Relaxing Resource-Sharing Constraints for Improved Hardware Management and Schedulability *

Bryan C. Ward
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Modern computer architectures, particularly multicore systems, include shared hardware resources such as caches and interconnects that introduce timing-interference channels. Unmanaged access to such resources can adversely affect the execution time of other tasks, and lead to unpredictable execution times and associated analysis pessimism that can entirely negate the benefits of a multicore processor. To mitigate such effects, accesses to shared hardware resources should be managed, for example, by a real-time locking protocol*. However, accesses to some hardware resources can be managed with more relaxed* sharing constraints *than mutual exclusion while still mitigating timing-interference channels. This paper presents two new classes of sharing constraints,* preemptive mutual exclusion*, and* half-protected sharing*, which are motivated by the sharing constraints of buses and caches, respectively. Synchronization algorithms are presented for both sharing constraints, where applicable, on both uni- and multi-processor systems. A fundamentally new analysis technique called* idleness analysis *is presented to account for the effects of blocking in globally scheduled multiprocessor systems. Experimental results suggest that these relaxed synchronization requirements and improved analysis techniques can improve schedulability by up to 250%. Furthermore, idleness analysis can be applied to existing locking protocols to improve schedulability in many cases.*

1 Introduction

Through much of the history of real-time systems, processors could be modeled as a single schedulable entity. However, as hardware platforms have become increasingly complex with multicore-processor (MCP) architectures that include numerous shared hardware resources, this simple abstraction of a processor is no longer sufficient—it does not model lower-level hardware interactions. Interactions through shared hardware can cause timing interference that is unpredictable or difficult to tightly analyze. In safety-critical domains, such interference can lead to analysis pessimism so great that it negates any benefits of the more complex architectures. Notably, the current best practice in safety-critical domains such as avionics is to turn off all but one core in a MCP. To address this issue, the U.S. Federal Aviation Administration (FAA) recently released a position paper (CAST-32) [10] on MCPs that identifies sources of shared-hardware interference, and suggests that the effects of such interference are identified, analyzed, and certifiably mitigated.

Several approaches have been presented for mitigating the effects of such interference, including static partitioning of resources (*e.g.*, caches) [9, 16, 24], time-division multiple access (*e.g.*, buses) [18, 19], and locking protocols (*e.g.*, I/O devices) [11, 21]. The efficacy of any of these approaches is ultimately measured in the resource utilization they enable. In the context of hard-real-time systems, higher resource utilization should enable greater schedulability, *i.e.*, task systems should have a better chance of provably satisfying all timing constraints.

In this paper, we argue that shared hardware resources such as caches and buses can be modeled as shared resources that can be managed using a synchronization algorithm with more relaxed *sharing constraints* than those previously mentioned, which can in turn lead to improved resource utilization and schedulability. Sharing constraints formalize the requirements of how resources can be safely shared. For example, memory objects require that accesses are serialized; once a critical section begins, it requires exclusive access to that memory objective until it completes, at which point the next critical section may commence. This is an example of a *non-preemptive mutual exclusion* sharing constraint, and is realized by ordinary mutex locking protocols. Reader/writer sharing is an example well-studied relaxed sharing constraint that can lead to improved performance and schedulability.

A locking protocol causes *priority inversion blocking* (*pi-blocking*) when a lower-priority task executes while a higher-priority task waits. Such blocking is detrimental to schedulability analysis. In contrast to memory objects, some hardware resources inherently have weaker sharing constraints, which can in turn reduce the adverse effects of blocking and yield better resource utilization. For example, we define the *preemptive mutual exclusion* sharing constraint, for resources for which accesses can be paused and later resumed, *i.e.*, *preempted*. We call such resources *preemptive resources*. Some buses can be modeled as preemptive resources. *Time division multiple access* (*TDMA*) is an example technique for managing a resource with a preemptive sharing constraint.

Further relaxed sharing constraints can be formalized for

*Work supported by an NSF Graduate Research Fellowship, NSF grants CNS 1115284, CNS 1218693, CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and a grant from General Motors.

other classes of resources, such as caches. Timing analysis tools such as AbsInt's *aiT Timing Analyzer* for ARM [12] can effectively classify cache blocks as either *useful* if they will be reused, or *evicting* if they will not be [1, 14]. This motivates a fundamentally new sharing constraint, which we call *half-protected* sharing. In the context of the caches, a useful cache block should be *protected* during execution to prevent it from being evicted, while evicting cache blocks should be *unprotected*, and can be preempted at any time without consequence. Half-protected sharing has parallels with reader/writer sharing, but is a weaker sharing constraint.

Contributions. In this work, we formalize two new sharing constraints: preemptive mutual exclusion (Sec. 3), and half-protected sharing (Sec. 4). We explore upon what platforms such constraints are useful. We present algorithms to realize these sharing constraints on preemptively scheduled uniprocessor and globally scheduled multiprocessor platforms, where applicable.

We also present relevant schedulability analysis for the aforementioned multiprocessor algorithms. This analysis is based on a fundamentally new analysis framework called *idleness analysis* for incorporating the effects of blocking into global schedulability tests. Idleness analysis bounds the maximum amount of idleness induced by synchronization, and is incorporated into a schedulability test by treating such idleness as demand. Idleness analysis is used in place of blocking analysis, which can be more pessimistic.

We conducted schedulability studies (Sec. 5) to demonstrate the schedulability benefits associated with our theoretical contributions. These studies show up to a 250% improvement in schedulability over previous real-time locking protocols that were proven optimal under common analysis assumption. Furthermore, our analysis framework can be applied to analyze those same protocols and in some cases yield improved schedulability.

Related work. Synchronization algorithms for non-preemptive mutual exclusion in real-time systems have been extensively studied, so we focus our discussion of such work to that which is most relevant. On uniprocessor systems, the *priority ceiling protocol* (*PCP*) [17] and *stack resource policy* (*SRP*) [2] support lock nesting and bound blocking to one outermost critical section. These algorithms have been extended to multiprocessors, in the MPCP [17] and MSRP [13], respectively. More recently, a number of multiprocessor protocols have been presented, including the FMLP [5], FMLP+ [6], and the OMLP [8], and have been proven optimal under different analysis assumptions described later. Ward and Anderson [20] presented the *real-time nested locking protocol* (*RNLP*), which is configurable to be optimal under most platform configurations, and different analysis assumptions while supporting nested locking. In this work, we develop protocols and analysis for resources with weaker sharing constraints. Note that the aforementioned protocols could be safely used to control resources with weaker sharing constraints at the expense of decreased schedulability, as shown in Sec. 5.

A number of novel approaches have been presented to more explicitly control access to hardware resources to mitigate the adverse affects of shared-hardware-based interference. For example, locking protocols have been used to arbitrate access to GPUs [11], as well as shared caches [21]. Others have studied partitioning-based approaches to reduce or eliminate sharing of hardware resources such as caches [9, 16] or memory banks [24]. Arbitration policies such as *time division multiple access* (*TDMA*) have also been used to predictably control memory bus accesses [18, 19]. TDMA exploits the preemptive sharing constraint of the bus to enable more fair and predictable behavior. Other work considers asynchronously scheduling both processing and shared hardware resources, such as scratchpad memory [22], or DRAM [23]. In contrast, this paper presents algorithms and analysis for synchronously scheduling both CPU and hardware resources, *i.e.*, a task accessing a hardware resource must also be scheduled on the processor. The implementation of the presented algorithms, and application to managing specific hardware resources is a significant implementation effort that is outside the scope of this paper, and deferred to future work.

2 Background

We consider a sporadic task system Γ composed of n tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, on m globally scheduled identical processors. Each task τ_i is composed of a potentially infinite sequence of jobs $\tau_{i,1}, \tau_{i,2}, \ldots$, which are repeatedly released. Each task $\tau_i = (e_i, d_i, p_i)$ is specified by a *worst-case execution requirement* e_i, *minimum inter-arrival time* (*i.e., period*), and a *relative deadline* d_i. We assume constrained deadlines, *i.e.*, $d_i \leq p_i$. Jobs are prioritized by the earliest-deadline first (EDF) scheduling policy. $\tau_{i,j}$ is released at time $r_{i,j}$, and must execute for at most e_i, before its deadline at $r_{i,j} + d_i$. We denote the time $\tau_{i,j}$ completes, or finishes its execution, as $f_{i,j}$. $\tau_{i,j+1}$ is released at or after $r_{i,j} + p_i$. The *processor utilization* of τ_i is $u_i^P = e_i/p_i$, and the total processor utilization is $U^P = \sum_{i=1}^{n} u_i^P$. Also, let $e_{\sum}$ be the largest $m-1$ execution times, and $e_{all} = \sum_{\tau_i \in \Gamma} e_i$.

The n tasks share n_r non-processor shared resources, $\ell_1, \ldots, \ell_{n_r}$. In subsequent sections, we consider different sharing constraints for these resources. A job safely accessing a resource given the assumed sharing constraints is said to be in a *critical section*. We assume that jobs occupy a processor while executing critical sections. For simplicity, critical sections are assumed to be non-nested, *i.e.*, only one resource may be accessed at a time. In Sec. 4, we briefly discuss how our analysis could be extended to nested requests. A job is composed of alternating critical sections and non-critical sections in which only a processor is used. A job of τ_i that requires access to ℓ_q must issue a *resource request* to a *locking protocol* before it can execute its critical section. A request may be *blocked*, or forced to wait, by the locking protocol to ensure that the sharing constraint is not violated.

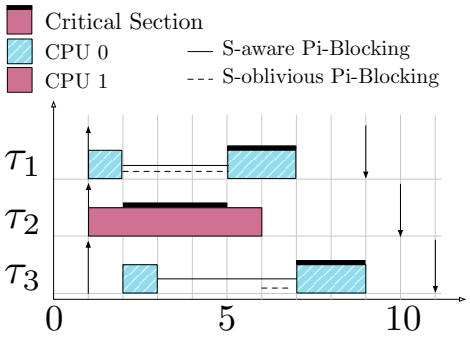


Figure 1: Illustration adapted from [8] of the difference between s-oblivious and s-aware analysis. Note that a job that is s-oblivious pi-blocked is also s-aware pi-blocked.

The total length of all critical sections for ℓ_q by a job of task τ_i is given by $L_{i,q}$. We make no other assumptions about the frequency or duration of critical sections within each job. Critical-section execution time is incorporated in e_i. A task τ_i's *per-resource utilization* of ℓ_q is defined as $u_i^{\ell_q} = L_{i,q}/p_i$, and *total resource utilization* is given by $u_i^R = \sum_{q=1}^{n_r} u_i^{\ell_q}$. The total resource utilization of all tasks is $U^R = \sum_{\tau_i \in \Gamma} u_i^R$. Also, let $L_{\sum}$ denote the sum of the $m-1$ largest $\sum_{q=1}^{n_r} L_{i,q}$, and $L_{all} = \sum_{\tau_i \in \Gamma} \sum_{q=1}^{n_r} L_{i,q}$.

Def. 1. Job $\tau_{l,j}$ is *pending* at time t if $r_{l,j} \leq t \leq f_{l,j}$.

Def. 2. Job $\tau_{l,j}$ is *ready* at time t if it is pending, and it is not blocked waiting for a shared resource.

Def. 3. Job $\tau_{l,j}$ is *CPU-preempted* at time t if it is ready, but it does not execute at t.

The relaxed sharing constraints we consider sometimes allow a job executing a critical section to be preempted with respect to the resource it is accessing, thereby allowing a higher-priority task to access the shared resource. Such a preemption is both a CPU preemption, as the job must suspend execution, as well as a *resource preemption*.

Blocking analysis. Brandenburg and Anderson [8] formalized two classes of techniques to incorporate the effects of *priority-inversion blocking* (*pi-blocking*) into schedulability analysis: *suspension-oblivious* (*s-oblivious*) analysis, in which suspensions due to pi-blocking is modeled as computation, and *suspension-aware* (*s-aware*) *analysis*, in which such suspensions due to pi-blocking are incorporated directly into schedulability analysis. These two techniques rely upon different definitions of pi-blocking, which are visually compared in Fig. 1.

Def. 4. Under **s-aware** schedulability analysis, a job τ_i incurs *s-aware pi-blocking* at time t if τ_i is pending but not scheduled and fewer than m higher-priority jobs are **ready**.

Def. 5. Under **s-oblivious** schedulability analysis, a job τ_i incurs *s-oblivious pi-blocking* at time t if τ_i is pending but not scheduled and fewer than m higher-priority jobs are **pending**.

Pi-blocking can occur in one of two scenarios. First, a job that has issued a request and is blocked waiting to acquire a resource, is said to experience *request blocking*. Second, many locking protocols employ a *progress mechanism* such as priority inheritance or priority boosting, to ensure that a resource-holding job is scheduled. This gives rise to *progress-mechanism-related pi-blocking*, as a lower-priority job can execute while a higher-priority job is forced to wait, even if it has not issued a resource request. This second source of blocking can introduce a great deal of analysis pessimism, as it affects all tasks, regardless of whether they access shared resources or not.

3 Preemptive Resources

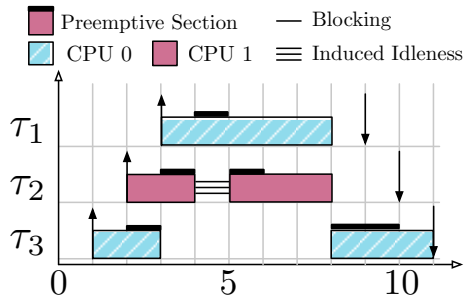
Locking protocols are often used to arbitrate access to non-processor shared resources such as memory objects. Such protocols guarantee that each critical section executes entirely with exclusive access to the locked resource before any other access is allowed. We call such a sharing constraint *non-preemptive mutual exclusion*. In this section, we consider a weaker sharing constraint called *preemptive mutual exclusion*, which is applicable to *preemptive resources*.

3.1 Problem description

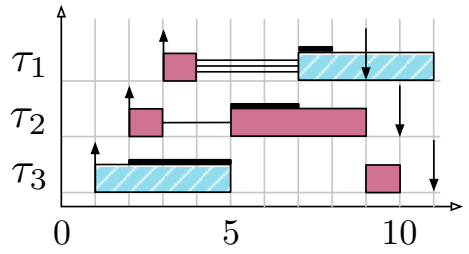
A preemptive resource is one for which at most one task can access the resource at any time t, but accesses to that resource may be preempted, or paused and later resumed. Preemptive resources differ from non-preemptive ones in that non-preemptive mutex resources require that no other job accesses the resource between the start and end of each critical section. Preemptive resources are motivated by bus scheduling, in which multiple tasks may need to transmit data on a bus such as the memory bus, PCIe bus, or a network link. Such transmissions can be "paused" to allow higher-priority transmissions to access the bus. We note that there may be systems-level considerations associated with pausing a transmission, just as there are systems-level considerations associated with scheduling real-time tasks on any physical processor. In this work we build a formal foundation for preemptive non-processor resources, and defer systems-level considerations to future work.

Preemptive resources on uniprocessors. On a uniprocessor platform, optimally synchronizing access to preemptive resources is trivial—the currently executing task implicitly has exclusive access to all preemptive resources. If a task executing a preemptive critical section is processor preempted, it is implicitly preempted from all preemptive resources. Therefore, preemptive resources on uniprocessors can be supported with zero pi-blocking (s-oblivious or s-aware), as they are trivially supported by existing uniprocessor scheduling algorithms.

Preemptive resources on multiprocessors. In contrast to the uniprocessor case, preemptive resources must be explicitly synchronized on multiprocessor platforms. If tasks exe-



(a) Fully preemptive resource example.



(b) Schedule from (a) if ℓ_p were non-preemptive.

Figure 2: Comparison of non-preemptive and preemptive scheduling of a shared resource. Induced idleness (Def. 8) occurs when a processor is idle and there are at least m pending jobs.

cuting concurrently on different processors both need access to the same preemptive resource, only one can execute at a time; the other task(s) must wait. We demonstrate this behavior in the following example.

Ex. 1. Consider three EDF-scheduled tasks on two processors and one EDF-prioritized preemptive resource, ℓ_p, as shown in Fig. 2a. At time $t = 1$, τ_3 is released, and at time $t = 2$, it acquires ℓ_p. At time $t = 2$, τ_2 is released. At time $t = 3$, τ_2 acquires ℓ_p, thereby preempting τ_3 from both the processor as well as ℓ_p. Also at time $t = 3$, τ_1 is released. At time $t = 4$, τ_1 preempts τ_2 w.r.t. ℓ_p. Note that during the interval $[4, 5)$, ℓ_p *induces idleness* on CPU 1, as there are two suspended tasks waiting to access ℓ_p. At time $t = 5$, τ_1 completes its preemptive critical section, and τ_2 resumes its access of ℓ_p. At time $t = 6$, τ_2 completes its preemptive critical section, but τ_3 does not resume its preemptive critical section until time $t = 8$, when τ_1 completes, relinquishing CPU 0 to τ_3.

Access to preemptive resources can be arbitrated by a non-preemptive mutex locking protocol, as the sharing constraint of a preemptive resource is weaker than a non-preemptive mutex. However, preemptive sharing can reduce blocking, and therefore improve schedulability. This is demonstrated in Fig. 2b, where the same task set shown in Fig. 2a is scheduled using a non-preemptive locking protocol,[1] and τ_1 misses a deadline on account of excessive blocking.

[1]This schedule would result from using either the FMLP$^+$ [6] or the global OMLP [8], which are optimal under s-aware and s-oblivious analysis, respectively.

3.2 Schedulability analysis.

We next provide schedulability analysis for scheduling with EDF-prioritized preemptive resources. This analysis is based on a novel technique called *idleness analysis* for accounting for the effects of blocking in schedulability analysis. In contrast to either s-oblivious or s-aware blocking analysis described in Sec. 2, in this work, we take a different approach entirely—we apply idleness analysis in place of blocking analysis. Instead of determining the worst-case per-request pi-blocking, we instead determine the maximum amount of induced idleness that can occur in an interval on account of synchronization. We then analytically treat idleness as demand, similar to s-oblivious analysis.

Our analysis builds upon the previous G-EDF schedulability analysis of Baruah [3] and Liu and Anderson's [15] extension to self-suspending tasks. Due to space constraints, proofs easily derived from those works are omitted. We begin with several definitions.

Def. 6. A processor is *busy* if it is executing a job, and *idle* if it is not. A time instant t is said to be *busy* if all processors are busy, and *idle* if at least one processor is idle.

Def. 7. An idle instant t is *truly idle* if there are at most $m - 1$ pending tasks, and *effectively busy* if there are m or more pending tasks.

Def. 8. Idleness that occurs in an effectively busy instant is said to be *induced* by blocking. We call such idleness *induced idleness*.

In Ex. 1, the system is busy at time $t = 7$, as both processors are in use, and idle at time $t = 4.5$ and $t = 10$. At time $t = 4.5$, all three tasks are pending, but there is idleness induced by blocking, so $t = 4.5$ is effectively busy. By comparison, at time $t = 10$, τ_3 is the only ready job, and therefore $t = 10$ is truly idle.

Next, we derive sufficient conditions that ensure that each task cannot miss any deadlines. Suppose that $\tau_{l,j}$ is the first job of τ_l to miss a deadline, $t_d = d_{l,j}$. Let $t_a = r_{l,j}$ denote the arrival time of $\tau_{l,j}$. $\tau_{l,j}$ necessarily misses its deadline if it executes for strictly less than e_l time units over the interval $[t_a, t_d)$. Job $\tau_{l,j}$ may be prevented from executing if it is preempted or blocked by higher-priority work. Therefore, we disregard all jobs with deadlines later than t_d, as they do not affect the scheduling of $\tau_{l,j}$. (Note that with alternative critical-section prioritizations, discarding lower-priority jobs is not safe, as will be considered in Sec. 4.) Consequently, any blocking of $\tau_{l,j}$ is induced idleness.

Def. 9. Let t_o denote the last truly idle time instant before t_a. Let $[t_o, t_d)$ be the *analysis interval*, and let $A_l = t_a - t_o$.

Over the interval $[t_o, t_a)$, $\tau_{l,j}$ is not pending, and therefore cannot execute. By construction, the entire interval $[t_o, t_a)$ is effectively busy.

Def. 10. Let Θ denote a collection of (possibly non-contiguous) intervals contained within $[t_a, t_d)$, in which $\tau_{l,j}$ does not execute, such that the total length of Θ is $d_l - e_l$.

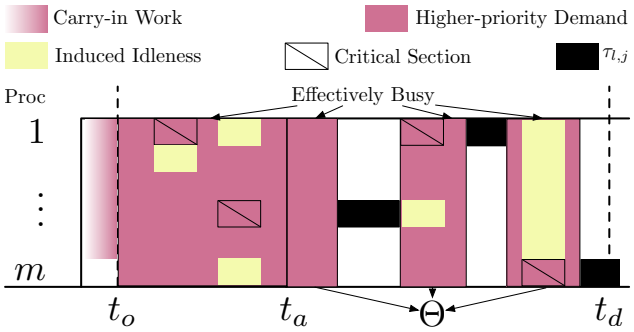


Figure 3: Depiction of the overall proof framework.

By Def. 10, the total length of the intervals in $[t_o, t_a) \cup \Theta$ is $A_l + d_l - e_l$. At any time instant $t \in [t_o, t_a) \cup \Theta$, all processors are effectively busy, *i.e.*, either executing work, or idle on account of blocking. Let $W(\tau_i)$ denote the higher-priority work contributed by τ_i to the interval $[t_o, t_a) \cup \Theta$. Similarly, let $I_q(\tau_l)$ denote the total amount of idleness induced in $[t_o, t_a) \cup \Theta$ by resource ℓ_q. When quantifying idleness, we sum across all processors, *i.e.*, if idleness is induced on two processors during $[0, 1]$, this is two units of idleness, not one. At any time instant $t \in [t_o, t_a) \cup \Theta$, the m processors can either be busy executing demand or idle on account of induced idleness. Thus, in order for $\tau_{l,j}$ to miss a deadline, it is necessary that the total amount of work and induced idleness in the interval $[t_o, t_a) \cup \Theta$ satisfies

$$\sum_{q=1}^{n_r} I_q(\tau_l) + \sum_{\tau_i \in \Gamma} W(\tau_i) > m(A_l + d_l - e_l). \quad (1)$$

To ensure that no task will ever miss a deadline, Condition (1) must not hold for any τ_l or value of A_l. The general proof framework codified by Condition (1) is depicted in Fig. 3. We next derive a corresponding schedulability test. Specifically, we review previous work that considers the work from higher-priority jobs, $W(\tau_i)$ in Sec. 3.3, and then present new analysis for induced idleness $I_q(\tau_l)$ in Sec. 3.4.

3.3 Demand

This subsection is largely a review of previous results, and therefore proofs are omitted. Similar to previous work (*e.g.*, [3]), there are two types of tasks to consider, those with *carry-in* work, *i.e.*, demand that had been released before t_o, and those without carry-in work. First, we review the classic demand-bound function [4].

Lemma 1. *The maximum cumulative execution requirement by jobs of a sporadic task τ_i that both arrive in, and have deadlines within any interval of length t is given by the demand-bound function*

$$DBF(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) e_i\right) \quad (2)$$

We denote the workload contributed by τ_i over the interval $[t_o, t_a) \cup \Theta$ assuming no carry-in job as $W_{nc}(\tau_i)$. Using the demand-bound function, we can derive the following bound on the work contributed by τ_i over $[t_o, t_a) \cup \Theta$.

Lemma 2.

$$W_{nc}(\tau_i) = \begin{cases} \min(DBF(\tau_i, A_l + d_l), A_l + d_l - e_l) & i \neq l \\ \min(DBF(\tau_i, A_l + d_l) - e_l, A_l) & i = l \end{cases} \quad (3)$$

For tasks with carry-in jobs, the demand-bound function is modified, to account for carry-in demand.

$$DBF'(\tau_i, t) = \left\lfloor \frac{t}{p_i} \right\rfloor e_i + \min(e_i, t \mod p_i) \quad (4)$$

Using (4), we can derive the workload $W_c(\tau_i)$ contributed to the interval $[t_o, t_a) \cup \Theta$ by a carry-in task τ_i.

Lemma 3.

$$W_c(\tau_i) = \begin{cases} \min(DBF'(\tau_i, A_l + d_l), A_l + d_l - e_l) & i \neq l \\ \min(DBF'(\tau_i, A_l + d_l) - e_l, A_l) & i = l \end{cases} \quad (5)$$

Next, we determine the maximal total demand $\sum_{\tau_i \in \Gamma} W(\tau_i, t)$ by considering which tasks have carry-in work and which do not. In previous analysis of independent task system [3], t_o is chosen to be the last idle instant before t_a, and therefore at most $m - 1$ tasks can have carry-in work. Similarly, by our assumption that t_o is truly idle, then at most $m - 1$ tasks can have carry-in work. Note that this choice of t_o limits carry-in work to $O(m)$, as compared to the best known global s-aware schedulability test [15], which has $O(n)$ carry-in work.

Let $W_d(\tau_i) = W_c(\tau_i) - W_{nc}(\tau_i)$ be the amount of carry-in work τ_i could contribute. Then the total demand contributed by all higher-priority tasks τ_i is given by

$$\sum_{\tau_i \in \Gamma} W(\tau_i) = \sum_{\tau_i \in \Gamma} W_{nc}(\tau_i) + \sum_{\text{the } (m-1) \text{ largest}} W_d(\tau_i). \quad (6)$$

This concludes our review of prior demand-based analysis. Next, we consider the idleness induced within $[t_o, t_a) \cup \Theta$, which is the novel aspect of our analytical contributions.

3.4 Idleness

At any time instant $t \in [t_o, t_a) \cup \Theta$ in which a processor is not busy executing higher-priority work, idleness must be induced. Here we bound the amount of such idleness.

Traditionally, pi-blocking, which can cause idleness, is accounted for by computing the maximum duration of pi-blocking for each request, and incorporating that into a compatible schedulability test. Such approaches suffer from analysis pessimism in both the blocking analysis, as well as the associated schedulability test. In this work, we take a more holistic analysis approach, and incorporate the critical-

section behavior into the schedulability analysis itself. We do so by effectively inverting the analysis logic—instead of analyzing the worst-case blocking a request may experience, we instead analyze the worst-case idleness that the same request can induce. To do so, we first consider the demand for shared resources within the analysis interval, using similar analysis techniques as presented for processor demand.

Lemma 4. *The maximum cumulative critical-section execution requirement by jobs of a sporadic task τ_i for resource ℓ_q that both arrive in and have deadlines within any interval of length t is given by the critical-section-bound function*

$$CSBF_q(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) L_{i,q}\right). \quad (7)$$

Proof. We consider only jobs that are released and have deadlines within an interval of length t. The total demand can be bounded by the scenario in which some job $\tau_{i,k}$ has a deadline at the end of the interval, and jobs are released periodically. There can be at most $\lfloor \frac{t-d_i}{p_i} \rfloor$ jobs released before $\tau_{i,k}$ that also have deadlines within the interval. Each job can execute at most $L_{i,q}$ within all of its critical sections for ℓ_q. The lemma follows. □

At the beginning of the analysis interval, carry-in jobs may also carry in critical sections that must be executed during the analysis interval. We must account for those critical sections as well, and again, we can apply similar reasoning as in the ordinary demand arguments. For the carry-in case, at most $L_{i,q}$ units of critical-section workload carry-in to the analysis interval.

Lemma 5.

$$CSBF'_q(\tau_i, t) = \left\lfloor \frac{t}{p_i} \right\rfloor L_{i,q} + \min(L_{i,q}, t \mod p_i) \quad (8)$$

Using Lemmas 4 and 5, we can quantify demand for each resource within the analysis interval. Next, we bound idleness induced by such critical sections.

Lemma 6. *The maximum total amount of induced idleness w.r.t. τ_l within $[t_o, t_a) \cup \Theta$ by requests for resource ℓ_q is*

$$I_q(\tau_l) = (m-1)\left(\sum_{\tau_i \in \Gamma} CSBF_q(\tau_i, A_l + d_l) + \sum_{\substack{\text{the } (m-1) \\ \text{largest}}} \delta_i\right) \quad (9)$$

where $\delta_i = CSBF'_q(\tau_i, A_l + d_l) - CSBF_q(\tau_i, A_l + d_l)$

Proof. There can be at most $m-1$ carry-in jobs, each of which contributes at most δ_i to the demand for resource ℓ_q in $[t_o, t_a) \cup \Theta$. All tasks contribute at most $CSBF_q(\tau_i, A_l + d_l)$ non-carry-in demand for ℓ_q. For each time unit of critical-section demand within $[t_o, t_a) \cup \Theta$, at most $m-1$ processors can have idleness induced on account of blocking for ℓ_q. □

From the preceding discussions and lemmas, we have the following Theorem.

Theorem 1. *A task system Γ is G-EDF schedulable on m processors sharing n_r preemptive resources if for all tasks $\tau_l \in \Gamma$ and for all $A_l \geq 0$,*

$$\sum_{q=1}^{n_r} I_q(\tau_l) + \sum_{\tau_i \in \Gamma} W(\tau_i) \leq m(A_l + d_l - e_l). \quad (10)$$

3.5 Run-time complexity

Similar to previous tests, (10) can be evaluated in time polynomial in n for each combination of τ_l and A_l. The following theorem, which is proven in the appendix using a similar proof strategy as prior work [3, 15], demonstrates that only a pseudo-polynomial number of values of A_l need to be evaluated to determine if (10) holds for all A_l.

Theorem 2. *If $(U^P + (m-1)U^R) < m$ and (10) is violated for any A_l, then it is violated for some A_l satisfying the following condition:*

$$A_l \leq \frac{\xi}{m - (U^P + (m-1)U^R)} \quad (11)$$

where $\xi = \sum_{\tau_i \in \Gamma}(d_l - d_i)((m-1)u_i^R + U^P) + (m-1)(L_{\sum} + L_{all}) + e_{\sum} + e_{all} - m(d_l - e_l)$.

Discussion. Since idleness analysis is a radically different approach to accounting for the effects of blocking in schedulability, it is useful to compare and contrast it to previous methods to better understand its merits. Under traditional blocking analysis, either s-oblivious or s-aware, blocking is quantified *horizontally*, or over time w.r.t. the blocked task. For example, under the global OMLP [8] a request can be s-oblivious pi-blocked by $2m-1$ other requests. This blocking is incorporated into the task as additional time it must wait for the request to be satisfied. In contrast, in idleness analysis the effects of blocking are quantified *vertically*, or with respect to processors. A request may delay higher-priority work, but if it does not induce idleness, that blocking is analytically *irrelevant*—the processors are busy executing other higher-priority demand. If a critical section is executing, it is occupying one processor, and in the worst case idleness is induced on all of the remaining $m-1$ processors. By analyzing the impacts of critical sections vertically instead of horizontally, we have essentially reduced the analytical impact of each critical section from $2m-1$ (OMLP) or n (FMLP) to $m-1$ via idleness analysis and relaxed sharing constraints. (In practice, constant factors make this comparison more difficult, but this high-level comparison gives insight into the differences.)

Under s-aware schedulability analysis, blocking does not contribute any demand to the analysis interval. However, the main source of pessimism in prior G-EDF s-aware schedulability analysis [15] is in bounding carry-in work. Any suspending (resource-requesting) task may contribute carry-

in work, instead of only $m-1$ as is possible using either s-oblivious or idleness analysis. In fact, Brandenburg [6] showed that in some cases treating s-aware blocking bounds as demand and using s-oblivious schedulability analysis yielded far greater schedulability than s-aware schedulability tests, likely due to the issue of carry-in work. Idleness analysis allows for the analysis interval to be extended to a point where by definition only $m-1$ jobs carry-in work, which can significantly reduce carry-in workload.

Under idleness analysis, the utilization loss due to a critical section is inflated by a factor of m. This is similar to a bus-arbitration policy such as TDMA that fairly shares bandwidth among cores. However, in practice many commercial off-the-shelf (COTS) memory controllers do not implement a fair arbitration policy, and therefore modeling bus accesses as preemptive critical sections can be used to achieve similar theoretical results on a less predictable hardware platform. Furthermore, in practice, lower-priority jobs could execute during induced idleness thereby improving response times, or idled processor could lead to power savings.

4 Half-Protected Resources

In this section, we consider a weaker sharing constraint called *half-protected*, which is motivated by managing caches. During program execution, a job may access many cache blocks, only some of which are reused. Blocks that can be shown to be reused are called *useful cache blocks* (*UCBs*), and the remaining accessed blocks are called *evicting cache blocks* (*ECBs*). By managing cache accesses, we want to ensure that UCBs are *protected*, or not evicted before they are reused. However, regions of code in which evicting cache blocks may be accessed may be *unprotected*, and only prevented from interfering (co-scheduling or preempting) with a protected section. Importantly, unprotected sections can be interrupted at any time with no negative consequences, since they do not reuse cache blocks.

In the remainder of this section, we formalize the half-protected sharing constraint, and present algorithms for both uniprocessors and multiprocessors.

Problem description. There are two classes of requests for half-protected resources: *protected requests*, which require non-preemptive mutual exclusion, and unprotected requests, which have no sharing constraints. Therefore, a protected request can interrupt an unprotected request at any time, but the converse is not true. On a multiprocessor, multiple unprotected sections can execute concurrently, but only one protected request can execute at a given time, and it must execute non-preemptively w.r.t. the resource.

To better understand the half-protected sharing constraint, we compare it to reader/writer sharing. Under reader/writer sharing, there are also two classes of requests: reads, which can execute concurrently or processor-preempt one another, and writes, which have the same sharing constraint as protected sections. However, both reads and writes are non-preemptive w.r.t. the resource. Therefore, a write request cannot be satisfied until all incomplete reads have completed their critical sections. In half-protected sharing, protected requests are identical to writes, while unprotected requests are a weaker version of reads. Unprotected requests are effectively preemptive read requests.

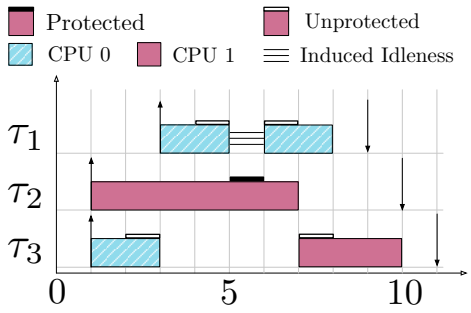


Figure 4: Example schedule depicting half-protected sharing.

Ex. 2. Consider the example in Fig. 4 in which three tasks are scheduled on two processors sharing a half-protected resource. At time $t=1$, both τ_2 and τ_3 are released and begin executing. At time $t=2$, τ_3 begins an unprotected section, and is processor-preempted by the release of τ_1 at time $t=3$. At time $t=4$, τ_1 also begins an unprotected section. Note that τ_3, while preempted is still in an unprotected section at $t=4$. With protected sections or non-preemptive sharing, two critical sections cannot be satisfied concurrently. At time $t=5$, τ_2 begins a protected section, which resource-preempts τ_1 and τ_3. This induces idleness during the interval $[5,6)$. At time $t=6$, τ_1 resumes its unprotected section. The remainder of the example follows G-EDF scheduling.

Scheduling upon uniprocessors. Half-protected resources can be supported with extensions to the Priority Ceiling Protocol (PCP) [17] or the Stack Resource Policy (SRP) [2]. Interestingly, unlike preemptive resources discussed previously, half-protected resources pose a new synchronization problem on uniprocessor platforms.

In a priority-ceiling-based synchronization protocol, the *current priority ceiling* $\hat{\Pi}(t)$ at time t is equal to the highest ceiling associated with any active critical section. For the simple case of non-preemptive mutual exclusion, the ceiling of each resource is defined to be the highest-priority task that accesses that resource. If a task does not have a priority higher than the current priority ceiling $\hat{\Pi}(t)$, then it is forced to block unless it is executing the critical section responsible for setting the current priority ceiling. Priority inheritance is used to ensure resource-holder progress if a higher-priority task is blocked waiting for a lower-priority one.

To realize any benefits of a more relaxed sharing constraint in a priority-ceiling-based protocol, the rules for setting the current priority ceiling need to be relaxed to lower the ceiling and reduce blocking. For example, for reader/writer sharing, the ceiling of a reader/writer resource ℓ_r during a read critical section is set to the highest priority

task that may read ℓ_r, while during a write critical section, the ceiling of ℓ_r is set to the highest priority task that may read or write ℓ_r [17]. By reducing the ceiling for read requests, some higher-priority read requests can processor-preempt tasks executing read critical sections, instead of blocking.

The resource ceiling Π_h of a half-protected resource ℓ_h depends upon whether it is currently in a protected or unprotected section, similar to reader/writer sharing. The *protected ceiling* or the ceiling during a protected section, is set to the priority of the highest-priority task that accesses ℓ_h. This ceiling value ensures that a protected section runs non-preemptively w.r.t. ℓ_h. The *unprotected ceiling* or the ceiling during an unprotected section, is set to the lowest priority in the system. This low ceiling ensures that any task can processor- or resource-preempt a job in an unprotected section, therefore preventing any task from blocking on an unprotected section. However, an unprotected section may still block on a protected section. Notably, similar to other priority-ceiling-based protocols, protected and unprotected sections can be nested using this approach.

When these resource-ceiling definitions are applied in the context of the SRP, there is an additional benefit: system calls are not necessary to enter or exit unprotected sections, thereby reducing overheads. In the SRP, all blocking is incurred before a job begins, and therefore all resources can be accessed immediately without blocking. System calls are required for protected sections, as the current priority ceiling may be increased to prevent higher-priority jobs from beginning execution when resources they may need are currently locked. However, unprotected sections will never increase the current priority ceiling, and therefore do not need to issue system calls. In fact, in the SRP no run-time knowledge of unprotected sections is necessary. Such knowledge is only necessary to set the protected-ceiling values offline. In contrast, in the PCP, entering and exiting unprotected sections must be handled at runtime to ensure that unprotected sections block on protected ones.

Scheduling upon multiprocessors. On a multiprocessor platform, half-protected resources pose new issues. Protected sections must execute non-preemptively w.r.t. the half-protected resource, as well as non-concurrently with any other protected or unprotected sections. In contrast, unprotected sections can execute concurrently, and be arbitrarily resource preempted. In fact, one protected request can resource-preempt multiple unprotected sections.

We consider the following half-protected synchronization policy. Protected requests are statically prioritized over unprotected ones, *i.e.*, a protected request will always preempt an unprotected request. Unprotected requests execute whenever they are not preempted. Protected requests are assumed to be prioritized against one another by non-preemptive EDF. Additionally, we assume that priority inheritance is used to ensure the progress of protected requests, *i.e.*, a protected request inherits the priority of the highest-priority protected or unprotected request that it blocks. Given this synchronization policy, we next extend our idleness analysis and derive an associated schedulability test.

Unprotected sections. Idleness analysis, as compared to blocking analysis, shifts the analysis burden from blocked requests to satisfied requests. This analysis technique is particularly well suited to unprotected sections.

Lemma 7. *Unprotected sections do not induce idleness.*

Proof. A satisfied request induces idleness if it blocks another request leaving a processor idle. Unprotected requests can be satisfied concurrently, and therefore do not block one another. By assumption, unprotected requests are statically prioritized lower than protected requests, and do not block protected requests. Therefore, unprotected requests do not block other requests and thus do not induce idleness. □

Given this result, we safely ignore unprotected requests in idleness analysis. Therefore, to avoid notational clutter, for the rest of this section, we assume that $L_{i,q}$ is the total length of τ_i's ℓ_q protected sections. Notably, these results also apply to non-preemptive resources also.

Carry-up sections. In idleness analysis for preemptive resources (Sec. 3.4), all critical sections and jobs with deadlines later than the analyzed job $\tau_{l,j}$ were ignored as they could not delay the execution of $\tau_{l,j}$. When considering non-preemptive resources, this is no longer true. A lower-priority job than $\tau_{l,j}$ may issue a request while $\tau_{l,j}$ is blocked or scheduled on another core that may subsequently non-preemptively block other higher-priority requests and induce idleness. We define a *carry-up request* to be a request that can execute in the analysis interval but that has a deadline after the analysis interval. In comparison to carry-in critical sections, which are issued by higher-priority jobs that were released before the analysis interval, a carry-up critical section can be released at any point (even before t_o) but has a lower priority than the analyzed job $\tau_{l,j}$. This behavior is depicted in Ex. 2 and Fig. 4. Observe that when analyzing τ_1, τ_2 has a deadline after d_1. However, τ_2 still induces idleness during time $[5, 6)$. In this case, the protected section of τ_2 is a carry-up section.

Carry-up critical sections must be accounted for in idleness analysis for protected sections to account for the additional sources of induced idleness. Importantly, such analysis requires more carefully considering jobs with deadlines after the end of the analysis interval, t_d. We therefore clarify several previous assumptions and definitions in this context.

Recall from Def. 9, that t_o is defined to be the last truly idle time instant before t_a, the release of $\tau_{l,j}$. However, we had previously discarded all jobs with deadlines after t_d. We use the same definition here, *i.e.*, t_o is the last truly idle time instant before t_a w.r.t. to jobs that have deadlines at or before t_d. Thus, at time t_o, there are at most $m-1$ pending jobs with deadlines at or before t_d, which are the *carry-in jobs*. Note that there could be additional jobs released before t_o but with deadlines after t_d that can contribute carry-up critical sections. Such critical sections are considered carry-

up critical sections, rather than carry-in.

Lemma 8. *Let $[t_0, t_1)$ be an interval of length t, and τ_i be a task without a carry-in job. The maximum cumulative execution requirement of protected sections of a half-protected resource ℓ_q by jobs of a sporadic task τ_i that are pending at any time during $[t_0, t_1)$ is given by the* protected-section bound function

$$PSBF_q(\tau_i, t) = \left\lceil \frac{t}{p_i} \right\rceil L_{i,q}. \tag{12}$$

Proof. There are two classes of tasks without carry-in work, those with all job releases after t_0 (Case 1), and those released before t_0, and with deadlines after t_1 (Case 2). Those with releases before t_0 and deadlines before t_1, by definition have a carry-in job.

Case 1. The maximum execution time of protected sections generated by jobs of τ_i released after t_0 is bounded by the case in which the first job of τ_i is released at t_0, and jobs are released periodically thereafter. There are at most $\left\lceil \frac{t}{p_i} \right\rceil$ such jobs that could be released before t_1, each of which executes protected sections for at most $L_{i,q}$ time.

Case 2. If a job of τ_i is released strictly before t_0, and has a deadline strictly after t_1, then $t < d_i \leq p_i$. Thus, there can be only one job of τ_i that is pending during $[t_0, t_1)$, and it executes protected sections for at most $L_{i,q} = \left\lceil \frac{t}{p_i} \right\rceil L_{i,q}$. □

Note that $PSBF_q(\tau_i, t)$ has many commonalities with the well-studied *request-bound function*, which bounds the execution requirement of jobs that can be released within an interval of length t. Similarly, we consider the protected sections of all jobs that can be released in an interval of length t, as even jobs with priority lower than $\tau_{l,j}$ can contribute carry-up sections, which can induce idleness.

Lemma 9. *Let $[t_0, t_1)$ be an interval of length t, and τ_i be a task with a carry-in job. The maximum cumulative execution requirement of protected sections of a half-protected resource ℓ_q by jobs of a sporadic task τ_i that are pending at any time during $[t_0, t_1)$ is given by the* protected-section bound function

$$PSBF'_q(\tau_i, t) = \left\lceil \frac{t + d_i}{p_i} \right\rceil L_{i,q}. \tag{13}$$

Proof. By Lemma 5.1 of [7], there are at most $\lceil \frac{t+R_i}{p_i} \rceil$ distinct jobs of a task τ_i that can execute in any interval of length t, where $R_i \leq d_i$ is the response time of τ_i. Thus, $\left\lceil \frac{t+d_i}{p_i} \right\rceil$ total jobs may execute within $[t_0, t_1)$, and each may execute protected sections for at most $L_{i,q}$. □

Any non-protected sections (*i.e.*, unprotected or non-critical) of jobs with deadlines after t_d that execute within the analysis interval execute either concurrently with $\tau_{l,j}$ or during blocking of higher-priority jobs. Therefore, such demand does not delay $\tau_{l,j}$ and need not be analyzed further.

We next bound the total induced idleness, similarly to Lem. 6, using the results from the previous two lemmas.

Lemma 10. *The maximum total amount of induced idleness w.r.t. τ_l within $[t_o, t_a) \cup \Theta$ by protected requests for a half-protected resource ℓ_q is given by*

$$I_q^h(\tau_l) = (m-1)\left(\sum_{\tau_i \in \Gamma} PSBF_q(\tau_i, A_l + d_l) + \sum_{\substack{\text{the } (m-1) \\ \text{largest}}} \delta_i\right) \tag{14}$$

where $\delta_i = PSBF'_q(\tau_i, A_l + d_l) - PSBF_q(\tau_i, A_l + d_l)$.

Carry-up sections contribute demand to the analysis interval, as well as induce idleness. If a carry-up section is executed during the analysis interval, priority inheritance ensures that it is scheduled if it would induce idleness. However, because carry-up sections are contributed by lower-priority jobs, that demand is not accounted for by the workload of higher-priority jobs. We quantify the total carry-up demand by computing the difference between the total length of all protected requests, including carry-up work, and the total duration of requests that are not carry-up, as computed previously in Lems. 4, and 5. We thus have the following lemma.

Lemma 11. *The maximum cumulative protected-section execution time of all carry-up requests w.r.t. τ_l for a half-protected resource ℓ_q that run in $[t_o, t_a) \cup \Theta$ is given by*

$$C_q(\tau_l) = \sum_{\tau_i \in \Gamma} (PSBF_q(\tau_i, t) - CSBF_q(\tau_i, t)) + \sum_{\substack{\text{the } (m-1) \\ \text{largest}}} \alpha_i - \sum_{\substack{\text{the } (m-1) \\ \text{largest}}} \beta_i \tag{15}$$

where $t = A_l + d_l$, $\alpha_i = PSBF'_q(\tau_i, t) - PSBF_q(\tau_i, t)$, and $\beta_i = CSBF'_q(\tau_i, t) - CSBF_q(\tau_i, t)$.

From these results and discussion, we have the following schedulability test.

Theorem 3. *A task system Γ is global EDF schedulable on m processors sharing n_r half-protected resources if for all tasks $\tau_l \in \Gamma$ and for all $A_l \geq 0$,*

$$\sum_{q=1}^{n_r} (C_q(\tau_l) + I_q^h(\tau_l)) + \sum_{\tau_i \in \Gamma} W(\tau_i) \leq m(A_l + d_l - e_l). \tag{16}$$

Again, we can bound the maximum number of testing points required to check schedulability. The following theorem can be proven through algebraic manipulation, similarly to Thm. 2, and is shown in the appendix.

Theorem 4. *If $(U^P + (m-1)U^R) < m$ and (16) is violated for any A_l, then it is violated for some A_l satisfying the*

following condition:

$$A_l < \frac{\phi}{m - (U^P + (m-1)U^R)} \tag{17}$$

where $\phi = m(L_{all} + L_{\sum}) + (m-1)d_l U^R + e_{\sum} - m(d_l - e_l) + \sum_{\tau_i \in \Gamma}(d_l - d_i)u_i^P + e_{all}$.

Discussion. Perhaps surprisingly, how protected requests are prioritized against one another does not affect schedulability under our idleness-analysis framework, so long as a progress mechanism such as priority inheritance ensures that a carry-up request that induces idleness at time t is scheduled at time t. This is due to how idleness analysis accounts for the effects of blocking vertically rather than horizontally. Even a random prioritization among protected requests could not induce more than $m-1$ processors of idleness during the execution of a critical section, though the worst-case blocking bound would be quite pessimistic. Thus, the above idleness analysis can be applied to existing non-preemptive locking protocols such as the global OMLP [8], or the FMLP [5] by treating all lock requests as protected requests. Furthermore, nested locking protocols could also be supported at the expense of more verbose notation, provided the protocol supported transitive priority inheritance and prevented deadlock. This would allow for multiple cache blocks to be locked simultaneously.

5 Evaluation

We next present evaluations of our proposed algorithms and analysis with respect to hard real-time *schedulability*, or the fraction of randomly generated task systems that are schedulable. We considered two alternative schemes for randomly generating task systems, one motivated by a preemptive bus or interconnect, and another similar to previous studies on real-time locking protocols [6]. These two generation schemes provide insights into the schedulability gains afforded by relaxed sharing constraints and idleness analysis.

Bus synchronization. We first consider a random task-system generation process that varies both the total processor utilization and the total resource utilization of one preemptive resource. We considered schedulability on $m \in \{2, 4, 8\}$ processor systems. We generated task systems for each processor utilization in $\{0.1, 0.2, \ldots, m\}$. The total resource utilization of the generated tasks was in $\{0.1, 0.2, 0.3, 0.4\}$. The per-task processor utilizations were chosen from one of two exponential distributions with mean 0.1 (*light*) and 0.25 (*medium*), and restricted to $[0, 1]$. The period of each task was uniformly chosen from $[10\,\mathrm{ms}, 100\,\mathrm{ms}]$. The resource utilization of each task was uniformly random, and normalized across all tasks to sum to the specific total resource utilization. We assume one critical section per job. For each combination of parameters, task systems were randomly generated until the schedulability ratio could be estimated with 95% confidence to within 0.05.

For each generated task system, we considered schedulability under the global OMLP [8]; the FMLP+ [6] assuming either Liu and Anderson's s-aware analysis [15] (FMLP-A), or by treating s-aware blocking bounds as s-oblivious ones and applying Baruah's [3] s-oblivious test (FMLP-O); preemptive sharing with idleness analysis (P-I); and non-preemptive sharing with idleness analysis (NP-I). Note that NP-I follows from treating all lock requests as protected requests. As a basis for comparison, we also plot schedulability without any synchronization (NOLOCK), which upper bounds all other considered cases. An example schedulability graph from this study is shown in Fig. 5a.

Obs. 1. Preemptive sharing and idleness analysis can greatly improve schedulability over non-preemptive sharing.

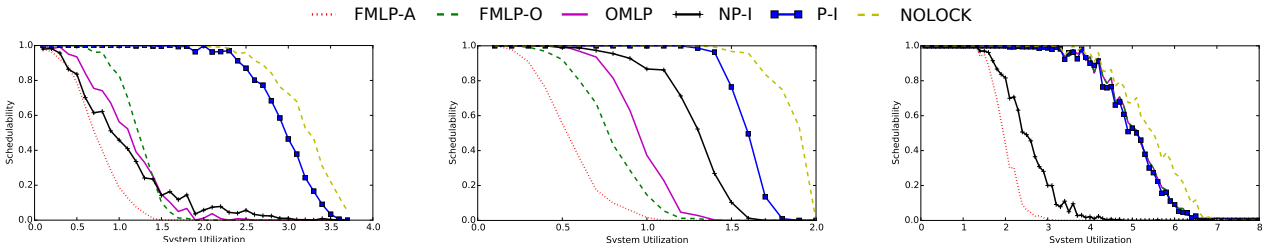
This observation is supported by Fig. 5a. In particular, with preemptive sharing most task systems with utilizations up to approximately 2.5 are schedulable, while most task systems with utilization greater than 1.0 are unschedulable by all considered non-preemptive approaches. This demonstrates how more relaxed sharing constraints can be exploited to improve schedulability.

Locking study As we discussed previously, idleness analysis can be used in place of blocking analysis to analyze schedulability for many existing real-time locking protocols, including the OMLP and the FMLP. We therefore considered an alternative task-system generation scheme similar to previous studies to evaluate the difference between idleness analysis and blocking analysis. In this study, tasks were generated as described above with the following exceptions. Total resource utilization was not directly controlled, instead, critical-section lengths were chosen uniformly among $[1\,\mu\mathrm{s}, 15\,\mu\mathrm{s}]$ (*short*), $[1\,\mu\mathrm{s}, 100\,\mu\mathrm{s}]$ (*moderate*), or $[5\,\mu\mathrm{s}, 1280\,\mu\mathrm{s}]$ (*long*). Each task had a uniform probability $p^{acc} \in \{0.1, 1.0\}$ of accessing one shared resource. Example schedulability graphs from this study can be seen in insets (b) and (c) of Fig. 5.

Obs. 2. Idleness analysis can improve schedulability over both s-aware and s-oblivious blocking analysis for existing real-time locking protocols.

This observation is supported by Fig. 5b, in which the curve NP-I has greater schedulability than all of the existing locking protocols. As discussed previously, idleness analysis effectively reduces the effect of blocking to $m-1$ from $2m-1$ or $O(n)$ for the OMLP and FMLP+, respectively. In comparison to s-aware schedulability analysis, by explicitly analyzing induced idleness, we are able to bound carry-in work to $m-1$ tasks instead of $O(n)$. Consequently, idleness analysis can provide improved schedulability. This is significant as the blocking bounds for the FMLP+ and the OMLP have been proven asymptotically optimal under s-aware and s-oblivious analysis assumptions, while idleness analysis applies to a very general class of mutex locking protocols, including random request prioritizations, which have very large worst-case blocking bounds.

Obs. 3. Idleness analysis is incomparable with both s-aware



(a) $m = 4$, $U^R = 0.1$. (b) $m = 2$, $p^{acc} = 1.0$. (c) $m = 8$, $p^{acc} = 0.1$.

Figure 5: Three example schedulability graphs. (a) is drawn from the bus synchronization experiment, and (b) and (c) are from the locking experiment. Task utilizations are light in all cases. Long critical sections are assumed in (b) and (c).

and s-oblivious blocking analysis.

This observation is supported by Fig. 5c, in which $p^{acc} = 0.1$. In task systems in which the number of resource-using tasks is small, or the number of processors is large, idleness bounds are most pessimistic. In these cases, s-aware and s-oblivious blocking analysis can yield better schedulability. Idleness analysis is therefore incomparable to blocking analysis.

Obs. 2 and 3 demonstrate that there may be interesting tradeoffs to explore at the intersection of idleness analysis and blocking analysis. For example, perhaps protocol-specific knowledge could be used to tighten idleness bounds.

6 Conclusion

In this paper, we argued that some shared hardware resources have weaker sharing constraints than non-preemptive mutual exclusion. We have formalized two such weaker sharing constraints, namely, preemptive mutual exclusive, which is motivated by bus synchronization, and half-protected sharing, which is motivated by cache management. We have presented simple algorithms to realize these sharing constraints. To analyze the effects of blocking on schedulability of these algorithms, as well as other synchronization algorithms, we have developed a novel analysis technique called idleness analysis. Idleness analysis is used in place of blocking analysis in demand-based multiprocessor schedulability tests. Instead of analyzing the delays due to blocking, idleness analysis quantifies the idleness induced in the analysis interval, which can significantly reduce analysis pessimism. Experimental results show up to 250% increase in schedulability as a result of our new analysis techniques and relaxed sharing constraints.

Future work. This work builds the foundation for a broad range of future research, ranging from theoretical problems to systems-level challenges. From a theoretical perspective, we would like to derive finer-grained idleness bounds that exploit protocol-specific properties to reduce idleness bounds. Also, we plan to explore alternative sharing constraints, such as preemptive k-exclusion. From a systems-level perspective, we would like to implement these algorithms, and apply them to applications such as managing data transfers in GPUs, for example in GPUSync [11], or for controlling cache evictions, similarly to [21].

References

[1] S. Altmeyer and C Maiza. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57(7):707–719, Aug. 2011.

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

[3] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS '07*.

[4] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS '90*.

[5] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07*, pages 47–56, Aug. 2007.

[6] B. Brandenburg. The FMLP$^+$: an asymptotically optimal real-time locking protocol for suspension-aware analysis. In *ECRTS '14*.

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

[8] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design automation for embedded systems*, 17:277–342, 2014.

[9] B. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA '08*.

[10] Certification Authorities Software Team (CAST). Position paper CAST-32 multicore processors. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf.

[11] G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS '13*.

[12] C. Ferdinand and R. Heckman. aiT: worst case execution time prediction by static program analysis. In *IFIP*, 2004.

[13] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS '01*.

[14] C.-G. Lee *et al.*. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computing*, 47(6):700–713, Jun 1998.

[15] C. Liu and J. Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. In *ECRTS '13*.

[16] F. Mueller. Compiler support for software-based cache partitioning. In *LCTRTS '95*.

[17] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[18] J. Rosén, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS '07*.

[19] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *DAC '10*.

[20] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS '12*.

[21] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*.

[22] S. Wasly and R. Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *ECRTS '13*.

[23] S. Wasly and R. Pellizzoni. Hiding memory latency using fixed priority scheduling. In *RTAS '14*.

[24] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicoore platforms. In *RTAS '14*.

Appendix

In this appendix, we give proofs for Thms. 2 and 4, which imply that there are a pseudo-polynomial number of testing points for the schedulability tests for both preemptive and half-protected resources. As these proofs are similar, we begin with common notation and observations.

For notational convenience, let $T = A_l + d_l$. Observe that $W_{nc}(\tau_i) \leq DBF(\tau_i, T)$, and $W_d(\tau_i) \leq e_i$. Also, a carry-in job can carry in at most one job's worth of critical sections, so $CSBF'_q(\tau_i, A_l + d_l) - CSBF_q(\tau_i, A_l + d_l) \leq L_{i,q}$, and $PSBF'_q(\tau_i, A_l + d_l) - PSBF_q(\tau_i, A_l + d_l) \leq L_{i,q}$.

Proof of Theorem 2. If Condition (10) is violated we have

$$\sum_{q=1}^{n_r} I_q + \sum_{\tau_i \in \Gamma} W(\tau_i) > m(T - e_l)$$

$$\Rightarrow \text{Substituting}$$

$$(m-1)(L_\sum + \sum_{\tau_i \in \Gamma} \sum_{q=1}^{n_r} CSBF_q(\tau_i, T)) +$$
$$\sum_{\tau_i \in \Gamma} DBF(\tau_i, T) + e_\sum > m(T - e_l)$$

$$\Rightarrow \text{By Equations (2) and (7)}$$

$$(m-1)(L_\sum + \sum_{\tau_i \in \Gamma} \sum_{q=1}^{n_r} (\lfloor \frac{T - d_i}{p_i} \rfloor + 1) L_{i,q} +$$
$$\sum_{\tau_i \in \Gamma} (\lfloor \frac{T - d_i}{p_i} \rfloor + 1) e_i + e_\sum > m(T - e_l)$$

$$\Rightarrow \text{Removing the floor}$$

$$(m-1)(L_\sum + \sum_{\tau_i \in \Gamma} \sum_{q=1}^{n_r} ((T - d_i) u_i^{\ell_q} + L_{i,q})) +$$
$$\sum_{\tau_i \in \Gamma} ((T - d_i) u_i^P + e_i) + e_\sum > m(T - e_l)$$

$$\Rightarrow \text{Rearranging}$$

$$(m-1)(L_\sum + L_{all}) + e_\sum + e_{all} + \sum_{\tau_i \in \Gamma} ((T - d_i) u_i^P) +$$
$$(m-1) \sum_{\tau_i \in \Gamma} \sum_{q=1}^{n_r} (T - d_i) u_i^{\ell_q}) > m(T - e_l)$$

$$\Rightarrow \text{Substituting T and Rearranging}$$

$$\sum_{\tau_i \in \Gamma} (d_l - d_i)((m-1) u_i^R + U^P) + (m-1)(L_\sum + L_{all}) +$$
$$e_\sum + e_{all} - m(d_l - e_l) > A_l(m - (U^P + (m-1)U^R))$$

$$\Rightarrow \text{Dividing}$$

$$A_l < \frac{\xi}{m - (U^P + (m-1)U^R)}$$

provided $m - (U^P + (m-1)U^R) < m$. □

Proof of Theorem 4. A task can only contribute carry-up critical sections from one job, thus $PSBF_q(\tau_i, T) - CSBF_q(\tau_i, T) \leq L_{i,q}$, and $PSBF'_q(\tau_i, T) - CSBF'_q(\tau_i, T) \leq L_{i,q}$. Therefore, $\sum_{q=1}^{n_r} C_q \leq L_{all} + L_\sum$. If Condition (16) is unsatisfied, we have

$$\sum_{q=1}^{n_r} (C_q + I_q^h) + \sum_{\tau_i \in \Gamma} W(\tau_i) > m(T - e_l)$$

$$\Rightarrow \text{Substituting as described above.}$$

$$L_{all} + L_\sum + (m-1) \sum_{q=1}^{n_r} \sum_{\tau_i \in \Gamma} (\left\lceil \frac{T}{p_i} \right\rceil L_{i,q} + L_\sum) +$$
$$\sum_{\tau_i \in \Gamma} (\left\lfloor \frac{T - d_i}{p_i} \right\rfloor + 1) e_i + e_\sum > m(T - e_l)$$

$$\Rightarrow \text{Removing floors and ceilings}$$

$$L_{all} + L_\sum + (m-1) \sum_{q=1}^{n_r} \sum_{\tau_i \in \Gamma} ((\frac{T}{p_i} + 1) L_{i,q} + L_\sum) +$$
$$\sum_{\tau_i \in \Gamma} (\frac{T - d_i}{p_i} + 1) e_i + e_\sum > m(T - e_l)$$

$$\Rightarrow \text{Rearranging and substituting}$$

$$m(L_{all} + L_\sum) + (m-1) T U^R + e_\sum + e_{all} +$$
$$\sum_{\tau_i \in \Gamma} (T - d_i) u_i^P > m(T - e_l)$$

$$\Rightarrow \text{Rearranging and substituting } T$$

$$m(L_{all} + L_\sum) + (m-1) d_l U^R + e_\sum - m(d_l - e_l) +$$
$$\sum_{\tau_i \in \Gamma} (d_l - d_i) u_i^P + e_{all} > A_l(m - (U^P + (m-1)U^R))$$

$$\Rightarrow \text{Substituting}$$

$$A_l < \frac{\phi}{m - (U^P + (m-1)U^R)},$$

provided $m - (U^P + (m-1)U^R) < m$.

□