A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications

Björn B. Brandenburg
Max Planck Institute for Software Systems (*MPI-SWS*)

Abstract

Independence preservation, a property in real-time locking protocols that isolates latency-sensitive tasks from delays due to unrelated critical sections, is identified, formalized, and studied in detail. The key to independence preservation is to ensure that tasks remain fully preemptive at all times. For example, on uniprocessors, the classic priority inheritance protocol is independence-preserving. It is shown that, on multiprocessors, independence preservation is impossible if job migrations are disallowed. The $O(m)$ independence-preserving protocol (OMIP)*, a new, asymptotically optimal binary sempahore protocol based on migratory priority inheritance, is proposed and analyzed. The OMIP is the first independence-preserving, real-time, suspension-based locking protocol for clustered job-level fixed-priority scheduling. It is shown to benefit latency-sensitive workloads, both analytically by means of schedulability experiments, and empirically using response-time measurements in LITMUS^{RT}.*

1 Introduction

A frequent, but challenging requirement in the design and implementation of practical real-time systems is the need for predictable, mutually exclusive access to *shared resources* such as I/O ports, network links, message buffers, or other shared data structures. To this end, virtually all modern real-time operating systems (RTOSs) offer some sort of locking primitives, most commonly *binary semaphores* (or *mutexes*), under which tasks suspend when trying to access already locked resources. The defining characteristics of a *real-time* semaphore protocol are that it **(i)** must guarantee bounded delays for *all* real-time tasks—which at times requires lock-holding tasks to be scheduled despite the presence of (otherwise) higher-priority tasks [31, 34]—while **(ii)** simultaneously ensuring that urgent, high-priority tasks are not unduly delayed by unrelated critical sections in lower-priority tasks.

In the *uniprocessor* case, the real-time locking problem has long been solved: the three classic uniprocessor real-time semaphore protocols, namely the *priority-inheritance protocol* (PIP) [31, 34], the *priority-ceiling protocol* (PCP) [31, 34], and the *stack resource policy* (SRP) [3], fully satisfy requirements (i) and (ii). Unfortunately, the same cannot be said of the *multiprocessor* case. While numerous protocols satisfying requirement (i) have been developed for various scheduling approaches (reviewed below), we show in this paper that *all* existing protocols fail to meet the second requirement when confronted with *latency-sensitive* workloads (*i.e.*, if some tasks have near-zero tolerance for scheduling delays) under *partitioned scheduling* (where each task is statically assigned to a processor and each processor is scheduled individually). In other words, there exist demanding, but common hard real-time workloads that currently simply cannot be provisioned on multiprocessors under partitioned scheduling (which is widespread in practice) due to the lack of appropriate synchronization primitives.

The primary contribution of this paper is a solution to this problem: we present the design, analysis, implementation, and evaluation of the first multiprocessor real-time semaphore protocol applicable to partitioned scheduling (and other policies) that guarantees that latency-sensitive applications are *never* delayed by unrelated lower-priority tasks.

The problem. To understand why existing multiprocessor semaphore protocols fail, first consider why the classic uniprocessor protocols *do* work. For example, consider the *de facto* industry standard of preemptive *fixed-priority* (FP) scheduling in conjunction with the PIP, and let T_h denote the highest-priority task in the system. *Priority inheritance* [31, 34] famously addresses requirement (i) by temporarily raising the priority of a lock-holding task to that of the highest-priority waiter (if any). One major advantage of this approach is that task T_h is *never* delayed by critical sections associated with resources shared *only* among lower-priority tasks, since such critical sections always execute at a lower priority. *As a result, such unrelated critical sections may be of arbitrary length without affecting the temporal correctness of T_h.* This isolation property, which we formalize as "independence preservation" in Sec. 3, crucially depends on the fact that lock-holding tasks remain fully preemptive.

Now consider the most straightforward (and widespread) multiprocessor extension of FP scheduling, namely *partitioned fixed-priority* (P-FP) scheduling, together with the *multiprocessor priority-ceiling protocol* (MPCP) [30, 31], a classic shared-memory semaphore protocol for P-FP scheduling. Under partitioned scheduling, it is particularly difficult to ensure that lock-holders are scheduled when remote tasks are blocked;[1] the MPCP—and all real-time semaphore protocols for partitioned scheduling developed since—employ *priority boosting*, that is, they simply raise the priority of lock-holding tasks above any "normal" priority, thereby ensuring that lock-holding tasks are *never*

[1] Priority inheritance is ineffective under partitioning, *e.g.*, see [8, 13].

preempted by non-lock-holding tasks. While effective, this simple technique has the unfortunate consequence that lock-holding tasks are effectively no longer fully preemptive: even the highest-priority task T_h cannot preempt all other tasks at all times. *As a result, there is no isolation: the temporal correctness of any task depends on the maximum critical section length of any other task (assigned to the same processor).* Crucially, this is unacceptable for latency-sensitive workloads, where lower-priority tasks must *not* affect the correctness of the highest-priority tasks.

The described phenomenon is not specific to the MPCP, but rather a consequence of priority boosting. While priority boosting can be acceptable if *all* critical sections can be ascertained with high confidence to always be "short enough," latency-sensitive workloads with highly heterogeneous timing requirements fundamentally require high-priority tasks to be shielded from unrelated critical sections. Such workloads are common in practice. For instance, in the automotive domain, it is not unusual for real-time constraints to span three orders of magnitude, ranging from a few milliseconds for high-frequency control tasks up to one second for less urgent tasks (*e.g.*, see [37]). However, in a task with a period of 1000 milliseconds, it is not unreasonable to encounter critical sections with *worst-case* lengths in excess of one millisecond, especially in embedded systems with relatively slow processors. Clearly such a critical section cannot be permitted to delay latency-sensitive tasks (*e.g.*, those with a one-millisecond period). However, as we discuss next, *none* of the real-time semaphore protocols for partitioned scheduling proposed to date can actually guarantee this—a major oversight that we rectify in this paper.

Related work. Multiprocessor semaphore protocols are typically designed for, and tightly integrated with, a specific scheduling policy. Most relevant to our work are prior semaphore protocols for partitioned scheduling. The first such protocols were proposed by Rajkumar *et al*. [30–32], who designed the aforementioned MPCP and the *distributed PCP* for P-FP-scheduled systems. In later work, Block *et al*. [7] proposed the *flexible multiprocessor real-time locking protocol* (FMLP), and an improved version for partitioned scheduling, termed FMLP$^+$, has since been developed in [8]. In recent work, Nemati *et al*. [28] proposed a semaphore protocol that simplifies the consolidation of independently developed (legacy) applications onto P-FP-scheduled multicore platforms. Crucially, all of the just-cited protocols employ priority boosting and are thus not suitable for latency-sensitive workloads, as explained above.

In work focused on *global scheduling*, under which (conceptually) all processors serve a single ready queue and tasks may migrate freely among cores, several semaphore protocols have been proposed in recent years, including a global variant of the FMLP and two extensions of the PCP [21, 27]. These protocols rely on priority inheritance [34] instead of priority boosting and allow tasks to remain fully preemptive; however, they apply *only* to global scheduling and cannot be easily transferred to the partitioned case, which is (more) commonly used in industrial applications (*e.g.*, partitioning is mandated by the Autosar 4.0 standard).

Most closely related to this paper are optimality results [8, 12, 13] pertaining to the intuitive goal of "minimal blocking" in multiprocessor real-time semaphore protocols under *job-level fixed-priority* (JLFP) scheduling, a broad category of schedulers that includes the FP and *earliest-deadline first* (EDF) policies. Specifically, recent work [12] investigated asymptotic bounds on maximum *priority-inversion blocking* (*pi-blocking*), which (intuitively) occurs whenever a high-priority task is delayed by a lower-priority one. Surprisingly, there exist two classes of schedulability analysis, called *suspension-aware* and *suspension-oblivious* analysis [12], resp., that give rise to different lower bounds on worst-case pi-blocking, namely $\Omega(n)$ and $\Omega(m)$, resp., where n denotes the number of tasks and m the number of processors. In brief, the difference arises due to how semaphore-induced self-suspensions are accounted for.

This paper pertains to the suspension-oblivious case, for which there already exists the asymptotically optimal family of *O(m) locking protocols* (OMLP) [8, 12, 13]. In particular, the OMLP family contains semaphore-based mutual exclusion protocols for global scheduling, partitioned scheduling, and for *clustered scheduling* [4, 17], which is a hybrid (or generalization) of global and partitioned scheduling where tasks are statically assigned to non-overlapping subsets (or *clusters*) of processors, and each cluster is scheduled using a global policy. We revisit these protocols, denoted as P-OMLP, G-OMLP, and C-OMLP, resp., in greater detail in Sec. 2. However, note that the OMLP family decidedly does *not* solve the problem addressed in this paper, which is to support *latency-sensitive* workloads under *clustered* scheduling, since the priority-inheritance-based G-OMLP applies *only* to global scheduling, whereas the P-OMLP and the C-OMLP use (a variant of) priority boosting, and thus are susceptible to potentially problematic latency increases.

A well-studied, sometimes-preferable alternative to semaphores are *non-preemptive spinlocks*, in which blocked jobs busy-wait by executing a delay loop (*e.g.*, see [7, 8, 18, 25]). While busy-waiting conceptually wastes processor cycles, non-preemptive spinlocks are an attractive choice in practice due to their simplicity and low overheads [8, 15]. Unfortunately, non-preemptive spinlocks can cause unacceptable delays for latency-sensitive workloads because tasks may remain non-preemptive for prolonged times. While it is possible to reduce the negative effects by allowing spinning tasks to remain preemptive (*e.g.*, see [2, 20, 35]), critical sections are still executed non-preemptively in such spinlocks and thus have a major impact on latency, not unlike the priority-boosting of critical sections under the MPCP.

The need for non-preemptive execution can be avoided altogether with *helping* mechanisms (*e.g.*, see [36]), where instead of executing a delay loop, "spinning" tasks attempt

to complete the critical sections of preempted lock holders. However, such helping-based approaches have two limitations in practice. First, they are considerably more complicated than regular spinlocks, which negates one of the key reasons to use spinlocks in the first place, and second, they are less general than regular locks. That is, "helping" cannot be offered as a generic OS primitive, but must be integrated individually into each application because critical sections must be implemented in a "helping-aware" manner such that they can be posted to a queue of pending operations.

Finally, in work aimed at temporal isolation among tasks in mixed real-time/non-real-time environments, Faggioli *et al*. [24] presented the *multiprocessor bandwidth-inheritance protocol* (MBWI), a scheduler-agnostic locking protocol based on preemptive spinning. Notably, to ensure progress, the MBWI allows lock-holding tasks to migrate to processors on which tasks are waiting for the lock, an effective technique that we reuse in Sec. 3.

Contributions. We provide a precise definition of *independence preservation*, which captures the idea that "tasks should be isolated from unrelated critical sections," and provide evidence of its considerable practical importance (Sec. 3). We show in Sec. 4.1 that this desirable property is impossible on multiprocessors unless jobs may (temporarily) migrate among all processors. Based on this observation, we present in Sec. 4.2 our main contribution, the $O(m)$ *independence-preserving locking protocol* (OMIP) for clustered scheduling, and show that it is optimal within a factor of approximately two under suspension-oblivious analysis (Sec. 4.3). Finally, in Sec. 5, we report on response-time measurements and schedulability experiments that show the OMIP to greatly benefit latency-sensitive workloads. We begin with a review of needed background.

2 Background and Definitions

This paper assumes the classic sporadic task model and follows the conventions from [12].

Tasks and jobs. The workload is specified as a set of n sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ that is to be scheduled on m identical processors. We let T_i denote a task with a *worst-case per-job execution time* e_i and a *minimum job separation* p_i, and let J_i denote a job of T_i. Our results do not depend on the type of deadline constraint; we assume implicit deadlines for simplicity. A task's *utilization* is the ratio $u_i = e_i/p_i$, and τ's *total utilization* is given by $U = \sum_{T_i \in \tau} u_i$.

A task's *slack* $p_i - e_i$ describes how much delay one of its jobs can tolerate without missing its deadline. A task is *latency-sensitive* if its slack does not exceed the maximum critical section length of other tasks (see below).

Jobs are *pending* from the time when they are released until they complete. A pending job can be in one of two 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.

Scheduling. Under *clustered scheduling* [4, 17], processors are grouped into $\frac{m}{c}$ non-overlapping sets (or *clusters*) of c processors each, where C_k denotes the k^{th} cluster. We require uniform cluster sizes and for simplicity assume that m is an integer multiple of c.[2] Each task is statically assigned to a cluster; we let $C(T_i)$ denote the cluster to which task T_i has been assigned. *Partitioned* ($c = 1$) and *global* ($c = m$) scheduling are special cases of clustered scheduling.

Priority. Each cluster is scheduled independently using a work-conserving *job-level fixed-priority* (JLFP) scheduler. Under a JLFP scheduler, each job is assigned a unique *base priority* that does not change while it is pending. However, a locking protocol (see below) may temporarily raise the *effective priority* of a job to ensure its timely completion. At any point in time and w.r.t. each cluster, the c highest-effective-priority ready jobs are scheduled.

JLFP scheduling is a broad category of scheduling policies that includes EDF and FP scheduling. Note that in the following, we use the terms "higher-priority" and "lower-priority" to refer to the generic JLFP notion of a job's base priority (and *not* to a task's priority under FP scheduling). For illustrative purposes, we consider *clustered earliest-deadline-first* (C-EDF) scheduling as a representative JLFP policy. (We chose C-EDF since it is the policy actually implemented in LITMUS$^{\text{RT}}$, the RTOS underlying the evaluation in Sec. 5.) We let P-EDF denote the special case of partitioned scheduling (*i.e.*, C-EDF scheduling with $c = 1$).

Real-time locking. Besides the m processors, tasks share r serially-reusable shared resources $\ell_1, \ldots, \ell_r$. Access to shared resources is governed by a *locking protocol* that ensures mutual exclusion by serializing conflicting requests.

A job J_i *requests* a resource ℓ_q at most $N_{i,q}$ times. If ℓ_q is already in use, J_i incurs *acquisition delay* until its request for ℓ_q is *satisfied*. In this paper, we focus on semaphore protocols, that is, jobs wait by self-suspending. Once J_i has finished using ℓ_q, its request is *complete* and J_i *releases* ℓ_q. We let $L_{i,q}$ denote the *maximum critical section length*, that is, the maximum time that J_i uses ℓ_q as part of a single request (or *critical section*). We assume that J_i must be scheduled in order to use ℓ_q; however, the analysis in Sec. 4 remains valid even if critical sections contain suspensions (*e.g.*, when performing I/O). For notational convenience, we define $L_q^{max} \triangleq \max_i\{L_{i,q}\}$ and $L^{max} \triangleq \max_q\{L_q^{max}\}$, and require that $L_{i,q} = 0$ if $N_{i,q} = 0$.

We assume that tasks do not hold resources across job boundaries and that jobs request at most one resource at any time. Analysis of nested critical sections is beyond the scope of this paper and the subject of future work (see Sec. 6).

Priority inversion. Intuitively, a *priority inversion* exists if a high-priority job that *should* be scheduled is *not* scheduled, which is typically due to acquisition delay. However,

[2] If c does not evenly divide m, there are $\lfloor \frac{m}{c} \rfloor$ clusters of size c and one cluster of size $m \bmod c$. This does not affect the asymptotic bounds.

priority inversion is not the same as acquisition delay: a suspended job does not suffer a priority inversion if it would not have been scheduled anyway (*i.e.*, in the presence of higher-priority jobs) [31, 34]. We let b_i denote the maximum cumulative duration of priority inversion incurred by any J_i.

Recent work [8, 12] showed that what exactly constitutes a priority inversion (Def. 1 below) depends on how self-suspensions are accounted for during schedulability analysis (*i.e.*, when checking whether all jobs will meet their deadline). Accurate analysis of self-suspensions is NP-hard [33] and complicates schedulability tests considerably. Therefore, many published tests (*e.g.*, [5, 6]) make the simplifying assumption that tasks never self-suspend. Such *suspension-oblivious* (*s-oblivious*) tests can still be used with semaphores by inflating each task's execution cost e_i by b_i prior to applying the test [12]. In contrast, *suspension-aware* (*s-aware*) schedulability tests explicitly account for self-suspensions. The focus of this paper is s-oblivious analysis; the s-aware case is subject of ongoing work. S-oblivious analysis yields the following definition of "priority inversion."

Def. 1. A pending job J_i suffers an *s-oblivious priority inversion* at time t iff J_i is **(i)** not scheduled and **(ii)** among the c highest-base-priority *pending* jobs in cluster $C(T_i)$ [12].

Note that clause (ii) pertains only to jobs in J_i's assigned cluster $C(T_i)$—the priorities of jobs in other clusters are irrelevant since each cluster is analyzed individually—and that it pertains to pending (and not just ready) jobs because self-suspensions are analyzed as execution time.

Priority inversions are considered to cause "blocking" because they lead to an undesirable increase in worst-case response time. To avoid confusing such locking-related delays with other uses of the word "blocking," we use the more specific term "pi-blocking" in this paper.

PI-Blocking optimality. The purpose of a real-time locking protocol is to minimize the occurrence and duration of priority inversions, which, however, cannot be avoided entirely. The significance of differentiating between s-oblivious and s-aware analysis is that they give rise to different lower bounds on *maximum pi-blocking*, formally $\max_\tau\{b_i\}$ [12]. In particular, the s-oblivious case is subject to an $\Omega(m)$ lower bound on maximum pi-blocking, whereas s-aware analysis yields an $\Omega(n)$ lower bound (where $n \geq m$). In other words, under s-oblivious analysis, there exist pathological task sets such that pi-blocking is linear in the number of processors under *any* semaphore protocol. To be asymptotically optimal, a locking protocol must ensure that maximum pi-blocking is *always* within a constant factor of m. In the suspension-oblivious case, $\max_i\{b_i\} = O(m)$ is hence asymptotically optimal. When stating asymptotic bounds, L^{max} and $\sum_q N_{i,q}$ are considered constant (*i.e.*, not a function of n or m) [12, 13]. The task execution costs e_i are *not* constants, and pi-blocking is considered "bounded" only if no b_i depends on any e_i.

The OMLP. The family of *O(m) locking protocols* is a collection of in total five asymptotically optimal locking protocols for global, partitioned, and clustered JLFP scheduling [8, 12, 13]. Most relevant to this paper are the two mutual exclusion protocols for partitioned [12] and clustered scheduling [13], which we denote as P-OMLP and C-OMLP, resp. Both use *priority boosting* to ensure lock-holder progress, which means that the effective priority of lock-holding jobs exceeds any possible base priority. Lock-holding jobs thus are never preempted by non-lock-holding jobs, and in particular not by newly-released jobs (which cannot yet have acquired any locks at the time of their release).

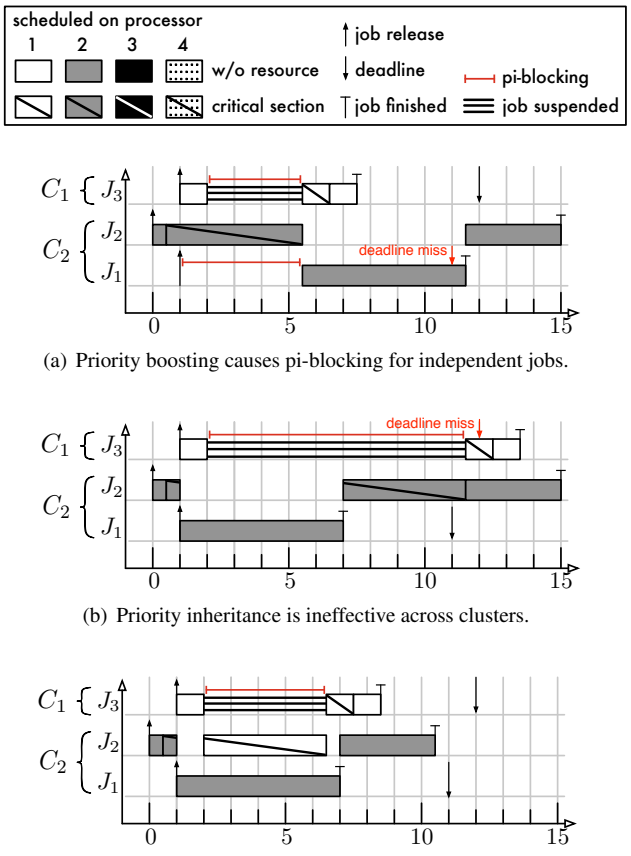
While we consider clustered JLFP scheduling with arbitrary cluster sizes, we focus on the P-OMLP [12] as a baseline because it is somewhat simpler and because it suffices to reveal the limitations of priority boosting. The key difference between the two protocols is that the P-OMLP is asymptotically optimal only for $c = 1$, whereas the C-OMLP uses a refined version of priority boosting called *priority donation* that allows it to cover the entire range of $1 \leq c \leq m$ [13] . For the purpose of this paper, the C-OMLP and the P-OMLP can be considered to be equivalent.

In brief, the P-OMLP works as follows. Requests for each ℓ_q are serialized with a FIFO queue FQ_q. However, before a job can be enqueued in any FQ_q, it must hold its processor's *contention token* CT_k. As there is only one such token per processor, at most m jobs are enqueued in any FQ_q at any time, which yields the desired $O(m)$ bound. Contention for each CT_k is managed using a per-processor priority queue PQ_k. The P-OMLP is simple, easy to implement [8], ensures asymptotically optimal maximum pi-blocking, and has been shown to perform well compared to the FMLP$^+$ and the MPCP for certain workloads [8, 13]. However, its chief limitation, which it shares with the FMLP$^+$ and the MPCP, is its reliance on priority boosting, as we show in detail next.

3 Independence Preservation

In this section, we first give a precise specification for the intuitive requirement that "high-priority jobs should be isolated from unrelated critical sections," and then provide evidence of its considerable practical importance.

To motivate the need for such isolation, consider the example P-OMLP schedule given in Fig. 1(a), which demonstrates that priority boosting is ill-suited to latency-sensitive workloads. In this particular example, task T_1 with parameters $e_1 = 6$ and $p_1 = 10$ is latency-sensitive since it has only four time units of slack, less than task T_2's maximum critical section length ($L_{2,1} = 5$). In the depicted schedule, job J_2 acquires resource ℓ_1 and becomes priority-boosted shortly after its release at time 0. Hence, job J_1 cannot preempt J_2 at time 1 and consequently misses its deadline at time 11. Notably, J_1 does not access shared resources and is continuously the highest-priority job while it is pending—clearly it *should* not incur *any* delays, but under the P-OMLP (or any other semaphore protocol based on priority boosting), J_1 is not isolated from J_2's long critical section because J_2



(a) Priority boosting causes pi-blocking for independent jobs.

(b) Priority inheritance is ineffective across clusters.

(c) Temporarily migrating J_2 to C_1 renders the task set schedulable.

Figure 1: P-EDF schedules ($c = 1$) of three tasks T_1, T_2, T_3 with execution costs $e_1 = 6$, $e_2 = 9$, $e_3 = 3$, and periods $p_1 = 10$, $p_2 = 40$, and $p_3 = 11$. Tasks T_2 and T_3 share one resource (ℓ_1), where $L_{2,1} = 5$ and $L_{3,1} = 1$. While P-EDF is assumed in this example, the same effect can arise under any partitioned JLFP scheduler (including P-FP). The legend applies to Fig. 2, too.

is effectively non-preemptable while accessing ℓ_1.

It bears repeating that classic priority inheritance [34], which allows jobs to remain fully preemptive, is unfortunately ineffective (analytically speaking) when applied *across* clusters or partitions, as shown in Fig. 1(b). At time 2, job J_3 attempts to lock ℓ_1, but must wait since J_2 already holds the resource. Assuming priority inheritance, this causes J_2's effective priority (*i.e.*, deadline under P-EDF) to be raised from time 40 (J_2's deadline) to time 12 (J_3's deadline). However, J_1 has an even earlier deadline at time 11 and J_2 remains preempted. As a result, J_3 incurs *unbounded* pi-blocking in the sense that it is no longer possible to bound pi-blocking only in terms of critical section lengths (b_3 depends on e_1). A deadline miss results.

However, a simple tweak renders this example schedulable: instead of boosting J_2's priority while holding ℓ_1, it should simply be migrated to J_3's cluster C_1 when J_3 requests ℓ_1. As illustrated in Fig. 1(c), this allows J_2 to complete its critical section in a timely fashion so that J_3 can meet its deadline. Notably, this is accomplished without causing *any* additional delay for the *independent*, latency-sensitive job J_1, and while some pi-blocking is unavoidable when using semaphores [12], it is only incurred by the job that actually requires a shared resource (J_3 in this example). We formalize this property as follows (first defined in [8]).

Def. 2. Let $b_{i,q}$ denote the maximum pi-blocking incurred by J_i due to requests by *any* task for resource ℓ_q. Under s-oblivious analysis, a locking protocol is *independence-preserving* if and only if $N_{i,q} = 0$ implies $b_{i,q} = 0$.

In other words, independence among tasks is preserved if the overall s-oblivious[3] pi-blocking $b_i = \sum_q b_{i,q}$ does not depend on resources that a task does not access.

Significance. We posit that independence preservation is indispensable for many workloads common in current practice in industry, as illustrated by the following use cases.

First, if the locks acquired (only) by lower-priority tasks protect data structures of potentially unbounded size (*e.g.*, lists or trees of unknown size), or if they serialize operations of potentially unbounded or unknown runtime (*e.g.*, vendor-provided "black-box" functionality), then it is not possible to derive *a priori* bounds on the maximum duration that lower-priority tasks remain effectively non-preemptive. This renders priority-boosting-based locking protocols unviable for real-time systems that need to be certified, even if the developer is certain that all lower-priority critical sections are "short enough." Due to the lack of isolation under priority boosting, a small, but hard-to-quantify risk remains, which can pose considerable challenges in the certification process. Use of a predictable, independence-preserving locking protocol allows to categorically rule out such concerns.

Second, in open systems (*i.e.*, in systems in which the final task set composition is unknown at design time), in RTOSs based on commodity OSs, and in real-time applications incorporating commodity libraries (*e.g.*, for common tasks such as encryption, compression, or signal processing), locks acquired by lower-priority (or even best-effort) tasks often stem from (a large volume of) untrusted code. In this case, it is (at least economically) impossible to rule out the existence of unknown, potentially dangerous critical sections. For instance, this is a major problem in real-time Linux variants, since the Linux kernel contains millions of lines of code of varying quality (device drivers are notorious for above-average defect densities [19]), and because real-time locking primitives are exported to *all*, potentially untrusted user-space tasks (via the *pthreads* library). With an independence-preserving locking protocol, tasks remain fully preemptive and it is sufficient to inspect and test only the (few) parts of the kernel actually used by latency-sensitive tasks.

Finally, current industrial practice heavily favors fully preemptive kernel designs and uses scheduling latency (*i.e.*, how quickly a newly released high-priority job can be sched-

[3] Def. 2 is specific to s-oblivious analysis. For technical reasons, s-aware analysis requires a slightly weaker definition (a subject of future work).

uled) as a key evaluation metric. For example, the non-profit *Open Source Automation Development Lab* reports long-term, large-scale scheduling latency measurements of Linux-based real-time systems to demonstrate Linux's viability as an RTOS [29]. Similarly, commercial RTOSs are commonly advertised as sporting fully preemptive designs and negligible scheduling latencies. Case in point, a prominently advertised feature of the commercial *ThreadX* uniprocessor RTOS is that "[a] high priority thread starts responding to an external event on the order of the time it takes to perform a highly optimized *ThreadX* context switch" [23]. Without an independence-preserving locking protocol, it will be impossible to extend such claims to partitioned scheduling.

Taken as a whole, these examples reveal a clear need for practical real-time multiprocessor semaphore protocols that are both *predictable* and *independence-preserving*. However, in the case of partitioned and clustered scheduling, no such protocols have been proposed to date. To address this gap, we present the first design and analysis of an independence-preserving real-time semaphore protocol for clustered scheduling in the remainder of this paper.

4 The OMIP for Clustered JLFP Scheduling

Real-time semaphore protocols must deal with two central issues: how to avoid excessive pi-blocking when lock-holding jobs are preempted (Sec. 4.1), and how to order conflicting requests such that pi-blocking is minimized to the extent possible (Sec. 4.2). We begin with the former aspect.

4.1 An Independence-Preserving Progress Mechanism

As reviewed in Sec. 1, all prior real-time semaphore protocols for partitioned and clustered scheduling use variants of priority boosting to ensure lock-holder progress, which is not independence-preserving. A new progress mechanism is thus required. Interestingly, the example schedule in Fig. 1(c) contains an *inter-cluster migration* (*i.e.*, job J_2 must temporarily execute in cluster C_1 despite T_2 being assigned to C_2), which runs counter to the idea of clustered scheduling. Such migrations are in fact unavoidable.

Theorem 1. *Under clustered JLFP scheduling with $c \neq m$, it is impossible for a semaphore protocol to simultaneously **(i)** prevent unbounded pi-blocking, **(ii)** be independence-preserving, and **(iii)** avoid inter-cluster migrations.*

Proof. Consider Fig. 1. When job J_3 issues its request at time 2, any locking protocol that does *not* employ inter-cluster migrations has only two options: either it forces J_2 to complete its critical section at the expense of delaying J_1, as shown in Fig. 1(a), or the locking protocol can isolate J_1 from any locking-related delays, as shown in Fig. 1(b). In the former case, the locking protocol is not independence-preserving, in the latter case, J_3 is subject to unbounded pi-blocking because b_3 then depends on J_1's execution cost e_1 (recall that pi-blocking is considered "bounded" only if it does not depend on job execution costs). □

Priority boosting satisfies properties (i) and (iii) at the expense of (ii), whereas priority inheritance satisfies properties (ii) and (iii) at the expense of (i). For our purpose, however, we require a progress mechanism that satisfies both properties (i) and (ii), which according to Theorem 1 is only possible at the expense of (iii). As it turns out, such a mechanism is easily obtained by generalizing the idea of priority inheritance to "cluster inheritance," that is, inheritance should not only pertain to scheduling priorities, but also to *where* a job is eligible to execute, a concept that we refer to as "migratory priority inheritance" [14].

Def. 3. Let W_i denote the set of jobs waiting for a job J_i to release a resource. Under *migratory priority inheritance*, whenever J_i is not scheduled (but ready) and there exists a job $J_x \in W_i \cup \{J_i\}$ such that J_x is eligible to be scheduled in its assigned cluster (*i.e.*, whenever J_i is preempted and there are fewer than c ready higher-effective-priority jobs in J_x's cluster), J_i migrates to J_x's cluster (if necessary) and assumes J_x's priority. After releasing all locks, J_i migrates back to its assigned cluster (if necessary).

Paraphrased, the idea is to move preempted, lock-holding jobs among clusters such that they are always local to a waiting job that would be scheduled if it were not suspended (if any). Migratory priority inheritance may be triggered whenever a lock-holding job J_i is preempted, when a new waiter is added to W_i, or when one of the jobs in W_i becomes one of the c highest-effective-priority jobs in its cluster (due to the completion or suspension of a higher-priority job). If there are multiple candidates in $W_i \cup \{J_i\}$, then J_x may be chosen arbitrarily among the candidates (*e.g.*, cache locality may be taken into consideration).

Migratory priority inheritance is a straightforward generalization of classic priority inheritance [34]; it can also be understood as a greatly simplified, reduced-to-the-core[4] version of Faggioli *et al.* multiprocessor bandwidth inheritance [24]. Notably, a progress mechanism akin to migratory priority inheritance appeared as early as 2001 in TU Dresden's Fiasco microkernel under the name "local helping" [26] (a more detailed discussion of these techniques can be found in [14]). While migratory priority inheritance has received surprisingly little attention in the real-time literature so far, it is ideal for our purposes as ready jobs are never pi-blocked since lock holders remain fully preemptive.

Lemma 1. *In a mutual-exclusion semaphore protocol based on migratory priority inheritance, if a job J_i is ready at time t, then it does not incur s-oblivious pi-blocking at time t.*

Proof. Follows from the fact that priorities are not "duplicated" in a mutual-exclusion protocol. For each job that inherits a priority higher than J_i's base priority in cluster $C(T_i)$, there exists at least one suspended, pending job local

[4] The MBWI [24] also uses (non-optimal) FIFO queueing and allows for budget overruns, which is orthogonal to the idea of migrating lock holders.

to $C(T_i)$ with base priority higher than J_i's base priority. If J_i is ready but not scheduled at time t, then there exist c jobs with higher effective priorities, which, due to the one-to-one inheritance relationship (and because priorities are inherited only locally after migrating to the blocked job's cluster), implies the existence of c pending higher-base-priority jobs local to $C(T_i)$, which precludes s-oblivious pi-blocking. □

However, migratory priority inheritance by itself is *not* sufficient to guarantee independence preservation. For example, if combined with the P-OMLP's contention token abstraction, requests of otherwise independent jobs could still conflict (*i.e.*, contention tokens create artificial dependencies). Rather, a new locking protocol designed specifically to take advantage of migratory priority inheritance is required.

4.2 Queue Structure for Optimal Waiting Times

Asymptotic optimality under s-oblivious analysis requires $\max_\tau b_i = O(m)$. Clearly, this cannot be achieved using only FIFO queues, which would yield $\Omega(n)$ pi-blocking, nor can it be achieved using only priority queues, which are susceptible to starvation and can be shown to yield $\Omega(m \cdot n)$ pi-blocking [13]. However, a hybrid FIFO/priority queue design *can* be used to obtain $O(m)$ pi-blocking bounds under *global* scheduling [12]. While such a hybrid queue cannot be *directly* applied to clustered scheduling (this would require comparing priorities across cluster boundaries, which, analytically, is meaningless), it can be used *within* each cluster. Further, while it is not possible to directly apply the P-OMLP's per-processor contention token design, it is beneficial to reuse the underlying idea of multiple "stages" of contention; that is, contention should be resolved first on an intra-cluster basis (where priorities can be safely compared) and only thereafter on an inter-cluster basis. Based on these considerations, the $O(m)$ *independence-preserving locking protocol* (OMIP) is defined as follows.

Structure. Each shared resource ℓ_q is protected by a global FIFO queue GQ_q of maximum length $\frac{m}{c}$. The job at the head of GQ_q holds ℓ_q. Access to GQ_q is resolved on a per-cluster and per-resource basis: in each cluster C_k, there exist another two queues for each ℓ_q, a bounded-length FIFO queue $\text{FQ}_{q,k}$ of maximum length c that feeds into GQ_q, and a priority queue $\text{PQ}_{q,k}$ that feeds into $\text{FQ}_{q,k}$.

Rules. Requests for each ℓ_q are satisfied as follows. Let J_i denote a job of a task assigned to cluster C_k. Conceptually, J_i first enters its local $\text{PQ}_{q,k}$ and then advances through the queues until it becomes the head of GQ_q.

- **R1** When J_i issues a request for ℓ_q and $\text{FQ}_{q,k}$ is empty, then J_i is enqueued in both GQ_q and $\text{FQ}_{q,k}$. Otherwise, if there are fewer than c jobs queued in $\text{FQ}_{q,k}$, then J_i is enqueued only in $\text{FQ}_{q,k}$. Finally, if there are already c jobs queued in $\text{FQ}_{q,k}$, then J_i is enqueued in $\text{PQ}_{q,k}$.
- **R2** J_i's request for ℓ_q is satisfied when it becomes the head of GQ_q. J_i is suspended while it waits (if necessary).

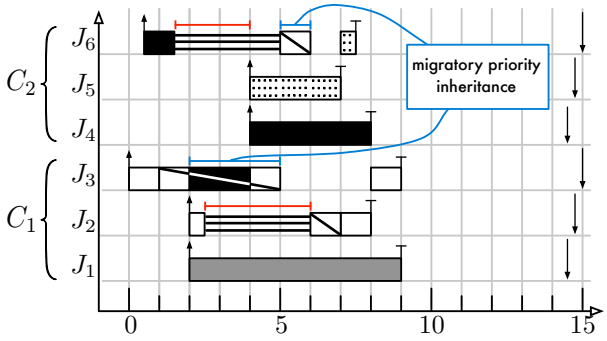


Figure 2: C-EDF schedule of six jobs in two two-processor clusters sharing one resource ℓ_1 under the OMIP. Changes in shade indicate migrations; jobs J_3 and J_6 migrate across cluster boundaries due to migratory priority inheritance. See Fig. 1 for a legend.

- **R3** While J_i holds ℓ_q, it benefits from migratory priority inheritance (w.r.t. any job waiting to acquire ℓ_q).
- **R4** When J_i releases ℓ_q, it is dequeued from both GQ_q and $\text{FQ}_{q,k}$. If $\text{PQ}_{q,k}$ is non-empty, then the head of $\text{PQ}_{q,k}$ is transferred to $\text{FQ}_{q,k}$. Further, if $\text{FQ}_{q,k}$ is non-empty, then the new head of $\text{FQ}_{q,k}$ is enqueued in GQ_q. The new head of GQ_q, if any, is resumed.

Rules R1–R4 together ensure $O(m)$ s-oblivious pi-blocking, which we show in Sec. 4.3 below after a brief example.

Example. Fig. 2 shows an example OMIP schedule under C-EDF scheduling with $m = 4$ and $c = 2$. Job J_3 enters both GQ_1 and $\text{FQ}_{1,1}$ and acquires ℓ_1 at time 1 since ℓ_1 is uncontested (Rule R1). Job J_6 requests ℓ_1 shortly thereafter and enqueues in GQ_1 and $\text{FQ}_{1,2}$ (Rule R1), but must suspend since J_3 holds ℓ_1 (Rule R2). At time 2, J_3 is preempted due to the release of J_1 and J_2. As J_3 holds ℓ_1, it is subject to migratory priority inheritance and migrates to J_6's cluster, where it is scheduled on processor 3 (Rule R3). J_2 proceeds to request ℓ_1 as well and enqueues in $\text{FQ}_{1,1}$, but suspends without enqueuing in GQ_1 because $\text{FQ}_{1,1}$ already contained J_3 (Rule R1). J_3 is now eligible to execute in C_1 again, but remains scheduled in C_2 since Def. 3 only takes effect when J_3 is preempted. At time 4, jobs J_4 and J_5 are released in C_2, which implies that J_6 is no longer among the $c = 2$ highest-priority pending jobs. J_3 is thus preempted and migrates back to C_1 (Def. 3), where it inherits J_2's priority (Rule R3). When J_3 releases ℓ_1 at time 5, it is removed from both GQ_1 and $\text{FQ}_{1,1}$, and the new head of $\text{FQ}_{1,1}$, which is J_2, is enqueued in GQ_1 (Rule R4). J_2 does not immediately acquire ℓ_1 because it is preceded by J_6 (Rule 2). However, due to the presence of J_4 and J_5, J_6 is no longer of sufficient priority to be scheduled in C_2. Hence, migratory priority inheritance takes effect (Rule 3) and J_6 migrates to C_1 to execute its critical section. At time 6, J_6 releases ℓ_1 and ceases to benefit from migratory priority inheritance; it is thus preempted and not scheduled until time 7 when J_5 completes. J_2 finally acquires ℓ_1 as it becomes the head of

GQ_1. Importantly, the example shows that jobs J_1, J_4, and J_5, which do not require ℓ_1, do not incur any pi-blocking.

4.3 PI-Blocking Optimality

We bound the maximum s-oblivious pi-blocking incurred by a given task T_i under the OMIP. In the following, let J_i denote a job in cluster C_k that has requested resource ℓ_q. We begin with the observation that migratory priority inheritance ensures lock-holder progress.

Lemma 2. *Let J_h denote the job at the head of GQ_q (i.e., the job that holds ℓ_q). If J_i incurs s-oblivious pi-blocking at time t, and if J_h requires processor service at time t (i.e., if J_h is ready), then J_h is scheduled.*

Proof. Since J_i incurs s-oblivious pi-blocking, it is among the c highest-priority jobs in C_k. Therefore, J_i would be eligible to execute if it were not suspended. By Rule R3, J_h is subject to migratory priority inheritance. Thus, by Def. 3, if J_h requires processor service to proceed (*i.e.*, if J_h is ready), then J_h migrates to $C(T_i)$ (if required), inherits J_i's priority, and is thus scheduled by a processor in $C(T_i)$. □

Lemma 2 implies that transitive pi-blocking due to processor unavailability is impossible, which allows us to bound maximum pi-blocking in terms of the number of critical sections that must complete before J_i acquires ℓ_q. Due to the multi-stage structure of the OMIP, we proceed in reverse order, starting with jobs already enqueued in GQ_q.

Lemma 3. *Once J_i is enqueued in GQ_q, at most $\frac{m}{c} - 1$ requests for ℓ_q must complete before J_i holds ℓ_q.*

Proof. Due to the interplay of Rules R1 and R4, at most one job per cluster is enqueued in GQ_q at any time. Thus, as there are $\frac{m}{c}$ clusters and since GQ_q is FIFO-ordered, at most $\frac{m}{c} - 1$ critical sections must complete before J_i becomes the head of the queue and its request is satisfied (Rule 2). □

This also bounds how long J_i remains the head of $\text{FQ}_{q,k}$.

Lemma 4. *Once J_i becomes the head of $\text{FQ}_{q,k}$, at most $\frac{m}{c}$ requests for ℓ_q must complete before J_i is dequeued.*

Proof. Follows from Lemma 3 and the fact that J_i must complete its own request before being dequeued. □

This yields a bound on how long it takes to traverse $\text{FQ}_{q,k}$.

Lemma 5. *Once J_i is enqueued in $\text{FQ}_{q,k}$, at most $\frac{m}{c} \cdot (c-1)$ requests for ℓ_q must complete before J_i enters GQ_q.*

Proof. J_i enters GQ_q when it becomes the head of $\text{FQ}_{q,k}$ (Rules R1 and R4). Due to the bounded length of $\text{FQ}_{q,k}$, at most $c-1$ jobs precede J_i in $\text{FQ}_{q,k}$. By Lemma 4, each such job, once it has become the head of $\text{FQ}_{q,k}$ is dequeued after at most $\frac{m}{c}$ requests complete. Thus there are no jobs remaining in front of J_i after $\frac{m}{c} \cdot (c-1)$ requests complete. □

The final step is to bound the pi-blocking that J_i incurs while waiting in $\text{PQ}_{q,k}$. Since priority queues do not offer strong progress guarantees, simply counting preceding jobs is insufficient in this case and an indirect proof is required.

Lemma 6. *While J_i waits in $\text{PQ}_{q,k}$, at most m requests for ℓ_q must complete until either J_i enters $\text{FQ}_{q,k}$ or J_i stops to incur s-oblivious pi-blocking.*

Proof. Suppose not. Then there exists a point in time t at which more than m requests have completed since J_i entered $\text{PQ}_{q,k}$ and J_i still waits in $\text{PQ}_{q,k}$ and incurs pi-blocking. By Lemma 4, a job at the head of $\text{FQ}_{q,k}$ is dequeued after at most $\frac{m}{c}$ requests complete. Thus, at time t, at least $m/\frac{m}{c} = c$ jobs have been dequeued from $\text{FQ}_{q,k}$ since J_i entered $\text{PQ}_{q,k}$. By Rule R4, the highest-priority job in $\text{PQ}_{q,k}$ is transferred to $\text{FQ}_{q,k}$ when the head of $\text{FQ}_{q,k}$ is dequeued. Since J_i is still in $\text{PQ}_{q,k}$ at time t, a job with higher priority has been moved to $\text{FQ}_{q,k}$ at least c times. Therefore, each job in $\text{FQ}_{q,k}$ has a higher priority than J_i, that is, there exist c higher-priority pending jobs in cluster C_k at time t, which precludes s-oblivious pi-blocking (recall Def. 1). Contradiction. □

Having bounded maximum pi-blocking during each of the OMIP's "stages" that a job traverses, we obtain the following overall bound on per-resource s-oblivious pi-blocking.

Lemma 7. *J_i incurs at most $b_{i,q} = N_{i,q} \cdot (2m-1) \cdot L_q^{max}$ s-oblivious pi-blocking due to requests by any task for ℓ_q.*

Proof. By Lemma 1, J_i incurs pi-blocking only while suspended. Consider a single request for ℓ_q. While J_i is suspended waiting to acquire ℓ_q, the amount of pi-blocking incurred is determined by the number of conflicting requests that precede J_i's request. By Lemma 6, at most m requests must complete before J_i enters $\text{FQ}_{q,k}$ or ceases to incur pi-blocking in $\text{PQ}_{q,k}$. By Lemmas 3 and 5, at most $\frac{m}{c} - 1 + \frac{m}{c} \cdot (c-1) = m-1$ requests must complete from the time that J_i enters $\text{FQ}_{q,k}$ until it holds ℓ_q. Lemma 2 implies that the job holding ℓ_q progresses towards completion of its request whenever J_i incurs s-oblivious pi-blocking. Thus, after J_i has incurred $(m + m - 1) \cdot L_q^{max}$ s-oblivious pi-blocking, at least $2m-1$ requests must have completed and J_i holds ℓ_q. The stated bound follows from considering the maximum number of requests issued by any J_i for ℓ_q. □

Note that Lemma 7 implies that the OMIP is indeed independence-preserving (recall Def. 2). Finally, we obtain the following bound on total s-oblivious pi-blocking.

Theorem 2. *Under clustered JLFP scheduling with the OMIP, a job J_i incurs in total at most $b_i = \sum_q b_{i,q} = \sum_q N_{i,q} \cdot (2m-1) \cdot L_q^{max} = O(m)$ s-oblivious pi-blocking.*

Proof. Follows from Lemma 7 since $b_i = \sum_q b_{i,q}$. □

Prior work has established $m-1$ as a simple per-request lower bound under s-oblivious analysis [12]; the OMIP is thus optimal within a factor of $\frac{2m-1}{m-1} = 2 + \frac{1}{m-1} \approx 2$.

5 Implementation and Evaluation

We modified the existing C-EDF plugin in LITMUS$^{\text{RT}}$ [1], a real-time extension of the Linux kernel, to incorporate migratory priority inheritance, the OMIP, and the OMLP. The OMIP's multi-level queue structure is easily realized by

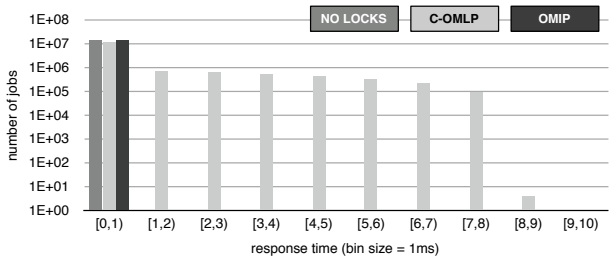
combining multiple *wait queues*, which are readily available in Linux. Most complexity stems from the need to coordinate scheduling decisions across cluster boundaries when a job is eligible to be scheduled in multiple clusters due to migratory priority inheritance. In particular, if a processor decides to *not* schedule a lock-holding job J_l because it observes that J_l is already scheduled in another cluster, the processor must be notified if J_l is subsequently preempted. In our prototype, this is realized with a per-semaphore bitmask that tracks on which processors a job holding the semaphore may currently be scheduled, and by sending an *inter-processor interrupt* (IPI) to available processors when a lock-holder preemption occurs. A migration, if possible, is then carried out by the first available processor to reschedule in response to the IPI. A detailed discussion of the implementation is omitted here due to space constraints; however, our changes are publicly available as a patch on the LITMUS$^{\mathrm{RT}}$ homepage [1].

5.1 Response-Time Measurements

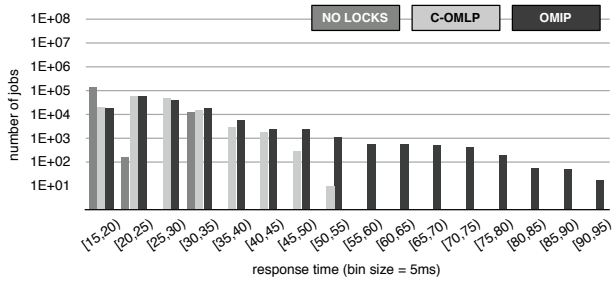
Based on LITMUS$^{\mathrm{RT}}$, we conducted an experiment to empirically demonstrate that the OMIP is effective at protecting latency-sensitive tasks, that is, to rule out the possibility that independence-preservation is merely an "analytical trick" without practical impact. To this end, we configured a simple synthetic task set on a 2.0 GHz Intel Xeon X7550 system with $m = 8$ cores using the modified LITMUS$^{\mathrm{RT}}$ C-EDF plugin with clusters defined by the L1 cache topology, which results in P-EDF scheduling ($c = 1$) on the test platform.

On each core, we launched four tasks with periods $1ms$, $25ms$, $100ms$, and $1000ms$ and execution costs of roughly $0.1ms$, $2ms$, $15ms$, and $600ms$, resp. The latency-sensitive, one-millisecond tasks did not access any shared resources. All other tasks shared a single lock with an associated maximum critical section length of approximately $L^{max} = 1ms$, and each of their jobs acquired the lock once. While the task set is synthetic, the chosen parameters are inspired by the range of periods found in automotive systems [16, 37] and we believe them to be a reasonable approximation of heterogenous timing constraints as they arise in practice.

We ran the task set once using the C-OMLP, once using the OMIP, and once using no locks at all (as a baseline assuming that all tasks are independent) for 30 minutes each. We traced the resulting schedules using LITMUS$^{\mathrm{RT}}$'s `sched_trace` facility and recorded the response times of more than $45,000,000$ individual jobs. Fig. 3 shows two histograms of recorded response times under each configuration. Fig. 3(a) depicts response times of the one-millisecond tasks. Due to the short period (and consistent deadline tie-breaking), their jobs always have the highest priority and thus incur no delays at all in the absence of locks (all response times are in the first bin). In contrast, under the C-OMLP, jobs are not protected from pi-blocking due to unrelated critical sections and response times exceeding $8ms$ were observed—priority boosting causes deadline misses in latency-sensitive tasks



(a) Response times of latency-sensitive tasks with period $p_i = 1ms$.



(b) Response times of regular tasks with period $p_i = 100ms$.

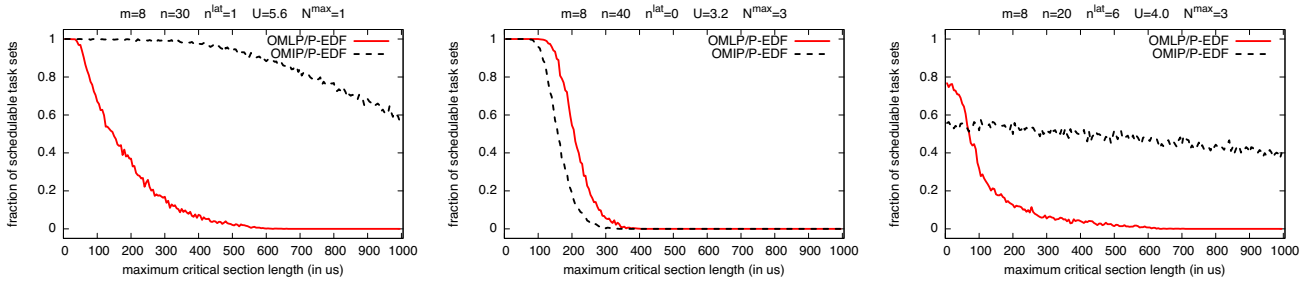
Figure 3: Response times measured in LITMUS$^{\mathrm{RT}}$ under C-EDF scheduling with the OMIP, the C-OMLP, and without locks.

(not coincidentally, $8ms \approx m \cdot L^{max}$). Not so under the OMIP, where the response-times are identical to the case without locks—task independence was indeed preserved.

However, mutual exclusion invariably causes some delays, and if such delays do not manifest in the highest-priority jobs, then they will be necessarily observable elsewhere. This is apparent in Fig. 3(b), which depicts the response-time distribution of the $100ms$-tasks. For these tasks with considerable slack, worst-case response-times are noticeably shorter under the C-OMLP than under the OMIP. This emphasizes that there is an obvious tradeoff between not penalizing higher-priority jobs and rapidly completing critical sections. To explore this tradeoff, we conducted schedulability experiments, which allow *analytical* differences w.r.t. response-time *guarantees* to be quantified (as opposed to the differences in *observed* response times discussed so far).

5.2 Schedulability Experiments

In preparation, we derived and implemented fine-grained (*i.e.*, non-asymptotic) pi-blocking analysis of the OMIP suitable for schedulability analysis based on a recently developed analysis technique using linear programming [11]; details can be found online [10]. To quantify each protocol's impact on schedulability, we generated task sets using Emberson *et al.*'s method [22] consisting of $n \in \{20, 30, 40\}$ tasks with total utilization $U \in \{0.4m, 0.5m, 0.7m\}$. Of the n tasks, $n^{lat} \in \{0, 1, \ldots, 8\}$ were chosen to be latency-sensitive. Latency-sensitive tasks were assigned a period randomly chosen from $[0.5ms, 2.5ms]$, and regular tasks were assigned



(a) A single latency-sensitive task dominates. (b) Overall pi-blocking dominates. (c) Combination of effects.

Figure 4: The effect of increasing maximum critical section length under P-EDF scheduling with the C-OMLP and the OMIP.

a period randomly chosen from $[10ms, 1000ms]$ (in both cases with a granularity of $500\mu s$).

The n^{lat} latency-sensitive tasks shared three resources, which each job accessed once each with maximum critical section lengths chosen uniformly from $[1\mu s, 15\mu s]$. The remaining $n - n^{lat}$ regular tasks shared 12 different resources, of which each task accessed $N^{max} \in \{1, 2, 3\}$, with a maximum critical section length randomly chosen from $[1, mcsl]$, where $mcsl$ is a variable parameter denoting the maximum critical section length. We varied $mcsl \in [5\mu s, 1000\mu s]$ in steps of $5\mu s$ on processor platforms with $m \in \{2, 4, 8, 16\}$ processors and generated for each combination of n, n^{lat}, U, N^{max}, and $mcsl$ at least 1000 task sets (more than $150,000,000$ in total). Each generated task set was schedulability tested under both the C-OMLP and the OMIP assuming P-EDF scheduling (*i.e.*, C-EDF with $c = 1$). The full set of results, 678 schedulability plots when visualized as a function of $mcsl$, is available online [10]; three illustrative examples highlighting two main "bottlenecks" that constrain schedulability are shown in Fig. 4.

Fig. 4(a) shows one of many cases where the OMIP is clearly preferable. In this scenario with one latency-sensitive task (*i.e.*, $n^{lat} = 1$, see figure for all parameters), schedulability under the C-OMLP decreases rapidly as critical section lengths start to exceed $\approx 50\mu s$. The cause is that starting with $mcsl = 50\mu s$, latency-sensitive tasks with a period of $p_i = 500\mu s$ are likely rendered unschedulable by delays due to unrelated critical sections in the range of $m \cdot L^{max} \approx 50\mu s = 400\mu s$, similar to the hand-crafted example discussed in Sec. 5.1. In contrast, the OMIP achieves high schedulability even in the presence of much longer critical sections. In this scenario, latency sensitivity is the bottleneck, and the OMIP is preferable for such workloads.

In contrast, Fig. 4(b) shows a scenario with the opposite outcome. In this case, no latency-sensitive tasks are included in the generated task sets ($n^{lat} = 0$) and there is a high degree of contention ($N^{max} = 3$). As a result, independence-preservation is not essential and constant factors in the pi-blocking analysis dominate: while the C-OMLP and the OMIP are both $O(m)$ protocols, the C-OMLP has a per-request bound of $(m-1) \cdot L^{max}$ [13], whereas the OMIP has a per-request bound of $(2m-1) \cdot L^{max}$ (Lemma 7). The C-OMLP thus achieves somewhat higher schedulability in this case, though long critical sections are infeasible under either protocol due to the high level of contention.

Finally, Fig. 4(c) shows a scenario with $n^{lat} = 6$ and $N^{max} = 3$ where both effects combine. Due to the high level of contention, schedulability under the OMIP is limited across the entire range of critical section lengths, whereas under the C-OMLP, schedulability for short critical sections is markedly higher than under the OMIP due to the C-OMLP's advantage in constant factors, but then decreases rapidly due to the lack of independence preservation under the C-OMLP.

In summary, our results show that independence preservation and the OMIP are clearly beneficial for latency-sensitive workloads. Further, the OMIP is competitive with and often performs similar to the C-OMLP even if independence preservation is not required, but may perform worse than the C-OMLP in the presence of heavy contention.

6 Conclusion

This paper is the first to investigate the challenge of supporting latency-sensitive real-time applications under partitioned and clustered scheduling. We have proposed independence preservation to capture the essential requirement of such workloads. The primary contribution of this paper is the OMIP, the first independence-preserving real-time semaphore protocol for clustered JLFP scheduling. Interestingly, the OMIP employs a novel three-stage queue, which yields asymptotic optimality under s-oblivious analysis.

In summary, latency-sensitive real-time applications have long been well-supported on uniprocessors, but could not be hosted on partitioned multiprocessors with prior semaphore protocols. The OMIP is the first semaphore protocol to overcome this limitation, at the expense of job migrations.

Naturally, job migrations do cause additional overheads, and current industry practice thus regards them critically. However, if job migration is disallowed or fundamentally infeasible (*e.g.*, if the processors in different clusters implement incompatible instruction sets), then Theorem 1 shows that there is little that can be done besides disallowing cross-cluster resource sharing altogether. In contrast, in shared-

memory systems in which global scheduling is technically possible, but a non-global scheduling approach is preferred for other reasons (*e.g.*, to improve cache affinity or to reuse existing uniprocessor algorithms), the OMIP is a viable compromise: first, migratory priority inheritance is triggered only if the lock-holding job is preempted, which means that its cache contents are likely evicted anyway, and second, it is cheaper to migrate critical sections than it is to migrate jobs since the cache footprint of a critical section is typically much smaller than that of an entire job. Concerning the number of times that a critical section may be preempted, migratory priority inheritance is no worse in this regard than the widely deployed PIP. Nontheless, it will be interesting to study the effects of the OMIP on overheads in more detail.

In future algorithmic work, we seek to lift the assumption that jobs lock only one resource at a time. Ward and Anderson [38] recently showed how to support nested critical sections without loss of asymptotic optimality. However, under s-oblivious analysis, their approach permits at most m concurrent (outermost) critical sections by means of a global token lock, which can serialize otherwise independent jobs. Ward and Anderson's technique [38] thus preserves asymptotic optimality when applied to the OMIP, but it is not independence-preserving. However, it may be possible to combine Ward and Anderson's approach with the FMLP's notion of resource groups [7] to obtain independence preservation and asymptotic optimality despite lock nesting.

Finally, we further seek to explore the design space of independence-preserving locking protocols for s-aware analysis, and are currently investigating synchronization mechanisms that provide "freedom from interference" (*i.e.*, temporal isolation) among mutually untrusting tasks, despite resource sharing in the presence of budget overruns, for which the OMIP is an essential building block [9].

References

[1] The LITMUS$^{\mathrm{RT}}$ project. `http://www.litmus-rt.org`.

[2] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proc. of the 19th Real-Time Systems Symposium*, 1998.

[3] T. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[4] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Systems*. Chapman Hall/CRC, 2007.

[5] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proc. of the 28th Real-Time Systems Symposium*, 2007.

[6] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proc. of the 28th Real-Time Systems Symposium*, 2007.

[7] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. of the 13th Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[8] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[9] B. Brandenburg. Virtually exclusive resources. Technical Report MPI-SWS-2012-005, MPI-SWS, May 2012.

[10] B. Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications (extended version). Technical Report MPI-SWS-2013-003, MPI-SWS, May 2013.

[11] B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *Proc. of the 19th Real-Time and Embedded Technology and Applications Symposium*, 2013.

[12] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proc. of the 31st Real-Time Systems Symposium*, 2010.

[13] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, online first, July 2012.

[14] B. Brandenburg and A. Bastoni. The case for migratory priority inheritance in Linux: Bounded priority inversions on multiprocessors. In *Proc. of the 14th Real-Time Linux Workshop*, 2012.

[15] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In *Proc. of the 14th Real-Time and Embedded Technology and Apps. Symposium*, 2008.

[16] D. Buttle. Real-time in the prime-time. Keynote presentation at the 24th Euromicro Conference on Real-Time Systems, July 2012.

[17] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proc. of the 19th Euromicro Conference on Real-Time Systems*, 2007.

[18] Y. Chang, R. Davis, and A. Wellings. Reducing queue lock pessimism in multiprocessor schedulability analysis. In *Proc. of the 18th Conference on Real-Time and Network Systems*, 2010.

[19] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. of the 18th Symposium on Operating Systems Principles*, 2001.

[20] T. Craig. Queuing spin lock algorithms to support timing predictability. In *Proc. of the 14th IEEE Real-Time Systems Symposium*, 1993.

[21] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proc. of the 30th Real-Time Systems Symposium*, 2009.

[22] P. Emberson, R. Stafford, and R. Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2010.

[23] Express Logic, Inc. ThreadX real-time operating system. `http://rtos.com/products/threadx/`, accessed 1/26/2013.

[24] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Proc. of the 22nd Euromicro Conference on Real-Time Systems*, 2010.

[25] P. Gai, M. di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In *Proc. of the 9th Real-Time And Embedded Technology Application Symposium*, 2003.

[26] M. Hohmuth and M. Peter. Helping in a multiprocessor environment. In *Proceeding of the Second Workshop on Common Microkernel System Platforms*, 2001.

[27] G. Macariu and V. Cretu. Limited blocking resource sharing for global multiprocessor scheduling. In *Proc. of the 23rd Euromicro Conference on Real-Time Systems*, 2011.

[28] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proc. of the 23rd Euromicro Conference on Real-Time Systems*, 2011.

[29] Open Source Automation Development Lab eG. Continuous worst-case latency monitoring. `https://www.osadl.org/Continuous-latency-monitoring.qa-farm-monitoring.0.html`, accessed 1/26/2013.

[30] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proc. of the 10th Conference on Dist Computing Systems*, 1990.

[31] R. Rajkumar. *Synchronization In Real-Time Systems—A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[32] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. of the 9th Real-Time Systems Symposium*, 1988.

[33] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proc. of 25th Real-Time Systems Symposium*, 2004.

[34] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[35] H. Takada and K. Sakamura. Predictable spin lock algorithms with preemption. In *Proc. of the 11th Workshop on Real-Time Operating Systems and Software*, 1994.

[36] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Proc. of the 18th Real-Time Systems Symposium*, 1997.

[37] K. Tindell and A. Burns. Guaranteeing message latencies on control area network (CAN). In *Proc. of the 1st Intl. CAN Conference*, 1994.

[38] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proc. of the 24th Euromicro Conference on Real-Time Systems*, 2012.