively. This leads to the following inflation.

$$L'_{i,q} \geq L_{i,q} + 2\Delta^{sch} + 2\Delta^{cxs} + \Delta^{sci} + \Delta^{sco} + \Delta^{out} + \Delta^{ipi} \tag{7.7}$$

Equation (7.7) charges the scheduling and context-switch costs to the preempting critical section, analogous to how regular preemptions are charged. This works because the pi-blocking bounds of jobs waiting for the lower-priority critical section to end include the request length of any preempting requests. Note that Equation (7.7) is identical to Equation (7.5), that is, when requests are executed preemptively, the effective critical section length is the same for both local and remote jobs.

Besides the MPCP and the preemptive FMLP$^+$, this analysis also applies to the global OMLP, which is based on priority inheritance. Like preemptive priority boosting, priority inheritance allows jobs that execute a critical section to be preempted by other resource-holding jobs. Further, priority inheritance also allows resource-holding jobs to be preempted by non-resource-holding jobs. However, this only occurs if the resource-holding job is not incurring s-oblivious pi-blocking (otherwise it would not be preempted). Therefore, the scheduling costs associated with such preemptions are not actually locking-related and have already been considered by regular release and preemption delay accounting (see Section 3.4.3), and hence do not have to be considered here.

7.1.5 Remote Procedure Calls

Recall from Section 2.4.4.2 that the DPCP is based on the RPC model: resources are assigned to processors, and jobs activate remote agents to access non-local resources. In a distributed memory system, agents are necessarily implemented as separate processes that await incoming requests. However, in a shared-memory system (such as LITMUS$^{\mathrm{RT}}$), remote requests can be realized by migrating the requesting job to the target processor, and back to the job's assigned processor when it has completed its request. In this model, agents are merely a modeling abstraction that simplifies reasoning about the protocol. The version of LITMUS$^{\mathrm{RT}}$ underlying this dissertation follows the latter approach since it avoids the need for additional processes, and because it lets the DPCP conform to LITMUS$^{\mathrm{RT}}$'s standard locking API (*i.e.*, if jobs are migrated directly, no DPCP-specific system calls are required to implement the DPCP scheduling rules). Regardless of how remote

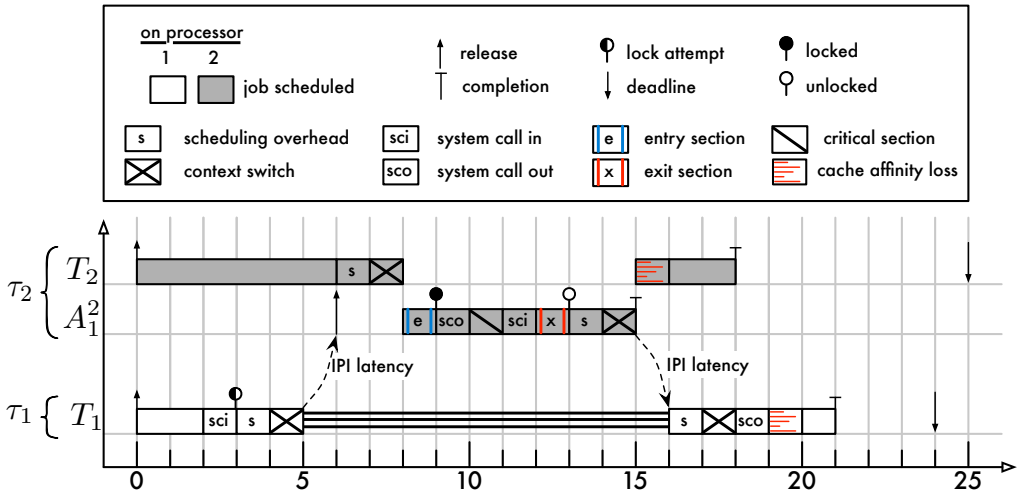


Figure 7.7: Schedule illustrating locking-relate overheads that arise under the DPCP. As the DPCP is based on the RPC model, the requesting job must first issue a request to its remote agent, which requires kernel support and hence incurs system call overhead. The remote processor is notified of the pending request with an IPI. Since IPIs are not delivered instantaneously, the requesting job is further twice delayed by IPI latency (once due to the request, and once due to the response).

agents are implemented, additional overheads arise due to the latencies inherent in cross-processor communication or job migrations.

Example 7.6. Figure 7.7 depicts a simple example DPCP schedule of two jobs (and one agent) sharing one resource on two processors. The single resource, ℓ_1, is assigned to processor 2.

Job J_1 on processor 1 requires ℓ_1 at time 2. As the DPCP requires kernel support, J_1 invokes a system call to request ℓ_1. The kernel then temporarily assigns J_1 to processor 2's ready queue and initiates J_1's migration by switching away from its stack, which requires invoking the scheduler to select another process to execute.

At time 5, it is safe for J_1 to migrate to processor 2 since its stack is no longer in use. Therefore, processor 2 is notified with an IPI to preempt the currently scheduled job J_2. Since the agent A_1^2 (implemented by J_1's process in LITMUS$^{\mathrm{RT}}$) is priority boosted, it is scheduled immediately and executes the entry section at time 8 to acquire ℓ_1 (recall that the DPCP uses the PCP locally). The agent then leaves the kernel, executes the request, and finally releases ℓ_1 with another system call. In LITMUS$^{\mathrm{RT}}$, sending a "response" involves re-assigning J_1 to its original processor and initiating a context switch to enable J_1 to be migrated. This allows J_2 to continue its execution at time 15. Since J_2 was preempted during [8,15), it is further (slightly) slowed down by CPMD during [15,16)—as

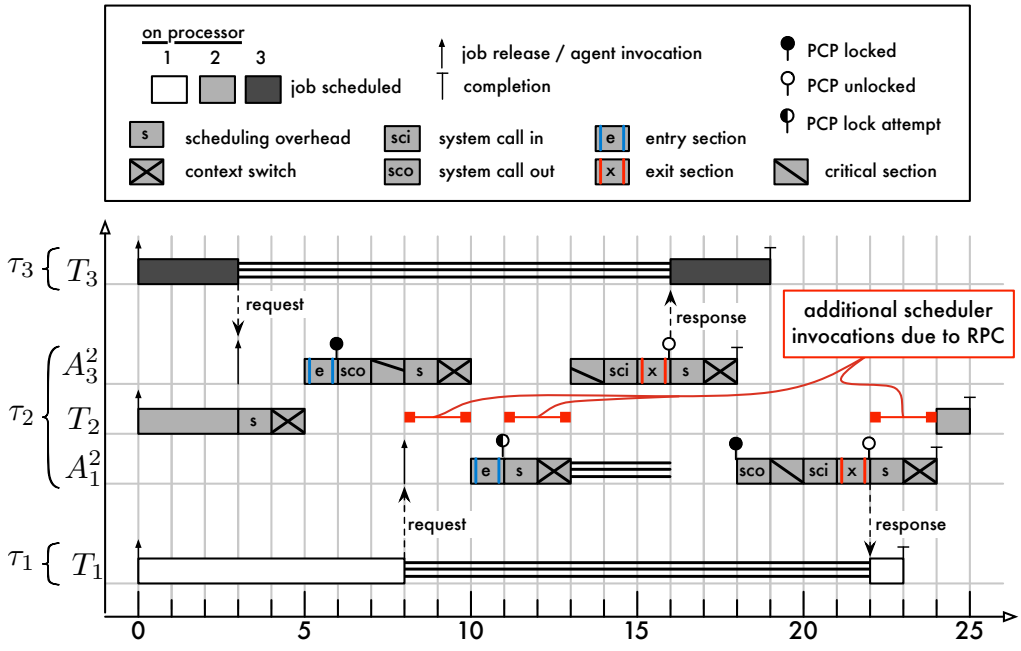


Figure 7.8: DPCP schedule demonstrating that a remote agent executing the PCP causes three additional scheduler invocations if the requested resource is currently unavailable. For the sake of clarity, overheads are shown only on processor 2 and have been omitted on processors 1 and 3.

discussed in Section 7.1.3 above, we consider CPMD caused by the small cache footprint of critical sections to be negligible.

When processor 1 receives the IPI notifying it of the need to reschedule, it selects J_1 to be scheduled. J_1 then leaves kernel mode and continues execution. However, since J_1 was suspended during [6,16), it likely lost cache affinity (*e.g.*, due to the execution of background processes) and incurs CPMD during [19,20) as it refetches its working set into processor 1's cache(s). ◊

In Figure 7.7, the remote resource requested by J_1 is not contested, and in fact does not depict a worst-case scenario. Higher overheads are possible if the resource is *not* available. A scenario that causes higher overheads is illustrated in Figure 7.8.

Example 7.7. The depicted DPCP schedule shows three jobs on three processors sharing one resource ℓ_1, which is local to processor 2. Job J_2 on processor 2 is delayed by requests issued by the remote jobs J_1 (on processor 1) and J_3 (on processor 3). Overheads on processors 1 and 3 are not shown since they are irrelevant to this example.

At time 3, J_3 is the first to activate its agent on processor 2. Since agents are priority boosted, A_3^2 preempts J_2, which causes a scheduler invocation and context switch during [3,5). A_3^2 then proceeds to lock ℓ_1 according to the rules of the PCP and begins to execute the request at time 7.

At time 8, agent A_1^2 is activated by a request from J_1 and preempts A_3^2. The resource-holding agent is preempted since requests are executed preemptively under the PCP (and hence the DPCP), and because A_1^2 acts on behalf of a job with base priority $\mathsf{Y}(J_1, t) = 1$, whereas A_3^2 acts on behalf of a job with base priority $\mathsf{Y}(J_3, t) = 3$. The preemption requires a scheduler invocation and context switch during [8,10).

Once scheduled, A_1^2 immediately attempts to lock ℓ_1. As ℓ_1 is unavailable, A_1^2 must suspend, thereby triggering another scheduler invocation and context switch at time 11.

At time 16, A_1^2 is resumed when A_3^2 releases ℓ_1, and can finally be scheduled to execute J_1's request during [19,20). At time 22, A_1^2 releases ℓ_1 and completes, which allows J_2 to continue its execution.

In total, J_2 is delayed by five scheduler invocations and context switches at times 3, 8, 11, 16, and 22. Of these, the initial invocation at time 3 and the invocation at time 16 can be attributed to J_3's request (*i.e.*, they would have occurred even if J_1 would not have issued a request). This leaves three scheduler invocations and context switches that are due to J_1's request. $\lozenge$

As with the semaphore-based protocols for shared-memory systems, the DPCP requires an inflation of each task's execution requirement, maximum self-suspension time, and each maximum request length to account for the overheads exhibited in Figures 7.7 and 7.8.

Execution requirement. Since the execution of agents is accounted for separately in the blocking analysis of the DPCP, only the overheads arising on a job's assigned processor must be reflected in its execution requirement. As can be seen in Figure 7.7, a job requesting a remote resource causes one additional system call, two scheduler invocations and context switches, and incurs CPMD on its assigned processor. Further, any additional CPMD that preempted lower-priority jobs incur due to suspensions (as illustrated in Figure 7.5) must be considered as well. This leads to the following

inflation of the effective execution requirement.

$$e'_i \geq e_i + \left(\sum_{1 \leq q \leq n_r} N_{i,q} \right) \cdot (\Delta^{sci} + \Delta^{sco} + 2\Delta^{sch} + 2\Delta^{cxs} + 2\Delta^{cpd}) \tag{7.8}$$

Request length. The effective maximum request length must reflect all of an agent's execution time since normal response-time analysis does not take priority-boosted agents into account (*i.e.*, all delays due to agents must be reflected by the DPCP's blocking terms). In the scenario shown in Figure 7.7, the agent's activation results in two scheduler invocations and context switches, one system call, and the execution of the PCP's entry and exit section. If the requested resource is unavailable at the time of the request, an agent can cause up to three scheduler invocation and context switches, as shown in Figure 7.8. The following inflation results.

$$L'_{i,q} \geq L_{i,q} + 3\Delta^{sch} + 3\Delta^{cxs} + \Delta^{sci} + \Delta^{sco} + \Delta^{in} + \Delta^{out} \tag{7.9}$$

Suspension time. Overheads already reflected by $L'_{i,q}$ do not have to be charged explicitly to each task's maximum suspension time $susp'_i$ since the regular DPCP blocking analysis (based on each $L'_{i,q}$) implicitly accounts for the increase in self-suspension time. However, $L'_{i,q}$ does not account for IPI latency related to agent activation.

Figure 7.7 shows two instances of IPI latency affecting J_1's response time. The first instance of IPI latency arises when the process implementing J_1 migrates to processor 2 to take on the role of A_1^2: the preemption of J_2 is delayed by IPI latency. Similarly, when the process implementing J_1 migrates back to processor 1 after relinquishing ℓ_1, IPI latency delays the resumption of J_1 on processor 1. This shows that each resource request is affected by IPI latency twice, which requires the following inflation of the maximum self-suspension time.

$$susp'_i \geq susp_i + \left(\sum_{1 \leq q \leq n_r} N_{i,q} \right) \cdot 2\Delta^{ipi} \tag{7.10}$$

The scenario depicted in Figure 7.7 shows a remote resource request. It is further possible for jobs to access local resources. For example, J_2 could also request ℓ_1. In this case, the access and the incurred overheads are similar, but no IPI latency is incurred.

7.1.6 Limitations

As mentioned in the beginning of Section 7.1, the presented analysis of locking-related overheads is not comprehensive. It is non-exhaustive in the sense that it accounts for all major overheads in common scenarios, but that the analyzed scenarios do not provably correspond to the worst case. There are two reasons for this.

First, the presented analysis assumes that interrupts are not an issue due to dedicated interrupt handling. This, however, is not entirely correct because there exists a slight chance of inadvertent IPIs delaying critical sections. To illustrate this problem, consider a protocol using priority boosting such as the MPCP or the FMLP$^+$. Under dedicated interrupt handling, IPIs are used to notify processors of required preemptions. While a job J_i is executing a request, it should normally not be preempted by non-resource-holding jobs. Consequently, the processor on which J_i is scheduled should not receive rescheduling IPIs. Unfortunately, because IPIs are not delivered instantaneously, a race condition exists: if a higher-priority job is released just before J_i enters its critical section, it could be the case that the processor dedicated to servicing ISRs observes J_i's base priority before J_i's effective priority is boosted. As a result, the dedicated processor will send an IPI to J_i's processor. If the IPI is sufficiently delayed, it could arrive (and be serviced) while J_i is already executing its critical section, thereby increasing the effective request length despite priority boosting.

In fact, the problem of a racing IPI affects all protocols equally, including spinlocks if non-preemptable sections are implemented by means of the control page mechanism (Section 3.3.2). The locking overhead analysis presented in this section does not reflect this (very unlikely) case. For our purpose, this is an acceptable limitation—all evaluated locking protocols are subject to the possibility of IPI races, and hence no bias in favor of any particular protocol is introduced. However, in a truly *safe* approximation, such race conditions will have to be considered. Alternatively, they would have to be avoided entirely. For example, if non-preemptive spinlocks are implemented by disabling interrupt delivery, then IPI races do not affect effective critical section lengths.

The second reason for why the presented analysis must be considered non-exhaustive is the difficulty of identifying worst-case scenarios. For example, suppose the presented analysis is extended to take IPI races into account. Even then it would not be clear if it addressed all possible interactions among processors. For example, if a job becomes the resource holder just after it began

to suspend, it must first finish suspending before it can be resumed (*i.e.*, job states cannot be changed instantaneously since state transitions take time), which could potentially cause an additional increase in effective critical section length. For this to be an actual problem, scheduling decisions and context switches would have to exceed maximum suspension times, which is unlikely. Nonetheless, the fundamental problem remains that locking protocols give rise to such a large number of potential executions that the manual identification and inspection of the worst case becomes a daunting task. We believe that the use of modeling and automated analysis tools will be essential for a future comprehensive analysis of locking protocol overheads.

This concludes our discussion of locking overheads. Major differences exist between spin-based and suspension-based locking protocols. In addition to typically having higher acquisition costs (due to the need to invoke system calls), suspension-based protocols also cause additional scheduler invocations, context switches, and cache affinity loss. We revisit these issues shortly in our comparison of locking protocols in Section 7.6. Next, we begin our overhead-aware evaluation of locking protocols with a comparison of spinlock choices.

7.2 Spinlock Implementation Efficiency

In Section 5.3, we introduced three implementations of phase-fair spinlocks: a simple ticket lock, denoted PF-T, a compact lock, denoted PF-C, and a list-based queue lock, denoted PF-Q. Since our experimental platform is not memory-constrained, there is little reason to prefer the PF-C lock over the PF-T lock as the PF-C lock requires more atomic instructions than the PF-T lock. It is not obvious, however, whether the PF-T or the PF-Q lock is preferable on our hardware platform.

Recall that the primary advantage of the PF-Q algorithm is that is has constant RMR complexity, that is, PF-Q locks implement local spinning to avoid repeated cache invalidations. In contrast, in a PF-T lock, each spinning processor suffers a cache invalidation each time the lock's state is updated (a PF-T lock fits into one cache line on our platform). Given that cache-coherency traffic can be a major source of overhead in shared-memory systems, low RMR complexity is certainly desirable. However, the PF-T entry and exit procedures contain fewer atomic instructions than the PF-Q entry and exit procedures. Since atomic instructions (such as fetch-and-add) are slower than regular, non-atomic instructions (such as regular add), either lock could be preferable in practice.

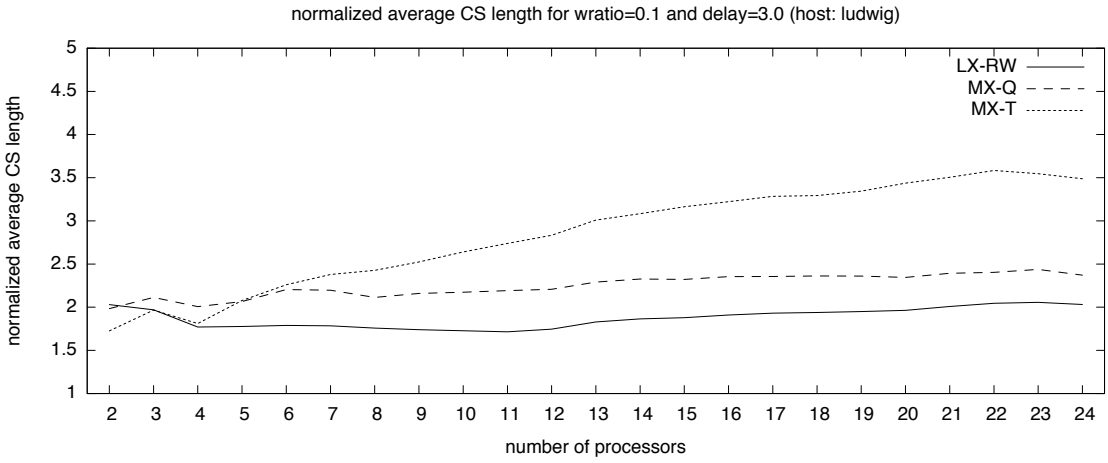
In a prior study on a single-chip, 8-core, 32-processor UltraSPARC T1 "Niagara" platform (Brandenburg and Anderson, 2010b), we found PF-T locks to be preferable to PF-Q locks since the high cost of atomic compare-and-exchange instructions on that platform outweighed the RMR complexity advantage of PF-Q locks. These prior results notwithstanding, an evaluation of the implementation efficiency of spinlock implementations is in order since the hardware platform used for the experiments in this dissertation is very different from the Niagara (in particular, our Xeon platform consists of multiple chips, which can be expected to increase the costs of cache coherency).

The preceding discussion applies equally to Mellor-Crummey and Scott's ticket-based and queue-based task-fair mutex and RW spinlocks (Mellor-Crummey and Scott, 1991a,b), after which our PF-T and PF-Q algorithms are modeled. In general, the choice of whether to use ticket or queue locks is a tradeoff between the cost of uncontested lock acquisition and the cost of spinning. Ticket locks are simpler, and hence faster if contention is low; queue locks have constant RMR complexity and thus reduce the cost of spinning. In this section, we present results from micro-benchmarks designed to measure each lock type's efficiency under a wide range of contention and write ratios. Interestingly, and in contrast to our earlier study on the Niagara platform, the observed results show queue locks to be preferable on our Xeon platform if locks are contended.

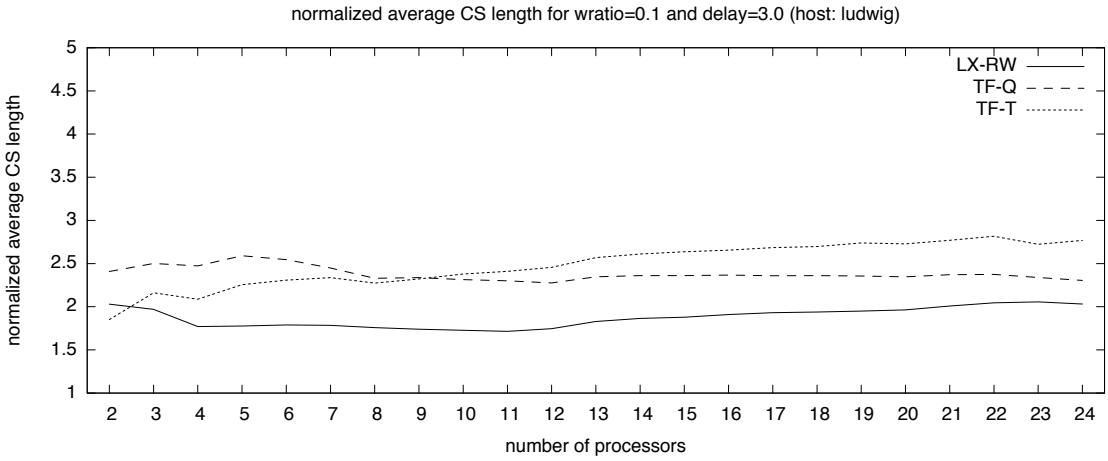
Setup. We implemented an experiment in which a configurable number of processors access a few shared variables—protected by one lock—repeatedly in a tight loop. The loop was configured via two parameters: $wratio$, which determined the fraction of updates, and $delay$, which determined the time between consecutive requests of one processor such that each processor spent approximately $\frac{1}{1+delay}$ of the total execution time in critical sections. We varied the number of processors from $2, \ldots, 24$ for each combination of $wratio \in \{0, 0.01, 0.05, 0.1, 0.2, 0.35, 0.5, 1\}$ and $delay \in \{1, \ldots, 7, 10\}$ and recorded the time required for each processor to (concurrently) execute 200,000 loop iterations (after an initial warm-up phase) for each of the locks listed in Table 7.2.

Results. Figures 7.9 and 7.10 show the measured results for $wratio = 0.1$ and $delay = 3.0$, that is, each processor spent about 25% of its execution time trying to access shared variables.

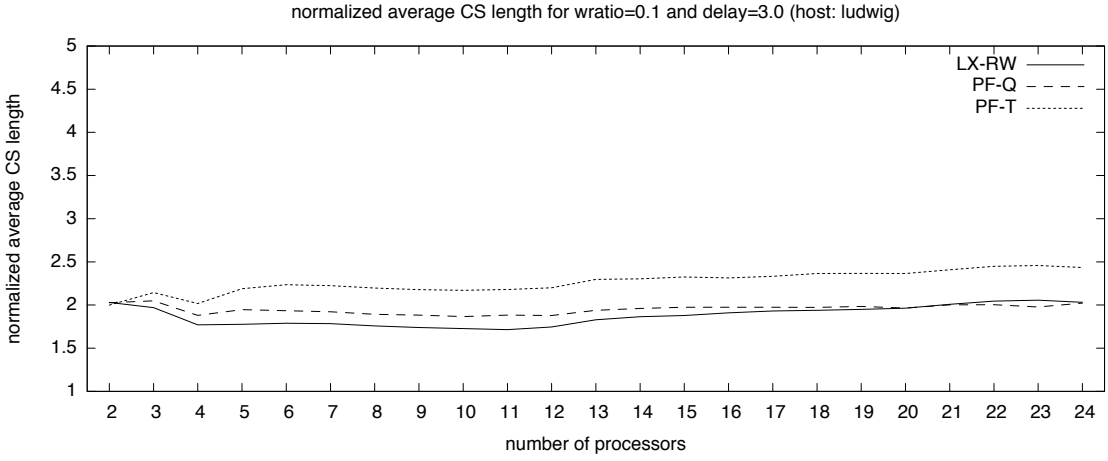
Figure 7.9 shows comparisons of a ticket-based to a queue-based implementation of task-fair mutex, task-fair RW, and phase-fair RW group locks respectively. Figure 7.10 shows the same data, however the curves are grouped differently to allow a comparison of all considered queue locks in



(a) Task-fair mutex spinlocks.

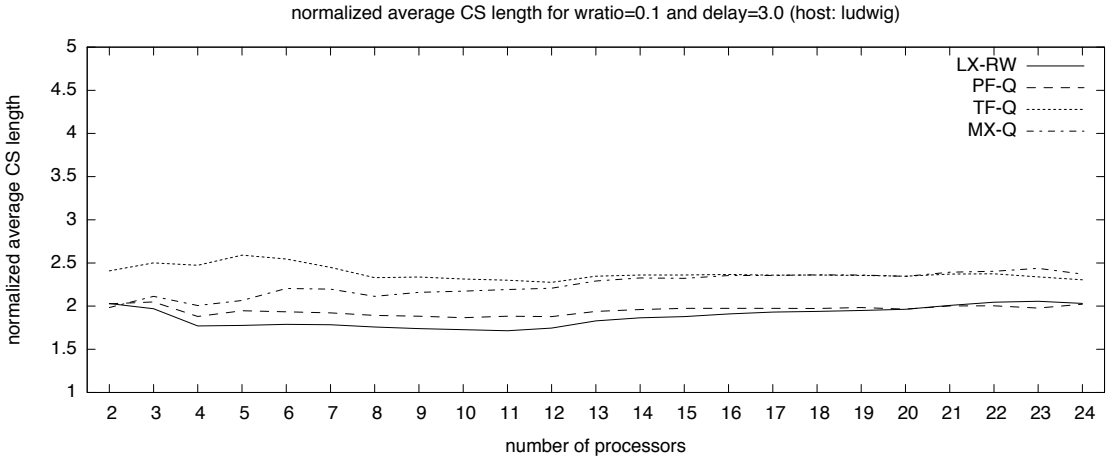


(b) Task-fair RW spinlocks.

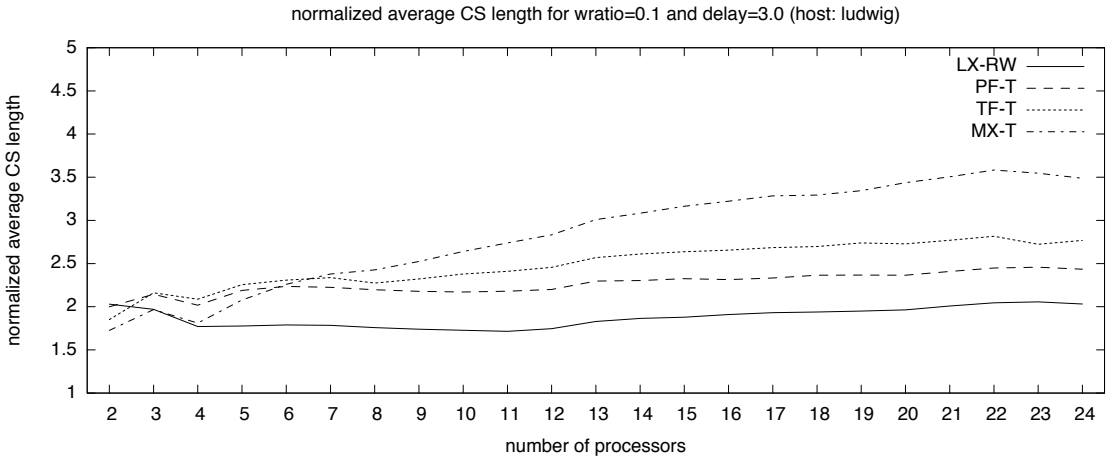


(c) Phase-fair RW spinlocks.

Figure 7.9: Average critical section length as a function of the number of contending processors, grouped by lock type with Linux as a baseline. See Table 7.2 for a list of the considered lock types.



(a) Queue locks.



(b) Ticket locks.

Figure 7.10: Average critical section length as a function of the number of contending processors, grouped by implementation technique with Linux as a baseline. See Table 7.2 for a list of the considered lock types.

Algorithm	Definition	Name	Complexity
Task-Fair Mutex Ticket Lock	(Mellor-Crummey and Scott, 1991a)	MX-T	$O(m)$
Task-Fair Mutex Queue Lock	(Mellor-Crummey and Scott, 1991a)	MX-Q	$O(1)$
Task-Fair RW Ticket Lock	(Mellor-Crummey and Scott, 1991b)	TF-T	$O(m)$
Task-Fair RW Queue Lock	(Mellor-Crummey and Scott, 1991b)	TF-Q	$O(1)$
Phase-Fair RW Ticket Lock	Listing 5.1	PF-T	$O(m)$
Phase-Fair RW Queue Lock	Listings 5.3 and 5.4	PF-Q	$O(1)$
Linux RW Lock	Linux 2.6.24	LX-RW	—

Table 7.2: Implemented and evaluated spinlock algorithms. The complexity metric is remote memory references on m cache-coherent processors. Linux's RW lock implementation resembles a writer preference lock but does not offer strong progress guarantees; it serves only as a baseline for implementation performance.

inset (a), and all considered ticket locks in inset (b). Further, each inset also shows Linux's RW lock implementation as a baseline. Each graph shows the average critical section length (including synchronization overhead) normalized by the average critical section length if no locks are acquired (*e.g.*, a value of 1.5 means that locking overheads increased the critical section length by 50%).

The major trend apparent in our results is that, on our Xeon L7455 platform, task-fair RW queue locks are preferable to task-fair RW ticket locks if more than 10 processors access a lock. In the case of phase-fair RW locks, the queue-based implementation is virtually always preferable, and with task-fair mutex locks, the break-even point is around 5 processors, as can be seen in Figure 7.9(a). This is in contrast to our prior study on a Niagara, in which we found ticket locks to be preferable even under unrealistically high levels of contention. A likely explanation for this difference is that our Xeon platform consists of four chips, which implies that a large amount of off-chip cache coherence protocol traffic is generated when processors (of different chips) spin on the same memory location.

Interestingly, among the ticket and queue locks, the phase-fair RW lock implementations are the most efficient algorithm (with respect to each category) and closest to the performance to Linux's native RW lock. (Linux's RW lock itself is not an option since it does not offer strong progress guarantees.) In fact, the phase-fair RW queue lock is almost as efficient as Linux's native RW lock if at most 20 processors are contending for the lock, and even slightly more efficient otherwise.

While these are micro-benchmarks of limited scope, they do show that phase-fair RW spinlocks can be implemented efficiently. Further, these experiments show queue locks to be more efficient

```
72  generate_resource_model(T_i, n_r, pacc, wratio, csdist):
73      for q ← 1, 2, 3, ..., n_r
74          if prng_draw_next( [0, 1] ) ≤ pacc:
75              if prng_draw_next( [0, 1] ) ≤ wratio:
76                  // T_i is a writer with respect to ℓ_q
77                  set N^R_{i,q} ← 0
78                  set L^R_{i,q} ← 0
79                  set N^W_{i,q} ← prng_draw_next( {1, 2, 3, 4, 5} )
80                  set L^W_{i,q} ← prng_draw_next( csdist )
81              else:
82                  // T_i is a reader with respect to ℓ_q
83                  set N^R_{i,q} ← prng_draw_next( {1, 2, 3, 4, 5} )
84                  set L^R_{i,q} ← prng_draw_next( csdist )
85                  set N^W_{i,q} ← 0
86                  set L^W_{i,q} ← 0
87          else:
88              // T_i does not access ℓ_q
89              set N^R_{i,q} ← 0
90              set L^R_{i,q} ← 0
91              set N^W_{i,q} ← 0
92              set L^W_{i,q} ← 0
```

Listing 7.1: Resource model generation.

than ticket locks on our hardware platform. Therefore, we only consider queue locks in the remainder of this chapter. Next, we compare each spinlock type in terms of HRT and SRT schedulability.

7.3 Mutex and Reader-Writer Spinlocks

In the first of two schedulability studies, we compared mutex, task-fair, and phase-fair RW spinlocks in terms of schedulability. The experimental setup is in large parts the same as the one described in detail in Section 4.1. Task sets were generated as before using the nine utilization and three period distributions discussed in Section 4.2.3. Additionally, resource requirements were generated using the procedure documented in Listing 7.1.

Each task's parameters $L_{i,q}^R$, $L_{i,q}^W$, $N_{i,q}^R$, and $N_{i,q}^W$ were randomly chosen based on the following parameters: the number of resources, denoted n_r, the *access probability*, denoted $pacc$, the *write ratio*, denoted $wratio$, and the *critical section length distribution*, denoted $csdist$. A given task T_i accessed resource ℓ_q with probability $pacc$. If T_i accessed ℓ_q, then it was determined to be a writer

with probability $wratio$, and a reader otherwise. The number of accesses $N_{i,q}$ was uniformly chosen from $\{1, 2, 3, 4, 5\}$. The maximum request length $L_{i,q}$ was randomly chosen from $csdist$.

In our schedulability experiments comparing spinlocks, we considered

- $pacc \in \{0.1, 0.25, 0.4\}$,
- $n_r \in \{6, 12, 24\}$, and
- $wratio \in \{0.1, 0.2, 0.3, 0.5, 0.75\}$,

and considered three uniform critical section length distributions. When using *short* critical sections, each $L_{i,q}$ was chosen randomly from $[1\mu s, 15\mu s]$; with *intermediate* critical sections, each $L_{i,q}$ was randomly chosen from $[1\mu s, 100\mu s]$; and finally, when assuming *long* critical sections, each critical section length was randomly chosen from $[5\mu s, 1280\mu s]$.

It is a widely acknowledged design principle that critical sections should be short: from a throughput point of view, long critical sections impede scalability, and from a real-time point of view, long critical sections result in pessimistic upper bounds on blocking and thus limit schedulability. We therefore believe most critical sections to be short in practice (in well-designed systems). In fact, in a case study that examined critical section lengths in the Linux kernel (hosted on a four-processor 2.7 GHz Pentium 4 system), we found more than 90% of all spinlock-protected critical sections to be shorter than $5\mu s$, and more than 90% of all semaphore-protected critical sections to be shorter than $13\mu s$ (Brandenburg and Anderson, 2007a). Similarly, when measuring critical section lengths in multimedia SRT applications on the same platform, we found 99% of all critical sections to be shorter than $10\mu s$ (Brandenburg and Anderson, 2007a). Nonetheless, in the schedulability study reported on in this chapter, we also included intermediate critical section lengths to allow for pessimism when determining $L_{i,q}$ parameters in practice, and further considered long critical sections following Lakshmanan *et al.*, who assumed the stated distribution of critical section lengths in their evaluation of the MPCP and the MPCP-VS (Lakshmanan *et al.*, 2009).

For each combination of $pacc$, n_r, $wratio$, and $csdist$, we evaluated HRT and SRT schedulability as a function of $ucap$ as discussed in Section 4.1.5 using a step size of 0.5 (*i.e.*, $ucap \in \{1, 1.5, 2, \ldots, 23.5, 24\}$). We conducted the experiments twice, once assuming a WSS of 4 KB, and once assuming a WSS of 128 KB. We did not consider further WSSs (and did not compute weighted

schedulability scores) because spinlock overheads are not sensitive to WSS, and because the additional locking-related parameters caused the parameter space to become too large to be exhaustively sampled (*i.e.*, the runtime requirements of the schedulability experiments became infeasible). When assessing HRT schedulability, we assumed worst-case overheads and load CPMD; in the SRT case, we assumed average-case overheads and idle CPMD. Locking overheads were accounted for as discussed in Section 7.1. As discussed in Section 7.1, we focus on dedicated interrupt handling in this chapter since interrupts cause transitive delays when they stop a job while it is executing a critical section.

HRT and SRT schedulability of each generated task set was determined under P-EDF-R1, C2-EDF-R1, C6-EDF-R1, and G-EDF-R1 scheduling assuming task-fair mutex, task-fair RW, and phase-fair RW spinlocks. Maximum s-blocking and pi-blocking incurred by each task were bounded using holistic blocking analysis as described in Section 5.4. Additionally, each task set was tested for HRT and SRT schedulability under each of the four schedulers assuming that tasks incur no blocking at all (however, scheduling overheads were fully accounted for). While the latter configuration does not correspond to a safe system, it does provide a baseline of each plugin's performance in the absence of locking-related delays, and was thus included for comparison purposes.

In total, we considered 7,290 parameter combinations and generated and tested 34,263,000 task sets. Each generated task set was tested for both HRT and SRT schedulability under 16 spinlock and scheduler combinations, as listed in Table 7.3.

In the remainder of this section, we discuss the major trends apparent in our results and show supporting HRT and SRT schedulability graphs. The full set of schedulability results, including all 116,640 schedulability graphs, is available online (Brandenburg, 2011). We begin by contrasting task-fair mutex, task-fair RW, and phase-fair RW locks under P-EDF scheduling and HRT constraints (Section 7.3.1), and then compare the three lock choices under G-EDF scheduling and SRT constraints (Section 7.3.2). Finally, in Section 7.3.3, we reconsider the choice of scheduler (Question Q1 from Chapters 1 and 4) if locking is realized with spinlocks.

7.3.1 Hard Real-Time Spinlock Comparison

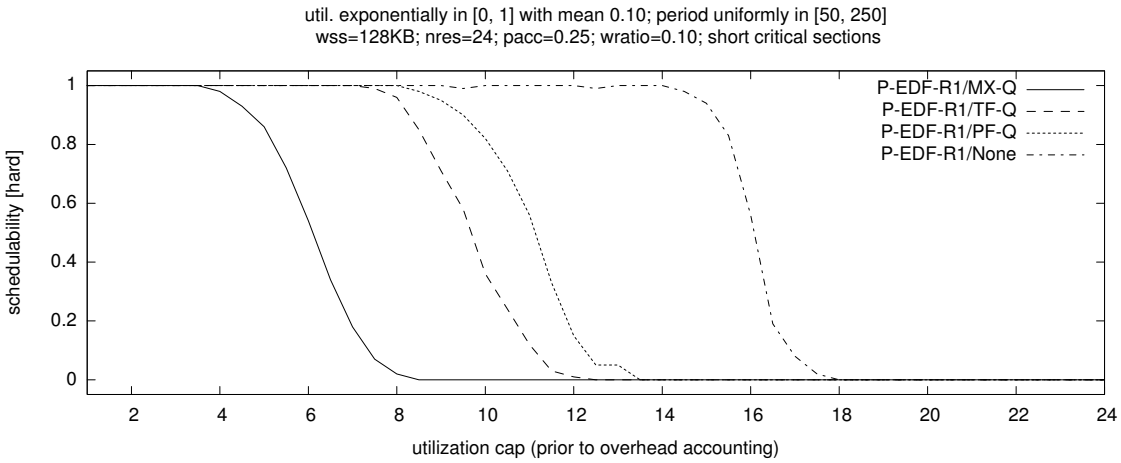
Our main result is that there is indeed a major analytical benefit to using RW locks (instead of mutex locks) if the write ratio is low. That is, in real-time systems, RW locks offer not just average-case

Scheduler/Locks	Spinlock Type	Algorithm	Blocking Analysis
P-EDF-R1/MX-Q	task-fair mutex	(Mellor-Crummey and Scott, 1991a)	Section 5.4.3
P-EDF-R1/TF-Q	task-fair RW	(Mellor-Crummey and Scott, 1991b)	Section 5.4.5
P-EDF-R1/PF-Q	phase-fair RW	Section 5.3.3	Section 5.4.6
P-EDF-R1/None	—	—	baseline w/o blocking
C2-EDF-R1/MX-Q	task-fair mutex	(Mellor-Crummey and Scott, 1991a)	Section 5.4.3
C2-EDF-R1/TF-Q	task-fair RW	(Mellor-Crummey and Scott, 1991b)	Section 5.4.5
C2-EDF-R1/PF-Q	phase-fair RW	Section 5.3.3	Section 5.4.6
C2-EDF-R1/None	—	—	baseline w/o blocking
C6-EDF-R1/MX-Q	task-fair mutex	(Mellor-Crummey and Scott, 1991a)	Section 5.4.3
C6-EDF-R1/TF-Q	task-fair RW	(Mellor-Crummey and Scott, 1991b)	Section 5.4.5
C6-EDF-R1/PF-Q	phase-fair RW	Section 5.3.3	Section 5.4.6
C6-EDF-R1/None	—	—	baseline w/o blocking
G-EDF-R1/MX-Q	task-fair mutex	(Mellor-Crummey and Scott, 1991a)	Section 5.4.3
G-EDF-R1/TF-Q	task-fair RW	(Mellor-Crummey and Scott, 1991b)	Section 5.4.5
G-EDF-R1/PF-Q	phase-fair RW	Section 5.3.3	Section 5.4.6
G-EDF-R1/None	—	—	baseline w/o blocking

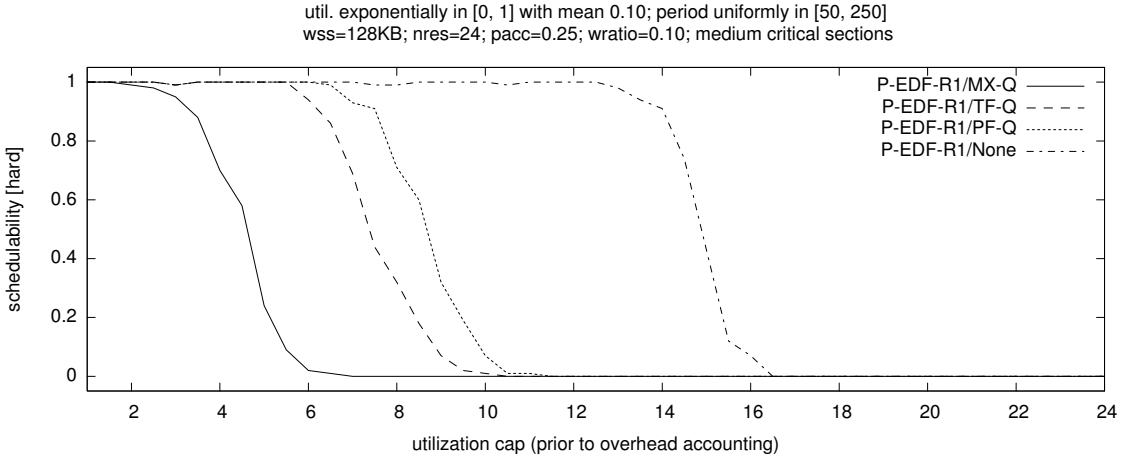
Table 7.3: List of evaluated spinlock and scheduler combinations.

improvements, but allow for substantially less-pessimistic worst-case analysis. Further, phase-fair RW locks offer an advantage over task-fair RW locks in many cases where there are some, but not "too many" writers (see below). However, while task-fair RW locks simply degrade to mutex-like performance, writers under phase-fair RW locks are subject to increased blocking if there are both many readers and writers because consecutive writer phases can be separated by a reader phase. In summary, the use of phase-fair RW locks is advisable when there are few writers (*i.e.*, in the primary use case for RW locks), whereas the task-fair mutex or RW locks result in higher schedulability if there are many writers. We illustrate these findings with representative examples in the following.

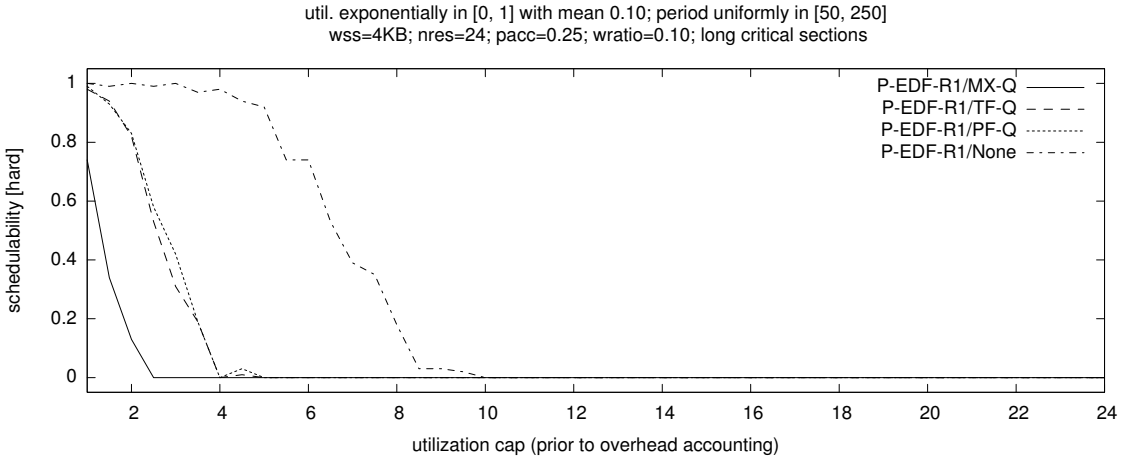
The effectiveness of RW locks is apparent in Figure 7.11, which shows HRT schedulability under each of the lock choices (and the baseline assuming no blocking) for light exponential utilizations and long periods under P-EDF-R1 scheduling. In the depicted scenarios, writes are infrequent ($wratio = 0.1$) and tasks require each of the $n_r = 24$ resources with probability $pacc = 0.25$. Inset (a) shows HRT schedulability assuming short critical sections, inset (b) depicts the same scenario assuming intermediate critical sections, and inset (c) likewise depicts performance for long critical sections. In each case, phase-fair RW locks are preferable from a schedulability perspective, although long critical sections result in poor performance under any of the lock choices.



(a) HRT schedulability for short critical sections.



(b) HRT schedulability for intermediate critical sections.



(c) HRT schedulability for long critical sections.

Figure 7.11: Graphs showing HRT schedulability under P-EDF-R1 scheduling with $n_r = 24$, $pacc = 0.25$, and $wratio = 0.1$ assuming light exponential utilizations and long periods, and **(a)** short, **(b)** intermediate, and **(c)** long critical sections.

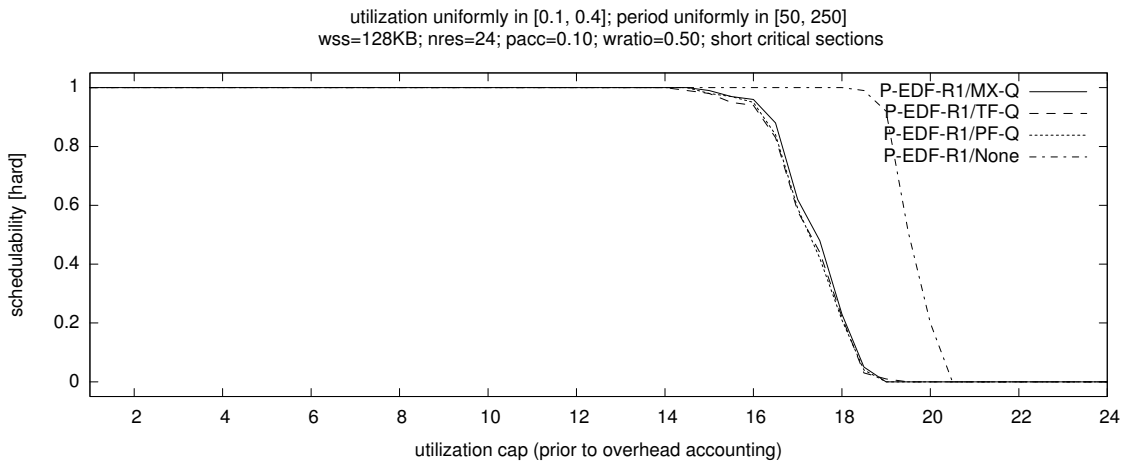


Figure 7.12: HRT schedulability under P-EDF-R1 scheduling with $n_r = 24$, $pacc = 0.1$, and $wratio = 0.5$ assuming medium uniform utilizations, long periods, and short critical sections. The curves corresponding to MX-Q, TF-Q, and PF-Q locks overlap.

Since reads are much more frequent than writes, performance under task-fair mutex locks is negatively affected by the needless serialization of (otherwise) concurrent reads, and the achieved schedulability decreases starting at $ucap \approx 4$ in Figure 7.11(a). In contrast, task-fair RW and phase-fair RW locks achieve high schedulability until $ucap \approx 8$ and $ucap \approx 9$, respectively.

Insets (a) and (b) demonstrate that phase-fair RW locks can enable higher schedulability than task-fair RW locks. This is because phase-fair RW locks offer lower bounds on s-blocking if there are multiple writers (compared to task-fair RW locks) as readers incur only $O(1)$ s-blocking, regardless of the number of competing writers. In inset (c), the difference is less pronounced since long critical sections reduce performance significantly, regardless of the choice of lock.

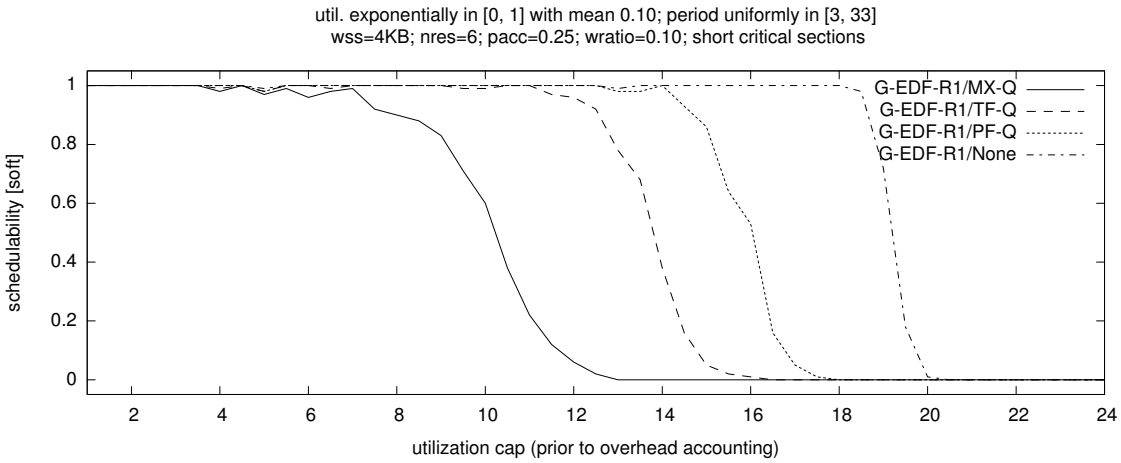
In general, the use of RW locks results in higher schedulability if there are more readers than writers. Figure 7.12 shows an example where there is no significant benefit to using RW locks. Since there are, on average, as many writers as readers ($wratio = 0.5$), reader parallelism cannot be analytically guaranteed. In this case, task-fair mutex locks are the better choice because they are simpler and thus incur somewhat lower overheads.

7.3.2 Soft Real-Time Spinlock Comparison

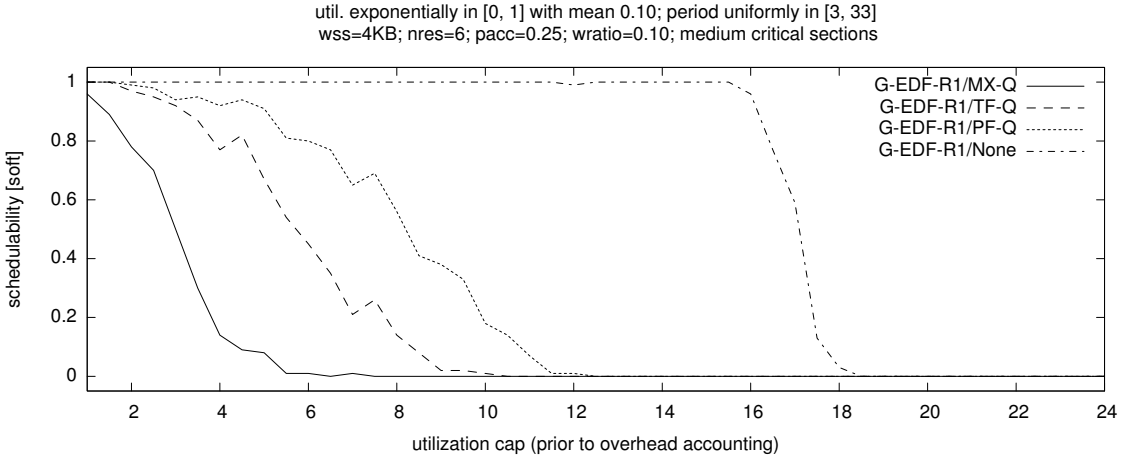
In the SRT case, the same trends as in the HRT case manifest. Average-case entry and exit overheads are almost negligible under each of the spinlocks (on the order of one microsecond or less), and average-case system call overhead is also in the range of a few microseconds (Brandenburg, 2011). Hence, jobs incur only little overheads under SRT assumptions when using spinlocks. As a result, SRT schedulability is uniformly higher than HRT schedulability under each lock type.

Figure 7.13 depicts SRT schedulability under G-EDF-R1 with light exponential utilizations, short periods, $pacc = 0.25$, $n_r = 6$, and $wratio = 0.1$ for short, intermediate, and long request lengths. From the significantly higher schedulability under task-fair RW and phase-fair RW locks in the range of approximately $6 \leq ucap \leq 17$ in Figure 7.13(a), it is apparent that RW locks offer significant analytical advantages in the SRT case as well. Again, phase-fair RW locks outperform task-fair RW locks due to the $O(1)$ delay of readers.

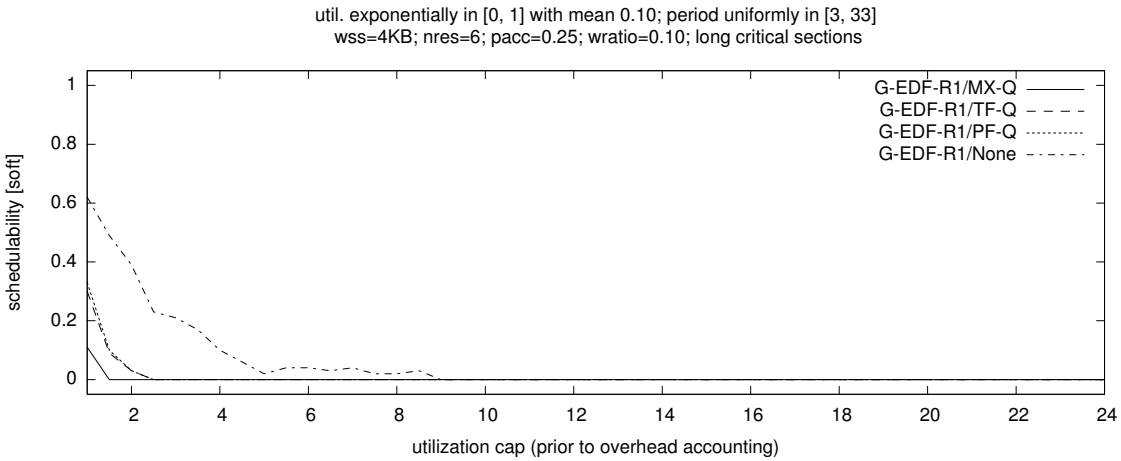
However, it should be noted that phase-fair RW locks are not always preferable to task-fair locks (under either HRT or SRT analysis). One example exhibiting this trend is shown in Figure 7.14, which depicts SRT schedulability under G-EDF-R1 with light exponential utilizations, moderate periods, $pacc = 0.1$, $n_r = 24$, and $wratio = 0.75$. As tasks are more likely to issue write requests than to issue read requests in this scenario, there is little advantage to using RW locks. Instead, task-fair mutex locks are a good choice because they are simpler than RW locks and therefore have much lower implementation overheads. Since task-fair RW locks reduce to mutex-like behavior in the worst case they achieve schedulability that is similar to that of task-fair mutex locks (asides differences in overhead). Phase-fair RW locks, however, can degenerate to worse than mutex-like performance. While non-preemptive request execution and the strict FIFO ordering of requests ensure that a writer is delayed by at most $m - 1$ earlier-issued requests (either reads or writes) under task-fair locks (either RW or mutex), a writer under phase-fair RW locks may be delayed by $m - 1$ earlier-issued writer phases, and by an *additional* $m - 1$ interspersed reader phases. Therefore, if there are many other competing writers, s-blocking for writers can be substantially worse under phase-fair RW locks than under task-fair locks (either RW or mutex). This indicates that phase-fair RW locks are an appropriate choice only in situations where writes are less frequent than reads. In other words, phase-fair RW locks favor readers at the expense of writers, which is a worthwhile



(a) SRT schedulability for short critical sections.



(b) SRT schedulability for intermediate critical sections.



(c) SRT schedulability for long critical sections.

Figure 7.13: Graphs showing SRT schedulability under G-EDF-R1 scheduling with $n_r = 6$, $pacc = 0.25$, and $wratio = 0.1$ assuming light exponential utilizations and short periods, and **(a)** short, **(b)** intermediate, and **(c)** long critical sections.

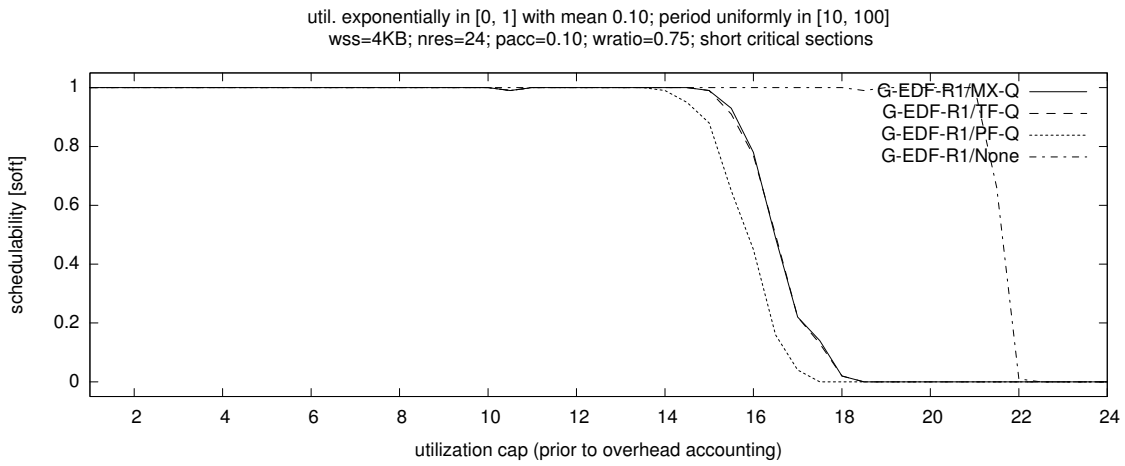


Figure 7.14: SRT schedulability under G-EDF-R1 scheduling with $n_r = 24$, $pacc = 0.1$, and $wratio = 0.75$ assuming light exponential utilizations, moderate periods, and short critical sections. The curves corresponding to MX-Q and TF-Q locks overlap.

tradeoff only if there are more readers than writers. However, note that it is trivial to mix task-fair mutex spinlocks and phase-fair RW spinlocks; that is, the choice of spinlock can be made on a resource-by-resource basis.

7.3.3 Scheduler Comparison

The preceding discussion has focused on comparing spinlock choices under partitioned and global scheduling. In this section, we consider the possibility that the relative performance of schedulers changes when locking constraints are introduced. In Chapter 4, we evaluated 22 schedulers in terms of HRT and SRT schedulability on our experimental platform. It is infeasible to replicate Section 4.5 in its entirety here; rather, we focus on the best-performing plugins (namely P-EDF in the HRT case and G-EDF and C-EDF variants in the SRT case), and illustrate the major trends with example graphs. Fortunately, introducing locking constraints does *not* invalidate our earlier findings: P-EDF remains the best choice in the HRT case, and larger cluster sizes remain effective at lessening the impact of bin-packing constraints in the SRT case.

This can be seen in Figure 7.15(a), which depicts HRT schedulability when locking constraints are realized with task-fair mutex locks under P-EDF-R1, C2-EDF-R1, C6-EDF-R1, and G-EDF-R1 for medium uniform utilizations, long periods, $pacc = 0.25$, $n_r = 12$, and $wratio = 0.2$.

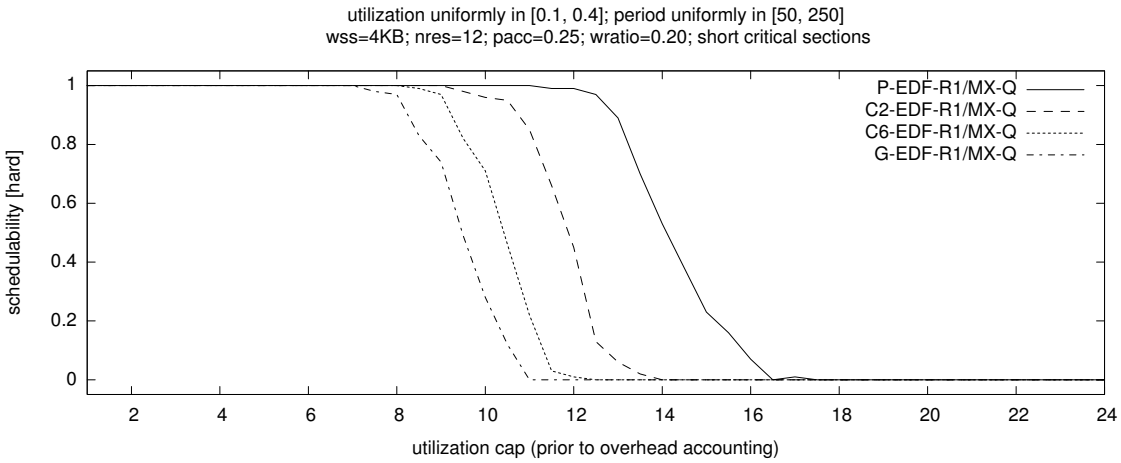
Figure 7.15(b) depicts the same scenario assuming task-fair RW locks, and Figure 7.15(c) does likewise assuming phase-fair RW locks. As is readily apparent, HRT schedulability is higher in smaller clusters, with the highest HRT schedulability being achieved by P-EDF-R1, which matches the relative performance when all tasks are independent (see Chapter 4). This observation is representative of our entire study and suggests that non-preemptive spinlocks affect the performance of each tested event-driven scheduler equally.

This is also true in the SRT case. Example graphs exhibiting this trend are shown in Figure 7.16, which is organized similarly to Figure 7.15. Each inset depicts SRT schedulability under P-EDF-R1, C2-EDF-R1, C6-EDF-R1, and G-EDF-R1 for light bimodal utilizations, short periods, $pacc = 0.4$, $n_r = 12$, and $wratio = 0.1$, and assuming resources are protected by task-fair mutex, task-fair RW, and phase-fair RW spinlocks, respectively. As can be seen in inset (a), the difference between schedulers is least pronounced when using task-fair mutex spinlocks. When using RW locks, however, G-EDF-R1 achieves the highest SRT schedulability. Generally speaking, the conclusions from Chapter 4 with regard to the SRT performance remain valid when jobs synchronize by means of non-preemptive spinlocks.

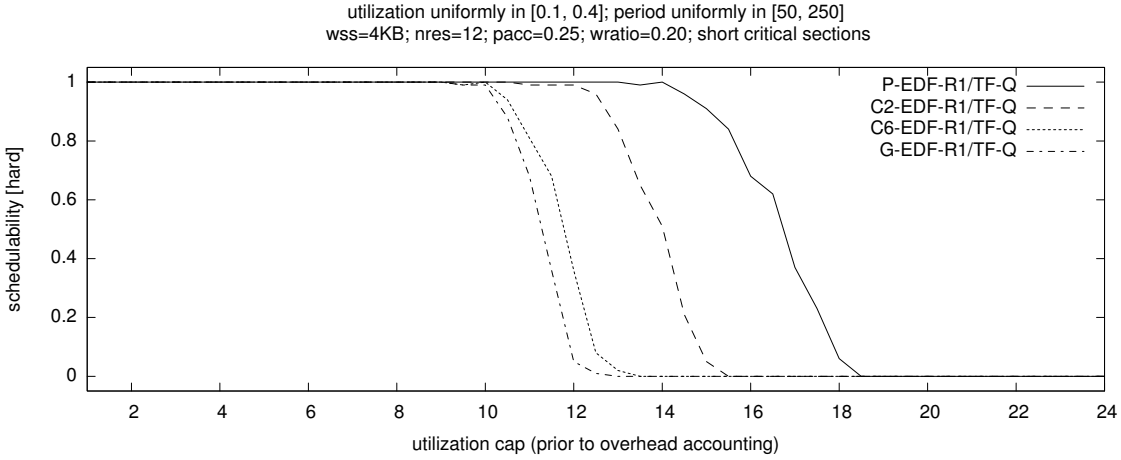
It should be noted, however, that no attempt was made to take resource-sharing considerations into account when partitioning task sets. That is, tasks were assigned using the normal worst-fit decreasing heuristic as in Chapter 4. Better results could likely be achieved by employing resource-sharing-aware partitioning heuristics. However, such heuristics are still an area of active research (Lakshmanan *et al.*, 2009; Nemati *et al.*, 2010; Hsiu *et al.*, 2011) and beyond the scope of this dissertation. It would be interesting to reevaluate the relative performance of partitioned, clustered, and global scheduling once it has become clear which partitioning heuristics are best suited to partitioning task sets with shared resources.

7.4 Semaphore Protocols for S-Oblivious Analysis

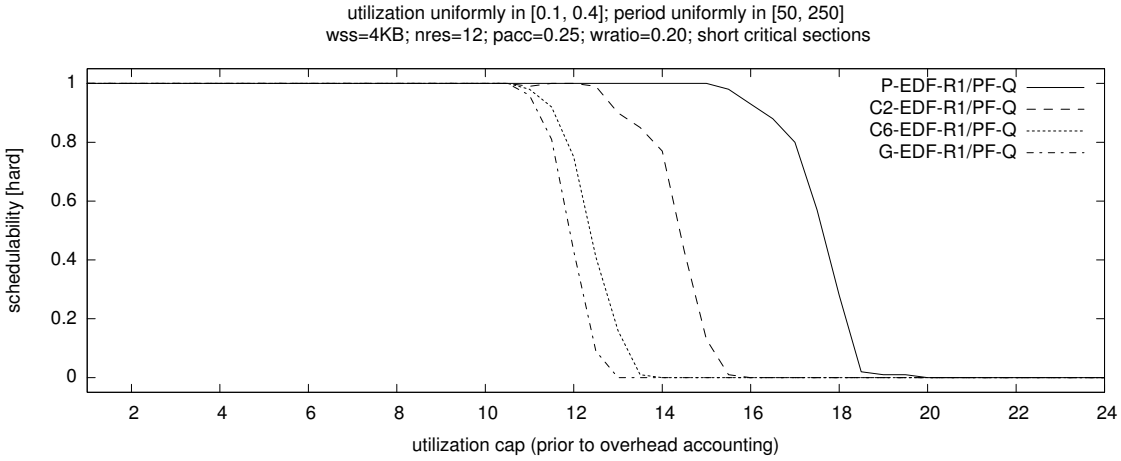
In a second schedulability study, we compared semaphore-based mutex protocols with each other and with task-fair mutex spinlocks. We used a setup similar to the one discussed in the preceding section, however, we only considered $wratio = 1$ since all locks that we considered in this study



(a) HRT schedulability under MX-Q locks.

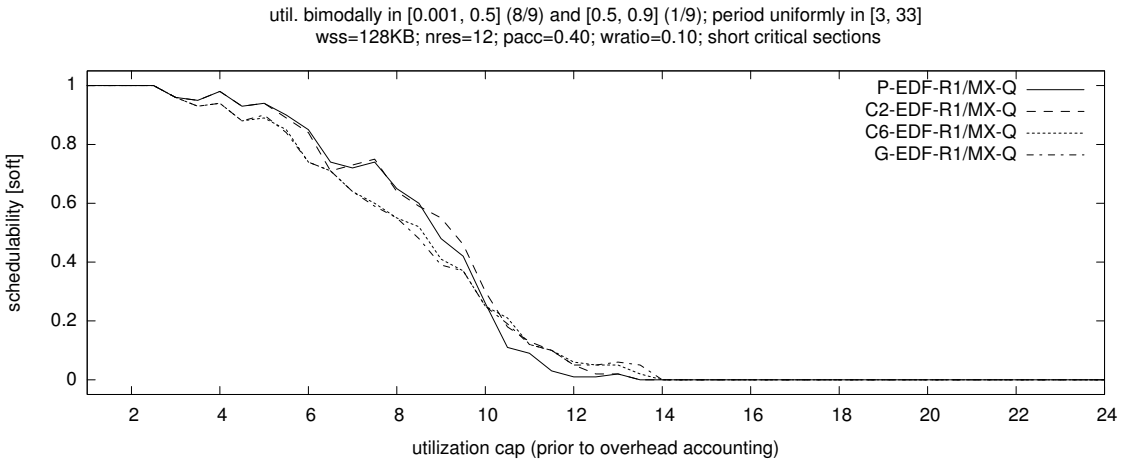


(b) HRT schedulability under TF-Q locks.

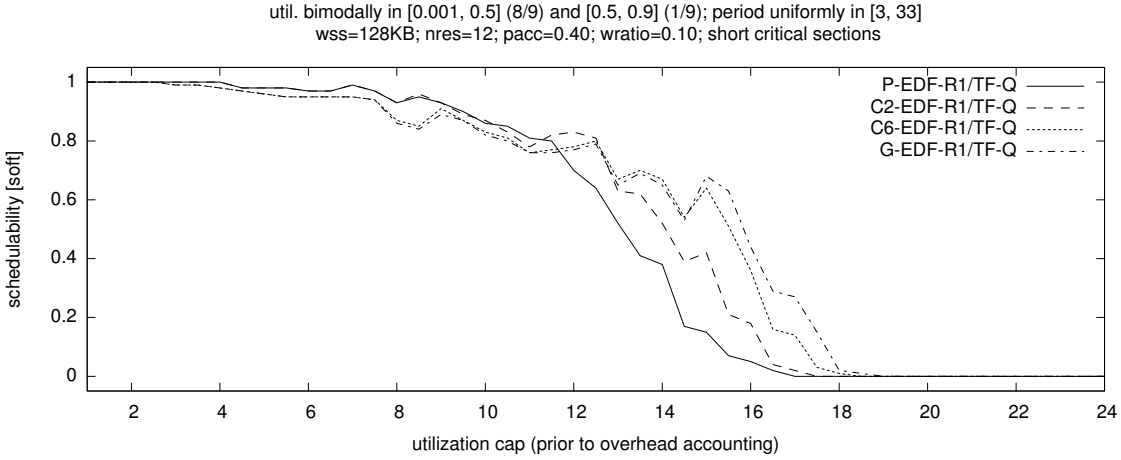


(c) HRT schedulability under PF-Q locks

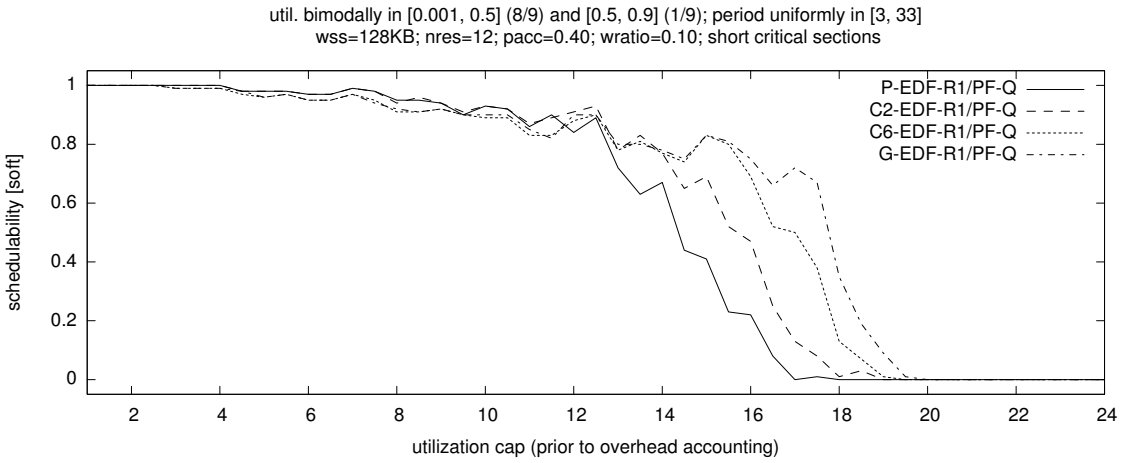
Figure 7.15: Graphs showing HRT schedulability under P-EDF-R1, C2-EDF-R1, C6-EDF-R1, and G-EDF-R1 scheduling with $n_r = 12$, $pacc = 0.25$, and $wratio = 0.2$ assuming medium uniform utilizations, long periods, and short critical sections.



(a) SRT schedulability under MX-Q locks.



(b) SRT schedulability under TF-Q locks.



(c) SRT schedulability under PF-Q locks

Figure 7.16: Graphs showing SRT schedulability under P-EDF-R1, C2-EDF-R1, C6-EDF-R1, and G-EDF-R1 scheduling with $n_r = 12$, $pacc = 0.4$, and $wratio = 0.1$ assuming light bimodal utilizations, short periods, and short critical sections.

are mutex locks. Since this reduced the parameter space considerably, we extended the considered ranges of the other parameters to $n_r \in \{1, 3, 5, 12, 24\}$, and $pacc \in \{0.1, 0.25, 0.4, 0.55, 0.7, 0.85\}$.

As in the previous schedulability study, we considered two WSSs, namely 4 KB and 128 KB. In contrast to spin-based protocols, suspension-based protocols are affected by the assumed WSS because jobs lose cache affinity when they suspend to wait for a resource. We chose the two evaluated WSSs for the following reasons. First, a WSS of 4 KB is quite small and thus represents a best-case scenario for suspension-based protocols. That is, suspension-based protocols are penalized by large WSSs; therefore, performance as observed with a small WSS is an upper bound on the performance (in terms of schedulability) of semaphore protocols. Second, we chose a WSS of 128 KB as another, more realistic WSS to be evaluated.

Since WSS-related overheads affect all suspension-based protocols equally, limiting our study to two WSSs does not bias it against or for any of the evaluated semaphore protocols. Nonetheless, in the future, it would be interesting to consider a larger range of WSSs to study the range in which semaphore-based locking protocols are competitive from a schedulability point of view (*i.e.*, to evaluate each semaphore-based protocol also in terms of its weighted schedulability score). However, it should be noted that such an expansion of the parameter space carries a large computational cost.[3]

In total, we evaluated 22,842,000 task sets corresponding to 4,860 parameter combinations under each of the 13 protocol and scheduler combinations listed in Table 7.4 in terms of both HRT and SRT schedulability. Besides semaphore protocols, we also considered task-fair mutex spinlocks as a baseline. The entire set of schedulability results can be found online (Brandenburg, 2011), including 53,460 graphs visualizing HRT and SRT schedulability under each scheduler and locking protocol.

In this section, we discuss our results pertaining to s-oblivious analysis. Locking protocols for s-aware analysis are discussed thereafter in Section 7.5. Finally, we compare spin-based and suspension-based locking protocols in terms of HRT and SRT schedulability in Section 7.6.

We implemented and evaluated four locking protocols for s-oblivious analysis in LITMUS$^{\mathrm{RT}}$: the global FMLP and the global OMLP were implemented in the G-EDF plugin, the clustered

[3]Extrapolating from the runtimes of the presented schedulability study for two WSSs, evaluating all WSSs previously considered in Chapter 4 would require more than 10 days of continuous execution on 64 nodes of UNC's research cluster TOPSAIL (unless significant improvements in implementation efficiency of the experimental framework are made).

Scheduler/Locks	Protocol Definition	Blocking Analysis	Type
P-EDF-R1/OMLP	clustered OMLP (Section 6.2.2)	Section 6.4.2	s-oblivious
P-FP-R1/MPCP-VS	MPCP-VS (Lakshmanan *et al.*, 2009)	(Lakshmanan *et al.*, 2009)	s-oblivious
P-FP-R1/MPCP	MPCP (Rajkumar, 1990)	(Lakshmanan *et al.*, 2009)	s-aware
P-FP-R1/DPCP	DPCP (Rajkumar *et al.*, 1988)	(Rajkumar, 1991)	s-aware
P-FP-R1/FMLP$^+$	preemptive FMLP$^+$ (Section 6.3)	Section 6.4.3	s-aware
P-FP-R1/NP-FMLP$^+$	non-preemptive FMLP$^+$ (Section 6.3)	Section 6.4.3	s-aware
C2-EDF-R1/OMLP	clustered OMLP (Section 6.2.2)	Section 6.4.2	s-oblivious
C6-EDF-R1/OMLP	clustered OMLP (Section 6.2.2)	Section 6.4.2	s-oblivious
G-EDF-R1/FMLP	global FMLP (Block *et al.*, 2007)	(Block *et al.*, 2007)	s-oblivious
G-EDF-R1/OMLP	global OMLP (Section 6.2.2)	Section 6.4.1	s-oblivious
C24-EDF-R1/OMLP	clustered OMLP (Section 6.2.2)	Section 6.4.2	s-oblivious
G-EDF-R1/MX-Q	non-preemptive spinlocks	Section 5.4.3	—
P-EDF-R1/MX-Q	non-preemptive spinlocks	Section 5.4.3	—

Table 7.4: List of evaluated lock and scheduler combinations. C24-EDF-R1 denotes the C-EDF plugin instantiated as a global scheduler (*i.e.*, $c = m$) with dedicated interrupt handling. That is, C24-EDF-R1 is effectively an implementation of G-EDF, but the implemented locking protocol is the clustered OMLP using priority donation (and not the global OMLP using priority inheritance as implemented in the G-EDF plugin).

OMLP was implemented in the C-EDF plugin (which can also be configured to act as a global scheduler), and Lakshmanan *et al.*'s MPCP-VS was implemented in the P-FP plugin.

In the following, we first contrast the three protocol choices for G-EDF (namely the global FMLP, the global OMLP, and the clustered OMLP in the case of $c = m$), and then compare the two s-oblivious suspension-based locking protocols suitable to partitioned scheduling, namely the clustered OMLP and the MPCP-VS.

7.4.1 The Global FMLP vs. the Global OMLP vs. the Clustered OMLP

In this dissertation, we have discussed the global FMLP mainly in the context of s-aware schedulability analysis (Chapter 6). Nonetheless, the global FMLP was the first suspension-based locking protocol proposed for G-EDF. Due to the lack of s-aware schedulability analysis for G-EDF at the time of its publication, the published blocking analysis for the global FMLP assumes s-oblivious schedulability analysis (Block *et al.*, 2007).

Since the global FMLP uses priority inheritance with simple FIFO queues, jobs are subject to $\Omega(n)$ s-oblivious pi-blocking under it. It is hence not asymptotically optimal under s-oblivious analysis, in contrast to the global OMLP. Therefore, one might expect the global OMLP to always

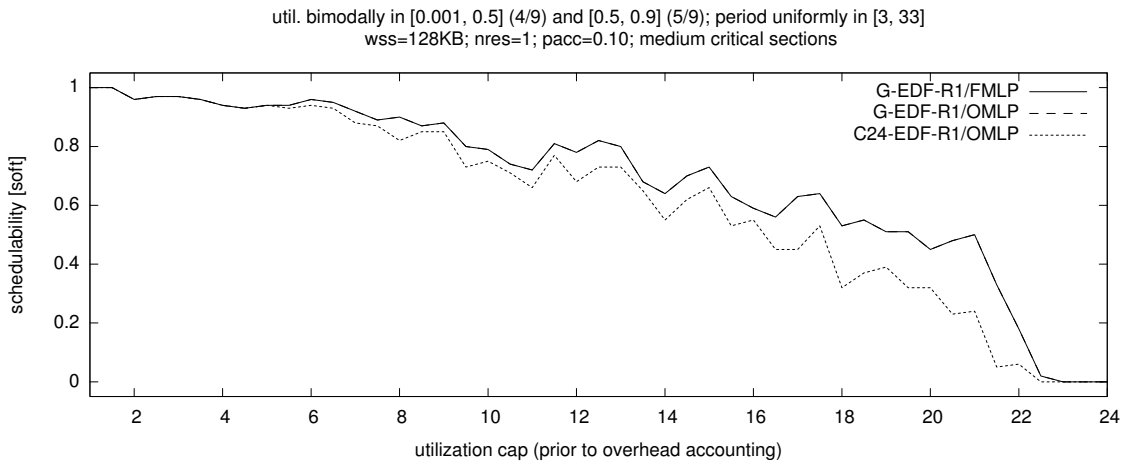


Figure 7.17: SRT schedulability under G-EDF-R1 scheduling with the global FMLP, G-EDF-R1 scheduling with the global OMLP, and C24-EDF-R1 scheduling with the clustered OMLP for heavy bimodal utilizations, short periods, $n_r = 1$, $pacc = 0.1$ and intermediate critical section lengths. The curves corresponding to global OMLP and the global FMLP overlap.

outperform the global FMLP under G-EDF. However, this is in fact not the case. One situation in which the bounds on blocking under both protocols are identical is when at most $m + 1$ tasks share a given resource since the global OMLP reduces to a simple FIFO protocol that is equivalent to the global FMLP in this case. This is a common situation in our experimental setup as there are $m = 23$ processors available to real-time tasks under G-EDF-R1 scheduling on our platform.

One example exhibiting this effect is shown in Figure 7.17, which depicts SRT schedulability under G-EDF-R1 with heavy bimodal utilizations, short periods, intermediate critical section lengths, $pacc = 0.1$, and $n_r = 1$ under the global FMLP, the global OMLP, and the clustered OMLP. Since each task accesses the single shared resource with low probability, only few tasks share the resource. Hence, there is no benefit to using the OMLP over the FMLP in this case, which is reflected by the observed schedulability as the curves for both protocols overlap. The clustered OMLP, which is based on priority donation, achieves lower schedulability than either of the protocols based on priority inheritance in this scenario because priority donation causes otherwise independent jobs to incur pi-blocking. Due to the small access probability $pacc$, there are many independent tasks in this particular scenario.

In situations with high contention for resources (*i.e.*, if $pacc$ is large and if there are many tasks), use of the global OMLP does result in higher schedulability, as expected. One such example

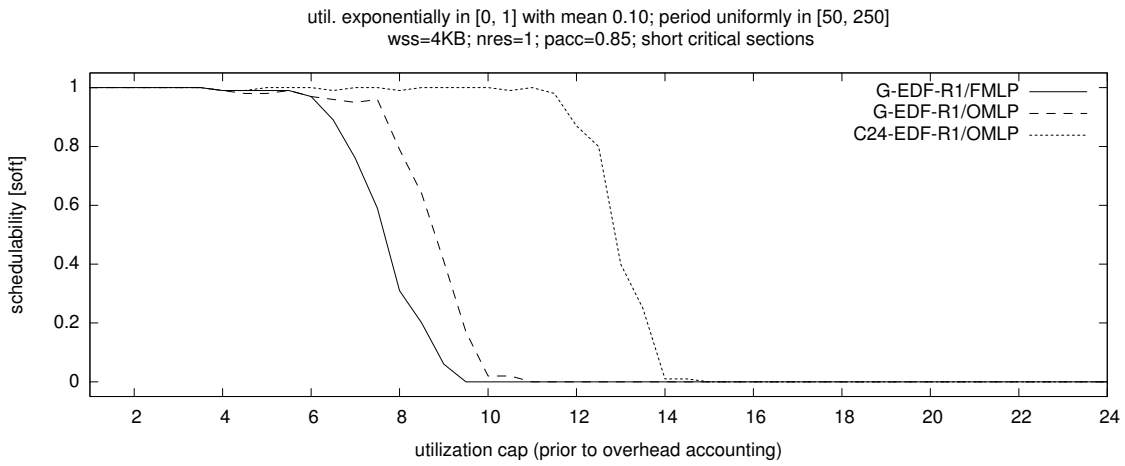


Figure 7.18: SRT schedulability under G-EDF-R1 scheduling with the global FMLP, G-EDF-R1 scheduling with the global OMLP, and C24-EDF-R1 scheduling with the clustered OMLP for light exponential utilizations, long periods, $n_r = 1$, $pacc = 0.85$ and short critical section lengths.

is shown in Figure 7.18, which depicts SRT schedulability for light exponential utilizations, long periods, $pacc = 0.85$, $n_r = 1$, and short critical sections. Since a majority of the tasks access the single resource (in the expected case), maximum queue length is long under the global FMLP. The global OMLP's bound of $2m - 1$ pi-blocking requests (per issued request) is lower in this case and thus yields higher schedulability. This illustrates that the global OMLP can be an improvement over the simpler, asymptotically sub-optimal FMLP. Moreover, since most tasks access the single shared resource in this scenario, priority donation causes only little additional capacity loss (if any). Therefore, the clustered OMLP is the best-performing locking protocol in this scenario since the maximum queue length is limited to m concurrent requests under it.

Perhaps counterintuitively, there also exist scenarios in which the global OMLP as discussed in Chapter 6 and implemented in LITMUS$^{\mathrm{RT}}$ can yield *worse* schedulability than the global FMLP. Let A_q denote the number of tasks that access a given resource ℓ_q, as in the analysis of the global OMLP (Definition 6.8). If $m + 1 < A_q < 2m - 1$, then the bound on the number of pi-blocking requests under the global OMLP is $2m - 1$ (per issued request), which, in this case, is in fact higher than that under the global FMLP (which is always $A_q - 1$ due to its reliance on FIFO queues). This can lead to unfavorable schedulability results under the global OMLP, as is apparent in Figure 7.19, which shows SRT schedulability under G-EDF-R1 with medium uniform utilizations, long periods,

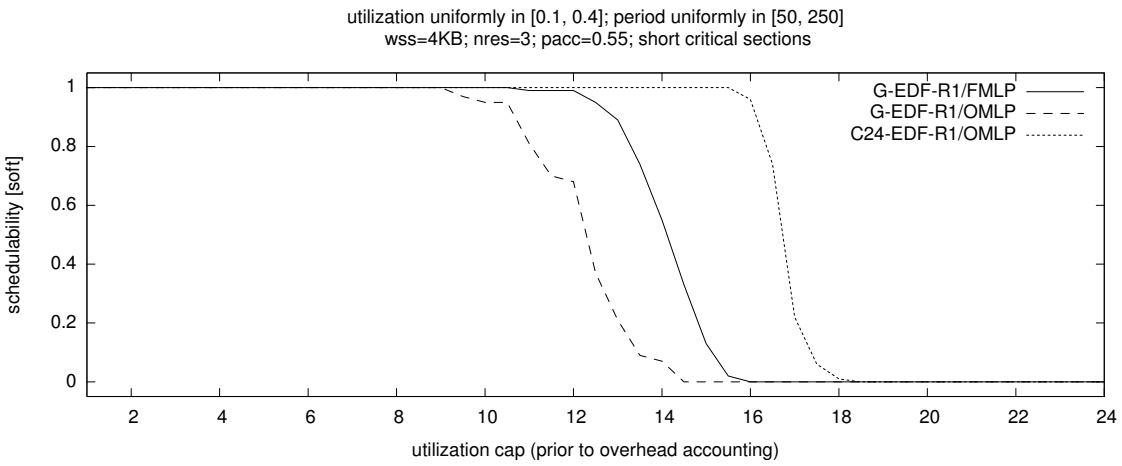


Figure 7.19: SRT schedulability under G-EDF-R1 scheduling with the global FMLP, G-EDF-R1 scheduling with the global OMLP, and C24-EDF-R1 scheduling with the clustered OMLP for medium uniform utilizations, long periods, $n_r = 3$, $pacc = 0.55$ and short critical section lengths.

$pacc = 0.55$, $n_r = 3$, and short critical sections. In this example, a moderate number of tasks share each resource in the expected case since $pacc = 0.55$, which produces the discussed effect. The clustered OMLP, however, still outperforms the FMLP by a large margin, showing the advantage of its $O(m)$ bound on maximum s-oblivious pi-blocking.

When using protocols based on priority inheritance, a hybrid of the global FMLP and the global OMLP would be preferable: if $A_q \leq 2m$, then resource requests are satisfied according to the FMLP's rules; otherwise, if $A_q > 2m$, then the global OMLP's rules have precedence. In LITMUS$^{\mathrm{RT}}$, the G-EDF plugin could be trivially modified to implement such a hybrid protocol since the FDSO layer (see Section 3.3.1.3) is aware of A_q (the number of tasks holding a reference to a semaphore is known since the semaphore's kernel memory must be deallocated when the last reference is dropped).

If contention is high and there are few independent tasks, however, the clustered OMLP achieves higher HRT and SRT schedulability than either the global FMLP or the global OMLP.

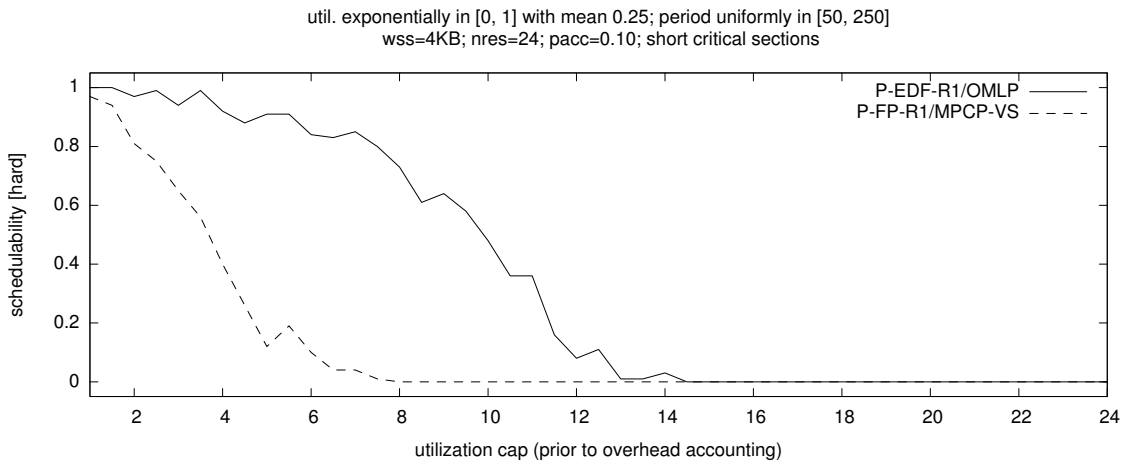


Figure 7.20: HRT schedulability under P-EDF with the clustered OMLP and under P-FP with the MPCP-VS for medium exponential utilizations, long periods, $n_r = 24$, $pacc = 0.1$ and short critical section lengths.

7.4.2 The Clustered OMLP vs. the MPCP-VS

Recall from Section 2.4.4.2 that the MPCP-VS is a variant of the MPCP in which suspension time is charged as execution time,[4] and concurrent resource requests originating from the same processor are disallowed. Under the clustered OMLP, priority donation similarly limits the number of concurrent requests to c per cluster, which under partitioning ($c = 1$) is equivalent to prohibiting concurrent requests (in each partition).

A major difference between the protocols exists with regard to queuing: the MPCP-VS uses priority queues, which, as discussed in Chapter 6, implies that it is asymptotically suboptimal under either s-aware and s-oblivious analysis, whereas the clustered OMLP uses a simpler FIFO queue, which ensures $O(m)$ s-oblivious pi-blocking. However, does this difference in asymptotic optimality affect actual protocol performance (as measured in terms of schedulability)? Our data clearly shows that it does. One typical example is shown in Figure 7.20, which depicts HRT schedulability under P-EDF-R1 scheduling with the clustered OMLP and under P-FP-R1 scheduling with the MPCP-VS for medium exponential utilizations, long periods, $pacc = 0.1$, $n_r = 24$, and short critical section lengths. From the depicted graph, it is clearly apparent that the OMLP under P-EDF-R1

[4]While the MPCP-VS uses s-oblivious schedulability analysis, it should be noted that it was proposed (Lakshmanan *et al.*, 2009) prior to the introduction of the term "s-oblivious analysis" (Brandenburg and Anderson, 2010a).

achieves higher HRT schedulability than the MPCP-VS under P-FP-R1 scheduling. This is in fact representative for all of our schedulability results, where the clustered OMLP outperforms the MPCP-VS in all scenarios where there are the significant differences among protocols.[5] While some of the observed performance gap is likely due to the locking-unrelated differences between P-EDF and P-FP, the conclusion remains that, when using s-oblivious analysis, P-FP scheduling with the MPCP-VS is not preferable to P-EDF scheduling with the clustered OMLP.

While it would be somewhat interesting to perform an "apples-to-apples" comparison of the MPCP-VS and the clustered OMLP in the future, there currently exists no analysis for the MPCP-VS under P-EDF scheduling, and the current implementation of LITMUS$^{\mathrm{RT}}$ only supports the OMLP in EDF-based plugins (for mundane reasons; support for the OMLP under P-FP scheduling is conceptually straightforward, but has simply not yet been implemented and evaluated in LITMUS$^{\mathrm{RT}}$ at this point). A primary feature of P-FP scheduling is that effective s-aware schedulability analysis is available for FP scheduling, which enables practical s-aware locking protocols, which we discuss next.

7.5 Semaphore Protocols for S-Aware Analysis

We implemented three suspension-based locking protocols for s-aware schedulability analysis in LITMUS$^{\mathrm{RT}}$'s P-FP plugin. Besides our partitioned FMLP^{+}, we implemented the two major suspension-based multiprocessor locking protocols proposed in prior work, namely the DPCP (Rajkumar *et al.*, 1988; Rajkumar, 1991) and the MPCP (Rajkumar, 1990, 1991).

Each of the three protocols uses priority boosting as the primary mechanism to ensure progress (locally, the DPCP also uses priority inheritance if agents are ceiling-blocked by the PCP). As was the case with the OMLP and the MPCP-VS, the main difference between the FMLP^{+} and the two PCP variants is the choice of queue. The DPCP and the MPCP rely on priority queues, which gives rise to non-optimal s-aware pi-blocking (recall Section 6.3.3), whereas the FMLP^{+} ensures $O(n)$ s-aware pi-blocking due to its use of FIFO queues (Section 6.3). Recall that the FMLP^{+} can use

[5] Some parameter combinations yield uniformly low schedulability under all protocols due to infeasibly high levels of contention. For example, this is frequently the case if $n_r = 24$ and $pacc = 0.85$.

either preemptive or non-preemptive request execution. Before comparing the FMLP$^+$ to the classic MPCP and DPCP, we first discuss which FMLP$^+$ variant is preferable on our platform.

7.5.1 The FMLP$^+$ with Preemptive and Non-Preemptive Execution

The original partitioned FMLP proposed by Block *et al*. (2007) used non-preemptive execution of resource requests to simplify blocking analysis and to reduce overheads. The FMLP$^+$ supports non-preemptive request execution as well, but is actually simpler to analyze when using preemptive request execution. The bound on worst-case s-aware pi-blocking (Section 6.4.3) is also lower when requests are executed preemptively. Therefore, whether to allow resource-holding jobs to be preempted by other resource-holding (and thus also priority-boosted) jobs is a tradeoff between pi-blocking bounds (which are lower if requests are executed preemptively) and overhead accounting (which is less pessimistic if requests are executed non-preemptively). Which variant is preferable hence depends on the overhead characteristics of the underlying platform.

In LITMUS$^{\mathrm{RT}}$, and on our hardware platform, the observed worst-case and average-case overheads do not justify making critical sections non-preemptive: in almost all of the evaluated scenarios (in which meaningful differences among locking protocols exist), the FMLP$^+$ with preemptive request execution achieves higher schedulability than the FMLP$^+$ with non-preemptive request execution. A representative example exhibiting this trend is shown in Figure 7.21, which depicts HRT schedulability under either variant for bimodal medium utilizations, moderate periods, $pacc = 0.25$, $n_r = 1$, and short critical sections.

Largely similar trends can also be observed in the SRT case, which is not depicted here but can be found online (Brandenburg, 2011). Notably, there exist a few exceptions where the FMLP$^+$ with non-preemptive request execution achieves slightly higher SRT schedulability than the FMLP$^+$ with preemptive request execution. However, this occurs only in few cases where there are many tasks with long periods, many resources, $pacc \leq 0.25$, and if critical sections are short. In these scenarios, overheads are the dominating factor; non-preemptive execution is hence preferable.

To summarize, in the SRT case, we found preemptive execution of requests to be preferable in a majority of the tested scenarios with short periods, and in all tested scenarios with moderate or long periods; in the HRT case, we found preemptive execution to be preferable in all tested scenarios

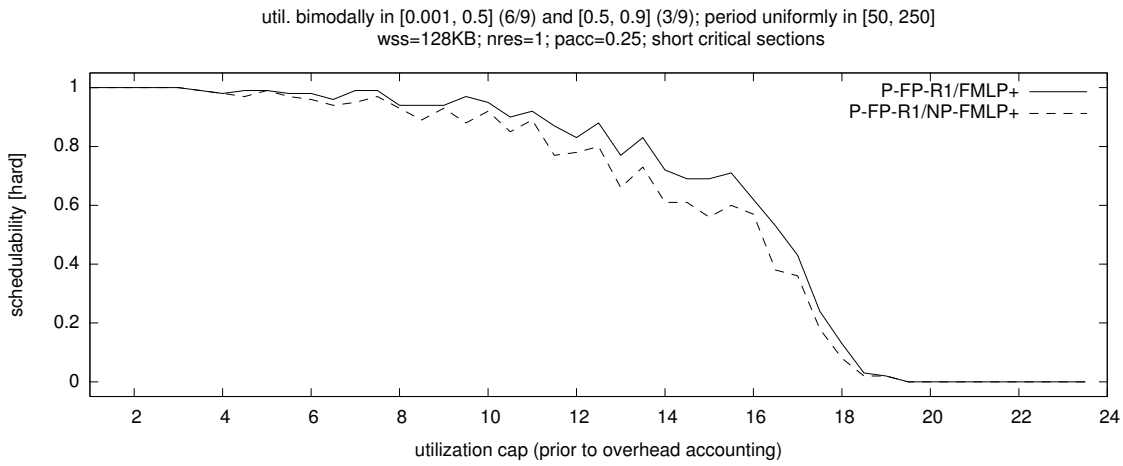


Figure 7.21: HRT schedulability under P-FP-R1 scheduling with two variants of the FMLP$^+$ for medium bimodal utilizations, moderate periods, $n_r = 1$, $pacc = 0.25$, and short critical sections.

regardless of the critical section length. We therefore do not consider the non-preemptive FMLP$^+$ in the remainder of this chapter.

7.5.2 The FMLP$^+$ vs. the MPCP vs. the DPCP

The FMLP$^+$, the MPCP, and the DPCP are quite different from one another. Of the three, the FMLP$^+$ is the only protocol to use FIFO queues, and the DPCP is the only protocol to use the RPC model where each resource is only accessed from a specific processor. These differences have noticeable impact on observed overheads.

The FMLP$^+$ generally incurs the lowest overheads due to its simple structure. In particular, the MPCP incurs higher acquisition overheads than the FMLP$^+$ when there are many waiting jobs because the MPCP requires jobs to wait in priority order, whereas the FMLP$^+$ uses low-overhead FIFO queues (in Linux, wait queues are implemented as doubly-linked lists, which enables efficient $O(1)$ enqueueing and dequeuing at either end).

The DPCP causes the highest overheads among all evaluated locking protocols for partitioned scheduling since a job that issues a resource request (*i.e.*, that locks a semaphore) must activate a "remote agent," which is implemented as temporary task migration in the version of LITMUS$^{\mathrm{RT}}$

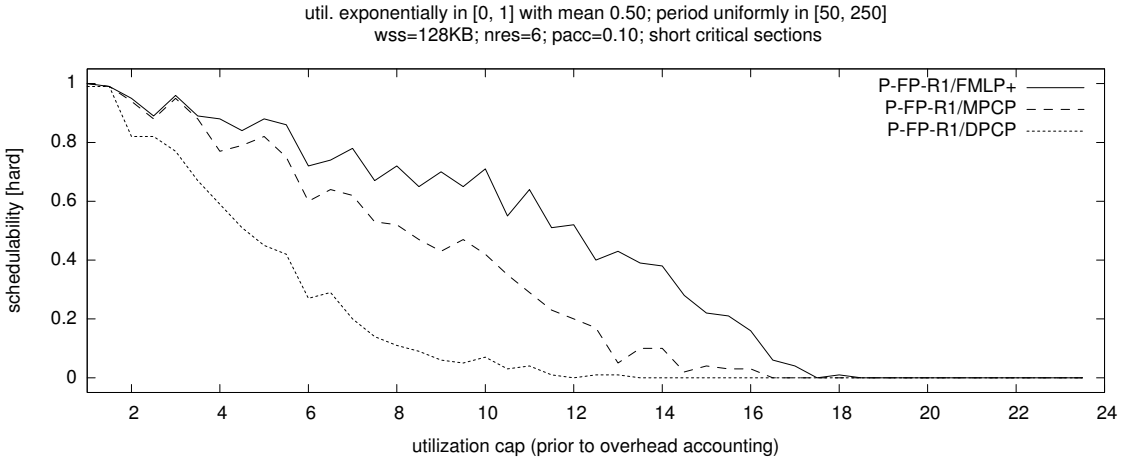
underlying this dissertation.[6] This effectively doubles the number of scheduler invocations and context switches that must be carried out, and hence results in significantly higher overheads.

In our experimental study, we found that the FMLP$^+$ outperforms both the MPCP and the DPCP in terms of HRT schedulability in all of the tested scenarios (aside those where schedulability is equally low under each of the tested protocols due to excessive contention). A typical example is shown in Figure 7.22, which depicts HRT schedulability under the FMLP$^+$, the MPCP, and the DPCP for heavy exponential utilizations, long periods, $pacc = 0.1$, and $n_r = 6$. In inset (a), which depicts HRT schedulability under each protocol assuming short critical sections, schedulability remains higher under the FMLP$^+$ until $ucap \approx 17$, whereas the curves corresponding to the DPCP and the MPCP reach zero already at $ucap \approx 12$ and $ucap \approx 16$, respectively. Insets (b) and (c) show that the same general trend manifests for longer critical sections as well. In general, long critical sections greatly reduce schedulability under any locking protocol. In this particular example, critical section lengths have only a minor impact due to the low access probability $pacc$ (*i.e.*, there are only few requests).

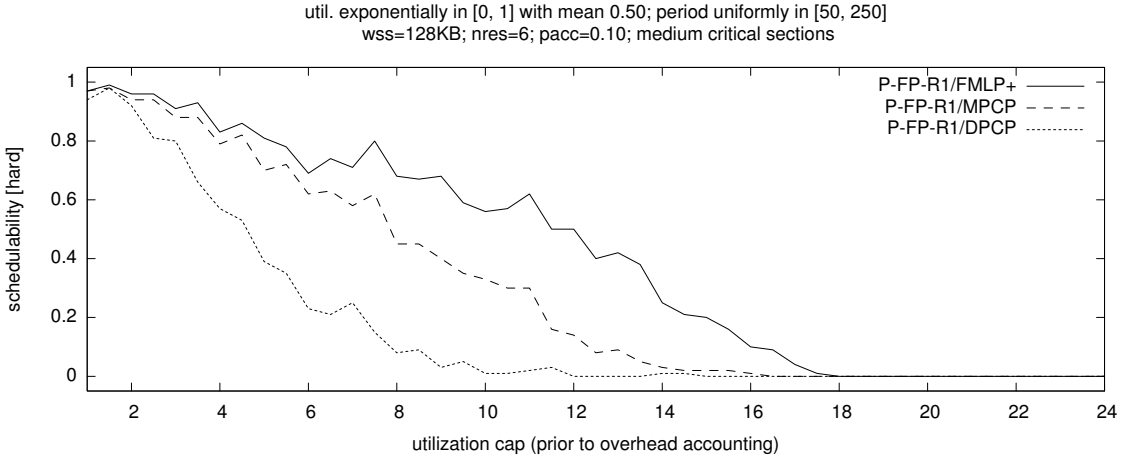
The FMLP$^+$ outperforms the MPCP in all tested scenarios in the SRT case as well. The FMLP$^+$ also outperforms the DPCP in the majority of the tested scenarios (again, aside those without meaningful differences in schedulability); however, there exist some cases where the DPCP achieves higher SRT schedulability than the FMLP$^+$. The scenario with the largest performance gap is exhibited in Figure 7.23, which depicts SRT schedulability under the FMLP$^+$, the MPCP, and the DPCP for uniform light utilizations, long periods, $pacc = 0.1$, and $n_r = 24$. The DPCP benefits from the fact that average-case overheads are much lower than worst-case overheads and achieves somewhat higher SRT schedulability than the FMLP$^+$ in this scenario with many tasks, many resources, and little contention. The MPCP is not competitive in this scenario. However, there exist many scenarios in which the MPCP performs better than the DPCP, and vice versa. The full set of results can be found online (Brandenburg, 2011).

Overall, our results show the FMLP$^+$ to yield improved schedulability in a large majority of the evaluated scenarios when compared to the two classic locking protocols for P-FP scheduling. The FMLP$^+$ benefits to some extent from its simplicity and its hence lower overheads; however,

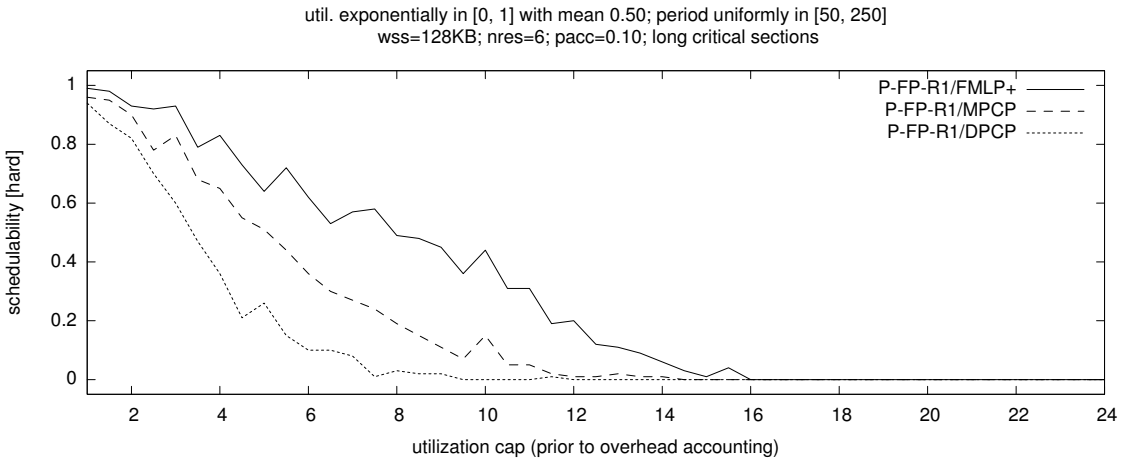
[6] An earlier version of LITMUS$^{\mathrm{RT}}$ required a separate process to act as a task's agent (Brandenburg and Anderson, 2008a,b). The current DPCP implementation is simpler to use since it implements LITMUS$^{\mathrm{RT}}$'s regular locking API.



(a) HRT schedulability for short critical sections.



(b) HRT schedulability for intermediate critical sections.



(c) HRT schedulability for long critical sections.

Figure 7.22: Graphs showing HRT schedulability under P-FP-R1 scheduling with the FMLP$^+$, the MPCP, and the DPCP for heavy exponential utilizations, long periods, $n_r = 6$, $pacc = 0.1$, and **(a)** short, **(b)** intermediate, and **(c)** long critical sections.

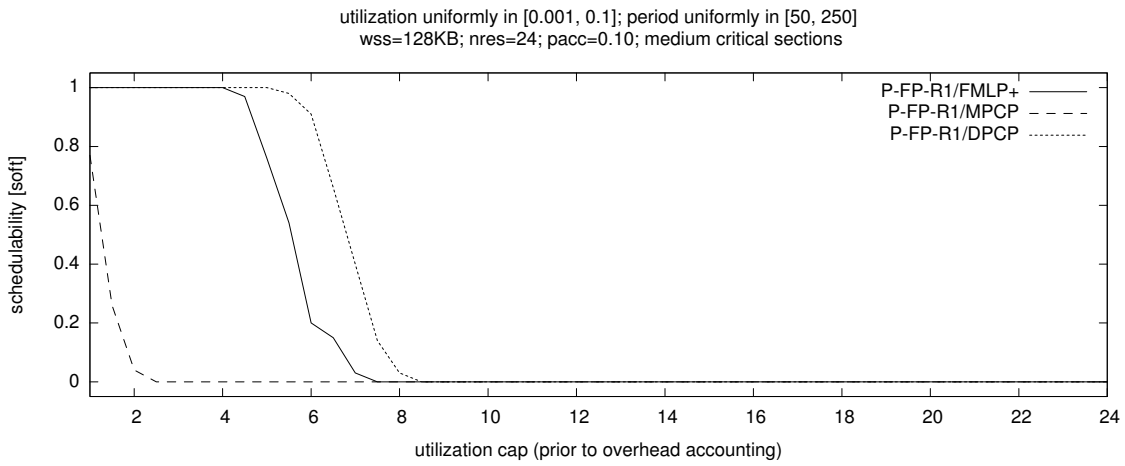


Figure 7.23: SRT schedulability under P-FP-R1 scheduling with the preemptive FMLP$^+$, MPCP, and DPCP for light uniform utilizations, long periods, $n_r = 24$, $pacc = 0.10$ and intermediate critical section lengths.

it remains competitive even if all overheads are assumed to be negligible (*i.e.*, set to zero). This emphasizes that while the FMLP$^+$'s simplicity is certainly beneficial in practice, its optimality with regard to maximum s-aware pi-blocking is equally important to achieving good performance in a practical system (in the tested configuration). The DPCP's performance also improves markedly if overheads are assumed to be negligible; this suggests that it may be worthwhile to study distributed locking approaches in systems with low inter-processor communication costs.

7.6 Spinning vs. S-Oblivious Analysis vs. S-Aware Analysis

In the final comparison, we contrast the performance of spin-based and suspension-based multiprocessor locking protocols (both s-aware and s-oblivious) in terms of HRT and SRT schedulability to answer Questions Q1 and Q2 from Chapter 1: which combination of scheduler and locking protocol is best suited to satisfying HRT and SRT constraints on our platform?

As discussed at length in Chapters 5 and 6, spin-based protocols can result in some processor capacity being wasted while jobs spin. Intuitively, this suggests that suspension-based locking protocols should be preferable. However, suspensions cary two penalties: first, jobs potentially lose cache affinity whenever they issue a resource request since they might suspend, and, second, priority inheritance or priority boosting is required. The former requires potentially pessimistic overhead

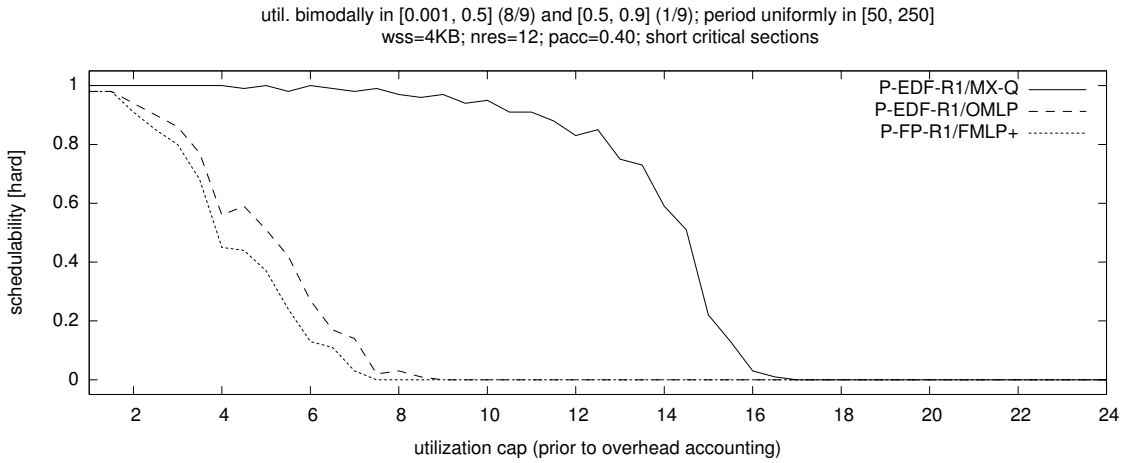
accounting, and the latter implies that real-time jobs must invoke system calls to notify the scheduler of priority changes and to enact suspensions. In particular, the lock and unlock system calls for semaphore protocols must acquire scheduler locks when changing priorities, which can result in significant delays in global schedulers where lock contention can be high.

As implemented in LITMUS$^{\mathrm{RT}}$, spin-based protocols do not issue any system calls in the common case (*i.e.*, if no higher-priority job was released during the critical section) due to the use of the control page to indicate non-preemptivity (see Section 3.3.2). Even in the worst case, that is, when a delayed preemption is required, only a single system call, sched_yield(), is required. In contrast, jobs issue two system calls for each critical section under suspension-based locking protocols.

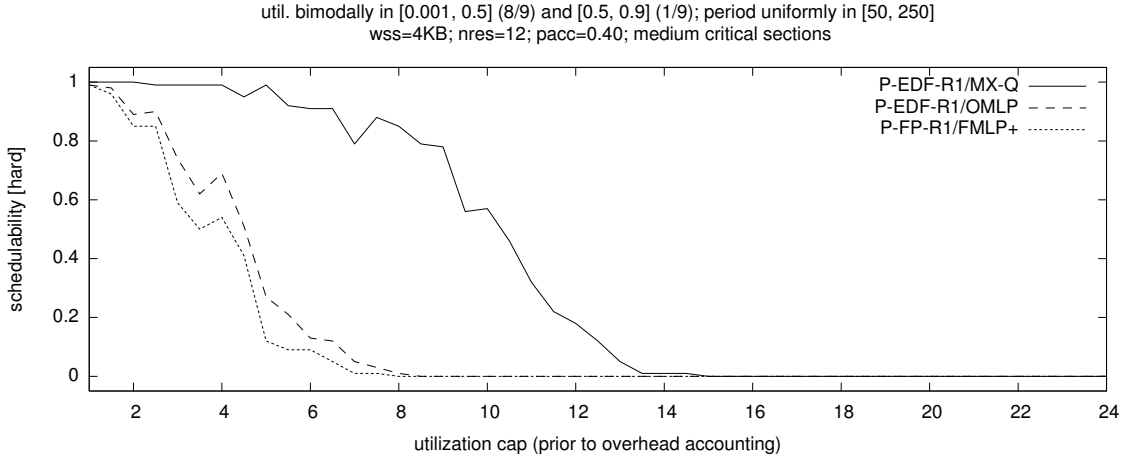
For these reasons, suspension-based locking protocols incur generally much higher overheads than the vastly simpler spin-based locking protocols. Further, in the case of s-aware schedulability analysis, spinlocks additionally avoid all pessimism related to the analysis of suspensions. Consequently we found spinlocks to be always preferable to semaphores (in terms of schedulability) in the HRT case, where high worst-case overheads must be considered.

A representative example can be seen in Figure 7.24, which depicts HRT schedulability under P-EDF-R1 scheduling with the clustered OMLP, under P-FP scheduling with the partitioned FMLP^{+}, and under P-EDF-R1 scheduling with non-preemptive mutex spinlocks (*i.e.*, under the best-performing s-oblivious, s-aware, and spin-based protocol, respectively) for light bimodal utilizations, long periods, $pacc = 0.4$, $n_r = 12$. Inset (a) exhibits the case of short critical sections, where spinlocks outperform either semaphore protocol by a large margin. Since the blocking analysis of the clustered OMLP and non-preemptive task-fair mutex spinlocks is structurally identical, the difference in achieved HRT schedulability is entirely caused by higher overheads under the suspension-based OMLP. Note that Figure 7.24 depicts results assuming a small WSS of only 4 KB, that is, it is biased *in favor* of suspension-based locking protocols. If larger, more realistic WSSs are assumed, then the gap between spinlocks and semaphores increases.

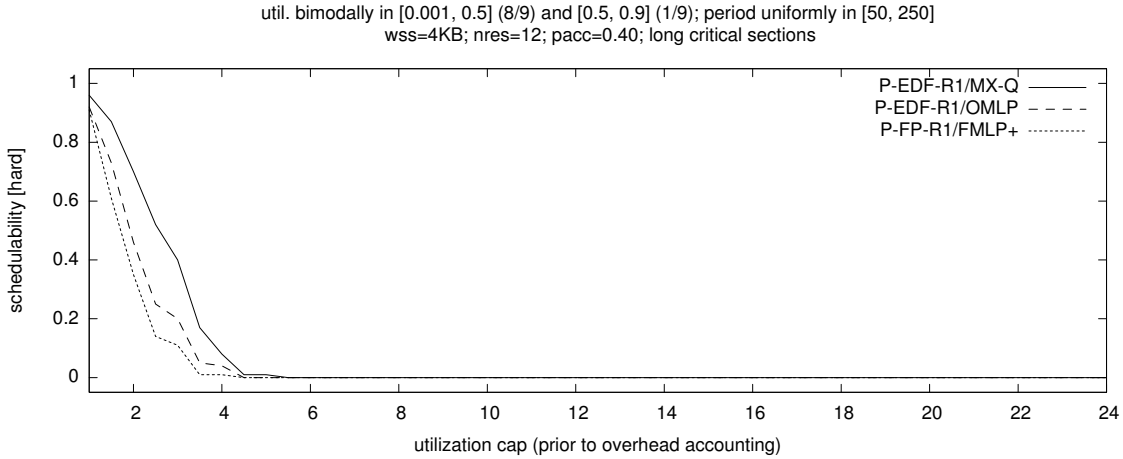
Another interesting trend is that the clustered OMLP achieves higher schedulability than the partitioned FMLP^{+} in this case. This highlights that protocols that take advantage of s-oblivious analysis are in fact competitive with (or even superior to) locking protocols for the best currently available s-aware schedulability analysis, namely response-time analysis for FP scheduling. In other words, while s-oblivious analysis appears very pessimistic on first sight, it can yield overall higher



(a) HRT schedulability for short critical sections.



(b) HRT schedulability for intermediate critical sections.



(c) HRT schedulability for long critical sections.

Figure 7.24: HRT schedulability under P-EDF-R1 scheduling with non-preemptive task-fair mutex spinlocks, under P-EDF-R1 scheduling with the clustered OMLP, and under P-FP-R1 scheduling with the FMLP$^+$ for light bimodal utilizations, long periods, $n_r = 12$, $pacc = 0.4$.

schedulability because it allows pessimism related to the analysis of suspensions to be avoided (since suspensions are effectively analyzed as execution time). Graphs comparing the clustered OMLP to the DPCP and MPCP are available online (Brandenburg, 2011) and similarly reflect that using the s-oblivious approach is beneficial for many of the considered parameter combinations.

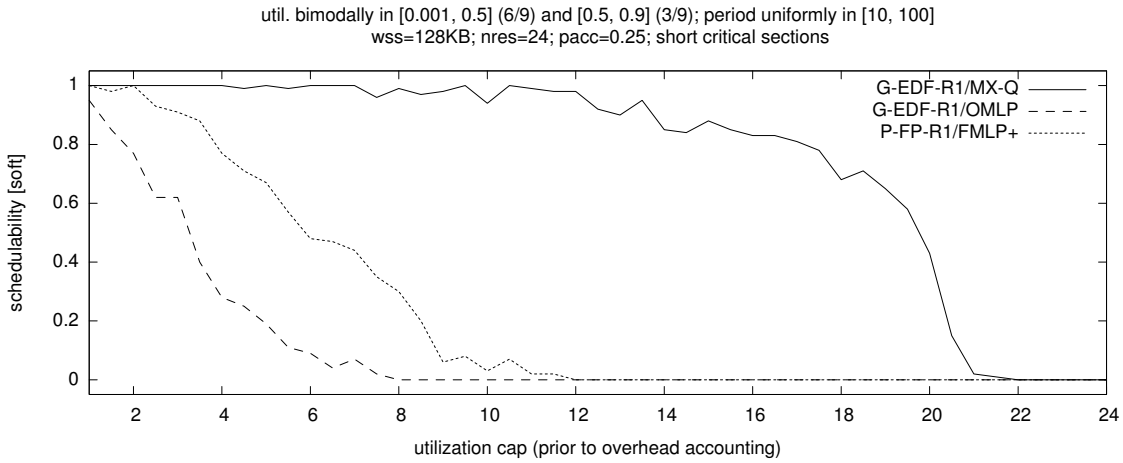
Insets (b) and (c) of Figure 7.24 show that similar trends manifest for intermediate and even long critical sections; however, schedulability is typically uniformly low under all locking protocols if critical sections are long. Notably, suspension-based locking protocols do *not* outperform spin-based locking protocols in the HRT case even if critical sections are long.

We found spinlocks to also be preferable in the SRT case in all tested scenarios if critical section lengths are short, in most of the tested scenarios if critical section lengths are intermediate. While the difference in average-case overheads of spin-based protocols and suspension-based protocols is much smaller than the difference in maximum overheads, semaphores are still subject to additional overheads that do not affect spinlocks (foremost CPMD, additional scheduling decisions, and system calls). Therefore, from an SRT schedulability point of view, spinlocks are in most cases preferable (on our platform).

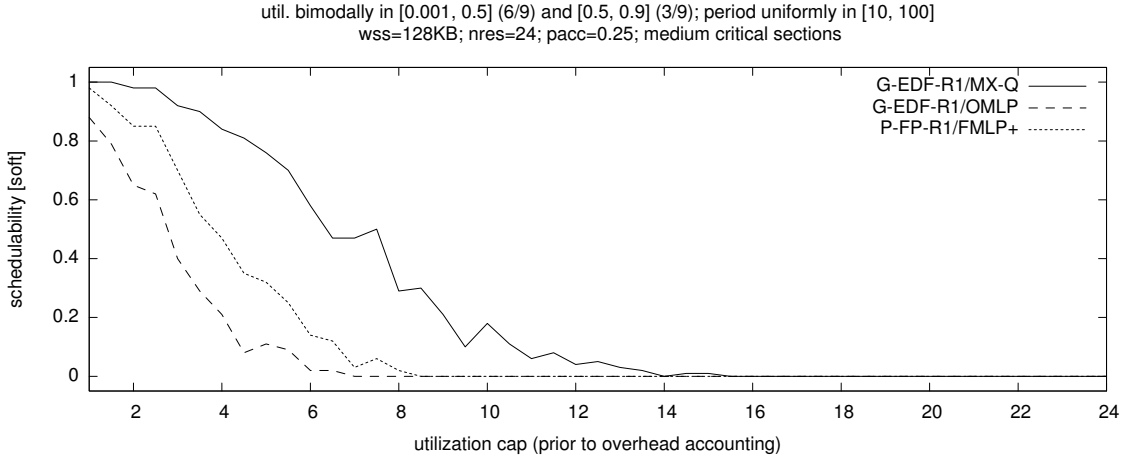
This is apparent in Figure 7.25, which depicts SRT schedulability under G-EDF-R1 scheduling with the clustered OMLP, under G-EDF-R1 scheduling with non-preemptive task-fair mutex spinlocks, and under P-FP-R1 scheduling with the partitioned FMLP$^+$ for medium bimodal utilizations, moderate periods, $pacc = 0.25$, $n_r = 24$, and a WSS of 128 KB.

Inset (a) shows SRT schedulability in the case of short critical sections. As discussed, spinlocks achieve much higher schedulability than either suspension-based protocol due to their overhead advantage. Further, the s-aware FMLP$^+$ outperforms the s-oblivious clustered OMLP; however, this is due to the higher scheduling overheads under G-EDF-R1 scheduling compared to the more efficient P-FP plugin. Inset (b) shows SRT schedulablity in the case of intermediate critical section lengths and exhibits similar, albeit compressed, trends. Inset (c) demonstrates that excessively long critical sections result in low schedulability regardless of the employed locking protocol.

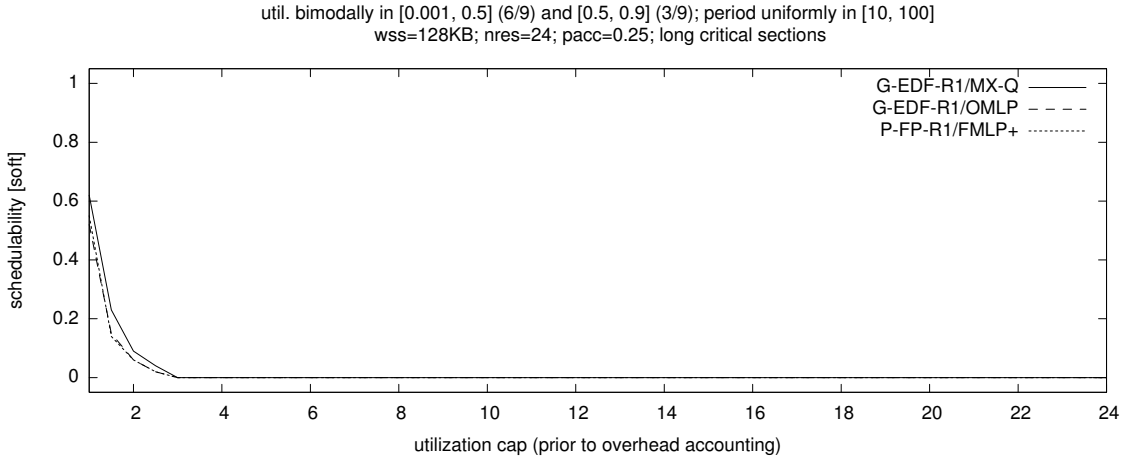
While we found spinning to always be preferable (in terms of SRT schedulability) if critical sections are short, we observed somewhat less uniform trends in the SRT case if critical section lengths are intermediate or long. In fact, there exist parameter choices for which the FMLP$^+$ achieves higher SRT schedulability. The case with the largest gap in SRT schedulability is depdicted



(a) SRT schedulability for short critical sections.



(b) SRT schedulability for intermediate critical sections.



(c) SRT schedulability for long critical sections.

Figure 7.25: SRT schedulability under G-EDF-R1 scheduling with non-preemptive task-fair mutex spinlocks, under G-EDF-R1 scheduling with the clustered OMLP, and under P-FP-R1 scheduling with the FMLP$^+$ for medium bimodal utilizations, moderate periods, $n_r = 24$, $pacc = 0.25$.

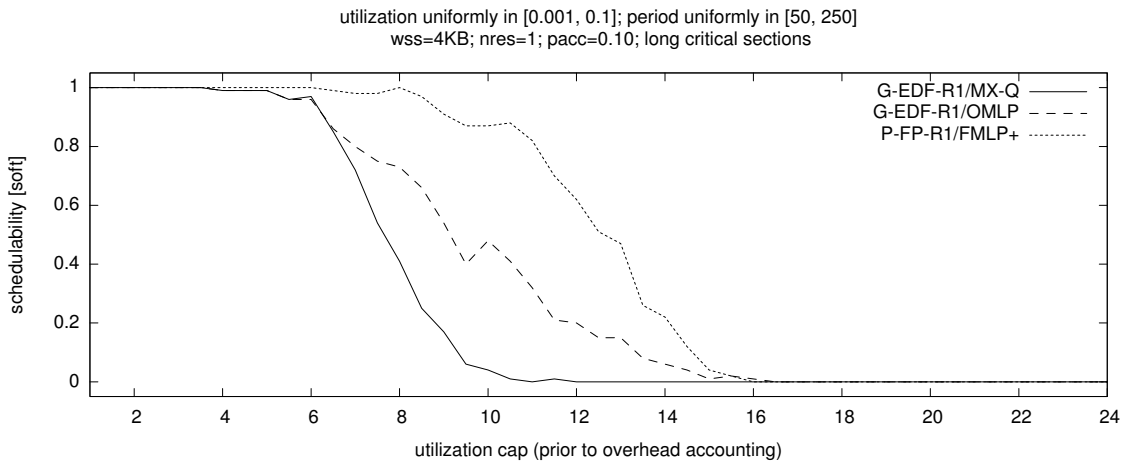


Figure 7.26: SRT schedulability under G-EDF-R1 scheduling with non-preemptive task-fair mutex spinlocks, under G-EDF-R1 scheduling with the clustered OMLP, and under P-FP-R1 scheduling with the FMLP$^+$ for uniform light utilizations, long periods, $n_r = 1$, $pacc = 0.10$.

in Figure 7.26, which shows SRT schedulability under G-EDF-R1 scheduling with the clustered OMLP, under G-EDF-R1 scheduling with non-preemptive task-fair mutex spinlocks, and under P-FP-R1 scheduling with the partitioned FMLP$^+$ for uniform light utilizations, long periods, $pacc = 0.10$, $n_r = 1$, and a WSS of 4 KB. Due to the light utilization distribution, many tasks that share a single resource. Under spinlocks and s-oblivious analysis, this results in significant utilization loss. While the FMLP$^+$ is also affected by large utilization loss, it is less so because suspended tasks do not contribute to processor demand under s-aware analysis. As a result, SRT schedulability is higher under P-FP-R1 scheduling with the FMLP$^+$. It should be noted, however, that, in most cases, SRT schedulability is uniformly low under all tested locking protocols if critical sections are long. The scenario depicted in Figure 7.26 is an unusual exception since $pacc = 0.10$ and $n_r = 1$ (*i.e.*, there is only one shared resource that is, on average, accessed by only ten percent of the tasks).

7.7 Summary

This concludes our overhead-aware evaluation of the real-time multiprocessor locking protocols considered in this dissertation. To enable this evaluation, we have discussed methods that allow locking overheads to be considered during schedulability analysis. Due to the complex interactions that arise when overheads are non-negligible, the presented overhead accounting methods are non-

exhaustive. We have further determined using micro-benchmarks that list-based queue locks offer the best scalability (*i.e.*, lowest overheads under high contention) on our hardware platform.

In the main part of this chapter, we have reported upon two schedulability studies that compared the locking protocols proposed in this dissertation with each other and with three protocols proposed in prior work. Generally speaking, we found that RW locks offer significant analytical benefits if reads are more frequent than writes, and that spinlocks are preferable to suspension-based locking protocols if critical sections are short (as they can be expected to be in well-designed systems).

CHAPTER 8

CONCLUSION

The main objectives of the research presented in this dissertation were to determine how sporadic real-time workloads should be scheduled to efficiently utilize current multiprocessors (Question Q1 in Chapter 1), and which locking primitives should be used to coordinate access to shared resources without exposing jobs to unpredictable delays (Question Q2). To answer these questions, we have built LITMUS$^{\mathrm{RT}}$, devised an overhead-aware evaluation methodology, designed several new real-time locking protocols, and conducted two large case studies on a 24-core Intel Xeon platform. In the following, we first summarize our results (Section 8.1), then discuss open questions and future work (Section 8.2), and finally conclude (Section 8.3).

8.1 Summary of Results

Our research makes novel contributions in three areas, namely RTOS design and implementation, the design and analysis of real-time locking protocols, and the empirical performance evaluation of real-time systems. In the following, we briefly recapitulate the key points of Chapters 3–7.

8.1.1 Theory, Practice, and Overheads

The design and implementation of LITMUS$^{\mathrm{RT}}$ is described in detail in Chapter 3. The purpose of LITMUS$^{\mathrm{RT}}$ is to facilitate the evaluation of real-time scheduling algorithms and locking protocols on actual hardware platforms, so that realistic overheads can be observed and taken into account. Towards this, LITMUS$^{\mathrm{RT}}$ extends the Linux kernel's scheduling architecture with a flexible plugin framework that allows the modular development of new scheduling policies. Notably, it is possible to switch the active scheduling policy at runtime.

In the design of LITMUS$^{\mathrm{RT}}$, we have chosen to realize the sporadic task model following the reservation interpretation (see Section 3.1), which allows tasks to be implemented either as processes (with address space protection) or as threads (without address space protection). An important difference between Linux's stock scheduler and LITMUS$^{\mathrm{RT}}$ is the way in which global scheduling is implemented. In particular, the Linux scheduler is not necessarily correct in all cases, as explained in Section 3.2. LITMUS$^{\mathrm{RT}}$'s link-based implementation of global scheduling in the G-EDF and C-EDF plugins is, to the best of our knowledge, correct for all job arrival sequences (see Section 3.3.3) , albeit at the expense of serializing all scheduling decisions and thus increased overheads. As discussed in Section 8.2.3 below, this is an area for future improvements.

LITMUS$^{\mathrm{RT}}$ has been essential to our research in two important ways. First, by implementing schedulers in a real OS such as Linux, the constraints imposed by the kernel serve as a valuable "reality check" that may be lacking when using simulators or a "toy OS" instead. Second, without the flexibility offered by LITMUS$^{\mathrm{RT}}$, the scope of our evaluation would necessarily have been much more constrained—in our experience, implementing a new scheduler or locking protocol within the LITMUS$^{\mathrm{RT}}$ framework is significantly simpler than directly modifying stock Linux.

Another key feature of LITMUS$^{\mathrm{RT}}$ is that most relevant kernel overheads (scheduler invocation, context switch, release ISR execution, *etc.*) can be measured individually. In accordance with the *principle of least measurement* (Section 3.1), this allows the combined effect of all overhead sources to be determined analytically. The overhead accounting approach used in this dissertation is based on the derivation of *safe approximations* of task sets, as explained in great detail in Sections 3.4 and 3.5. This approach enables the application of overhead-unaware schedulability tests.

8.1.2 Overhead-Aware Evaluation of Real-Time Schedulers

Chapter 4 consists of two parts. In Section 4.1, a novel methodology for conducting overhead-aware schedulability experiments is proposed—this is our approach to answering Question Q1 from Chapter 1. Thereafter, starting in Section 4.2, the applicability of the proposed evaluation method is demonstrated with a large-scale comparison of 22 scheduler configurations on a 24-core Intel Xeon platform. This study answers Question Q1 for our particular platform, and also suggests general trends that would likely arise on other platforms as well.

The defining characteristic of our evaluation methodology is that it incorporates both runtime overheads and schedulability tests. In the OS phase of the evaluation, models of average-case and worst-case overheads under each of the schedulers under evaluation are constructed from large numbers of overhead samples that are collected during the execution of synthetic benchmark tasks. The overhead models are then integrated with overhead-unaware schedulability tests from prior work, as described in Sections 3.4 and 3.5. We used observed maximum overheads when applying HRT schedulability tests and observed average-case overheads when applying SRT schedulability tests, which reflects that SRT applications are more tolerant to occasional delays as long as tardiness remains bounded. Finally, millions of task sets covering a wide range of task parameter choices (*i.e.*, execution cost, period, and WSS) are generated and tested for HRT or SRT schedulability under each of the schedulers under evaluation. The fraction of task sets that are found to be schedulable under each scheduler is an empirical performance measure that reflects both algorithmic and overhead-related capacity loss. We applied this methodology to LITMUS$^{\mathrm{RT}}$ on our 24-core Intel Xeon test platform, which is described in detail in Chapter 2. The major observations are as follows.

First, as expected, we observed lower average-case and worst-case kernel overheads under partitioned scheduling than under global scheduling (see Section 4.3). In the case of clustered scheduling, overheads increase with increasing cluster sizes. Further, while we found staggered quanta to reduce average-case overheads by a large amount, staggered quanta did not consistently reduce worst-case overheads.

Second, in terms of CPMD incurred by a job continuing execution after a preemption or migration, we found that migrations and preemptions cause similar delays if there is strong cache contention (which we induced with a cache-thrashing background workload in our experiments). We also observed that CPMD is markedly lower and that there are benefits to maintaining cache affinity if there is little contention (*i.e.*, if the system is mostly idle). This shows that migrations are not necessarily more costly than preemptions when making worst-case assumptions, but that cache affinity is beneficial in the average case.

Third, P-EDF is generally the best-performing HRT scheduler on our platform, closely followed by P-FP. Despite the bin-packing limitations of partitioned scheduling, P-EDF consistently outperforms both PD^2 (which is handicapped by high overheads) and G-EDF (which suffers from both high overheads and algorithmic limitations). Overheads are markedly lower in smaller clusters, but

C-EDF is still limited by the non-optimality of EDF-based schedulers on multiprocessors (in the HRT case).

Fourth, in the SRT case, P-EDF is the best-performing scheduler only in scenarios in which task sets are easy to partition and overheads have a significant impact (*i.e.*, if there are many low-utilization tasks). Otherwise, G-EDF and C-EDF variants with large clusters are effective at overcoming bin-packing limitations and hence outperform P-EDF.

Fifth, interrupt accounting is a significant source of capacity loss. In particular, in the case of larger clusters and under global scheduling, we found dedicated interrupt handling to be almost always preferable to global interrupt handling. Global interrupt handling is only preferable under P-FP (in all tested scenarios) and P-EDF (in about half of the scenarios), where release ISRs are short and per-processor task counts comparatively low.

Based on these results, our answer to Question Q1 from Chapter 1 is as follows: RTOSs should implement a configurable C-EDF scheduling framework to enable P-EDF scheduling ($c = 1$) for HRT workloads and G-EDF scheduling ($c = m$) or C-EDF scheduling ($1 < c < m$) scheduling for SRT applications.

8.1.3 Real-Time Spinlock Protocols

Our work pertaining to the design and analysis of spinlock protocols for multiprocessor real-time systems is presented in Chapter 5. To support the nested acquisition of resources, Section 5.1 introduces group locking, which is a simple mechanism to automatically transform nested, fine-grained resource requests into un-nested, coarse-grained lock acquisitions. Notably, group locking can be used with both spinlock and semaphore protocols. A limitation of group locking is that it requires all nesting to be known *a priori*; however, such knowledge is required anyway for the derivation of safe upper bounds on worst-case blocking.

The central contribution of Chapter 5 is the introduction of phase-fair reader-writer locks. The key property of phase-fairness (see Section 5.2) is that read requests incur only $O(1)$ acquisition delay without the risk of writer starvation. For context, the only other type of lock with $O(1)$ acquisition delay for readers is a reader preference lock, in which the maximum acquisition delay incurred by writers is potentially unbounded. Besides offering analytical advantages, phase-fair RW locks are

also highly practical. Three efficient implementations are presented in Section 5.3: one requiring little memory, one requiring few atomic instructions, and one with constant RMR complexity.

Finally, improved blocking analysis for task-fair mutex, task-fair RW, and phase-fair RW spinlocks is presented in Section 5.4. In the case of task-fair mutex spinlocks, the presented bounds improve upon prior work since they are based on *holistic blocking analysis* (Section 5.4.1), which takes both the protocol's constraints and the frequency in which individual tasks issue requests into account (prior bounds reflected protocol properties, but not task-set-specific limits). In the case of task-fair and phase-fair RW locks, we have presented the first (published) analysis of RW locks in multiprocessor real-time systems.

8.1.4 Real-Time Semaphore Protocols

Chapter 6 presents our work concerning suspension-based locking protocols. We have made three major contributions. First, we have studied the notion of "blocking optimality" and presented the first definition of a complexity measure for multiprocessor real-time locking protocols, namely maximum pi-blocking. Importantly, we have found that there are two fundamental classes of schedulability analysis that yield different lower bounds on maximum pi-blocking. Under s-oblivious analysis, suspensions cannot be modeled explicitly and suspending jobs are analyzed as if they were executing, that is, processor demand is generally overestimated. In contrast, s-aware schedulability analysis considers suspensions explicitly. While s-aware analysis is preferable in principle, most existing multiprocessor schedulability tests are s-oblivious since the analysis of suspensions is notoriously difficult. Under s-oblivious analysis, $\Omega(m)$ maximum pi-blocking is unavoidable in the general case; under s-aware analysis, the lower bound on maximum pi-blocking is $\Omega(n)$.[1]

The second major contribution is the design and analysis of the OMLP family of protocols (see Section 6.2). The global OMLP (for mutex constraints) ensures $O(m)$ maximum s-oblivious pi-blocking under global JLFP schedulers and is hence asymptotically optimal under s-oblivious analysis. Similarly, the clustered OMLP for mutex constraints ensures $O(m)$ maximum s-oblivious pi-blocking for clustered scheduling with $1 \leq c \leq m$. Further, we also developed two variants of the clustered OMLP for relaxed-exclusion synchronization: the clustered OMLP for RW constraints

[1] Recall that m denotes the number of processors and that n denotes the number of real-time tasks.

implements phase-fairness and thus ensures $O(1)$ maximum s-oblivious pi-blocking per read request (although readers may still incur $O(m)$ s-oblivious pi-blocking due to priority donation—see Section 6.2.6); and the clustered OMLP for k-exclusion ensures $O(m/k)$ maximum s-oblivious pi-blocking per request. The significance of the OMLP family of protocols derives from the fact that the OMLP's mutex protocols are the first asymptotically optimal locking protocols for s-oblivious analysis, the clustered OMLP is the first suspension-based locking protocol for clustered scheduling with $1 < c < m$, and the RW and k-exclusion variants are the first suspension-based locking protocols of their kind.

The third major contribution is the design and analysis of the FMLP$^+$, which ensures $O(n)$ maximum s-aware pi-blocking under partitioned JLFP scheduling, which implies that it is asymptotically optimal under s-aware analysis. Notably, we proposed the first asymptotically optimal real-time locking protocol for partitioned scheduling (Brandenburg and Anderson, 2010a), despite the fact that the first real-time locking protocols for partitioned scheduling were proposed more than 20 years ago.

Furthermore, two impossibility results are presented in Chapter 6: in Section 6.3.3, we show that it is not possible to design asymptotically optimal locking protocols for s-aware analysis using (only) priority queues; and in Section 6.3.2, we show that it is impossible to design asymptotically optimal locking protocols for s-aware analysis under global JLFP scheduling using either priority boosting or priority inheritance if tasks have arbitrary deadlines or may be tardy.

8.1.5 Overhead-Aware Evaluation of Real-Time Locking Protocols

An empirical evaluation of the locking protocols introduced in Chapters 5 and 6 is presented in Chapter 7. The performance comparison is based on overhead-aware schedulability experiments that were conducted using the overhead-aware approach described in Chapter 4.

Since locking protocols expose jobs to additional overheads, corresponding overhead accounting techniques are required to inflate the effective execution requirements and maximum request lengths. To the best of our knowledge, our analysis of locking-related overheads presented in Section 7.1, though still non-exhaustive (see Section 8.2.4 below), is the first systematic discussion of such overheads in multiprocessor real-time systems.

In Chapter 5, we presented three implementations of phase-fair RW locks that target different efficiency considerations: compact locks for memory-constrained systems, simple ticket locks with

few atomic operations, and list-based queue locks with constant RMR complexity. In Section 7.2, results from simple micro-benchmarks comparing the latter two variants (since our platform has ample memory, compact locks were not evaluated) are presented. Overall, our data shows that, among the locks that offer strong progress guarantees, list-based queue locks offer the best scalability on the tested hardware platform.

The remainder of Chapter 7 reports upon two schedulability studies. In the first study, mutex and RW spinlock choices are compared assuming a wide range of write ratios; in the second study, the clustered OMLP, the global OMLP, and the FMLP$^+$ are compared with each other, with the MPCP-VS, the MPCP, and the DPCP (three semaphore protocols proposed in prior work), and with task-fair mutex spinlocks. The major observations are as follows.

First, the use of RW spinlocks can yield significant analytical advantages over mutex spinlocks if the write ratio is low, and phase-fair RW locks offer advantages over task-fair RW locks if multiple writers access a resource. However, task-fair RW locks have the advantage of degrading to mutex-like performance in the worst case, whereas phase-fair RW locks sometimes yield lower schedulability than task-fair mutex locks if there are both many readers and many writers.

Second, in terms of HRT schedulability for the considered range of parameters, the partitioned FMLP$^+$ performs as well or better than either the MPCP or the DPCP, the two classic multiprocessor real-time locking protocols. This shows that the proposed FMLP$^+$ is not just of theoretic interest, but that it is practical as well.

Third, in the case of $c = 1$, the clustered OMLP for s-oblivious analysis is in many cases competitive with the partitioned FMLP$^+$ for s-aware analysis. Still, the FMLP$^+$ is the overall best-performing s-aware suspension-based locking protocol on our platform. This highlights that the analysis of suspension-based locking protocols is very pessimistic in general. This limitation manifests either as pessimistic analysis assumptions (in the s-oblivious case) or in large bounds on maximum pi-blocking (in the s-aware base). In the future, novel s-aware schedulability analysis should be compared against the simpler s-oblivious approach to quantify any reduction in pessimism.

Fourth, we found spinlocks to be generally preferable to suspension-based locking protocols: in all evaluated scenarios in which meaningful differences in locking protocol performance exist, task-fair mutex spinlocks achieved higher HRT schedulability than either the clustered OMLP or the partitioned FMLP$^+$. This is despite the fact that the clustered OMLP and the partitioned

FMLP$^+$ are both asymptotically optimal with regard to maximum pi-blocking (under s-oblivious and s-aware analysis, respectively) and also, in most cases, empirically better performing than previously-proposed suspension-based locking protocols. Further, on our platform, spinlocks are preferable even when assuming small WSSs such as 4 KB, and of course remain preferable if larger WSSs are assumed. In practice, few applications will have a WSS of only 4 KB since most non-trivial programs consist of more than 4 KB of instructions.

Suspension-based protocols were superior to spin-based protocols only in terms of SRT schedulability, only when assuming average overheads, and only in some scenarios involving predominantly intermediate or long critical section lengths. To reiterate, we believe that it can reasonably be expected that most critical sections are short in well-designed systems. A notable exception may be co-processors such as GPUs that result in inherently long critical sections. For such cases, semaphore protocols are likely the only acceptable option.

To summarize, while spinlocks do result in a waste of processor cycles when jobs busy-wait, suspension-based locking protocols result in a waste of processor cycles due to higher overheads. Further, suspension-based locking protocols give rise to additional pessimism that does not affect spinlocks. While long critical sections may necessitate the use of semaphores instead of spinlocks (if only to make processor capacity available to background tasks while real-time tasks wait), such resource-conserving choices do not result in higher real-time schedulability. In fact, to ensure high schedulability long critical sections must be avoided, regardless of the choice of locking protocol.

Finally, we found the conclusions regarding the relative performance of event-driven schedulers in Chapter 4 to remain valid even if tasks are not independent. Generally speaking, on our platform, the highest HRT schedulability is achieved when using P-EDF with non-preemptive task-fair mutex or phase-fair RW spinlocks. In the SRT case, clustered scheduling with $c > 1$ remains effective at overcoming bin-packing limitations.

Based on these results, our answer to Question Q2 from Chapter 1 is that real-time tasks should use non-preemptive spinlock protocols to coordinate access to shared resources. This concludes the summary of the main results of this dissertation. Next, we look at the road ahead and discuss opportunities for future work.

8.2 Assumptions, Open Questions, and Future Work

As is the case with any implementation-based empirical work, our results are necessarily contingent on the underlying hardware platform and on assumptions concerning the evaluation methodology and tested workloads. In the following, we first discuss how changes in the underlying hardware platform might affect our results (Section 8.2.1), then revisit several key assumptions of our evaluation methodology (Section 8.2.2), and finally discuss open questions and opportunities for future work concerning practical real-time scheduling (Section 8.2.3) and real-time locking (Section 8.2.4).

8.2.1 Hardware Platform

As described in depth in Chapter 2, the hardware platform used in the case studies reported upon in Chapters 4 and 7 is a UMA architecture consisting of 24 identical and relatively fast Intel Xeon cores. Strictly speaking, our performance results apply to only this particular platform, although it is reasonable to assume that many current platforms exhibit similar trends. In fact, as discussed in Section 4.5.4, we have observed trends closely matching those reported in this dissertation in several preceding studies on various UMA platforms with identical processors. Going forward, however, it is likely that platforms with NUMA architectures, heterogeneous processors, or particularly slow cores will become more common. In the following, we briefly conjecture how our work could apply to such platforms.

NUMA. Recall from Section 2.1.1 that, in a NUMA architecture, the cost of a memory reference depends on the distance between the accessing processor to the backing memory. That is, reads and writes to processor-local memory are typically much faster than those to remote memory. This has major implications for scheduling: to minimize the time that processors stall on memory references, a job should always be scheduled "close" to the physical memory that holds its code and data.

This constraint effectively rules out global scheduling: as a job may execute on any processor under global scheduling, it cannot be guaranteed that it executes closely to its code and data. Consequently, very pessimistic execution time requirements would have to be assumed, which renders global scheduling undesirable on NUMA platforms.

In contrast, under partitioned scheduling, memory locality is trivial to support. After partitioning a task set, each task's implementing process is simply allocated from memory local to the processor to which the task has been assigned. However, unless memory is abundant, it is further necessary to consider the available local memory during partitioning. That is, the constraint that no processor may be overloaded must be enforced with regard to both processor and memory utilization. This, of course, increases the difficulty of finding a valid task assignment (Fisher *et al.*, 2005).

In multicore designs, strict partitioning may not always be required, because each memory node is likely local to a number of cores (*e.g.*, in Intel's Nehalem processors, each chip contains one memory controller and two to six processors). Consequently, we expect clustered scheduling to be the best choice for multicore NUMA platforms. Assuming clusters are defined to match the underlying NUMA topology, clustered scheduling satisfies the memory locality constraint while also simplifying the bin-packing problem. In such a setup, the scheduling issues discussed in this dissertation are likely to arise in each NUMA node. Therefore, C-EDF scheduling is likely the best tradeoff in the SRT case, whereas in the HRT case, P-EDF may remain preferable to C-EDF if processor utilization is a more limiting constraint than memory utilization (since C-EDF is affected by algorithmic capacity loss if $c > 1$).

Non-identical processors. As discussed in Section 2.1.1, in a platform comprised of identical processors, all processors have the same speed and capabilities. In the case of uniform heterogeneous processors, all processors have the same capabilities, but differ in execution speed (*e.g.*, this is common when using frequency scaling on x86 platforms). Finally, in the case of unrelated heterogenous multiprocessors, individual cores differ in capabilities. We consider the uniform case first.

If processor speeds are static (though not identical), then partitioned scheduling can be used as with identical multiprocessors. However, when assigning tasks to processors, one must consider each task's processor-specific execution requirement, that is, when applying bin-packing heuristics, an item's "size" is a function of both the item and the "bin." Notably, it is not correct to simply linearly scale the execution requirement of a task by the relative execution speed of a processor. For example, a job requiring 30 ms on 1 GHz does not necessarily complete in 15 ms on a 2 GHz processor. This is because a task's execution requirement is determined not only by the processor speed, but also by memory access latencies, pipeline stalls, and other factors independent of processor speed.

If processor speeds are dynamic (*i.e.*, if speeds may change at runtime, for example to lower energy consumption), then partitioned scheduling is unattractive because it could require re-partitioning the entire task set when individual processors change their speed. More realistically, tasks likely would only be assigned once (or when new tasks are admitted), and a processor could only enter a slower low-energy mode if its local task set remains schedulable. This approach, however, would rule out optimal energy management: if re-partitioning is not an option, then there exist task assignments such that it is impossible for any processor to reduce speed without risking deadline misses even though the task set is feasible on slower processors.

As a result, global scheduling may be attractive on uniform heterogeneous multiprocessors, and more so than on identical multiprocessors. Global schedulers for uniform heterogeneous multiprocessors have been examined in detail in prior work by Funk (2004) and could be easily supported in LITMUS$^{\mathrm{RT}}$ by tracking each processor's current execution speed as part of the processor mapping (recall Section 3.3.3). On large multiprocessor platforms, clustered scheduling most likely is the best tradeoff between implementation overhead and energy-saving concerns.

In the case of unrelated heterogeneous multiprocessors, global scheduling is not an option since jobs that require particular processor capabilities cannot be migrated to processors that lack those capabilities. In fact, a heterogeneous multiprocessor platform may have different instruction sets for different processors (*e.g.*, an embedded system might consist of two fast x86 cores and several slower ARM cores), which makes job migration impossible. However, such platforms are essentially a collection of several smaller multiprocessors (or even uniprocessors) that are scheduled individually. Such systems can be managed with clustered scheduling, whereby each "island of uniformity" is considered to be a cluster. Further, it may also be beneficial to allow different policies in different clusters (*e.g.*, it might be preferable to use C-EDF for fast x86 cores and P-FP for slower ARM cores). In general, our results can be applied to unrelated heterogeneous multiprocessors by considering each kind of processor individually.

Many-core platforms. While the number of cores in the evaluated platform is relatively large when compared to current embedded systems, it is not difficult to envision a future in which "large" commodity platforms consist of several hundred processors with limited (or even no) cache coherency. From an implementation point of view, the partitioned scheduling of 24 cores is no different than

partitioned scheduling of 256 cores. However, finding *optimal* task assignments becomes more difficult as the number of cores grows. In practice, optimal assignments are not necessarily required, but even finding merely *good* task assignments for hundreds of cores will still be a significant challenge when issues such as resource-sharing and blocking are taken into account.

However, partitioning will be required to some extent: obviously, even the best-possible implementation of global scheduling cannot scale to arbitrary core counts. Therefore, clustered scheduling will likely increase in relevance if core counts grow drastically. In future work, it would be interesting to study task assignment heuristics for many-core platforms with variable, non-uniform cluster sizes. That is, if "most" tasks are easy to partition and only a "small" subset of the tasks makes the bin-packing instance difficult to solve, it may be beneficial to use partitioned scheduling for most processors and tasks, and to resort to larger clusters only for the tasks that are difficult to partition.

An interesting side effect of ever-increasing core counts is that cores become "cheap," that is, it will be possible to dedicate several cores to interrupt handling and scheduling without significantly affecting the overall system capacity.

Slow cores. The cores in our x86 platform are relatively fast, which de-emphasizes the impact of scheduling overheads. In systems with much simpler and slower processors, it may be the case that HRT and SRT schedulability are significantly more sensitive to overheads. In platforms where this is the case, it may be beneficial to use schedulers that avoid the need for priority queues with arbitrary priorities (such as EDF-based schedulers). Instead, if core speeds are very low, then P-FP scheduling is likely a good option in the HRT case, and global FIFO scheduling—which ensures bounded tardiness (Leontyev and Anderson, 2007, 2010; Leontyev, 2010)—may be preferable to G-EDF in the SRT case.

To summarize, as platforms grow in size and become less uniform, we expect clustered scheduling to become increasingly attractive, but simpler partitioned or global schedulers may be preferable on small platforms with slow processors. This concludes our conjectures concerning future hardware platforms. Next, we consider opportunities for improvement of our evaluation methodology.

8.2.2 Evaluation Methodology

The key advantage of schedulability experiments, as opposed to other, more direct metrics such as measuring the deadline miss ratio, is that schedulability evaluates an algorithm's ability to provide *guarantees* (instead of characterizing average-case behavior). The primary contribution of our extended evaluation methodology—described in detail in Chapter 4—is that it incorporates realistic overheads as observed on *real hardware*, in an *actual RTOS*. This allows us to evaluate a platform's ability to make worst-case guarantees for the assumed workloads *in practice*. We believe this integration of engineering concerns to be a significant improvement over prior evaluation approaches. Nonetheless, our evaluation methodology could be improved in several ways, which we discuss in the following.

Realistic workloads. A key assumption underlying the experimental setup is the choice of workload that is being assumed. As discussed in Section 4.2.3, we chose synthetic workloads that stress both implementation bottlenecks and algorithmic weaknesses of each tested scheduler. In future work, it would be interesting to augment our approach with a systematic evaluation of actual real-time applications.[2] However, a practical limitation is that it is generally difficult to obtain real-time workloads for multiprocessors, and particularly workloads with HRT constraints. This is due to (at least) three reasons: first, HRT applications are typically developed for a specific hardware platform with specific sensors and actuators, so that it is difficult to faithfully reproduce them in a lab setting; second, embedded systems software is typically not made public and only infrequently shared with researchers as most companies consider such software to be a trade secret; and third, existing workloads often do not yet take full advantage of multiprocessors since the adoption of multicore systems in the embedded systems industry is still at an early stage. Nonetheless, in future schedulability studies, it would still be interesting to use parameter distributions that are derived from actual workloads.

CPMD. As with execution times and periods, several alternatives for obtaining realistic cache-related overheads should be evaluated in future work. Instead of assuming a range WSSs, it would be interesting to profile actual real-time applications to determine typical WSSs of HRT and SRT

[2]For example, we were recently involved in an evaluation of video decoding and playback (Kenna *et al.*, 2011).

workloads. Further, instead of measuring CPMD for each WSS directly as a difference in execution times, as described in Section 4.4, it would also be possible to measure CPMD indirectly by observing cache misses with hardware performance counters, which are available in many modern processors. It would also be interesting to study CPMD on non-x86 embedded systems platforms with smaller and slower caches, and on recent x86 platforms with cache prefetching (the latter is only appropriate for SRT applications). Unfortunately, CPMD experiments are currently very time consuming; better automation is required to study a wider range of platforms.

Independent execution times. Recall from Section 4.1 that schedulers are compared based on the fraction of task sets that can be claimed HRT or SRT schedulable under it. To enable a fair comparison, each scheduler is tested using the *same* task sets. That is, once a task set has been generated by choosing a period and utilization for each task, it is tested whether it is schedulable under each of the evaluated schedulers. Importantly, while different overhead magnitudes (as measured in LITMUS$^{\mathrm{RT}}$) are used for each scheduler, the task parameters prior to inflation for overheads are identical under each scheduler.

We adopted this approach to avoid introducing a bias in task parameter selection, and because prior schedulability studies (Baker, 2005; Calandrino *et al.*, 2006) proceeded similarly. However, it is not clear that a task's worst-case and average execution times are actually identical under all schedulers. In fact, it is rather unlikely. For example, it is conceivable that WCETs are larger under global scheduling than under partitioned scheduling if migrations cause increased memory bus contention. It could, however, also be the case that average execution times are shorter under global scheduling than under partitioned scheduling—if all jobs can be scheduled on any processor, then jobs are preempted less frequently and for shorter times, which benefits cache affinity. In other words, it is not obvious how actual execution requirements under different schedulers relate. Similar concerns affect critical section lengths under various locking protocols (*e.g.*, they could conceivably be shorter under spinlocks than under semaphores) and quantum staggering (*e.g.*, staggering of quantum boundaries might increase memory bus contention throughout the quantum). In future work, it would be worthwhile to investigate if there are meaningful differences in execution time requirements under different scheduling approaches and locking protocols (aside of the overhead sources already accounted for), and if so, whether they affect relative scheduler performance.

Partitioning of malleable tasks. As discussed in Sections 2.3.2 and 2.3.4, clustered scheduling requires that a given set of tasks is assigned to the available clusters such that (i) no cluster is overloaded and (ii) each task is assigned to exactly one cluster. As discussed at length in Chapter 4, this can be a limitation, and particularly so under partitioned scheduling. Of the two constraints, Constraint (i) is fundamental, but it may be possible to violate Constraint (ii) in the design of certain embedded systems.

In *open* real-time systems, that is, when tasks are opaque and become known only at runtime, sporadic tasks cannot be split across multiple clusters since they are sequential. However, many embedded systems are *closed* (or static) in the sense that the workload to be supported is determined (and analyzed) at design time and then remains unchanged during operation. In closed systems, tasks may in fact be malleable and, to a certain extent, divisible across multiple processors or clusters. That is, if an embedded systems engineer is faced with a situation where no valid task assignment can be found, then the engineer may be able to split out some of the functionality of a difficult-to-assign, high-utilization task into several lower-utilization tasks for which a valid assignment can be found. Bin-packing limitations may therefore be less severe in the design of embedded systems with static workloads than assumed in this dissertation. Note, however, that splitting tasks may *increase* the total utilization due to added communication costs and duplicated functionality. In future work, it may be interesting to study bin-packing heuristics that, if no assignment can be found, support splitting a task T_i into two tasks $T_{i,1}$ and $T_{i,2}$, where $u_i < u_{i,1} + u_{i,2}$.

Avoiding measurements. The *principle of least measurement* (see Section 3.1) is of great importance to the future development of truly predictable multiprocessor RTOSs, with profound implications for RTOS design and implementation. Ideally, once timing analysis tools are sufficiently mature to handle multicore platforms, worst-case kernel overheads (*e.g.*, scheduler invocation, context switch, *etc.*) should be determined analytically. However, for the foreseeable future, this will likely not be possible in complex kernels such as Linux. Instead, it would be beneficial to develop (or extend existing) microkernels of much simpler design with LITMUS$^{\mathrm{RT}}$-like functionality. Still, even with the complexity of Linux removed, significant changes will be required to the way multiprocessor schedulers are implemented to make the code compatible with WCET analysis (*e.g.*, loops and priority queue lengths must have constant bounds).

Another promising substitution of measurements with analytical bounds that is possible even if WCET analysis is unavailable concerns lock contention within the kernel. In LITMUS$^{\mathrm{RT}}$, scheduler lock contention is currently measured indirectly and reveals itself as high worst-case overheads in global schedulers (and also in clustered schedulers with large clusters). However, in principle, there is little reason as to why worst-case lock contention should be *measured* when effective blocking analysis for spinlocks is available (see Chapter 5). Instead, critical section lengths should be measured individually and worst-case lock acquisition times should be *derived analytically*. In practice, however, measuring each critical section individually would lead to an unmanageably large number of samples; novel tracing techniques will thus have to be developed and integrated with LITMUS$^{\mathrm{RT}}$ (or other kernels) to facilitate such analysis.

As the preceding discussion shows, there exist numerous ways in which our evaluation methodology could be extended to reflect real-world concerns to an even larger degree. In particular, it would be interesting to refine our evaluation methodology in future work by incorporating more detailed models of overhead sources and task execution, by studying more realistic workloads, and by further emphasizing analytical approaches where possible to reduce the dependence on measured overheads. Next, we discuss scheduling-related issues.

8.2.3 Scheduler Design and Implementation

Our results have shown partitioned scheduling to be the best choice for HRT systems, and global scheduling and clustered scheduling with $c > 1$ to be effective at overcoming bin-packing issues in the SRT case. Our observations further raise questions concerning the practicality of pfair scheduling. In the following, we highlight several avenues for future work in the area of global and clustered scheduling and SRT applications, as well as pfair scheduling and periodic HRT applications.

Efficient and correct global scheduling. One benefit of global scheduling not reflected by our evaluation is that it avoids the need to re-partition task sets when tasks join or leave the workload in open real-time systems. That is, global scheduling is self-adjusting in the sense that it avoids the need to perform load balancing, which reduces overheads if workloads change frequently—Block (2008) discusses this aspect of global scheduling in detail. Global scheduling and clustered scheduling with large clusters are hence likely to remain relevant, at least for adaptive SRT applications.

However, our results also show that LITMUS$^{\mathrm{RT}}$'s implementation of global scheduling suffers from high runtime overheads and lock contention. The key problem with the current implementation is that all scheduling events are serialized by a single global lock, which is an obvious bottleneck. This serialization, however, is currently required to ensure that the processor mapping at the heart of link-based scheduling remains consistent (see Section 3.3.3). In contrast, Linux's implementation of global scheduling allows for parallel scheduling decisions and thus incurs lower overheads, but does so at the expense of correctness (recall Section 3.2). In future work, it would be ideal to develop a parallel version of link-based scheduling without contention bottlenecks. Crucially, such parallelized link-based scheduling should match Linux's low runtime overheads without sacrificing correctness.

Predictable deadline miss ratios. This dissertation assumes the bounded tardiness definition (Devi and Anderson, 2005; Devi, 2006; Devi and Anderson, 2008) of "soft" temporal correctness. As elaborated upon in Section 2.2.2, bounded tardiness is a useful property for SRT applications, and especially for those in which tardiness can be masked using input and output buffering—that is, assuming the actual tardiness bounds are reasonably small. In future work, it would thus be beneficial to continue the study of G-EDF and similar schedulers to obtain tighter tardiness bounds.

Going beyond bounded tardiness, however, there are SRT applications that require additional assurances. For example, when processing latency-sensitive live streams (*e.g.*, in video conferencing systems or surveillance applications), excessive buffering could introduce unacceptable delays. Similarly, buffering may not be an option in memory-constrained systems (*e.g.*, large memories consume significant amounts of energy, and should thus be avoided in battery-powered devices). For such applications, a bounded deadline miss ratio (with respect to reasonably short reference intervals) may be more useful than bounded tardiness.

Deadline miss ratios are straightforward to measure, but there currently exists no analysis for global multiprocessor schedulers that allows bounding worst-case deadline miss ratios *a priori*. Indeed, it is trivial to construct G-EDF schedules where all jobs of a task miss their deadline, that is, the worst-case deadline miss ratio under G-EDF is 1 in the general case. Therefore, there are two major issues that future algorithmic work on SRT scheduling on multiprocessors should investigate: first, is it possible to derive non-trivial, per-task deadline miss ratio bounds for G-EDF (for non-pathological task sets); and second, do there exist JLFP schedulers that provably ensure

lower deadline miss ratios than G-EDF? An ideal SRT scheduler would ensure both low tardiness bounds and low deadline miss ratio bounds for arbitrary sporadic workloads without incurring high overheads. Unfortunately, such an algorithm has yet to be developed.

Practical pfair scheduling. One of the major trends discussed in Chapter 4 is that none of the PD^2 variants performed particularly well due to high overheads. Does this imply that there is no practical application for PD^2? This is not necessarily the case. While it is true that PD^2 did not perform well for sporadic, interrupt-driven workloads, PD^2 is well-suited to periodic, polling HRT tasks. In fact, in HRT systems in which all tasks are periodic, PD^2 could be implemented more efficiently, and thus offer much improved HRT schedulability.

A key limitation in LITMUS$^{\mathrm{RT}}$'s implementation of PD^2 is the high cost of scheduling at each quantum boundary, which could be reduced using two techniques. First, as core counts increase, it may be beneficial to dedicate a core to scheduling, which would eliminate the need to compute the job assignment for the next quantum on one of the application processors serving real-time tasks.

Second, the dedicated "scheduling processor" could compute each quantum in advance, that is, during the x^{th} quantum, the scheduling processor could pre-compute the job assignment for the $(x+1)^{\text{th}}$ quantum. For sporadic workloads, this would increase the release delay by Q time units (*i.e.*, by one quantum), but for periodic workloads (where all job releases are pre-determined), such "negative staggering" of the scheduling processor incurs no penalty. The kernel overhead at each quantum boundary could thereby be reduced to a single pointer update and context switch; timer ticks could thus be almost as efficient as timer ticks in event-driven plugins.

Another limitation of PD^2 is that it frequently preempts jobs, which makes it vulnerable to high CPMD costs. As CPMD costs for non-trivial WSSs are indeed relatively high on our platform, the performance of PD^2 is negatively affected. However, in embedded system platforms, caches might be smaller and slower, to the effect that the worst-case effect of preemptions might have a significantly lower impact. Further, in true HRT systems, it may be necessary to disable caches altogether to enabled WCET analysis. In such an environment, PD^2 could be much more competitive.

As briefly discussed in Section 2.1.2, a WCET-friendly alternative to using processor caches is to use scratchpad memories, which can be thought of as OS-controlled caches. PD^2 may be ideally suited to periodic, scratchpad-based HRT workloads: since the schedule can be computed one or

several quanta in advance, it should be possible to prefetch scratchpad contents needed in subsequent quanta in advance. For example, the scratchpad memory could be split in half such that one half is used by the currently scheduled subtask, whereas the other half is used to prefetch the contents required by the *next* subtask. At each quantum boundary, the roles of the two scratchpad partitions would be flipped (which would require MMU support) such that no subtask is ever delayed by the cost of restoring scratchpad contents. Conceptually, this approach could eliminate CPMD almost entirely and is thus a promising direction for future work.

Overhead accounting vs. budget enforcement. A cornerstone of predictable real-time system design is that overheads must be accounted for prior to applying a schedulability test. We have presented such overhead analysis in Sections 3.4, 3.5, and 7.1. Based on the concept of *safe approximations*, overheads are accounted for by inflating the execution costs of tasks to account for all relevant overheads. The rationale behind safe approximations is that the inflated task set is at least as difficult to schedule in the absence of overheads as the actual task set in the presence of overheads. Importantly, each task is inflated both for overhead that it incurs itself (*e.g.*, system calls) and for overheads that it inflicts upon others (*e.g.*, CPMD).

Budget enforcement is required to guarantee temporal isolation: if a job misbehaves and fails to complete, budget enforcement ensures that the processor allocation of other jobs is not affected—budget overruns are prevented altoghether. Unfortunately, inflation-based overhead accounting conflicts with budget enforcement. For example, consider how CPMD is accounted for under event-driven scheduling, and let J_h denote a job that preempts a lower-priority job J_l. As described in Section 3.4.3, J_h's worst-case execution cost is inflated by the worst-case magnitude of CPMD to account for the slowdown in J_l's execution rate. However, if the resulting inflated budgets are enforced, then J_h would simply underutilize its budget (since it is not actually affected by CPMD itself), whereas J_l might exceed its budget due to the CPMD—which is reflected in J_h's budget, and not J_l's. In other words, the safe approximation of CPMD is based on an "accounting trick" that is unknown to the RTOS at runtime. Ultimately, budget enforcement could be triggered by the (small) budget overruns anticipated and accounted for by overhead analysis even when it is not required, thereby causing more harm than good.

There are three ways in which this could be addressed. First, if each task's budget is inflated only directly, that is, if each e_i' reflects *all* overheads incurred by J_i, then budget enforcement would work as expected. However, this would result in much more pessimistic accounting of CPMD and is thus undesirable. Second, budget enforcement could simply be disabled, which is undesirable from the points of view of temporal isolation and reliability. Third, the budget enforcement mechanism could be extended to take overhead accounting into account. That is, the RTOS would have to track each job's budget and *transfer* "execution time credit" from the preempting job's budget to the preempted job's budget. In general, budget transfers would be required whenever a job causes another to (indirectly) incur overheads. To the best of our knowledge, such a budget transfer mechanism has not been implemented in any RTOS to date. In future work, it will be necessary to develop such an overhead-aware budget tracking framework—and supporting analysis—to integrate temporal isolation with overhead accounting.

In summary, there are multiple avenues for future work in the area of practical multiprocessor real-time scheduling, including implementation improvements, analysis improvements, and the design of SRT schedulers that offer additional guarantees beyond bounded tardiness.

8.2.4 Future Directions in Real-Time Locking

Last but not least, our work has exposed a number of fundamental questions concerning blocking analysis, locking optimality, and overhead accounting that remain open.

Nested critical sections. Foremost, there currently exists no analysis of multiprocessor locking protocols that allows critical sections to be nested (*i.e.*, jobs already holding a resource may not request another resource). In particular, none of the protocols proposed to date prevents deadlock (without resorting to coarse-grained group locking). Further, even if all nesting is assumed to be well-ordered (which precludes deadlock), complex blocking dependencies arise that cannot be handled with existing analysis. In contrast, the PCP and SRP (recall Section 2.4.3) prevent deadlock and ensure provably optimal blocking even if critical sections are nested. Without doubt, fine-grained locking and nesting of critical sections occurs in real applications; further research is thus required to enable *predictable* locking for such applications on multiprocessors as well.

Nested critical sections also raise interesting questions with regard to locking optimality. It is easy to construct examples that show that the lower bounds established in Chapter 6 are not tight, that is, there exist task sets for which maximum pi-blocking worse than $O(m)$ (in the s-oblivious case) and $O(n)$ (in the s-aware case) seems unavoidable. In future work, it will be interesting to determine lower bounds on maximum pi-blocking, possibly as a function of the nesting degree.

Distributed real-time locking protocols. To our surprise, in the schedulability study reported upon in Chapter 7, we found the DPCP to perform much better than the MPCP in some (but not all) scenarios. This is despite the fact that the DPCP incurs higher overheads due to its reliance on the RPC model, and despite the fact that the MPCP's analysis was recently improved (Lakshmanan *et al.*, 2009). The strong performance of the DPCP warrants a closer inspection of distributed locking protocols in future work. For one, large many-core platforms are unlikely to provide a cache-consistent shared memory for all cores, which introduces aspects of distributed systems. But the success of the DPCP shows that, even in shared-memory systems, it may be worthwhile to adopt a more distributed protocol structure.

We plan to investigate three issues related to distributed locking protocols in future work. First, we seek to identify how the FMLP$^+$ can be applied to a distributed environment. The DPCP is not asymptotically optimal (it implicitly uses priority queues to order requests); it should be possible to design an asymptotically optimal protocol using FIFO queues instead. Further, as is the case with the FMLP$^+$ and the MPCP, it may be possible in many cases to achieve better schedulability, too.

Second, the DPCP only applies to P-FP scheduling. In the multicore age, it is likely that distributed real-time systems consist of multiple separate multiprocessors. For such systems, it may be beneficial to use clustered scheduling instead (with $c > 1$ or non-uniform cluster sizes). Therefore, it would be interesting to see if it is possible to modify the clustered OMLP to obtain a distributed locking protocol that is asymptotically optimal under s-oblivious schedulability analysis.

Third, it may be beneficial to adopt techniques from locking protocols with distributed lock state to real-time systems. For example, Hsieh and Weihl (1992) proposed a distributed RW lock in which there is a processor-local "reader lock" on each processor, and one global "writer lock." Readers acquire only their processor-local reader lock, whereas writers are forced to acquire the global "writer lock" and each "reader lock" on each processor (Hsieh and Weihl, 1992). For workloads in

which reads are much more frequent than writes, this approach results in low overheads for readers. However, it is not immediately clear how overly pessimistic worst-case blocking bounds for writers can be avoided in such a design.

Tool-supported analysis. In this dissertation, we have presented the first systematic analysis of locking-related overheads (see Section 7.1). However, as previously noted, the presented analysis is not a truly *safe* approximation since it does not handle rare corner cases, and even if it did, we currently have no method for showing that the worst case scenario has indeed been considered. The former could be addressed with additional analysis, but the latter is a significant roadblock. In a sense, our overhead analysis is necessarily non-exhaustive because the considerable complexity and timing uncertainty that overheads introduce into blocking analysis prevents effective *manual* analysis.

The same problem is also holding back advances in locking protocols that support fine-grained locking and nesting of critical sections. While it is straightforward to define simple protocols for nested locking (*i.e.*, the FMLP^{+} could trivially support well-ordered nesting), it is very difficult to analyze *worst-case* blocking under such protocols. Again, the complexity arises because it is difficult to identify the worst case, and even more challenging to prove that the worst case has indeed been accounted for. With considerable effort, it would likely be possible to derive coarse bounds; however, such analysis would likely be pessimistic due to the need to simplify the problem.

The root problem is that overheads and nesting both give rise to non-trivial transitive blocking, which stresses the limits of manual analysis. Therefore, we believe that future locking protocol analysis should be based on automated tools to derive less-pessimistic, yet sound upper bounds. In the case of nesting, one possible approach could be to formulate the derivation of blocking bounds as an optimization problem: the goal is to maximize a task's blocking term subject to the constraints imposed by the protocol (queue ordering, *etc.*) and the task set (the frequency in which requests are issued, *etc.*). This problem structure lends itself to a formulation as an *integer linear program* (ILP), for which highly optimized solvers exist. In the case of overhead accounting, it may be possible to model request execution using *timed automata* (Alur and Dill, 1994), for which powerful model checkers exist that could be used to derive bounds on maximum effective critical section lengths. However, while tool-based approaches seem promising on first sight, there are certainly details and challenges that must be resolved first.

Non-blocking synchronization. Non-blocking synchronization primitives such as wait-free concurrent queues or the *read-copy update* (RCU) mechanism (McKenney, 2004), which is used extensively in the Linux kernel, had to remain beyond the scope of this dissertation due to time and resource constraints. Nonetheless, there are certainly applications in which the use of non-blocking synchronization can result in improved performance (both in terms of schedulability and throughput).

In future work, our evaluation of locking protocols should therefore be extended to incorporate RCU and other non-blocking primitives. However, to do so, appropriate analysis will have to be developed for each candidate algorithm. For example, when using lock-free algorithms (in which update operations that were interfered with may be retried repeatedly), appropriate retry limits must be obtained analytically. Similarly, RCU requires shared objects that are no longer being referenced to be garbage-collected. The garbage collection task must be accounted for during schedulability analysis, which raises the question of how it should be provisioned. That is, the garbage collector could execute frequently, thereby reducing the maximum memory requirement at the expense of a higher processor utilization, or it could execute infrequently to minimize its processor utilization at the cost of inefficient memory use. We hope to explore this and related questions in future work.

8.3 Closing Remarks

The emergence of multicore platforms holds great promise for embedded real-time systems: in the future, instead of using hundreds of uniprocessors, it should be possible to use fewer, computationally powerful, yet energy-efficient multiprocessors to co-host and consolidate a wide range of demanding HRT and SRT applications. However, given the complexities inherent in parallelism and real-time computing, multiprocessor real-time systems constructed with *ad hoc* "build it first, test and debug later" approaches (which are still common in practice) are bound to be wasteful, unreliable, inflexible, and expensive. Instead, future software engineering methods should increasingly incorporate *a priori* performance modeling, analysis, and optimization—which fundamentally requires *predictable resource allocation* and *strong isolation*, and hence a new kind of multiprocessor RTOS.

In our research, we have closely studied two issues at the very core of such next-generation RTOSs: scheduling and locking. Most importantly, we have shown that algorithmic limitations and overhead concerns both strongly affect the ability to make temporal guarantees in practice. Future

RTOSs should thus be constructed upon *analytically-sound algorithmic foundations*, and future algorithmic-oriented research should be *informed by real-world engineering concerns*. It is our hope that this dissertation will prove useful in both regards.

BIBLIOGRAPHY

Abeni, L. and Buttazzo, G. (1998). Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 4–13.

Abeni, L. and Buttazzo, G. (2004). Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167.

AbsInt Angewandte Informatik GmbH (2011). aiT worst-case execution time analyzers. Product web site, `http://www.absint.com/ait/`.

Agarwal, A., Hennessy, J., and Horowitz, M. (1989). An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215.

Albers, K. and Slomka, F. (2005). Efficient feasibility analysis for real-time systems with EDF scheduling. In *Proceedings of the Design, Automation and Test in Europe Conference on and Exhibition*, pages 492–497.

Alur, R. and Dill, D. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

Anderson, J., Bud, V., and Devi, U. (2005). An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208.

Anderson, J. and Holman, P. (2000). Efficient pure-buffer algorithms for real-time systems. In *Proceedings of the 7th International Conference on Real-Time Systems and Applications*, pages 57–64.

Anderson, J., Jain, R., and Jeffay, K. (1998). Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355.

Anderson, J., Kim, Y., and Herman, T. (2003). Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2-3):75–110.

Anderson, J. and Ramamurthy, S. (1996). A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 92–105.

Anderson, J., Ramamurthy, S., and Jeffay, K. (1995). Real-time computing with lock-free shared objects. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 28–37.

Anderson, J., Ramamurthy, S., and Jeffay, K. (1997). Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, pages 28–37.

Anderson, J. and Srinivasan, A. (2000). Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43.

Anderson, J. and Srinivasan, A. (2004). Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204.

Anderson, T. (1990). The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16.

Andersson, B. (2008). Global static-priority preemptive multiprocessor scheduling with utilization bound 38%. In *Proceedings of the 12th International Conference On Principles Of Distributed Systems*, LNCS 5401, pages 73–88.

Andersson, B., Baruah, S., and Jonsson, J. (2001). Static-priority scheduling on multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 193–202.

Andersson, B. and Easwaran, A. (2010). Provably good multiprocessor scheduling with resource sharing. *Real-Time Systems*, 46(2):153–159.

Andersson, B. and Jonsson, J. (2003). The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 33–40.

Arthur, S., Emde, C., and McGuire, N. (2007). Assessment of the realtime preemption patches (RT-Preempt) and their impact on the general purpose performance of the system. In *Proceedings of the 9th Real-Time Linux Workshop*.

Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A. (1993). Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292.

Audsley, N., Burns, A., Richardson, M., and Wellings, A. (1991). Hard real-time scheduling: the deadline-monotonic approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137.

Aydin, H., Melhem, R., Mosse, D., and Alvarez, P. (2001). Optimal reward-based scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 50(2):111–130.

Baer, J.-L. (2010). *Microprocessor Architecture: from Simple Pipelines to Chip Multiprocessors*. Cambridge University Press.

Baker, T. (1990). A stack-based resource allocation policy for realtime processes. *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 191–200.

Baker, T. (1991). Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99.

Baker, T. (2003). Multiprocessor EDF and deadline monotonic schedulability analysis. *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 120–129.

Baker, T. (2005). A comparison of global and partitioned EDF schedulability tests for multiprocessors. Technical Report TR-051101, Florida State University.

Baker, T. (2010). What to make of multicore processors for reliable real-time systems? In *Proceedings of the 15th Ada-Europe International Conference on Reliable Software Technologies*, LNCS 6106, pages 1–18.

Baker, T. and Baruah, S. (2007). Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Systems*. Chapman Hall/CRC.

Baker, T. and Baruah, S. (2009a). An analysis of global EDF schedulability for arbitrary-deadline sporadic task systems. *Real-Time Systems*, 43:3–24.

Baker, T. and Baruah, S. (2009b). Sustainable multiprocessor scheduling of sporadic task systems. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 141–150.

Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P. (2002). Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, pages 73–78.

Barabanov, M. (1997). A linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology.

Barre, J., Rochange, C., and Sainrat, P. (2008). A predictable simultaneous multithreading scheme for hard real-time. In *Proceedings of the 21st International Conference on Architecture of Computing Systems*, LNCS 4934, pages 161–172.

Baruah, S. (2006). Resource sharing in EDF-scheduled systems: A closer look. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 379–387.

Baruah, S. (2007). Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119–128.

Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., and Stiller, S. (2010). Improved multiprocessor global schedulability analysis. *Real-Time Systems*, 46(1):3–24.

Baruah, S. and Burns, A. (2006). Sustainable scheduling analysis. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 159–168.

Baruah, S., Cohen, N., Plaxton, C., and Varvel, D. (1996). Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625.

Baruah, S., Gehrke, J., and Plaxton, C. (1995). Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288.

Baruah, S., Goossens, J., and Lipari, G. (2002). Implementing constant-bandwidth servers upon multiprocessor platforms. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 154–163.

Baruah, S. and Lipari, G. (2004a). Executing aperiodic jobs in a multiprocessor constant-bandwidth server implementation. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 109–116.

Baruah, S. and Lipari, G. (2004b). A multiprocessor implementation of the total bandwidth server. In *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium*, pages 40–49.

Baruah, S., Mok, A., and Rosier, L. (1990). Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190.

Bastoni, A. (2011). *Towards the Integration of Theory and Practice in Multiprocessor Real-Time Scheduling*. PhD thesis, Universita' degli Studi di Roma "Tor Vergata".

Bastoni, A., Brandenburg, B., and Anderson, J. (2010a). Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–44.

Bastoni, A., Brandenburg, B., and Anderson, J. (2010b). An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 14–24.

Bastoni, A., Brandenburg, B., and Anderson, J. (2011). Is semi-partitioned scheduling practical? In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 125–135.

Bell, S., Edwards, B., Amann, J., Conlin, R., Joyce, K., Leung, V., MacKay, J., Reif, M., Bao, L., Brown, J., Mattina, M., Miao, C.-C., Ramey, C., Wentzlaff, D., Anderson, W., Berger, E., Fairbanks, N., Khan, D., Montenegro, F., Stickney, J., and Zook, J. (2008). TILE64 - processor: A 64-core SoC with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 88–89,589.

Bernat, G., Burns, A., and Liamosi, A. (2001). Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321.

Bertogna, M. and Baruah, S. (2011). Tests for global EDF schedulability analysis. *Journal of Systems Architecture*, 57(5):487–497.

Bertogna, M. and Cirinei, M. (2007). Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 149–160.

Bertogna, M., Cirinei, M., and Lipari, G. (2005). Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218.

Bertogna, M., Cirinei, M., and Lipari, G. (2009). Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–666.

Bertogna, M., Fisher, N., and Baruah, S. (2007). Static-priority scheduling and resource hold times. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, page 154, Los Alamitos, CA, USA. IEEE Computer Society.

Bini, E. and Baruah, S. (2007). Efficient computation of response time bounds under fixed-priority scheduling. In *Proceedings of the 15th International Conference on Real-Time and Network Systems*, pages 95–104.

Bjerregaard, T. and Mahadevan, S. (2006). A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38(1).

Block, A. (2008). *Adaptive Multiprocessor Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill.

Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57.

Bovet, D. and Cesati, M. (2005). *Understanding the Linux Kernel*. O'Reilly, 3rd edition.

Brandenburg, B. (2011). Companion website to this dissertation. All data sets, graphs, and software publically available at `http://www.cs.unc.edu/~bbb/diss`.

Brandenburg, B. and Anderson, J. (2007a). Feather Trace: A light-weight event tracing toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 19–28.

Brandenburg, B. and Anderson, J. (2007b). Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 61–70.

Brandenburg, B. and Anderson, J. (2008a). A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 12th International Conference On Principles Of Distributed Systems*, pages 105–124.

Brandenburg, B. and Anderson, J. (2008b). An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 185–194.

Brandenburg, B. and Anderson, J. (2009a). On the implementation of global real-time schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 214–224.

Brandenburg, B. and Anderson, J. (2009b). Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *Proceedings of the 21th Euromicro Conference on Real-Time Systems*, pages 184–193.

Brandenburg, B. and Anderson, J. (2010a). Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st Real-Time Systems Symposium*, pages 49–60.

Brandenburg, B. and Anderson, J. (2010b). Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87.

Brandenburg, B. and Anderson, J. (2011). Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the International Conference on Embedded Software*.

Brandenburg, B., Block, A., Calandrino, J., Devi, U., Leontyev, H., and Anderson, J. (2007). LITMUS$^{\mathrm{RT}}$: A status report. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123.

Brandenburg, B., Calandrino, J., and Anderson, J. (2008). On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

Brandenburg, B., Leontyev, H., and Anderson, J. (2009). Accounting for interrupts in multiprocessor real-time systems. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 273–283.

Brandenburg, B., Leontyev, H., and Anderson, J. (2011). An overview of interrupt accounting techniques for multiprocessor real-time systems. *Journal of Systems Architecture: Embedded Software Design*, 57(6):638–654.

Brucker, P. (2007). *Scheduling Algorithms*. Springer, 5th edition.

Burns, A. and Baruah, S. (2008). Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):74–97.

Burns, A. and Wellings, A. (2009). *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, 4th edition.

Buttazzo, G. (2005). Rate monotonic vs. EDF: Judgment day. *Real-Time Systems*, 29:5–26.

Calandrino, J. (2009). *On the Design and Implementation of a Cache-Aware Soft Real-Time Scheduler for Multicore Platforms*. PhD thesis, University of North Carolina at Chapel Hill.

Calandrino, J., Anderson, J., and Baumberger, D. (2007). A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–256.

Calandrino, J., Leontyev, H., Block, A., Devi, U., and Anderson, J. (2006). LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.

Chandra, D., Guo, F., Kim, S., and Solihin, Y. (2005). Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351.

Checconi, F., Cucinotta, T., Faggioli, D., and Lipari, G. (2009). Hierarchical multiprocessor CPU reservations for the linux kernel. *Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 14–22.

Chen, C. and Tripathi, S. (1994). Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, Univ. of Maryland.

Chen, M. and Lin, K. (1990a). Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346.

Chen, M. and Lin, K. (1990b). A priority ceiling protocol for multiple-instance resources. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, pages 140–149.

Child, J. (2010). Multicore processing becomes the new mainstream. *COTS Journal*, April issue.

Childs, S. and Ingram, D. (2001). The Linux-SRT integrated multimedia operating system: Bringing QoS to the desktop. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 135–140.

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. (2001). An empirical study of operating systems errors. In *Proceedings of the 18th ACM symposium on Operating Systems Principles*, pages 73–88.

Cloutier, P., Mantegazza, P., Papacharalambous, S., Soanes, I., Hughes, S., and Yaghmour, K. (2000). DIAPM-RTAI position paper. In *Real-Time Linux Workshop*, volume 3.

CMU Real-Time and Multimedia Systems Lab (2006). Linux/RK. Project web site, `http://www.cs.cmu.edu/~rtmach/`.

Coffman, Jr., E., Garey, M., and Johnson, D. (1997). *Approximation algorithms for bin packing: a survey*, pages 46–93. PWS Publishing Company.

Corbet, J. (2010). KS2010: Deadline scheduling. Linux Weekly News, trade press, `http://lwn.net/Articles/412745/`.

Courtois, P., Heymans, F., and Parnas, D. (1971). Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668.

Crago, S. (2009). Mission-critical space software for multi-core processors. In *2009 Workshop on Spacecraft Flight Software (FSW-09)*.

Cucinotta, T. (2011). Adaptive quality of service architecture (for the linux kernel). Project web site, `http://aquosa.sourceforge.net/`.

Cucinotta, T., Palopoli, L., Marzario, L., Lipari, G., and Abeni, L. (2004). Adaptive reservations in a Linux environment. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 238–245. IEEE.

Davis, R. and Burns, A. (2011a). Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, pages 1–40.

Davis, R. and Burns, A. (2011b). A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (to appear)*.

Demers, A., Keshav, S., and Shenker, S. (1989). Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12.

Denning, P. (1968). The working set model for program behavior. *Communications of the ACM*, 11(5):323–333.

Dertouzos, M. (1974). Control robotics: The procedural control of physical processes. In *Information Processing 74*, pages 807–813.

Devi, U. (2003). An improved schedulability test for uniprocessor periodic task systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 23–30.

Devi, U. (2006). *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill.

Devi, U. and Anderson, J. H. (2005). Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341.

Devi, U. and Anderson, J. H. (2008). Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189.

Devi, U., Leontyev, H., and Anderson, J. (2006). Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 75–84.

Dhall, S. and Liu, C. (1978). On a real-time scheduling problem. *Operations Research*, 26(1):127–140.

Easwaran, A. and Andersson, B. (2009). Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 377–386.

Elliott, G. and Anderson, J. (2011). An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. In *Proceedings of the 19th International Conference on Real-Time and Network Systems*.

Erickson, J. and Anderson, J. (2011). Beating G-EDF at its own game: New results on G-EDF-like schedulers. Manuscript, The University of North Carolina at Chapel Hill.

Erickson, J., Devi, U., and Baruah, S. (2010a). Improved tardiness bounds for global EDF. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 14–23.

Erickson, J., Guan, N., and Baruah, S. (2010b). Tardiness bounds for global EDF with deadlines different from periods. In *Proceedings of the 14th Conference on the Principles of Distributed Systems*, LNCS 6490, pages 286–301.

Faggioli, D., Checconi, F., Trimarchi, M., and Scordino, C. (2009a). An EDF scheduling class for the linux kernel. In *Proceedings of the 11th Real-Time Linux Workshop*.

Faggioli, D., Lipari, G., and Cucinotta, T. (2010). The multiprocessor bandwidth inheritance protocol. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 90–99.

Faggioli, D., Trimarchi, M., and Checconi, F. (2009b). An implementation of the earliest deadline first algorithm in Linux. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1984–1989.

Fisher, N. (2007). *The Multiprocessor Real-Time Scheduling of General Task Systems*. PhD thesis, University of North Carolina at Chapel Hill.

Fisher, N., Anderson, J., and Baruah, S. (2005). Task partitioning upon memory-constrained multiprocessors. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 416–421.

Fisher, N., Bertogna, M., and Baruah, S. (2007a). The design of an EDF-scheduled resource-sharing open environment. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 83–92.

Fisher, N., Bertogna, M., and Baruah, S. (2007b). Resource-locking durations in EDF-scheduled systems. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 91–100.

Fisher, N., Goossens, J., and Baruah, S. (2010). Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1):26–71.

Funk, S. (2004). *EDF Scheduling on Heterogeneous Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill.

Gai, P., di Natale, M., Lipari, G., Ferrari, A., Gabellini, C., and Marceca, P. (2003). A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time And Embedded Technology Application Symposium*, pages 189–198.

Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman.

Gerum, P. (2008). The Xenomai real-time system. In *Building Embedded Linux Systems*, chapter 13, pages 365–385. O'Reilly Media, 2nd edition.

Gleixner, T. (2006). The real-time preemption patch (PREEMPT_RT). Keynote, 8th Real-Time Linux Workshop, `http://www.kernel.org/pub/linux/kernel/people/tglx/preempt-rt/rtlws2006.pdf`.

Gleixner, T. and Niehaus, D. (2006). Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the 2006 Ottawa Linux Symposium*, pages 333–346.

Goossens, J. (2008). (m, k)-firm constraints and DBP scheduling: impact of the initial k-sequence and exact schedulability test. In *Proceedings of the 16th Internation Conference on Real-Time and Network Systems*.

Goossens, J., Funk, S., and Baruah, S. (2003). Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205.

Gore, P., Pyarali, I., Gill, C., and Schmidt, D. (2004). The design and performance of a real-time notification service. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 112–120.

Goyal, P., Vin, H., and Chen, H. (1996). Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *ACM SIGCOMM Computer Communication Review*, 26(4):157–168.

Green Hills Software (2011). Everything you need to develop embedded software for aerospace & defense. Marketing material, `http://www.ghs.com/AerospaceDefense.html`.

Guan, N., Stigge, M., Yi, W., and Yu, G. (2009). Cache-aware scheduling and analysis for multicores. In *Proceedings of the 7th ACM International Conference on Embedded Software*, pages 245–254.

Hamdaoui, M. and Ramanathan, P. (1995). A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451.

Hamdaoui, M. and Ramanathan, P. (1997). Evaluating dynamic failure probability for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 46(12):1325–1337.

Hardy, D. and Puaut, I. (2009). Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *Proceedings of the 17th International Conference on Real-Time and Network Systems*, pages 45–54.

Harris, M. (2005). Shrinking slices: Looking at real time for Linux, PowerPC, and Cell—an interview with Paul E. McKenney. Technical report, IBM developerWorks. `http://www.ibm.com/developerworks/power/library/pa-nl14-directions.html`.

Härtig, H., Baumgartl, R., Borriss, M., Hamann, C., Hohmuth, M., Mehnert, F., Reuther, L., Schönberg, S., and Wolter, J. (1998). DROPS: Os support for distributed multimedia applications. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 203–209.

Härtig, H., Hohmuth, M., and Wolter, J. (1998). Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems*.

Härtig, H. and Roitzsch, M. (2006). Ten years of research on L4-based real-time systems. In *Proceedings of the 8th Real-Time Linux Workshop*.

Hennessy, J. and Patterson, D. (2006). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition.

Hergenhan, A. and Heiser, G. (2008). Operating systems technology for converged ECUs. In *Proceedings of the 6th Embedded Security in Cars Conference*.

Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba (2010). *Advanced Configuration and Power Interface Specification 4.0a*. `http://www.acpi.info/`.

Hohmuth, M. (1996). Linux-Emulation auf einem Mikrokern. Master's thesis, Technische Universität Dresden.

Holman, P. (2004). *On the Implementation of Pfair-scheduled Multiprocessor Systems*. PhD thesis, University of North Carolina at Chapel Hill.

Holman, P. and Anderson, J. (2002a). Locking in pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 149–158.

Holman, P. and Anderson, J. (2002b). Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 111–120.

Holman, P. and Anderson, J. (2005). Adapting Pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4):543–564.

Holman, P. and Anderson, J. (2006). Locking under Pfair scheduling. *ACM Transactions on Computer Systems*, 24(2):140–174.

Holmes, V. and Harris, D. (1989). A designer's perspective of the Hawk multiprocessor operating system kernel. *ACM SIGOPS Operating Systems Review*, 23(3):158–172.

Holmes, V., Harris, D., Piorkowski, K., and Davidson, G. (1987). Hawk: An operating system kernel for a real-time embedded multiprocessor. Technical report, Sandia National Laboratories.

Horn, W. (1974). Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185.

House, S. and Niehaus, D. (2000). Kurt-linux support for synchronous fine-grain distributed computations. In *Proceedings of the 6th IEEE Technology and Applications Symposium*, pages 78–87. Published by the IEEE Computer Society.

Hsieh, W. and Weihl, W. (1992). Scalable reader-writer locks for parallel systems. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 216–230.

Hsiu, P.-C., Lee, D.-N., and Kuo, T.-W. (2011). Task synchronization and allocation for many-core real-time systems. In *Proceedings of the 9th ACM International Conference on Embedded Software*, pages 79–88.

IEEE (1993). *IEEE Standard for Information Technology - Portable Operating System Interfaces (POSIX) - Part 1: System Application Program Interface - Amendment 1: Realtime Extension*. Number Std 1003.1b-1993. IEEE Computer Society.

IEEE (2003). *IEEE Standard for Information Technology - Standardized Application Environment Profile (AEP) - POSIX Realtime and Embedded Application Support*. Number Std 1003.13-2003. IEEE Computer Society.

IEEE (2008a). *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Number Std 1588-2008. IEEE Computer Society.

IEEE (2008b). *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) - Base Specifications*. Number Std 1003.1-2008. IEEE Computer Society.

Intel Corporation (1996). 82093AA I/O advanced programmable interrupt controller. Datasheet 290566, Intel Corporation.

Intel Corporation (2007). Intel 7300 chipset memory controller hub (MCH). Datasheet 318082, Intel Corporation.

Intel Corporation (2008a). *Basic Architecture*, volume 1 of *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

Intel Corporation (2008b). Intel Xeon processor 7400 series. Datasheet 320335, Intel Corporation.

Intel Corporation (2008c). *System Programming Guide*, volume 3 of *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

Intel Corporation (2008d). TLBs, paging-structure caches, and their invalidation. Application note 317080-003, Intel Corporation.

Ishiwata, Y. and Matsui, T. (1998). Development of Linux which has advanced real-time processing function. In *Proceedings of the 16th Annual Conference of Robotics Society of Japan*, pages 355–356.

Jackson, J. (1954). Scheduling a production line to minimize maximum tardiness. Research Report 43, Management Science Research Project, University of California at Los Angeles.

Jain, R., Hughes, C., and Adve, S. (2002). Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 134–145.

Jeffay, K. (1992). Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89–99.

Jeffay, K. and Stone, D. (1993). Accounting for interrupt handling costs in dynamic priority task systems. *Proceedings of the 14th Real-Time Systems Symposium*, pages 212–221.

Jensen, E., Locke, C., and Tokuda, H. (1985). A time-driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, pages 112–122.

Johnson, D. (1973). *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology.

Johnson, D. (1974). Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314.

Joseph, M. and Pandya, P. (1986). Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395.

Kanehiro, F., Hirukawa, H., and Kajita, S. (2004). OpenHRP: Open architecture humanoid robotics platform. *The International Journal of Robotics Research*, 23(2):155–165.

Kaneko, K., Kanehiro, F., Morisawa, M., Miura, K., Nakaoka, S., and Kajita, S. (2009). Cybernetic human HRP-4C. In *Proceedings of the 9th IEEE-RAS International Conference on Humanoid Robots*, pages 7–14. IEEE.

Kato, S., Ishikawa, Y., and Rajkumar, R. (2011). CPU scheduling and memory management for interactive real-time applications. *Real-Time Systems*, to appear.

Kato, S., Rajkumar, R., and Ishikawa, Y. (2010). AIRS: Supporting interactive real-time applications on multicore platforms. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 47–56.

Kenna, C., Herman, J., Brandenburg, B., Mills, A., and Anderson, J. (2011). Soft real-time on multiprocessors: Are analysis-based schedulers really worth it? In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*.

Kenyon, C. (1996). Best-fit bin-packing with random order. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 359–364.

Kirk, D. (1989). SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real Time Systems Symposium*, pages 229–237.

Kopetz, H. and Ochsenreiter, W. (1987). Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8).

Koren, G. and Shasha, D. (1995). Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 110–117.

Kurzak, J., Buttari, A., Luszczek, P., and Dongarra, J. (2008). The PlayStation 3 for high-performance scientific computing. *Computing in Science & Engineering*, 10(3):84–87.

Labetoulle, J. (1974). Some theorems on real time scheduling. In Gelenbe, E. and Mahl, R., editors, *Computer Architecture and Networks*, pages 285–298. North-Holland Publishing Company.

Lackorzynski, A. (2004). L^4Linux porting optimizations. Master's thesis, Technische Universität Dresden.

Lackorzynski, A. (2011). Running Linux on top of L4. Project web site, `http://os.inf.tu-dresden.de/L4/LinuxOnL4/`.

Lakshmanan, K. (2007). Alpha release of Linux/RK for 2.6.18 Linux kernel. Project web site, `http://www.andrew.cmu.edu/user/klakshma/rtml/`.

Lakshmanan, K., Niz, D., and Rajkumar, R. (2009). Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 469–478.

Lamastra, G., Lipari, G., and Abeni, L. (2001). A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 151–160.

Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455.

Lehoczky, J., Sha, L., and Ding, Y. (1989). The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 166–171.

Lehoczky, J., Sha, L., and Strosnider, J. (1987). Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 261–270.

Lehoczky, J., Sha, L., Strosnider, J., and Tokuda, H. (1991). Fixed priority scheduling theory for hard real-time systems. In van Tilborg, A. and Koob, G., editors, *Fondations of Real-Time Computing: Scheduling and Resource Management*, chapter 1, pages 1–30. Kluwer Academic Publishers.

Leontyev, H. (2010). *Compositional Analysis Techniques For Multiprocessor Soft Real-Time Scheduling*. PhD thesis, University of North Carolina at Chapel Hill.

Leontyev, H. and Anderson, J. (2007). Tardiness bounds for FIFO scheduling on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 71–80.

Leontyev, H. and Anderson, J. (2008). A unified hard/soft real-time schedulability test for global EDF multiprocessor scheduling. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 375–384.

Leontyev, H. and Anderson, J. (2010). Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1):26–71.

Leontyev, H., Chakraborty, S., and Anderson, J. (2009). Multiprocessor extensions to real-time calculus. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 410–421.

Leung, J. and Merrill, M. (1980). A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118.

Leung, J. and Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250.

Li, J., Song, Y., and Simonot-Lion, F. (2004). Schedulability analysis for systems under (m, k)-firm constraints. In *Proceedings of the 2004 IEEE Internation Workshop on Factory Communication Systems*, pages 22–30.

Liebemann, E., Meder, K., Schuh, J., and Nenninger, G. (2004). Safety and performance enhancement: The Bosch electronic stability control (ESP). In *Proceedings of the SAE Convergence Congress & Exposition On Transportation Electronics*.

Liedtke, J. (1995). On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250.

Liedtke, J., Härtig, H., and Hohmuth, M. (1997). OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, pages 213–224.

Lin, C. and Brandt, S. (2005). Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 410–421.

Lin, K. and Wang, Y. (2003). The design and implementation of real-time schedulers in RED-Linux. *Proceedings of the IEEE*, 91(7):1114–1130.

Linux Devices (2007). Snapshot of the embedded linux market. eWeek LinuxDevices.com, `http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Snapshot-of-the-embedded-Linux-market-April-2007/`.

Liu, C. (1969a). Scheduling algorithms for hard-real-time multiprogramming of a single processor. In *JPL Space Programs Summary 37-60*, volume II, pages 31–37. Jet Propulsion Laboratory.

Liu, C. (1969b). Scheduling algorithms for multiprocessors in a hard real-time environment. In *JPL Space Programs Summary 37-60*, volume II, pages 28–31. Jet Propulsion Laboratory.

Liu, C. and Anderson, J. (2011). Multiprocessor schedulability analysis for self-suspending task systems. Manuscript, The University of North Carolina at Chapel Hill.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61.

Liu, J. (2000). *Real-Time Systems*. Prentice Hall.

Liu, J., Lin, K.-J., Shih, W.-K., and Yu, A. (1991). Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):48–68.

López, J., Díaz, J., and García, D. (2004a). Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):642–653.

López, J., Díaz, J., and García, D. (2004b). Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68.

Lu, C., Stankovic, J., Abdelzaher, T., Tao, G., Son, S., and Marley, M. (2000). Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 13–23.

Lu, C., Stankovic, J., Son, S., and Tao, G. (2002). Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1):85–126.

Lv, M., Yi, W., Guan, N., and Yu, G. (2010). Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 339–349.

Macariu, G. and Cretu, V. (2011). Limited blocking resource sharing for global multiprocessor scheduling. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 262–271.

Malone, M. (2009). OPERA RHBD multi-core. In *Military/Aerospace Programmable Logic Device Workshop (MAPLD 2009)*.

Manolache, S., Eles, P., and Peng, Z. (2004). Optimization of soft real-time systems with deadline miss ratio constraints. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 562–570.

Masrur, A., Chakraborty, S., and Färber, G. (2010). Constant-time admission control for partitioned EDF. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 34–43.

Masrur, A., Drössler, S., and Färber, G. (2008). Improvements in polynomial-time feasibility testing for EDF. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1033–1038.

Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30.

Maurer, W. (2008). *Professional Linux Kernel Architecture*. Wiley.

McKenney, P. (2004). *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University.

McKenney, P. (2005). A realtime preemption overview. Linux Weekly News, trade press, `http://lwn.net/Articles/146861/`.

McKenney, P. (2007). SMP and embedded real-time. *Linux Journal*, 2007(153).

Mellor-Crummey, J. and Scott, M. (1991a). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65.

Mellor-Crummey, J. and Scott, M. (1991b). Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113.

Mercer, C. and Rajkumar, R. (1995). An interactive interface and RT-Mach support for monitoring and controlling resource management. In *Proceedings of the 1st IEEE Real-Time Technology and Application Symposium*, pages 134–139.

Mills, A. and Anderson, J. (2010). A stochastic framework for multiprocessor soft real-time scheduling. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 311–320.

Mills, A. and Anderson, J. (2011). A multiprocessor server-based scheduler for soft real-time tasks with stochastic execution demand. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 207–217.

Milojičić, D., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. (2000). Process migration. *ACM Computing Surveys*, 32(3):241–299.

Ming, L. (1994). Scheduling of the inter-dependent messages in real-time communication. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications*.

Mok, A. (1983). *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology.

Mok, A. and Wang, W. (2001). Window-constrained real-time periodic task scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 15–24.

Mollison, M., Brandenburg, B., and Anderson, J. (2009). Towards unit testing real-time schedulers in LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 5th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–39.

Molnar, I. (2004). CONFIG_PREEMPT_REALTIME, 'fully preemptible kernel', vp-2.6.9-rc4-mm1-t4. email to Linux Kernel Mailing List (LKML), `http://lwn.net/Articles/105948/`.

Monden, H. (1987). Introduction to ITRON the industry-oriented operating system. *IEEE Micro*, 7(2):45–52.

Müller, F. (1995). Compiler support for software-based cache partitioning. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 125–133.

Musial, M., Remuß, V., Deeg, C., and Hommel, G. (2006). Embedded system architecture of the second generation autonomous unmanned aerial vehicle MARVIN MARK II. In *Proceedings of the 7th International Workshop on Embedded Systems-Modeling, Technology and Applications*, pages 101–110.

NASA (2008). The Altair lunar lander. Fact sheet FS-2008-09-007-JSC, National Aeronautics and Space Administration.

Nelis, V., Goossens, J., and Andersson, B. (2009). Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 151–160.

Nemati, F., Nolte, T., and Behnam, M. (2010). Partitioning real-time systems on multiprocessors with shared resources. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, LNCS 6490, pages 253–269.

NIST/SEMATECH (2010). e-Handbook of statistical methods. `http://www.itl.nist.gov/div898/handbook/`.

Oh, D.-I. and Baker, T. (1998). Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15(2):183–192.

Oikawa, S. and Rajkumar, R. (1998). Linux/RK: A portable resource kernel in Linux. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (Work in Progress Session)*.

Oikawa, S. and Rajkumar, R. (1999). Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the 5th Real-Time Technology and Applications Symposium*, pages 111–120.

Olukotun, K., Nayfeh, B., Hammond, L., Wilson, K., and Chang, K. (1996). The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11.

Palopoli, L., Cucinotta, T., Marzario, L., and Lipari, G. (2009). AQuoSA—adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31.

Paoloni, G. (2010). How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. Whitepaper 324264-001, Intel Corporation.

Pellizzoni, R. and Caccamo, M. (2008). M-CASH: A real-time resource reclaiming algorithm for multiprocessor platforms. *Real-Time Systems*, 40(1):117–147.

Phillips, C. A., Stein, C., Torng, E., and Wein, J. (1997). Optimal time-critical scheduling via resource augmentation. In *Proceedings of the 29th annual ACM Symposium on Theory of Computing*, pages 140–149.

Phillips, C. A., Stein, C., Torng, E., and Wein, J. (2002). Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200.

QNX Software Systems (2011). QNX developer support. Documentation for QNX Neutrino 6.3.0SP3, `http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/smp.html#BMP`.

Quan, G. and Hu, X. (2000). Enhanced fixed-priority scheduling with (m, k)-firm guarantee. In *Proceedings of the 21st IEEE Real Time Systems Symposium*, pages 79–88.

Rajkumar, R. (1990). Real-time synchronization protocols for shared memory multiprocessors. *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123.

Rajkumar, R. (1991). *Synchronization In Real-Time Systems—A Priority Inheritance Approach*. Kluwer Academic Publishers.

Rajkumar, R., Juvva, K., Molano, A., and Oikawa, S. (1998). Resource kernels: a resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*.

Rajkumar, R., Juvva, K., Molano, A., and Oikawa, S. (2001). Resource kernels: A resource-centric approach to real-time and multimedia systems. In Jeffay, K. and Zhang, H., editors, *Readings in multimedia computing and networking*, pages 476–490. Morgan Kaufmann.

Rajkumar, R., Sha, L., and Lehoczky, J. (1988). Real-time synchronization protocols for multiprocessors. *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 259–269.

Ramanathan, P. (1999). Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559.

Ramaprasad, H. and Mueller, F. (2010). Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, 10(2):27:1–27:34.

Ravindran, B., Jensen, E., and Li, P. (2005). On recent advances in time/utility function real-time scheduling and resource management. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 55–60.

Ruocco, S. (2008). A real-time programmer's tour of general-purpose L4 microkernels. *EURASIP Journal on Embedded Systems*, 2008:1–14.

Saulsbury, A. (2008). *UltraSPARC Virtual Machine Specification*. Sun Corporation.

Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185.

Short, M. (2010). Improved task management techniques for enforcing EDF scheduling on recurring tasks. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 56–65.

Simchi-Levi, D. (1994). New worst-case results for the bin-packing problem. *Naval Research Logistics*, 41(4):579–585.

Sjödin, M. and Hansson, H. (1998). Improved response-time analysis calculations. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 399–408.

Sodan, A., Machina, J., Deshmeh, A., Macnaughton, K., and Esbaugh, B. (2010). Parallelism via multithreaded and multicore CPUs. *Computer*, 43(3):24–32.

Spuri, M. and Buttazzo, G. (1994). Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 2–11.

Spuri, M. and Buttazzo, G. (1996). Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210.

Srinivasan, A. (2003). *Efficient and Flexible Fair Scheduling of Real-time Tasks on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill.

Srinivasan, A. and Anderson, J. (2002). Optimal rate-based scheduling on multiprocessors. In *In Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198.

Srinivasan, A. and Anderson, J. (2003). Efficient scheduling of soft real-time applications on multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 51–59.

Srinivasan, A. and Anderson, J. (2005). Efficient scheduling of soft real-time applications on multiprocessors. *Journal of Embedded Computing*, 1(2):285–302.

Srinivasan, A. and Anderson, J. (2006). Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117.

Srinivasan, A., Holman, P., and Anderson, J. (2002). Integrating aperiodic and recurrent tasks on fair-scheduled multiprocessors. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 19–28.

Srinivasan, B., Pather, S., Hill, R., Ansari, F., and Niehaus, D. (1998). A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, pages 112–119.

Stankovic, J. and Rajkumar, R. (2004). Real-time operating systems. *Real-Time Systems*, 28(2-3):237–253.

Stankovic, J. and Ramamritham, K. (1991). The Spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72.

Stanovich, M., Baker, T., Wang, A., and Harbour, M. (2010). Defects of the POSIX sporadic server and how to correct them. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 35–45.

Staschulat, J. and Ernst, R. (2007). Scalable precision cache analysis for real-time software. *ACM Transactions on Embedded Computing Systems*, 6(4):25:1–25:39.

Steinberg, U., Böttcher, A., and Kauer, B. (2010). Timeslice donation in component-based systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 16–23.

Steinberg, U., Wolter, J., and Härtig, H. (2005). Fast component interaction for real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 89–97.

Stern, J. (2010). HP TouchSmart tm2t review. `http://www.engadget.com/2010/06/27/hp-touchsmart-tm2-review/3`.

Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S., Gehrke, J., and Plaxton, C. (1996). A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299.

Strosnider, J., Lehoczky, J., and Sha, L. (1995). The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91.

Sundaram, V., Chandra, A., Goyal, P., Shenoy, P., Sahni, J., and Vin, H. (2000). Application performance in the qlinux multimedia operating system. In *Proceedings of the 8th ACM International Conference on Multimedia*, pages 127–136.

SYSGO AG (2011). Customer successes. Promotional case studies, `http://www.sysgo.com/en/services-solutions/sysgo-solutions-at-work/customer-successes/`.

Takada, H. and Sakamura, K. (1991). ITRON-MP: an adaptive real-time kernel specification for shared-memory multiprocessor systems. *IEEE Micro*, 11:24–27, 78–85.

Takada, H. and Sakamura, K. (1994). Predictable spin lock algorithms with preemption. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 2–6.

Takada, H. and Sakamura, K. (1995). μITRON for small-scale embedded systems. *IEEE Micro*, 15(6):46–54.

Takada, H. and Sakamura, K. (1997). A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 134–143.

Tanenbaum, A. S. (2005). *Structured Computer Organization*. Pearson Prentice Hall, 5th edition.

Thiebaut, D. and Stone, H. (1987). Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329.

Tijdeman, R. (1980). The chairman assignment problem. *Discrete Mathematics*, 32(3):323–330.

Tokuda, H., Nakajima, T., and Rao, P. (1990). Real-time Mach: Towards a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, pages 73–82.

Topolsky, J. (2010). Nook Color review. `http://www.engadget.com/2010/11/16/nook-color-review/`.

Torvalds, L. (1997). Linux: A portable operating system. Master's thesis, University of Helsinki.

Torvalds, L. and contributors (2010). The Linux kernel 2.6.36. Source code available at `http://www.kernel.org`.

Turley, J. (2005). Embedded systems survey: Operating systems up for grabs. EE Times, trade press, `http://www.eetimes.com/discussion/other/4025539/Embedded-systems-survey-Operating-systems-up-for-grabs`.

van Rossum, G. and contributors (2010). The Python 2.6.6 distribution. Source code available at `http://www.python.org`.

Varghese, G. and Lauck, T. (1987). Hashed and hierarchical timing wheels: data structures for the efficient implementation of a timer facility. *SIGOPS Operating Systems Review*, 21(5):25–38.

Villalpando, C., Johnson, A., Some, R., Oberlin, J., and Goldberg, S. (2010). Investigation of the Tilera processor for real time hazard detection and avoidance on the Altair lunar lander. In *Proceedings of the 2010 IEEE Aerospace Conference*.

Vuillemin, J. (1978). A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315.

Wang, C.-D., Takada, H., and Sakamura, K. (1996). Priority inheritance spin locks for multiprocessor real-time systems. In *Proceedings of the 2nd International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 70–76.

Wang, S., Li, K., and Wang, Y. (2002). Hierarchical budget management in the RED-Linux scheduling framework. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 76–83. IEEE Computer Society.

Wang, Y. and Lin, K. (1999). Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 246–255.

Wang, Y.-C. and Lin, K.-J. (1998). Enhancing the real-time capability of the Linux kernel. In *Proceedings of the 5th International IEEE Conference on Real-Time Computing Systems and Applications*, pages 11–20.

West, R. and Poellabauer, C. (2000). Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 239–248.

West, R. and Schwan, K. (1999). Dynamic window-constrained scheduling for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 87–91.

Wikipedia (2011). List of real-time operating systems. http://en.wikipedia.org/w/index.php?title=List_of_real-time_operating_systems&oldid=455438897.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. (2008). The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53.

Yan, J. and Zhang, W. (2008). WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89.

Yang, T., Liu, T., Berger, E., Kaplan, S., and Moss, J. (2008). Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 73–86.

Yodaiken, V. and Barabanov, M. (1997). A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*.

Zijlstra, P. (2008). Linux-rt: Turning a general purpose OS into a real-time OS. Keynote, Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, http://www.cs.unc.edu/~anderson/meetings/ospert08/zijlstra.pdf.