Virtually Exclusive Resources

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

Technical Report MPI-SWS-2012-005
Completed: May 2012   Published: December 2012

Abstract

Independence preservation, a desirable property in real-time locking protocols that isolates tasks from delays due to unrelated critical sections, is formalized. It is shown that independence preservation is impossible if temporary job migrations are disallowed. The OMIP, a new, asymptotically optimal, independence-preserving locking protocol for clustered scheduling based on migratory priority inheritance, is proposed and analyzed. By extending the OMIP to tolerate budget overruns, it is shown that shared resources can be abstracted into virtually exclusive resources (VXRs)—tasks can be provisioned and composed as if they had exclusive access to each resource, even when sharing resources with erroneous tasks. Several practical extensions and applications of VXRs are described. Notably, VXRs enable predictable resource sharing in mixed-criticality systems.

1 Introduction

Open real-time systems [10, 17], in which the final task set composition and configuration is unknown at development time, are of considerable economic importance. For example, in size-, weight-, and power-constrained domains such as automotive systems or avionics, overall hardware costs may be reduced by integrating independently-developed applications of potentially multiple component vendors on a shared multicore processor. Such workloads are also frequently subject to *heterogenous* timing constraints, that is, open real-time systems often consist of a mix of *hard real-time* (HRT), *soft real-time* (SRT), and *best-effort* (BE) tasks.

Both aspects, compositional system design and timing heterogeneity, require strict *temporal isolation* among tasks: the real-time operating system (RTOS) must ensure that a component's temporal correctness depends only on its own resource consumption. For instance, critical HRT tasks should clearly be immune to faults in tasks of lower importance, even if erroneous tasks exhibit undue resource requirements.

Temporal isolation is typically achieved using *reservation-based scheduling*, under which each task is carefully monitored and policed to not exceed its provisioned processor budget.[1] Even under reservation-based scheduling, however, temporal isolation is difficult to guarantee if tasks access *shared resources* protected by locks (*e.g.*, a shared network stack) since tasks may incur significant blocking when required resources are unavailable. Worse, even when employing a predictable real-time locking protocol that allows bounding maximum blocking *a priori*, the ability to easily compose tasks is lost: with existing locking protocols (see below), it is not possible to determine fixed budgets *at development time* that will always be sufficient *at runtime* because worst-case blocking may increase when additional tasks are admitted. In other words, existing locking protocols fail to temporally isolate tasks that share resources, which renders compositional system design exceedingly difficult.

This paper. The primary contribution of this paper is a new resource-sharing environment that restores temporal isolation by providing the illusion that each task has exclusive access to each resource, which enables effortless, *resource-agnostic* system composition. To this end, we study why certain locking protocols are problematic in open real-time systems (Sec. 2.3) and identify "independence preservation" as a key property (Sec. 3), which, however, is only possible if jobs may (temporarily) migrate to any processor. Based on the observation that migrations are essential, we propose and analyze a new asymptotically optimal, "independence-preserving" locking protocol for clustered scheduling that isolates tasks from unrelated critical sections (Secs. 3.2 and 3.3). This protocol is extended to tolerate budget overruns in Sec. 4, which yields the desired temporal isolation: a task that requires a shared resource with maximum access cost x can be provisioned as if it had a private, but slower replica with maximum access cost $(2m+1) \cdot x$, where m denotes the number of processors (Theorem 3). Finally, several extensions and applications are discussed in Sec. 5, among them predictable locking in mixed-criticality systems [4, 25] and average-case performance improvements (Sec. 5.3).

Related work. Predictable locking in multiprocessor real-time systems has received considerable attention in recent years. Numerous locking protocols have been proposed for partitioned [14, 16, 22, 23], global [7, 11, 12, 18], and clustered real-time multiprocessor scheduling [8]. However, in the context of open, heterogeneous workloads, these protocols for classic HRT systems are subject to two limitations: they do not tolerate budget overruns, and tasks are not temporally isolated (*i.e.*, unrelated critical sections may cause blocking, as explained in Sec. 2.3).

Closely related to this paper is the MSOS protocol proposed by Nemati *et al.* [21], who similarly study the problem

[1]See Sec. 2 for a review of key concepts.

of composing independently-developed real-time applications. However, their approach differs strongly from ours: whereas Nemati *et al*. require each component's resource requirements to be explicitly specified as part of its interface and a non-linear optimization problem to be solved, we focus on *hiding* the effects of resource sharing. Moreover, their MSOS protocol is not resilient to budget overruns.

Our work is inspired in part by Faggioli *et al*.'s *multiprocessor bandwidth inheritance protocol* (MBWI) [13]. Notably, the MBWI is designed for reservation-based scheduling and tolerates budget overruns (Sec. 2.2); we adopt similar mechanisms in Secs. 3 and 4. Our protocol improves upon the MBWI with asymptotically lower interference bounds (Sec. 4.2) and by simplifying task set composition (Sec. 4.4). Another difference is that the MBWI is a spin-based protocol, whereas our protocol is suspension-based since spinning bears the risk of pathological capacity loss in open systems.

Finally, this paper reuses two protocols of the $O(m)$ *locking protocol* (OMLP) family [8] and the associated "suspension-oblivious" analysis [7]. These prior results are central to this paper and reviewed in detail next.

2 Background and Definitions

We consider a workload consisting of n sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ scheduled on m processors. We let T_i denote a task with a *worst-case per-job execution time* and a *minimum job separation* p_i, and let J_i denote a job of T_i. The results presented in this paper do not depend on the type of deadline constraint; we assume implicit deadlines for simplicity.

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.

Under *clustered scheduling* [3, 9], processors are grouped into $\frac{m}{c}$ non-overlapping sets (or *clusters*) of c processors each, where C_k denotes the k^{th} cluster. For simplicity, we assume uniform cluster sizes and that m is an integer multiple of c. Each task is statically assigned to a cluster. *Partitioned* and *global* scheduling are special cases of clustered scheduling with $c = 1$ and $c = m$, respectively.

Each cluster is scheduled independently using a work-conserving *job-level fixed-priority* (JLFP) scheduler, that is, each job is assigned a unique *base priority* and the c highest-priority ready jobs (w.r.t. each cluster) are scheduled at any point in time. We consider *clustered earliest-deadline-first* (C-EDF) scheduling as a representative policy of this class.

2.1 Real-Time Locking

Besides the m processors, tasks share r serially-reusable shared resources $\ell_1, \ldots, \ell_r$ (*e.g.*, network links, sensors, or shared data structures). Access to shared resources is governed by a *locking protocol* that ensures mutual exclusion.

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 locking protocols in which jobs wait by 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 request 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 Secs. 3 and 4 remains valid even if critical sections contain suspensions (*e.g.*, when accessing a disk). For convenience, we define $L_q^{max} \triangleq \max_i\{L_{i,q}\}$ and $L^{max} \triangleq \max_q\{L_q^{max}\}$. We assume that tasks do not hold resources across job boundaries and, in Secs. 3 and 4, that jobs request at most one resource at any time. We discuss supporting nested resource requests in Sec. 5.4.

When locks are used, bounds on *priority inversion blocking* (*pi-blocking*) are required during schedulability analysis. Intuitively, pi-blocking occurs when a high-priority job that should be scheduled is delayed because it incurs acquisition delay. We let b_i denote a bound on the total pi-blocking incurred by any J_i. Previous work has shown that there are two notions of pi-blocking, depending on how job suspensions are accounted for [7]. This work pertains to *suspension-oblivious* (*s-oblivious*) schedulability analysis, where each task's execution cost e_i is inflated by b_i prior to applying a schedulability test [7]. Pi-blocking is defined as follows.

Def. 1. A pending job J_i in cluster C_k incurs *s-oblivious pi-blocking* if J_i is not scheduled and there are fewer than c higher-priority pending jobs (w.r.t. tasks assigned to C_k).

Unless mentioned otherwise, "pi-blocking" refers to the s-oblivious definition in this paper. We base our work on s-oblivious analysis since it is well-suited to providing temporal isolation, as will become apparent in Secs. 3 and 4.

Pi-blocking is considered bounded if b_i can be expressed as a function of L^{max}. To ensure bounded pi-blocking, a locking protocol may have to force a low-priority, resource-holding job to be scheduled if it is causing other jobs to incur pi-blocking. This is accomplished by temporarily raising the resource holder's *effective priority* using either *priority inheritance* [24], where a resource holder's effective priority is the maximum priority of any job that it blocks, or *priority boosting* [22, 23], where resource holders are unconditionally prioritized over non-resource-holding jobs.

Real-time locking protocols aim to minimize b_i; however, some amount of pi-blocking is unavoidable when using locks. Prior work has shown that, under s-oblivious analysis, $\Omega(m)$ *maximum pi-blocking* is generally unavoidable, that is, there exist task sets such that $\max_i\{b_i\} = \Omega(m)$ under any locking protocol [7, 8]. Note that L^{max} and $\sum_q N_{i,q}$ are assumed to be constants (*i.e.*, not a function of n or m) in asymptotic blocking bounds [7, 8].

2.2 Reservation-Based Scheduling

The system model described so far, which underlies Sec. 3, describes classical closed (hard) real-time systems for which all task parameters are known *a priori*. For the heterogeneous

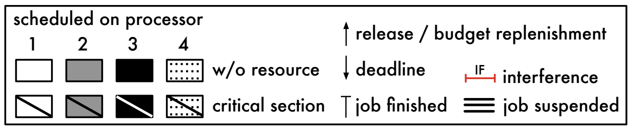


Figure 1: Legend for Figs. 2–5. Up- and down-arrows denote job releases and deadlines (resp.) in Figs. 2–4, and server budget replenishments and server deadlines (resp.) in Fig. 5.

workloads considered in Secs. 4 and 5, it is ill-suited because *reliable* execution cost estimates are likely not available for SRT and BE tasks. Instead, specified parameters are interpreted as permissible budgets (or *reservations*), which are enforced by the RTOS to *temporally isolate* tasks.

Under reservation-based scheduling,[2] each task T_i is encapsulated in a *server* S_i with a specified *budget* Q_i and a *replenishment interval* P_i. Server S_i is *active* when a job of its client T_i is pending. Servers are scheduled like equivalent sporadic tasks (server activation and budget replenishment are analogous to job releases), but prevent client jobs from imposing undue processor demand: if a job J_i requires more than Q_i processor time, it is preempted when S_i exhausts its budget and not scheduled until S_i's budget is replenished. J_i depletes S_i's budget as follows.

Def. 2. An active server *consumes its budget* when it is **(i)** scheduled and a client job is executing or **(ii)** among the c highest-priority servers, but no client job is ready.

Clause (ii) is required to isolate tasks from suspension-related scheduling anomalies. Def. 2 ensures that a HRT task T_i will not miss a deadline if S_i is provisioned with $Q_i \geq e_i$ and $P_i \leq p_i$, provided that the set of servers passes a schedulability test and that T_i does not share resources.

Reservation-based scheduling complicates locking since even a high-priority resource holder may run out of budget, that is, priority inheritance or boosting is insufficient. Instead, *bandwidth inheritance* (BWI) [13, 17, 20] allows a resource holder J_h to consume the budget of a waiting job's server S_w by making J_h temporarily a client of S_w. Such budget transfers cause "interference" [13] because they reduce the budget available to the waiting job, akin to pi-blocking.

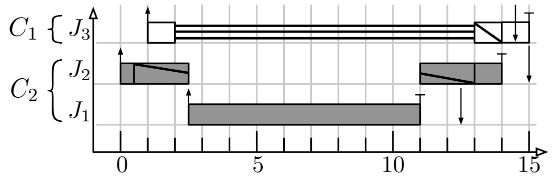
Def. 3. Server S_i experiences *interference* when it consumes budget, but its client job J_i is not executing.

To appropriately provision S_i such that T_i is temporally isolated, it is necessary to bound worst-case interference.

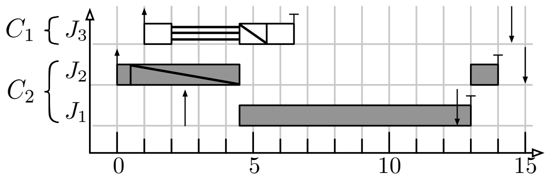
2.3 The Global and the Partitioned OMLP

The protocol proposed in Sec. 3 reuses both the *global* and the *partitioned OMLP* [7] (denoted as G-OMLP and P-OMLP, resp.); we therefore review both variants in detail.

[2] Reservation-based scheduling has been studied extensively for more than 20 years (*e.g.*, [20]) and a comprehensive review is beyond the scope of this paper. For the sake of simplicity, we instead describe a basic approach suitable for any JLFP scheduler and note that the protocol in Secs. 4 and 5 is compatible with more-sophisticated algorithms (*e.g.*, CBS [1]).



(a) Priority inheritance is ineffective across clusters.



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

Figure 2: Schedules of three jobs on two processors under P-EDF scheduling ($c = 1$). Jobs J_2 and J_3 share a resource.

The G-OMLP combines priority inheritance with *hybrid wait queues* [7]: requests for each resource ℓ_q are serialized with a FIFO queue FQ_q of length at most m and a priority queue PQ_q, which is used only when FQ_q is full. When a job is dequeued from FQ_q, the highest-priority job in PQ_q (if any) is transferred from PQ_q to FQ_q (*i.e.*, PQ_q is the tail of FQ_q). Key to the G-OMLP is that even if a job waits for an extended time in PQ_q, it ceases to incur pi-blocking after m higher-priority jobs have entered FQ_q (since the waiting job is no longer among the $c = m$ highest-priority jobs, recall Def. 1) [7, 8]. As a result, at most $m - 1$ requests for ℓ_q cause pi-blocking while a job J_i waits in FQ_q, and at most m requests cause pi-blocking while J_i waits in PQ_q. Maximum pi-blocking is thus bounded by $b_i = \sum_q N_{i,q} \cdot (2m - 1) \cdot L_q^{max} = O(m)$.

Unfortunately, priority inheritance does not generalize to clustered scheduling with $c \neq m$ if applied across cluster boundaries. This limitation arises because priorities are, from an analytical point of view, incomparable across clusters. A simple example of this effect assuming P-EDF is shown in Fig. 2(a), which uses the notation defined in Fig. 1. In Fig. 2(a), even if J_2 inherits J_3's priority (*i.e.*, deadline), J_2 has insufficient priority to prevent preemption by J_1; J_3 thus incurs pi-blocking until J_2 finishes. If priority boosting is used instead as shown in Fig. 2(b), b_3 is bounded by the length of J_2's request, but a deadline is still missed in this example (discussed below). Priority-boosting is the *de facto* standard progress mechanism in real-time locking protocols for partitioned scheduling (*e.g.*, see [7, 16, 22, 23]).

However, as evident in Fig. 2(b), priority boosting may cause pi-blocking as well and should be carefully controlled. Under the P-OMLP, this is achieved with *contention tokens*, of which there is only one per partition. Each resource ℓ_q is protected by a FIFO queue FQ_q, but before a job may enqueue in any FQ_q, it must first acquire its local contention token. This ensures that at most m jobs (one per partition) compete globally at any time. A job may incur additional pi-blocking due to competition for its local contention token, but this can be managed to not exceed the duration of m requests for any resource (see [7, 8] for details). Maximum pi-blocking is thus bounded by $b_i = m \cdot L^{max} + \sum_q N_{i,q} \cdot (m-1) \cdot L_q^{max} = O(m)$ under the P-OMLP (if $c = 1$). In the case of clustered scheduling, a similar bound is ensured by the *clustered OMLP* (C-OMLP) using *priority donation* [8], which is a refinement of priority boosting required for clusters of size $c > 1$.

Note that even though the G-OMLP and the P-OMLP each ensure $O(m)$ pi-blocking, the two bounds are structurally quite different. Whereas independent tasks do not incur any pi-blocking under the G-OMLP (since the G-OMLP uses per-resource queues and priority inheritance), *any* job may incur pi-blocking due to priority boosting under the P-OMLP (and, similarly, under the C-OMLP). For example, in Fig. 2(b), job J_1 misses its deadline due to J_2's request even though J_1 does not require any shared resources.

As argued in Sec. 1, many applications cannot tolerate such lack of isolation. Under global scheduling, it can be avoided (*e.g.*, with the G-OMLP), but all prior suspension-based locking protocols for clustered and partitioned scheduling expose independent tasks to pi-blocking. In the next section, we explore this limitation and determine when it is possible to avoid it, and then present a new locking protocol that protects independent tasks for $1 \leq c \leq m$.

3 Independence-Preserving Locking

The key limitation of the P-OMLP—and any other protocol using priority boosting—is that it creates a dependency among a task's pi-blocking bound and the maximum request length of *any* other task. In contrast, under the G-OMLP, tasks are shielded from requests for resources they do not require. That is, the G-OMLP is "independence-preserving," whereas the P-OMLP is not, which we formalize as follows.

Def. 4. Let $b_{i,q}$ denote an upper bound on pi-blocking incurred by J_i due to requests by any job of any task for resource ℓ_q. A locking protocol is *independence-preserving* if and only if $N_{i,q} = 0$ implies $b_{i,q} = 0$.

Unfortunately, independence preservation is not possible in all circumstances. In fact, the examples shown in Fig. 2 demonstrate that independence preservation is generally *impossible* if job migrations are constrained.

Theorem 1. *If $c \neq m$ and jobs may not migrate across cluster boundaries, then no locking protocol that prevents unbounded pi-blocking is independence-preserving.*

Proof. Consider Fig. 2. When job J_3 issues its request at time 2, any locking protocol has two options: either it forces J_2 to complete its critical section at the expense of delaying J_1, as shown in Fig. 2(b), or the locking protocol can isolate J_1 from any delays due to resource sharing, as shown in Fig. 2(a). In the former case, the locking protocol is not independence-preserving, in the latter case, J_3 is subject to unbounded pi-blocking because it depends on J_1's execution requirement e_1, which can be arbitrarily large in general. □

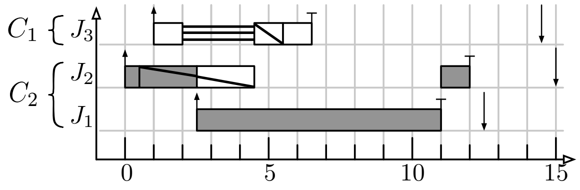


Figure 3: In the same scenario as depicted in Fig. 2, no deadlines are missed if job J_3 temporarily migrates to job J_2's processor.

For systems in which job migration is fundamentally infeasible (*e.g.*, if the processors in different clusters implement incompatible instruction sets), Theorem 1 points a bleak picture: besides using blocking-aware partitioning heuristics, there is little that can be done to improve isolation.

However, this is emphatically not the case 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). In such systems—where job migrations are not impossible, but merely discouraged—independence preservation *is* possible, as we show next.

3.1 Migratory Priority Inheritance

It is easy to see that migrations enable independence preservation. For example, a straightforward "fix" for the scenario shown in Fig. 2 is for J_2 to migrate to J_3's assigned processor while it is blocking J_3, as illustrated in Fig. 3.

This observation immediately leads to a simple extension of priority inheritance for the case $c \neq m$: inheritance should not only apply to *scheduling priority*, but also to a job's *cluster assignment*. That is, instead of inheriting priorities across cluster boundaries, jobs should migrate to ensure that priority inheritance occurs only *locally* within a cluster.

Def. 5. Let W_i denote the set of jobs waiting for a job J_i to release a resource. Under *migratory priority inheritance* (MPI), 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.*, there are fewer than c ready higher-priority jobs in J_x's cluster), J_i migrates to J_x's cluster (if necessary) and inherits J_x's priority.

Paraphrased, MPI moves resource holders among clusters such that resource-holding jobs are always local to a waiting job that would be eligible to be scheduled (if it were not suspended), which is similar to BWI in the MBWI [13]. After releasing all resources, jobs migrate back to their assigned clusters (if necessary). Crucially, MPI ensures progress without causing additional pi-blocking.

Lemma 1. *Under MPI, if a job J_x incurs s-oblivious pi-blocking at time t while waiting for a resource ℓ_q held by job J_i and J_i is ready, then J_i is scheduled at time t.*

Proof. Since J_x incurs s-oblivious pi-blocking at time t, it is among the c highest-priority pending jobs in its cluster. Hence, there are also fewer than c higher-priority ready jobs in J_x's cluster. J_i has thus either migrated to J_x's cluster and is inheriting J_x's priority, or J_i is scheduled elsewhere. □

The following lemma shows that MPI is suitable for the design of independence-preserving locking protocols.

Lemma 2. *In an MPI-based mutex locking protocol, a job J_i incurs s-oblivious pi-blocking only if it is suspended.*

Proof. Let C_k denote J_i's assigned cluster. If J_i is ready and not scheduled, then there are $h \geq c$ ready jobs with higher effective priorities scheduled in C_k. Each of these jobs either belongs to a task assigned to C_k, or is blocking a higher-priority job of a task assigned to C_k. Thus, there exist $h \geq c$ jobs of tasks assigned to C_k that have higher priorities than J_i, which rules out pi-blocking (recall Def. 1). □

Of course, MPI is by itself not sufficient to guarantee bounded pi-blocking or independence preservation. For example, if MPI is combined with the P-OMLP, requests of otherwise independent jobs could still conflict due to the limited number of available contention tokens. Similarly, if the G-OMLP is applied across cluster boundaries using MPI, the protocol would remain independence-preserving, but it would not ensure $O(m)$ pi-blocking since the OMLP's hybrid queues fundamentally require comparing priorities among *all* queued jobs, which remains analytically meaningless if jobs stem from multiple clusters even when using MPI. A new locking protocol designed specifically to take advantage of MPI is thus required.

3.2 An Independence-Preserving Locking Protocol

The $O(m)$ *independence-preserving multiprocessor locking protocol* (OMIP) combines aspects of both the G-OMLP and the P-OMLP. From the P-OMLP, it borrows the concept of contention tokens to limit access to global queues, but uses them on a per-resource basis to preserve task independence. Additionally, it reuses the G-OMLP, but only within each cluster, to serialize requests for each contention token.

Structure. Each shared resource ℓ_q is protected by a global FIFO queue GQ_q. The job at the head of GQ_q holds ℓ_q. Further, for each resource and each cluster, there is a contention token; we let $\mathrm{CT}_{q,k}$ denote the contention token for ℓ_q in cluster C_k. Each contention token is a virtual local resource that is shared among local jobs using the G-OMLP.

Rules. Requests for each ℓ_q are satisfied as follows. Let J_i denote a job of a task assigned to cluster C_k.

I1 To issue a request for ℓ_q, job J_i must first acquire $\mathrm{CT}_{q,k}$ according to the rules of the G-OMLP.

I2 Once J_i holds $\mathrm{CT}_{q,k}$, it is added to GQ_q. J_i's request for ℓ_q is satisfied when it becomes the head of GQ_q.

I3 While J_i holds ℓ_q, it benefits from MPI (with regard to any job currently waiting to acquire ℓ_q, including those not holding a contention token yet).

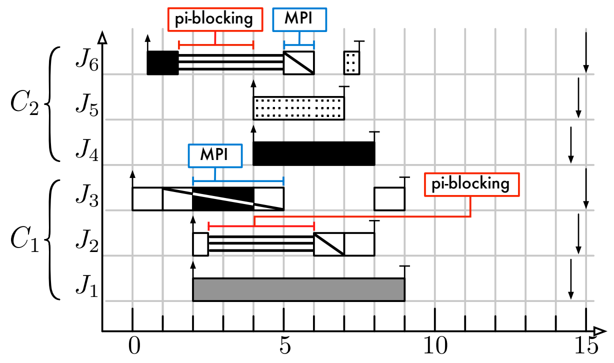


Figure 4: C-EDF schedule of six jobs in two two-processor clusters sharing one resource ℓ_1 under the OMIP.

I4 When J_i releases ℓ_q, it also releases $\mathrm{CT}_{q,k}$ and is dequeued from GQ_q. J_i ceases to benefit from MPI and ℓ_q is passed on to the new head of GQ_q (if any).

Rules I1–I4 ensure $O(m)$ s-oblivious pi-blocking, which is established in Sec. 3.3 below after a brief example.

Example. Fig. 4 shows an example OMIP schedule under C-EDF scheduling with $m = 4$ and $c = 2$. Job J_3 acquires $\mathrm{CT}_{1,1}$ and ℓ_1 at time 1 since ℓ_1 is uncontested. Job J_6 requests ℓ_1 shortly thereafter, acquires $\mathrm{CT}_{2,1}$, and enqueues in GQ_1, but must suspend since J_3 holds ℓ_1. 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 MPI and migrates to J_6's cluster, where it is scheduled on Processor 3. J_2 proceeds to request ℓ_1 as well, but suspends without enqueuing in GQ_1 because $\mathrm{CT}_{1,1}$ is not available. 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 jobs. J_3 is thus preempted and migrates back to C_1, where it inherits J_2's priority instead. When J_3 releases ℓ_1 and $\mathrm{CT}_{1,1}$ at time 5, J_2 acquires $\mathrm{CT}_{1,1}$ and enqueues in GQ_1. J_2 does not acquire ℓ_1 because it is preceded by J_6. However, J_6 is not of sufficient priority to be scheduled in C_2. Hence, due to MPI, J_6 also migrates to C_1 to complete its critical section. At time 6, J_6 releases ℓ_1 and ceases to benefit from MPI; it is thus not scheduled until time 7 when J_5 completes. J_2 finally acquires ℓ_1. Notice how MPI ensures resource holder progress while J_2 incurs pi-blocking and that jobs J_1, J_4, and J_5, which do not require ℓ_1, do not incur any blocking.

3.3 Schedulability Analysis

In general, Rules I1–I4 ensure $O(m)$ pi-blocking under any clustered JLFP scheduler with $1 \leq c \leq m$. Due to the nested structure of the OMIP, the following analysis proceeds in reverse order, starting with jobs already holding a contention token. In the following, let J_i denote a job in cluster C_k that has requested resource ℓ_q.

Lemma 3. *Once J_i holds $\mathrm{CT}_{q,k}$, it incurs pi-blocking for the length of at most $\frac{m}{c} - 1$ earlier-satisfied requests for ℓ_q.*

Proof. Due to MPI (Rule I3), the job holding ℓ_q is scheduled whenever J_i incurs s-oblivious pi-blocking (Lemma 1).

Since each job in GQ_q holds a contention token (Rules I1 and I2), and because there is only one contention token for ℓ_q per cluster, at most $\frac{m}{c} - 1$ jobs precede J_i in GQ_q. □

This in turn bounds the maximum time that a job will hold a contention token if a waiting job incurs pi-blocking.

Lemma 4. *Let J_h denote a job holding $\text{CT}_{q,k}$ (where $J_h \neq J_i$). While J_h holds $\text{CT}_{q,k}$, J_i incurs s-oblivious pi-blocking for the length of at most $\frac{m}{c}$ requests for ℓ_q.*

Proof. Analogously to Lemma 3, J_i incurs pi-blocking for the length of at most $\frac{m}{c} - 1$ requests while J_h traverses GQ_q. Further, J_i incurs pi-blocking for at most the length of J_h's request while J_h holds ℓ_q. Hence, at most $(\frac{m}{c} - 1) + 1 = \frac{m}{c}$ requests for ℓ_q delay J_i while J_h holds $\text{CT}_{q,k}$. □

In other words, the maximum request length with respect to each virtual resource $\text{CT}_{q,k}$ is $\frac{m}{c} \cdot L_q^{max}$.

Lemma 5. *J_i incurs s-oblivious pi-blocking for at most the length of $(2m - 1)$ requests for ℓ_q each time it requests ℓ_q.*

Proof. By Lemma 3, *after* J_i acquires $\text{CT}_{q,k}$, it incurs pi-blocking for the duration of at most $\frac{m}{c} - 1$ request lengths (with respect to ℓ_q). Additionally, *before* J_i acquires $\text{CT}_{q,k}$, job J_i incurs s-oblivious pi-blocking for the duration of at most $2c - 1$ "token request lengths" (*i.e.*, request lengths with regard to the virtual resource $\text{CT}_{q,k}$) since access to $\text{CT}_{q,k}$ is governed by the G-OMLP and because J_i's cluster contains c processors. By Lemma 4, the maximum "token request length" (in terms of pi-blocking incurred by J_i) is bounded by $\frac{m}{c}$ requests for ℓ_q, for a total of $(2c - 1) \cdot \frac{m}{c}$ blocking requests due to $\text{CT}_{k,q}$. Thus, maximum pi-blocking is bounded by the lengths of $(\frac{m}{c} - 1) + (2c - 1) \cdot (\frac{m}{c}) = (\frac{m}{c} - 1) + (2m - \frac{m}{c}) = 2m - 1$ requests for ℓ_q. □

The OMIP is thus independence-preserving and asymptotically optimal under s-oblivious analysis.

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

Proof. Follows from Lemmas 2 and 5 and the analysis assumptions that $\sum_q N_{i,q}$ and L_q^{max} are constant. □

As an aside, the presented analysis does not require the employed JLFP policy to be the same in all clusters. It is not assumed that deadlines are met (*i.e.*, the presented analysis is valid even if jobs are tardy).

The OMIP preserves independence among tasks and ensures predictable pi-blocking for non-independent tasks, provided that tasks do not hold resources across job boundaries and that jobs do not exceed their specified execution cost. The OMIP is hence well-suited for closed HRT and SRT environments. However, as discussed in Sec. 1, stronger resiliency to faults is required in open and heterogenous real-time systems. We present a new OMIP-based solution suitable for such demanding environments next.

4 The Illusion of Exclusive Access

Open and heterogenous real-time systems are challenging from a resource-sharing perspective because the final task set composition is unknown at analysis time (*i.e.*, the number of tasks n and some of the $N_{i,q}$ parameters may be unknown or wrong), and because shared resources may be accessed by SRT and BE tasks for which only imprecise execution time estimates are known (*i.e.*, some of the e_i and $L_{i,q}$ parameters may underestimate actual demand). To guarantee temporal isolation in such environments, each task must be protected against unexpected resource requirements by any other task.

To this end, we introduce *virtually exclusive resources* (VXRs)—a predictable resource-sharing environment that ensures complete temporal isolation even if resources are shared with erroneous or unpredictable tasks. In particular, the VXR environment *hides* resource sharing and allows tasks to be provisioned as if they had exclusive access to a private (but slower) replica of each shared resource.

The VXR environment combines the OMIP with reservation-based scheduling, which requires augmenting MPI with BWI [13, 17] and adding rules to cope with budget overruns, which we introduce next.

4.1 Tolerating Budget Overruns

Recall from Sec. 2.2 that reservation-based scheduling enforces temporal isolation by delaying the completion of a job that overruns its specified execution cost until its server's budget is replenished. If a job J_i exhausts its server's budget during a request for a resource ℓ_q, there are two possibilities: either J_i already holds ℓ_q (*i.e.*, it is currently the head of GQ_q), or it is still suspended and waiting to acquire ℓ_q. The latter case is easier to handle since J_i has not yet started executing its critical section; J_i can thus be safely dequeued. In the former case, if J_i already holds a resource, then J_i's request must continue since the shared resource may be in an inconsistent state. A resource holder must thus finish its critical section using the budget of waiting jobs (if any).

Recall that each contention token $\text{CT}_{q,k}$ is protected by the G-OMLP, which uses a FIFO and a priority queue to serialize requests for $\text{CT}_{q,k}$. In the following, let $\text{FQ}_{q,k}$ denote the FIFO and $\text{PQ}_{q,k}$ the priority queue corresponding to $\text{CT}_{q,k}$. Budget overruns are handled as follows.

B1 If J_i's server S_i exhausts its budget while J_i is waiting to acquire ℓ_q, then J_i's request for ℓ_q is *aborted*. This causes J_i to be dequeued from all queues related to ℓ_q (*i.e.*, GQ_q, $\text{FQ}_{q,k}$, and $\text{PQ}_{q,k}$), and to no longer participate in BWI (*i.e.*, the job holding ℓ_q may no longer consume S_i's budget). When S_i's budget is replenished, J_i's request is automatically *reissued* (*i.e.*, it is processed as a new request).

B2 If J_i's server S_i exhausts its budget while J_i holds ℓ_q, then J_i is removed from $\text{FQ}_{q,k}$ without releasing $\text{CT}_{q,k}$. The highest-priority job in $\text{PQ}_{q,k}$ (if any) is transferred to $\text{FQ}_{q,k}$, but the new head of $\text{FQ}_{q,k}$ (if any) does not yet enter GQ$_q$. J_i remains an eligible BWI recipient until it releases ℓ_q, at which point it also releases CT$_{q,k}$, which allows the head of FQ$_{q,k}$ (if any) to enter GQ$_q$.

Example. Rules B1 and B2 are illustrated in Fig. 5, which shows a schedule of four servers under reservation-based P-EDF scheduling sharing a resource in the VXR environment. Job J_3 (in server S_3) acquires ℓ_1 at time 1, which causes S_1 to experience interference when its client job J_1 requests ℓ_1 at time 1.5. J_2 (in S_2) requests ℓ_1 shortly thereafter, but cannot enter GQ$_1$ since J_1 holds CT$_{1,2}$. At time 2, S_3 exhausts its budget, which gives S_4 a chance to be scheduled. Since J_3 holds ℓ_1, Rule B2 applies and J_3 continues to execute its critical section using S_1's budget on Processor 2. This in turn exhausts S_1's budget at time 3. Since J_1 does not yet hold ℓ_1, Rule B1 applies and J_1's request is aborted. Hence J_2 acquires CT$_{1,2}$ and enters GQ$_1$ at time 3. J_3 continues to execute since it now inherits budget from S_2. At time 4, J_3 releases ℓ_1 and ceases to benefit from BWI. At time 8, S_1's budget is replenished and J_1's request for ℓ_1 is reissued. J_1 acquires ℓ_1 immediately, but S_1 is not scheduled until time 9 since the higher-priority S_2 must complete first.

Rules B1 and B2 prevent the accumulation of budget-less jobs in queues, which limits interference for jobs that do *not* exhaust their server's budget, which we show next.

4.2 Bounding Interference

Recall from Sec. 2.2 that, under reservation-based scheduling, interference reflects the budget loss that a server experiences while its job waits for a resource. Since interference is closely related to pi-blocking, the following analysis is in large parts analogous to the analysis of the OMIP in Sec. 3.3, with the key difference that budget overruns must be taken into account. In the following, we let J_i denote a job that has requested ℓ_q and assume that its server S_i does not deplete its budget during J_i's request.

Lemma 6. *While J_i is waiting in GQ$_q$, S_i experiences interference due to at most $\frac{m}{c} - 1$ requests for ℓ_q.*

Proof. If no job preceding J_i in GQ$_q$ exhausts its server's budget, the bound follows analogously to Lemma 3. If a preceding job is removed from GQ$_q$ due to Rule B1, then J_i's acquisition delay and hence interference is only reduced. If a job preceding J_i in GQ$_q$ exhausts its budget while holding ℓ_q, J_i's worst-case interference is not affected since resource holders are eligible to execute using S_i's budget anyway. □

Budget overruns are thus unproblematic for jobs already waiting in GQ$_q$. This allows bounding the amount of interference while J_i waits for CT$_{q,k}$.

Lemma 7. *While J_i is waiting in FQ$_{q,k}$, S_i experiences interference due to at most $m - \frac{m}{c} + 1$ requests for ℓ_q.*

Proof. Analogously to Lemma 4, Lemma 6 implies that a job J_h at the head of FQ$_{q,k}$ holding CT$_{q,k}$ will release CT$_{q,k}$ after S_i experiences interference due to at most $\frac{m}{c}$ requests while J_h traverses GQ$_q$. If J_i enters FQ$_{q,k}$ normally (*i.e.*, not due to Rule B2), then at most $c-1$ jobs precede J_i in FQ$_{q,k}$ (and fewer if any are removed due to Rule B1) and S_i experiences interference due to at most $(c-1)\cdot\frac{m}{c}$ requests while jobs in FQ$_{q,k}$ hold CT$_{q,k}$. Otherwise, in the worst case, J_i enters FQ$_{q,k}$ when the current token holder is removed from FQ$_{q,k}$ without releasing CT$_{q,k}$ due to Rule B2. In this case, S_i experiences additional interference for at most one request length until the resource holder relinquishes CT$_{q,k}$. Hence, in total, at most $(c-1)\cdot\frac{m}{c}+1 = m-\frac{m}{c}+1$ earlier-satisfied requests interfere while J_i traverses FQ$_{q,k}$. □

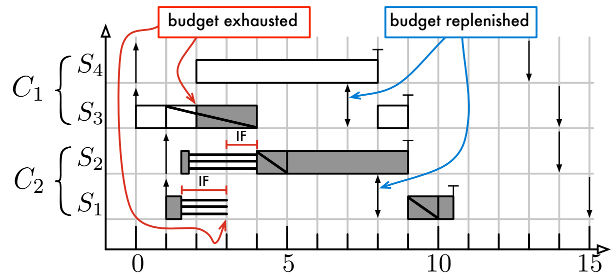


Figure 5: P-EDF schedule of four servers (S_1–S_4) sharing one resource ℓ_1. Servers S_1 and S_3 have parameters $Q_1 = Q_3 = 2$ and $P_1 = P_3 = 7$; servers S_2 and S_4 have parameters $Q_2 = Q_4 = 10$ and $P_2 = P_4 = 14$. The request of job J_1 is aborted at time 3 and reissued at time 8 (Rule B1).

Finally, interference while J_i waits in PQ$_{q,k}$ can be bounded analogously to the argument given in [7].

Lemma 8. *While J_i waits in PQ$_{q,k}$, S_i experiences interference due to at most m requests for ℓ_q.*

Proof. Suppose J_i is still queued in PQ$_{q,k}$ and that S_i experiences interference after at least c jobs in higher-priority servers have entered FQ$_{q,k}$. Since both Rules B1 and B2 remove a job immediately from FQ$_{q,k}$ when its server exhausts its budget, each job in FQ$_{q,k}$ implies the existence of a higher-priority server in S_i's cluster. There are thus c higher-priority servers in S_i's cluster, which precludes interference (recall Defs. 2 and 3). Hence, while J_i waits in PQ$_{q,k}$, S_i does not experience interference after at least c higher-priority servers have entered FQ$_{q,k}$, which requires at most $c\cdot\frac{m}{c} = m$ requests to complete. □

Lemmas 6–8 together bound the maximum per-request interference and hence yield the desired temporal isolation.

Theorem 3. *In the VXR environment, if the set of servers is schedulable, then all jobs of a real-time task T_i with worst-case execution time e_i and period p_i meet their deadline if T_i's server S_i is provisioned with parameters $P_i \le p_i$ and $Q_i \ge e_i + I_i$, where $I_i = \sum_q N_{i,q}\cdot 2\cdot m\cdot L_q^{max}$.*

Proof. Since the OMIP is independence-preserving, servers experience interference only when its client job waits for a resource. By Lemmas 6–8, each time J_i requests a resource ℓ_q, at most $(\frac{m}{c}-1)+(m-\frac{m}{c}+1)+m = 2m$ earlier-satisfied requests for ℓ_q consume S_i's budget while J_i waits for ℓ_q, assuming S_i does not exhaust its budget before J_i

completes. Since each J_i issues at most $N_{i,q}$ requests for each ℓ_q, S_i experiences at most $I_i = \sum_q N_{i,q} \cdot 2 \cdot m \cdot L_q^{max}$ interference. Each J_i requires at most e_i processor time to complete; a budget of $e_i + I_i$ is thus sufficient to ensure that each J_i finishes before S_i exhausts its budget. □

There are several points to observe. First, tolerating budget overruns increases worst-case interference by only one critical section length per request (in comparison to s-oblivious pi-blocking under the OMIP), which seems a low price to pay for the gain in robustness. Second, interference is asymptotically optimal in the VXR environment since it is trivial to show that $\max\{I_i\} = \Omega(m)$ is unavoidable in the general case (analogously to s-oblivious pi-blocking [7]). And finally, note that Rule B2 is essential: if a resource holder that overruns its budget were not removed from $\mathrm{FQ}_{q,k}$, then Lemma 8 would not be true; similarly, if a resource holder would release its contention token immediately, Lemma 6 would not hold.

Theorem 3 expresses strong isolation: maximum interference is bounded regardless of **(i)** the number of tasks sharing a resource, **(ii)** how many times other tasks request any resource, and **(iii)** whether any other task exceeds its allocated budget. That is, the VXR environment allows provisioning a task's server *a priori*, based alone on the number of processors m and each maximum request length L_q^{max}.

It is reasonable to assume that the target processor platform, and hence m, is known at analysis time—this is required in any case to bound the execution cost e_i, using either worst-case execution time analysis or empirical estimates. However, safely bounding L_q^{max} *a priori* requires imposing restrictions on how resources are accessed, as discussed next.

4.3 Bounding L_q^{max}

Lock-based synchronization inherently implies trust, as other tasks may fail to release locks in a timely fashion or even entirely. In particular, if tasks have direct access, then it is generally not possible to derive a safe upper bound on each resource's maximum request length L_q^{max} in open real-time systems (*i.e.*, if the final task set composition is unknown in advance). This implies that untrusted tasks should not have direct access to shared resources, but rather issue requests through a common, trusted mediation layer.

For example, suppose tasks share a device such as a sensor or network link. In a UNIX-like RTOS (such as QNX or Linux variants), tasks typically do not access a device directly; rather, they invoke system calls to perform operations upon the device. The OS kernel and device driver provide a trusted mediation layer that ensures, if implemented properly, that the maximum request length can be bounded *a priori*, regardless of which tasks access the device at runtime.

More generally, if resources are shared among mutually untrusted tasks, each resource should be encapsulated in a server process (in the microkernel sense) and only be accessed using a well-defined set of operations implemented by the server process. Such a mediation layer can also enforce access control policies and guard against certain logical errors (*e.g.*, it can reject nonsensical requests).

If resource access is mediated such that L_q^{max} can be determined at design time (or if all tasks are trusted), then the VXR environment ensures strong temporal isolation that greatly simplifies task admission in open real-time systems.

4.4 Resource-Agnostic System Composition

As pointed out in Sec. 1, the defining characteristic of an open real-time system is that not all tasks are known at design time. Instead, systems are composed from independently-developed components, where each component consists of one or more tasks encapsulated in servers. Strict temporal isolation makes compositional system design possible: if each to-be-added component passes an admission test (*i.e.*, if the set of reservations remains schedulable), then a *properly provisioned* real-time task is guaranteed to satisfy its temporal specification regardless of failures in other tasks. Determining proper budgets for each server is thus essential.

However, using prior protocols, this is challenging if tasks share resources. For example, the MBWI [13] uses FIFO queuing and thus is fundamentally an $\Theta(n)$ protocol, that is, the maximum interference depends on the number of tasks sharing a given resource. This has major implications: if a task requires resources that might be used by other components, then it is impossible to determine a budget in advance that is sufficient in all cases (since the number of tasks sharing a resource is unknown at design time). Instead, each component must declare all its resource requirements (which is tedious for developers) and the admission control mechanism must determine appropriate budgets based on all the specified resource requirements of all admitted servers. Worse, admitting a new component may *invalidate* the budgets of previously-admitted servers that share resources with the newly-admitted server; adding one server may thus require changing the budgets of *all* servers, which in turn may require load balancing (readjusting budgets at runtime is itself a challenging problem [5]). Composition under the MBWI is thus a fairly difficult affair.

In contrast, the VXR environment ensures firm bounds on the maximum interference incurred due to resource sharing *regardless of the number of tasks sharing a particular resource*: a server's budget provisioned according to Theorem 3 is guaranteed to be sufficient in *all* cases. This simplifies system composition greatly since a task's resource requirements do *not* have to be part of its public interface and adding servers does *not* require changing any previously-admitted server's budget. That is, the VXR environment provides the illusion that each task uses a private replica of each resource that is $2m + 1$ times slower than the shared instance—sharing is hence effectively hidden within each server as tasks access only (virtually) exclusive resources.

5 Extensions and Applications

The high degree of predictability in the VXR environment makes it an attractive choice for applications in which isolation is paramount. To emphasize its versatility, we provide an overview of practical extensions and possible applications.

5.1 Resource Sharing in Mixed-Criticality Systems

The VXR environment not only isolates tasks from temporal faults, but also from (otherwise) incompatible differences in analysis assumptions. For example, if a SRT task T_s and a HRT task T_h share a resource ℓ_q, it is possible to use *different* estimates of L_q^{max} when provisioning S_s and S_h (*e.g.*, observed maxima vs. analytic bounds). In the VXR environment, T_h's temporal correctness is not at risk even if the estimate used to provision S_s turns out to be wrong.

The option to mix components that were developed with different levels of rigor is useful in many scenarios, ranging from low-end SRT applications (*e.g.*, isolating multimedia playback tasks from GUI tasks) to safety-critical HRT systems subject to stringent certification requirements.

With regard to the latter, the VXR environment is applicable to the emerging field of *mixed-criticality systems* [4, 25], in which tasks of different *criticalities* (*i.e.*, importance) are co-hosted on a shared hardware platform. For example, consider the hierarchical mixed-criticality scheduler proposed by Anderson *et al.* [2, 15], which, inspired by avionics workloads, supports five criticality levels (denoted A–E). Higher-criticality tasks are subject to much more stringent certification requirements (and hence use more pessimistic cost estimates) than lower-criticality tasks (which may use measured or even estimated parameters).

It is often (economically) infeasible to certify all tasks at the highest level. Instead, certification seeks to establish that level-X tasks are correct if each task's actual execution is consistent with level-X assumptions. For example, if level-C assumptions are exceeded but level-B assumptions are met, then level-C tasks may fail, but level-B tasks must continue to function correctly. VXRs are well-suited for such conditional guarantees: when provisioning servers of level-X tasks using level-X estimates of L_q^{max}, Theorem 3 implies temporal correctness if level-X estimates are met, *even when sharing resources with tasks of lower criticality*.

To the best of our knowledge, the VXR environment is the first to allow predictable sharing of resources among tasks of different criticalities in multiprocessor systems. Notably, Anderson *et al.*'s hierarchical mixed-criticality framework can be interpreted as a JLFP scheduler since higher-criticality servers are statically prioritized over lower-criticality servers, and various JLFP policies may be used to schedule servers within each level; VXR support can thus be easily integrated.

5.2 Resource Access for Budget-Less Tasks

The VXR environment as described so far allows real-time and best-effort tasks to share resources, but only if best-effort tasks are provisioned in servers with non-zero scheduling budgets. However, practical systems often also contain best-effort background jobs scheduled at idle priorities, which are budget-less, that is, for the purpose of Rules B1 and B2, their budget is "continuously exhausted." Such jobs cannot be queued in any $\mathrm{FQ}_{q,k}$ or $\mathrm{PQ}_{q,k}$ since the VXR analysis does not allow budget-less *waiters*. It does, however, already account for budget-less *resource holders*.

This allows background jobs to be integrated as follows.

B3 When a budget-less job J_b in cluster C_k requests a shared resource ℓ_q, it immediately acquires $\mathrm{CT}_{q,k}$ and ℓ_q if ℓ_q is currently available and uncontested; otherwise, J_b is enqueued in the *background queue* BQ_q (no particular ordering of BQ_q is required).

B4 When a resource holder J_h (with or without budget) releases ℓ_q, the global FIFO queue GQ_q is checked: if GQ_q is empty after J_h releases ℓ_q and $\mathrm{CT}_{q,k}$ (*i.e.*, if there are no unsatisfied requests by jobs with budgets), then the head of BQ_q (if any) is dequeued and acquires ℓ_q and the local contention token for ℓ_q.

Note that background jobs still benefit from bandwidth inheritance if they block jobs in servers with non-zero budgets. Rules B3 and B4 ensure that budget-less background jobs acquire a resource only if the resource would otherwise be idle, which reflects their low priority. With this extension in place, the VXR environment facilitates predictable resource sharing among truly *all* tasks; it can thus be employed as the sole locking protocol in an RTOS for mixed workloads.

5.3 Throughput and Interference Improvements

Since a majority of critical sections are short in practice, throughput typically improves if jobs spin briefly before suspending (*e.g.*, Solaris uses such *adaptive mutexes* [19]). While such a spin-first policy does not improve the worst case (in which jobs are assumed to exceed the spinning threshold and eventually suspend anyway), it likely improves average-case resource access costs. The VXR is trivially compatible with this throughput optimization: if the interference definition (Def. 3) is augmented to include budget lost to busy-waiting, then the interference bound derived in Sec. 4.2 remains valid, provided that waiting jobs cease spinning immediately if the resource-holder is preempted. In an RTOS such as LITMUS$^{\mathrm{RT}}$, this can be easily supported.

A reduction in worst-case interference is possible for resources that are shared by fewer than $2c$ tasks in a given cluster. Let $A_{q,k}$ denote the set of tasks assigned to cluster C_k that access ℓ_q. If all jobs wait simply in FIFO order for $\mathrm{CT}_{q,k}$ (instead of using the G-OMLP), then a request for $\mathrm{CT}_{q,k}$ is preceded by at most $A_{q,k}$ earlier-issued requests. Thus, if $A_{q,k} < 2c$, it is favorable to use a FIFO queue instead of a hybrid queue. If resource access is mediated, the kernel is typically aware of $A_{q,k}$ since tasks must open "handles" to shared resources; the kernel can thus adapt transparently and switch from a simple FIFO queue to the G-OMLP when

$A_{q,k}$ exceeds $2c$. This ensures that interference is no worse than in a FIFO protocol like the MBWI.

A key advantage of the VXR environment is that it provides strict temporal isolation without making any assumptions about the composition and resource requirements of a task set (Theorem 3). However, it is of course possible to derive more accurate interference bounds using holistic blocking analysis [6]. That is, by considering per-task maximum critical section lengths and the frequency of resource requests, less-pessimistic, task-set-specific interference bounds can be established. Such fine-grained bounds, expressed as a linear program, can be found in Appendix A.

5.4 Nested Requests

The isolation provided by Theorem 3 implicitly supports nested requests if deadlock is avoided, that is, if the nesting graph is acyclic, which can be enforced by the mediation layer. If a request for a resource ℓ_a may contain a request for another resource ℓ_b, then Theorem 3 remains valid if L_a^{max} is inflated to account for $2 \cdot m \cdot L_b^{max}$ maximum *transitive* interference. It is likely possible to reduce the pessimism inherent in inflating request lengths for all nested requests; this is an interesting opportunity for future work.

6 Conclusion

This paper breaks new ground in several ways. We have presented the OMIP, the first suspension-based independence-preserving real-time locking protocol for clustered JLFP scheduling. The OMIP generalizes the G-OMLP and reduces to it in the case of global scheduling ($c = m$). The distinguishing feature of the OMIP is that it ensures $O(m)$ pi-blocking and independence preservation for *any* cluster size. In contrast, the G-OMLP only applies if $c = m$, and neither the P-OMLP nor the C-OMLP are independence-preserving. Based on the OMIP, we have presented the VXR environment, which is the first locking protocol for reservation-based scheduling to ensure temporal isolation with asymptotically optimal interference. This enables numerous interesting applications: to the best of our knowledge, the VXR environment is the first lock-based approach **(i)** to enable resource-agnostic system composition, **(ii)** suitable for mixed-criticality systems, and **(iii)** to allow budget-less background jobs to share resources with real-time tasks.

In future work, we seek to improve support for nested requests. Further, while implementation concerns had to remain beyond the scope of this paper, they are clearly of great importance. We plan to support the OMIP and VXRs in LITMUS$^{\mathrm{RT}}$ to evaluate the protocols under consideration of realistic overheads in comparison with the MBWI [13].

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. 19th IEEE Real-Time Sys.Symp.*, pages 4–13, 1998.

[2] J. Anderson, S. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proc. Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

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

[4] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proc. 20th Euromicro Conf. on Real-Time Sys.*, pages 147–155. IEEE, 2008.

[5] A. Block. *Adaptive Multiprocessor Real-Time Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2008.

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

[7] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proc. 31st Real-Time Sys.Symp.*, pages 49–60, 2010.

[8] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Sys.*, 2012 (to appear).

[9] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proc. 19th Euromicro Conf. on Real-Time Sys.*, pages 247–256, 2007.

[10] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Proc. 18th Real-Time Sys.Symp.*, pages 308–319. IEEE, 1997.

[11] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proc. 30th IEEE Real-Time Sys.Symp.*, pages 377–386, 2009.

[12] G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU Sys.. In *Proc. 19th Intl.Conf. on Real-Time and Network Sys.*, 2011.

[13] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Proc. 22nd Euromicro Conf. on Real-Time Sys.*, pages 90–99, 2010.

[14] 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. 9th IEEE Real-Time And Embedded Technology Application Symp.*, pages 189–198, 2003.

[15] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS support for multicore mixed-criticality systems. In *Proc. 18th IEEE Real-Time and Embedded Technology and Apps.Symp.*, pages 197–208, 2012.

[16] K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proc. 30th IEEE Real-Time Sys.Symp.*, pages 469–478, 2009.

[17] G. Lamastra, G. Lipari, and L. Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proc. 21st IEEE Real-Time Sys.Symp.*, pages 151–160, 2001.

[18] G. Macariu and V. Cretu. Limited blocking resource sharing for global multiprocessor scheduling. In *Proc. 23rd Euromicro Conf. on Real-Time Sys.*, pages 262–271, 2011.

[19] J. Mauro and R. McDougall. *Solaris internals: core kernel components*. Prentice Hall, 2001.

[20] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: an abstraction for managing processor usage. In *Proc. Fourth Workshop on Workstation Operating Systems*, pages 129–134, 1993.

[21] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proc. 23rd Euromicro Conf. on Real-Time Sys.*, pages 251–261, 2011.

[22] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *Proc. 10th Intl.Conf. on Distributed Comp.Sys.*, pages 116–123, 1990.

[23] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. *Proc. 9th IEEE Real-Time Sys.Symp.*, pages 259–269, 1988.

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

[25] S. Vestal. Preemptive scheduling of multi-criticality Sys. with varying degrees of execution time assurance. In *Proc. 28th IEEE Real-Time Sys.Symp.*, pages 239–243. IEEE, 2007.

A Detailed Blocking Analysis

In this appendix, we derive task-set-specific bounds on s-oblivious pi-blocking and interference. For many task sets, these bounds are less pessimistic than the coarse-grained bounds given in Secs. 3 and 4 because they take each task's resource requirements into account (*i.e.*, they reflect each $N_{i,q}$ and $L_{i,q}$). This avoids overestimating worst-case contention if most resources are shared only among few tasks.

To obtain a task-set-specific bound, we derive a linear program (LP) that, when maximized, yields a safe upper bound on maximum blocking (resp., interference). We use linear programming as the underlying formalism in the interest of conciseness and to avoid using *ad-hoc* notation. We begin with bounding pi-blocking under the OMIP.

A.1 Maximum PI-Blocking under the OMIP

Recall from Sec. 3 that we let $b_{i,q}$ denote the maximum s-oblivious pi-blocking incured by an arbitrary job J_i due to requests for a shared resource ℓ_q. Further, recall from Lemma 5 that $b_{i,q} \leq N_{i,q} \cdot (2m-1) \cdot L_q^{max}$, assuming a uniform cluster size c and that m is an integer multiple of c. The bound derived in this section is both more accurate and more general, in the sense that it is in many cases less pessimistic and that it requires neither cluster sizes to be uniform nor m to be a multiple of c. To account for the latter, we redefine c to denote only the number of processors in T_i's assigned cluster C_k and make no assumption about the number of processors in any of the other clusters.

We further assume that the "FIFO queue optimization" from Sec. 5.3 is employed. That is, access to the contention token $\mathrm{CT}_{q,k}$ is governed by the OMIP if more than $2c$ tasks assigned to C_k access ℓ_q, and by a simple FIFO queue otherwise. Recall that we let $A_{q,k}$ denote the number of tasks assigned to C_k that require ℓ_q. To simplify the LP constraints given below, we define the following shorthand notation.

Def. 6. We let $A'_{q,k} \triangleq \min(A_{q,k},\ 2c) - 1$ denote the maximum number of jobs other than J_i that hold $\mathrm{CT}_{q,k}$ while J_i incurs pi-blocking (w.r.t. a single request of J_i).

Recall that the OMIP as described in Sec. 3 is primarily targeted at systems in which jobs can be assumed to not overrun their budget. The following analysis is based on this assumption. We begin by expressing $b_{i,q}$ as a linear equation to obtain an objective function for the LP that bounds $b_{i,q}$.

A.2 Linear Program Setup

For an arbitrary, but fixed job J_i, the bound $b_{i,q}$ can be expressed exactly as the sum of all pi-blocking that J_i incurs due to each request issued by jobs of other tasks while J_i is pending. This requires bounding the maximum number of requests that are issued while J_i is pending. Let r_x denote the maximum response time of task T_x.

Lemma 9 (from [6], p. 406)**.** *At most $\left\lceil \frac{t+r_x}{p_x} \right\rceil$ distinct jobs of a task T_x execute during any interval of length t.*

Since J_i is pending during an interval of length at most r_i (*i.e.*, T_i's maximum response time), Lemma 9 bounds the maximum number of conflicting requests. We use the following shorthand notation.

Def. 7. Let $N^i_{x,q} \triangleq N_{x,q} \cdot \left\lceil \frac{r_x+r_i}{p_x} \right\rceil$ denote the maximum number of requests for a shared resource ℓ_q issued by jobs of T_x while J_i is pending.

This allows enumerating all conflicting requests.

Def. 8. Let $\mathcal{R}_{x,q,v}$ denote the v^{th} request issued by jobs of T_x while J_i is pending, where $v \in \{1, \ldots, N^i_{x,q}\}$.

To express $b_{i,q}$ as a linear function, we introduce the concept of a "blocking fraction." The *blocking fraction* $X_{x,q,v}$ denotes the amount of pi-blocking incurred by J_i due to request $\mathcal{R}_{x,q,v}$, expressed as a fraction of the maximum request length $L_{x,q}$. For example, suppose $L_{x,1} = 3$ and that, in a specific, fixed schedule, J_i incurs pi-blocking for two time units while $\mathcal{R}_{x,1,1}$ is being executed: in this case $X_{x,1,1} = \frac{2}{3}$. Similarly, if a request $\mathcal{R}_{x,q,v}$ does not block J_i at all (w.r.t. a specific schedule), then $X_{x,q,v} = 0$.

This formalization allows expressing $b_{i,q}$ as a function of each $X_{x,q,v}$, with each $L_{x,q}$ serving as a coefficient.

$$b_{i,q} = \sum_{\substack{x=1 \\ x \neq i}}^{n} \sum_{v=1}^{N^i_{x,q}} X_{x,q,v} \cdot L_{x,q} \tag{1}$$

Since J_i is an arbitrary job of T_i, and since no specific assumptions are made about the fixed schedule under analysis, a safe upper bound on $b_{i,q}$ for any J_i in any schedule can be obtained by *maximizing* Eq. (1) while interpreting each $X_{x,q,v}$ as a variable with $X_{x,q,v} \in [0,1]$.

Of course, unless additional constraints are imposed, Eq. (1) is maximized when each $X_{x,q,v} = 1$, that is, if each request is assumed to block J_i in its entirety. To avoid such pessimism, we next introduce linear constraints that derive from the OMIP's request rules.

A.3 OMIP Constraints

To begin, we restrict the maximum pi-blocking due to conflicting requests issued by jobs in remote clusters. In the following, let C_k denote T_i's assigned cluster, and let C_r denote a remote cluster (*i.e.*, $C_k \neq C_r$). Further, let τ_k and τ_r denote the sets of tasks assigned to clusters C_k and C_r, resp., and let $\tau^i_k \triangleq \tau_k \setminus \{T_i\}$.

Lemma 10. *Let $Q = \min(\sum_{T_l \in \tau^i_k} N^i_{l,q},\ N_{i,q} \cdot A'_{q,k})$. In any schedule of τ under the OMIP:*

$$\forall C_r : \sum_{T_x \in \tau_r} \sum_{v=1}^{N^i_{x,q}} X_{x,q,v} \leq N_{i,q} + Q.$$

Proof. Suppose not. Then there exists a schedule such that a job J_i incurs pi-blocking due to more than $N_{i,q} + Q$

requests issued by jobs of tasks assigned to a remote cluster C_r. Under the OMIP, a remote job J_x delays J_i only if it precedes a job holding $\text{CT}_{q,k}$ in GQ_q (*i.e.*, either J_x precedes J_i directly or J_x precedes another job that holds $\text{CT}_{q,k}$ while J_i is still waiting to acquire $\text{CT}_{q,k}$). By definition of $N_{i,q}$, J_i is at most $N_{i,q}$ times enqueued in GQ_q.

Additionally, if $Q = \sum_{T_l} N^i_{l,q}$, then other local jobs are at most $\sum_{T_l} N^i_{l,q}$ times enqueued in GQ_q. If $\sum_{T_x} \sum_v X_{x,q,v} > N_{i,q} + Q$, then a single request of some job in C_k is blocked by at least two requests of jobs in C_r.

Similarly, if $Q = N_{i,q} \cdot A'_{q,k}$, then other local jobs are at most $N_{i,q} \cdot A'_{q,k}$ times enqueued in GQ_q while J_i incurs pi-blocking. If $\sum_{T_x} \sum_v X_{x,q,v} > N_{i,q} + Q$, then a single request of some job in C_k is blocked by at least two requests of jobs in C_r while J_i incurs pi-blocking.

Since GQ_q is FIFO-ordered, and since there is only one $\text{CT}_{q,r}$ in C_r, a job in GQ_q is blocked by at most one request from each other cluster. Contradiction. □

Lemma 10 limits pi-blocking due to remote jobs only. Next, we introduce two constraints that limit pi-blocking due to local jobs. The first constraint applies in all cases, whereas the second constraint applies only if $A_{q,k} \le 2c$. The general case is constrained as follows.

Lemma 11. *In any schedule of τ under the OMIP:*

$$\sum_{T_x \in \tau^i_k} \sum_{v=1}^{N^i_{x,q}} X_{x,q,v} \le N_{i,q} \cdot A'_{q,k}.$$

Proof. Suppose not. Then there exists a schedule such that a job J_i is pi-blocked by more than $N_{i,q} \cdot A'_{q,k}$ requests issued by local jobs other than J_i. Since there is only one contention token for ℓ_q in C_k, local jobs never block while J_i traverses GQ_q. Hence, J_i is only pi-blocked while waiting to acquire $\text{CT}_{q,k}$, which it does at most $N_{i,q}$ times.

If $A_{q,k} \le 2c$, and if more than $N_{i,q} \cdot A'_{q,k} = N_{i,q} \cdot (A_{q,k} - 1)$ requests of local jobs cause J_i to incur pi-blocking, then jobs of some T_x must acquire $\text{CT}_{q,k}$ at least twice while J_i is waiting to acquire $\text{CT}_{q,k}$, which is impossible since $\text{CT}_{q,k}$ is protected by a FIFO queue if $A_{q,k} \le 2c$.

Otherwise, if $A_{q,k} > 2c$, $\text{CT}_{q,k}$ is shared according to the rules of the G-OMLP. Hence, at most $2c - 1$ requests by local jobs cause J_i to incur pi-blocking each time that it issues a request for ℓ_q, which contradicts the assumption that more than $N_{i,q} \cdot A'_{q,k} = N_{i,q} \cdot (2c - 1)$ requests by local jobs cause J_i to incur pi-blocking. □

Note that Lemma 11 does not constrain the number of pi-blocking requests due to any individual task. In the general case, if $A_{q,k} > 2c$, this cannot be avoided (without making further assumptions about the underlying JLFP scheduling policy) since J_i may be blocked by later-issued requests that enter $\text{FQ}_{q,k}$ while J_i is still waiting in $\text{PQ}_{q,k}$. However, if $A_{q,k} \le 2c$, then the stronger FIFO progress guarantee can be exploited to limit per-task pi-blocking.

Lemma 12. *If $A_{q,k} \le 2c$, then in any schedule of τ under the OMIP:*

$$\forall T_x \in \tau^i_k : \sum_{v=1}^{N^i_{x,q}} \le N_{i,q}.$$

Proof. Suppose not. Then there exists a schedule such that some job J_i incurs pi-blocking due to more than $N_{i,q}$ requests for ℓ_q by jobs of a local task T_x. Hence, one of J_i's $N_{i,q}$'s requests must be preceded by at least two requests issued by jobs of T_x, which is impossible since $\text{CT}_{q,k}$ is protected by a FIFO queue if $A_q \le 2c$. □

By maximizing Eq. (1) subject to the constraints stated in Lemmas 10 and 11 and, if applicable, Lemma 12, a safe upper bound $b_{i,q}$ is obtained that is suitable for use under s-oblivious analysis. Note that the generated LP is not difficult to solve; it is thus not necessary to use an LP solver in an actual implementation. However, specifying blocking bounds as an LP has the distinct advantage that it is trivial to obtain a correct reference implementation for testing purposes.

A.4 Maximum Interference in the VXR Environment

The bound on maximum interference experienced by S_i depends on the assumptions than can be reasonably made about *other* tasks. If a resource ℓ_q is shared only among HRT tasks for which all parameters are accurately known, Lemmas 10–12 apply analogously since budget overruns cannot occur (provided that the server of *each* task that T_i shares resources with is appropriately provisioned).

In contrast, if a resource is shared with untrusted tasks for which $N_{x,q}$ cannot be determined with certainty, then bounds more accurate than those presented in Sec. 4.2 cannot be established without risking the temporal isolation of T_i. We note, however, that $N_{x,q}$ could also be enforced by the access mediation layer by delaying unexpected requests, which leads to the following scenario.

In the case that $N_{x,q}$ and $L_{x,q}$ *are* known (or enforced), but jobs other than J_i may still overrun their budget (*e.g.*, if T_i shares a resource with SRT tasks that are provisioned using average-case estimates), Lemma 11 remains valid only if Def. 6 is slightly adjusted to take Rule B2 into account: if jobs holding ℓ_q might exhaust their budget, then $A'_{q,k} \triangleq \min(A_{q,k},\ 2c)$. This change accounts for $\text{CT}_{q,k}$ being unavailable when J_i enters $\text{FQ}_{q,k}$, analogously to Lemma 7. Lemmas 10 and 12 remain valid without change even if budget overruns are possible.