# Blocking Analysis for Spin Locks in Real-Time Parallel Tasks

Son Dinh, Jing Li, Kunal Agrawal, Chris Gill, *Senior Member, IEEE*, and Chenyang Lu, *Fellow, IEEE*

**Abstract**—In recent years, there has been significant interest in developing real-time schedulers for parallel tasks. Most of that research has concentrated on idealized task models where tasks do not access any shared resources protected with locks. In this paper, we consider the problem of scheduling parallel tasks which experience contention due to shared resources. In particular, we provide a schedulability test for federated scheduling by deriving blocking time analyses for parallel tasks that access shared resources protected by FIFO-ordered and priority-ordered spin locks. Our numerical evaluation on randomly generated task sets indicates that priority-ordered locks generally provide better schedulability results than FIFO-ordered locks. We also incorporated both FIFO-ordered and priority-ordered spin lock implementations into a federated scheduling platform, which is able to schedule parallel tasks written with OpenMP. Via empirical evaluations, we found that priority-ordered locks also have better performance than FIFO-ordered locks in practice.

**Index Terms**—Real-time synchronization, spin locks, parallel scheduling, blocking analysis

## 1 Introduction

The last few years have seen increasing interest in designing real-time schedulers and schedulability tests for *parallel tasks*—tasks with internal parallelism that can use multiple cores at the same time [1], [2], [3], [4], [5], [6], [7]. Such parallelism is essential for modern tasks with high computational demands, where the worst-case computational requirement exceeds the relative deadline [8], [9].

Many real-world applications use locks to synchronize accesses to shared resources, such as variables, data structures, or shared objects [10], [11]. Since tasks may have to wait to get these locks, this form of synchronization requires that the schedulability analyses take these waiting times into account. While researchers have developed many schedulability analyses for sequential tasks with locks (for both uniprocessors and multiprocessors) [12], [13], [14], [15], [16], [17], [18], [19], [20], no such analyses have been developed for parallel tasks.

Parallel tasks present additional challenges with respect to synchronization. Due to the parallel structure of a task, multiple cores running different parts of the same task can access the same resource concurrently, causing one core to block the others. Thus, even with a single parallel task, adding critical sections changes its schedulability, which is not the case for sequential tasks. Therefore, for parallel tasks, we must consider both inter-task and intra-task contentions for shared resources. For similar reasons, multiple concurrent critical sections of a task can block other tasks at the same time—making the blocking time worse. Finally, unlike sequential tasks, two critical sections $A$ and $B$ of the same task may execute in different orders: $A$ may appear before, after or concurrently with $B$, because internal scheduling of this task may vary from execution to execution (see further explanation in Section 3). This makes the blocking time analysis much harder for parallel tasks.

In this paper, we provide a schedulability analysis for task sets with parallel tasks that contain critical sections. Our schedulability analysis is designed for *federated schedulers* [7], [21], [22], [23], which generalize partitioned scheduling to parallel tasks, and execute each task with high-computational demand under a work-conserving scheduler on a set of dedicated cores. Specifically, federated schedulers use three parameters of each parallel task to calculate and assign the *minimum number of dedicated cores* to that task: *work*—its worst-case execution time on a single core, *critical-path length*—its worst-case execution time on hypothetically infinite number of cores, and *relative deadline*—a time interval from a job release of the task, within which it must complete.

Due to this dedicated core assignment, federated schedulers provide specific advantages when analyzing schedulability for tasks with critical sections. First, they eliminate priority inversions since tasks do not share cores and therefore do not need prioritization. Second, for the same reason, if tasks do not access the same lock, they do not interfere with each other for either CPU cycles or the shared resource. Lastly, with global scheduling, up to $m$ (the number of cores on the machine) critical sections of the same task may be concurrent in the case of spin locks; federated scheduling limits such blocking since tasks can only execute on limited numbers of assigned cores.

To incorporate blocking times under federated schedulers, we specifically use spin locks to synchronize accesses to shared resources—that is, in case of contention, each thread

- *S. Dinh, K. Agrawal, C. Gill, and C. Lu are with the Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO 63130. E-mail: {sonndinh, kunal, lu}@wustl.edu, cdgill@cse.wustl.edu.*
- *J. Li is with the Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102. E-mail: jingli@njit.edu.*

*Manuscript received 30 Mar. 2017; revised 4 Sept. 2017; accepted 31 Oct. 2017. Date of publication 28 Nov. 2017; date of current version 9 Mar. 2018. (Corresponding author: Son Dinh.)*
*Recommended for acceptance by J. N. Amaral.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TPDS.2017.2777454*

will spin until its turn to access the resource. Since parallel tasks are assigned dedicated cores, the blocking time of a task on a core cannot be utilized by other tasks. Moreover, Brandenburg et al. [10], [11] have studied the distribution of critical section length in the Linux kernel and in several real-time applications and found that most critical sections are short (e.g., 95 percent of them are shorter than $5\ \mu s$ in their benchmarks). Therefore, spin locks would be a better choice for most cases since unlike suspension-based locks, spin locks do not trigger making scheduling decisions, or preemptions, or migrations which are high overhead OS kernel operations. Spin locks instead can be implemented simply and efficiently in user-space and generally have lower overhead than suspension-based locks.

We derive the schedulability of tasks with critical sections under federated schedulers as follows. We calculate the worst-case blocking times of a task and use them to inflate its work and critical-path length, when calculating the number of assigned cores to this task. Since the blocking times of a task actually depend on the number of cores assigned to each task, we use a fixed-point algorithm to iteratively calculate the blocking times and core assignment.

Note that this schedulability test only requires basic information about the tasks including worst case execution time (work), critical-path length (span), relative deadline, the number of critical sections for each shared resource, and their lengths; it does not require the internal graph structure of the parallel task. This property confers a few advantages. First, this simplifies the test and makes it tractable. The required information about the parallel tasks also can be collected easily. Second, a common algorithm design techniques for parallel programs is divide-and-conquer in which data is divided into multiple portions and processed by multiple threads concurrently. Parallel programs written using this approach are data-dependent and the internal graph structure of such a program can only be unfolded at run time by the runtime system of the parallel language in which the program is written. Thus, the exact graph structure of a parallel task can vary from one release to the next. Finally, federated scheduling is flexible in that it can use any greedy (work conserving) scheduler; therefore, even for the same graph structure the execution can differ from one invocation to the next due to changes in the internal scheduling of the graph. If instead we wished to use information about the structure of the task, we would have to examine all of its possible internal graph structures, and for each graph structure, either we would have to fix the schedule statically, or the analysis would have to consider all possible internal schedules for that structure, which would make the analysis very expensive. Fixing the schedule also makes it difficult to guarantee greediness, which is required to ensure parallel tasks to meet their deadlines under federated scheduling. In addition, as a practical matter, it makes implementation more difficult: e.g., one would not be able to simply use OpenMP's work conserving scheduler out of the box as our current implementation does.

In this paper, we focus primarily on systems containing only high-utilizations tasks—tasks with utilization greater than 1—since these are the tasks that require parallel execution to meet their deadlines. In particular, these are tasks for which federated schedulers assign dedicated cores. Such systems are not uncommon in emerging real-time applications such as hybrid structural simulation [9] where each task is a numerical simulation and has high computation demand relative to its deadline. Towards the end of this paper, we will discuss the challenges involved in task sets with both low and high-utilization tasks as well as some potential strategies for addressing these challenges.

Our contributions in this paper are as follows. For parallel tasks with critical sections (described in Section 2), we consider tasks using two types of spin locks: (i) FIFO-ordered in which requests to a resource are satisfied in first-come-first-served order; and (ii) Priority-ordered in which requests to a resource are satisfied based on their priorities—higher priority requests are guaranteed to acquire the lock before lower priority ones. Note that these priorities are only used to arbitrate the locks—the tasks never share cores and therefore no priority is needed for scheduling tasks on their cores. We describe the priority for requests in Section 2. We provide a high-level overview of the schedulability analysis in Section 3. We first analyze *intra-task blocking*—the blocking experienced due to critical sections within the task (Section 4), and then analyze *inter-task blocking*—the blocking experienced due to critical sections from other tasks (Section 5). We then summarize and discuss the results of our analysis in Section 6. To compare the performance between FIFO-ordered and priority-ordered locks, we first conduct numerical evaluations and observe that priority-ordered locks generally provide better schedulability (Section 7). In addition, we implement both FIFO-ordered and priority-ordered spin locks in a federated scheduling platform that can execute parallel programs written in OpenMP. We run empirical experiments and show results that indicate that the platform with priority-ordered locks also can schedule parallel tasks better than the one with FIFO-ordered locks in practice (Section 8).

## 2 Model and Background

In this section, we describe our task model. We also briefly explain the federated scheduling algorithm.

*Parallel Task Model.* We consider a set $\tau$ containing $n$ implicit deadline, sporadic real-time tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$, scheduled on $m$ identical processors. Each task $\tau_i \in \tau$ has its minimum inter-arrival time (i.e., *period*) $T_i$ equal to its *relative deadline* $D_i$. We use $J_i$ to denote any job of the task $\tau_i$. For each task, a job must finish before the next job is released. Our model is thus different than the pipeline model, where multiple invocations of a task can run simultaneously. In the pipeline parallel model, the task graph is usually well defined, which is also different than our task model. Each job of a task $\tau_i$ can be described as a dynamically unfolding *directed acyclic graph (DAG)* in which each node (subjob) represents a sequence of instructions (a strand) and each edge represents a precedence constraint between nodes. A node is *ready* to be executed when all of its predecessors in the graph have been executed. The exact DAG of a task depends on the input data and is unfolded by the runtime system of the parallel programming language in which the task is written, such as OpenMP or Cilk Plus. Thus, the DAG of a task can be different for each release and is only known at run time. Each parallel task $\tau_i$ has two parameters:

- The *work* (or worst-case execution time) $C_i$ of task $\tau_i$ is the sum of execution times of all nodes of task $\tau_i$.

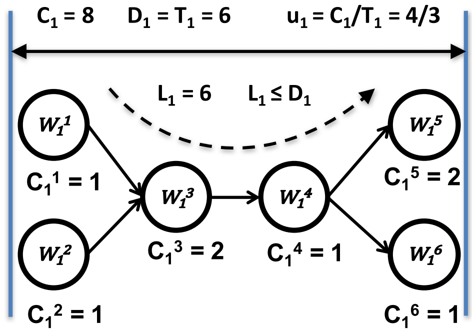


Fig. 1. Example parallel task $\tau_1$ with work $C_1 = 8$ and critical-path length $L_1 = 6$.

- The *critical-path length* (or span) $L_i$ of task $\tau_i$ is the length of the longest weighted path in the DAG of $\tau_i$, where the weight of each node is its execution time.

These two parameters can be obtained using profiling tools for the particular parallel language used to write the parallel tasks.

Utilization of task $\tau_i$ is defined as $u_i = \frac{C_i}{T_i} = \frac{C_i}{D_i}$. Fig. 1 shows an example of a parallel task. Its work is the sum of all nodes in the task $C_1 = 8$, and its critical-path is the path following the dashed line with the length of $L_1 = 6$. A job is said to be pending in an interval starting from its arrival until its completion. We use $njobs(\tau_j, t) = \left\lceil \frac{t+r_j}{T_j} \right\rceil$ [24], where $r_j$ is the response time of $\tau_j$, to denote the maximum number of jobs of $\tau_j$ that can be pending in an interval of length $t$.

*Resource Sharing.* We augment the parallel task model to include access to shared resources. Let $Q_i$ denote a set of resources that $\tau_i$ accesses. The maximum number of times a job of task $\tau_i$ accesses a resource $l_q \in Q_i$ is $R_{i,q}$.

We consider serially reusable and non-preemptable resources that require mutual exclusion to enforce the consistency of their state, such as shared objects. Each shared resource is protected by a distinct spin lock. Threads of a task must acquire the lock on a shared resource before accessing the resource. If the lock is already held by another thread, it must spin non-preemptively until its turn. Threads also run non-preemptively during critical sections. Outside of critical sections and spinning durations, threads are preemptible. We overload notation and use $l_q$ to denote both the resource and the spin lock that protects it. We also use *critical section* and *request* interchangeably—they both denote a code segment executed on a core that is required to be serialized with other code segments accessing the same resources executed on other cores. The maximum length of a critical section (the amount of time it holds a lock) of a job of $\tau_i$ for resource $l_q$ is denoted by $\Phi_{i,q}$. We assume there are no nested critical sections; that is, a task can only hold one lock at a time.

We consider two orders for spin locks: FIFO-ordered and priority-ordered. In priority-ordered spin locks, requests from a task to a resource have a priority, which we call *locking-priority* to distinguish them from conventional task priorities in the real-time systems literature. Requests to a shared resource from different tasks are satisfied in order of their locking-priorities. We assume that requests from a task have the same locking-priority; in other words, the locking-priorities are task-level fixed priorities. Requests for a shared resource issued by the same task are satisfied in FIFO order. Later, in our evaluation (Section 7), we investigate different methods to assign locking-priorities for resource requests.

*Federated Scheduling for Parallel Tasks.* Federated scheduling is a generalization of partitioned scheduling for parallel tasks. Li et al. [7] prove the following lemma.

**Lemma 1.** *Given a high-utilization, implicit deadline, parallel task ($u_i > 1$) with work $C_i$, critical-path length $L_i$ and deadline $D_i$, $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ dedicated cores are sufficient to guarantee that all jobs of this task will meet their deadlines, when scheduled with a greedy scheduler.*

Note that in federated scheduling, high-utilization tasks are scheduled with a greedy scheduler—one that never keeps a core idle if some node is ready to execute. For low-utilization tasks, federated scheduling treats them as sequential tasks and uses an existing scheduling algorithm such as P-EDF to schedule them on the remaining cores. They also prove a *capacity augmentation bound* of 2 for federated scheduling [7]. In particular, they state that a task set $\tau$ is schedulable under federated scheduling provided that it satisfies the following two conditions: (i) the total utilization of the task set is no more than half of the computing capacity of the system, $\sum_{i=1}^{n} u_i \leq \frac{m}{2}$, and (ii) for all tasks, the critical path length is no more than half the corresponding relative deadline of that task, $\forall i, L_i \leq \frac{D_i}{2}$. These two conditions can serve as a sufficient test for federated scheduling; however, task sets that do not satisfy these conditions may still be schedulable under federated scheduling if after the high-utilization tasks are allocated their cores, the low-utilization tasks can be scheduled on the remaining cores. In this work, we do not require the task sets to satisfy the two conditions above.

## 3 Schedulability Analysis

In this section, we present a schedulability analysis for federated scheduling with parallel real-time tasks sharing resources that are protected by FIFO-ordered and priority-ordered locks. We first calculate the number of dedicated cores $n_i$ for each task with no blocking delays. We then repeatedly use these assignments to calculate an upper bound on blocking delays experienced by each task and recalculate the new assignments (new sets of $n_i$ for all tasks) based on these blocking delays, until a fixed point is found.

*Work Blocking and Critical-Path Blocking.* The blocking experienced by a parallel task has two components: work blocking and critical-path blocking, defined as follows.

**Definition 1.** *Work blocking $B_i^C$ is an upper bound on the total amount of time that all the cores assigned to $\tau_i$ spend spinning (collectively), waiting for lock requests to be granted.*

**Definition 2.** *Critical-path blocking $B_i^L$ is an upper bound on the amount of spinning time accumulated along any single path of the DAGs of task $\tau_i$, waiting for lock requests on that path to be granted.*

Intuitively, each execution of a job $J_i$ of task $\tau_i$ can be mapped to a hypothetical DAG $G'$, which is constructed by augmenting $J_i$'s DAG $G$ with "spinning nodes"—nodes that represent spinning intervals on that specific execution. For all possible executions of jobs of task $\tau_i$, we have a set of such hypothetical DAGs. Then the accumulated spinning time along any single path of any hypothetical DAG in that set is bounded by the critical-path blocking $B_i^L$.

*Calculating the Core Assignments.* Given $B_i^C$ and $B_i^L$ for task $\tau_i$, we calculate the "inflated work" as $C_i + B_i^C$ and the "inflated critical-path length" as $L_i + B_i^L$.

**Lemma 2.** *Given a parallel task $\tau_i$ with $u_i = \frac{C_i}{D_i} > 1$, work blocking $B_i^C$, and critical-path blocking $B_i^L$ such that $D_i > L_i + B_i^L$, $n_i$ dedicated cores are sufficient to guarantee that all jobs of this task will meet their deadlines when scheduled with a greedy scheduler, where*

$$n_i = \left\lceil \frac{C_i + B_i^C - L_i - B_i^L}{D_i - L_i - B_i^L} \right\rceil. \tag{1}$$

**Proof.** Consider an arbitrary execution $E$ (i.e., schedule) of $\tau_i$ on its current number of dedicated cores. We construct a hypothetical DAG $G'$ from $E$ by inflating the nodes of the DAG $G$ of $\tau_i$ with the corresponding spinning times. By definition of $B_i^C$ and $B_i^L$, the work $C_i'$ of $G'$ is bounded by $C_i + B_i^C$, and the critical-path length $L_i'$ of $G'$ is bounded by $L_i + B_i^L$. Therefore, applying Lemma 1 gives us the number of cores required for $G'$: $\left\lceil \frac{C_i' - L_i'}{D_i - L_i'} \right\rceil \leq \left\lceil \frac{C_i + B_i^C - L_i - B_i^L}{D_i - L_i - B_i^L} \right\rceil$. Thus, the number of cores assigned to $\tau_i$ by Equation (1) is sufficient to schedule the DAG $G'$. In other words, it is sufficient to guarantee that all jobs of task $\tau_i$ with work blocking $B_i^C$, and critical-path blocking $B_i^L$ will meet their deadlines. □

*Schedulability Test.* Algorithm 1 shows the pseudocode for the schedulability test. We start by setting $B_i^C$ and $B_i^L$ to 0 for all tasks and calculate the set of $n_i$. We then repeatedly calculate new values of $B_i^C$ and $B_i^L$ and use Equation (1) to calculate the new $n_i$ for each task in alternating fashion. Note that both $B_i^L$ and $B_i^C$ depend on (i) the numbers of requests by all tasks that access the same resources that task $\tau_i$ accesses, and (ii) the numbers of cores allocated to all tasks. Meanwhile, the number of cores $n_i$ allocated to each task $\tau_i$ depends on both $B_i^L$ and $B_i^C$. Therefore, the algorithm runs iteratively until a fixed point is reached.

The algorithm declares a task set unschedulable if either the inflated critical-path length $L_i + B_i^L \geq D_i$ or the total number of cores allocated across all tasks ($\sum_i n_i$) exceeds $m$, the total number of available cores. Otherwise, the task set is declared schedulable once no task's core allocation changes. If at any step, the new calculated $n_i$ is ever smaller, we keep the previous value. This is because the dependency between the blocking delays ($B_i^L, B_i^C$) and the core allocation ($n_i$) is complex, it is hard to assure that the core allocation to each task increases monotonically after each iteration. However, this is safe because the task is assigned at least the number of cores required to satisfy its deadline, given the calculated blocking $B_i^C$ and $B_i^L$ at this step. When a fixed point is reached, i.e., the task set is deemed schedulable, both the blocking delays and the numbers of cores allocated to every tasks do not change anymore. This means that with this core allocation, every tasks in the task set meet their deadlines, with their blocking delays accounted.

In this algorithm, the crucial step is the calculation of $B_i^L$ and $B_i^C$—the blocking times for each task $\tau_i$ on the critical-path and work, respectively. We divide the blocking into two portions: *intra-task blocking* is caused by concurrent critical sections in the same task and *inter-task blocking* is caused by critical sections in other tasks. We describe this calculation in the next two sections (Sections 4 and 5), for both FIFO-ordered and priority-ordered locks. The overall work blocking and critical-path blocking for the two types of spin locks are presented in Section 6.

**Algorithm 1.** Schedulability Test

```
1: procedure CALCULATEBLOCK($\tau_i, n_1, \ldots, n_n$)
2:     Calculate and return $B_i^C$ and $B_i^L$
3: end procedure
4: procedure ISSCHEDULABLE($\tau$)
5:     for each $\tau_i \in \tau$ do $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$
6:     end for
7:     while (true) do
8:         for each $\tau_i \in \tau$ do
9:             $B_i^C, B_i^L$ = CALCULATEBLOCK($\tau_i, n_1, \ldots, n_n$)
10:            if ($L_i + B_i^L \geq D_i$) then
11:                Return Unschedulable
12:            end if
13:            $n_i' = \left\lceil \frac{C_i + B_i^C - L_i - B_i^L}{D_i - L_i - B_i^L} \right\rceil$
14:        end for
15:        if ($\sum_{i=1}^{n} n_i' > m$) then
16:            Return Unschedulable
17:        end if
18:        if (No task's core assignment changes) then
19:            Return Schedulable
20:        end if
21:        for each $\tau_i \in \tau$ do
22:            if $n_i' > n_i$ then
23:                $n_i = n_i'$
24:            end if
25:        end for
26:    end while
27: end procedure
```

## 4 Calculating Intra-Task Blocking

In this section, we present upper-bounds on the intra-task blocking components for the work-blocking $B_i^C$ and the critical-path blocking $B_i^L$ of task $\tau_i$. Since all requests from the same task have the same priority, they are satisfied in FIFO order for both FIFO-ordered and priority-ordered locks. Therefore, the bounds for intra-task blocking for both lock types are identical.

*Intra-Task Work Blocking.* We consider the intra-task blocking component for the work blocking $B_i^C$ of task $\tau_i$.

**Lemma 3.** *For a shared resource $l_q$, the intra-task blocking of task $\tau_i$ is upper bounded by: $Intra\_Wb = \big(\frac{\min(R_{i,q}, n_i)\cdot(\min(R_{i,q}, n_i)-1)}{2} + (n_i - 1)\cdot\max(R_{i,q} - n_i, 0)\big)\cdot\Phi_{i,q}$.*

**Proof.** Consider the case when the number of requests to $l_q$ from $\tau_i$ is larger than its number of cores, i.e., $R_{i,q} \geq n_i$. In this case, the above expression becomes $\big(\frac{n_i\cdot(n_i-1)}{2} + (n_i - 1)\cdot(R_{i,q} - n_i)\big)\cdot\Phi_{i,q}$. First, we observe that a particular request $r$ of a job of $\tau_i$ can wait for at most $n_i - 1$ other requests: the job has $n_i$ total cores and since a core cannot issue another request until its previous requests are satisfied, there can be at most $n_i - 1$ unsatisfied requests when $r$ arrives. Since the requests are satisfied in FIFO order, $r$ can only wait for requests that were already in the queue when it arrived. Second, not all requests can block for this long. In particular, the first request from the job does not block on any other request from this job. Similarly, the second request waits (blocks) only for the first request; the third waits for at most two, and so on. Thus, the first $n_i$ requests block for at most $1 + 2 + 3 + \cdots + (n_i - 1) = \frac{n_i \cdot (n_i - 1)}{2}$ requests. The remaining $R_{i,q} - n_i$ requests each may have to wait for $n_i - 1$ preceding requests, giving the total bound.

For the case when $R_{i,q} < n_i$, the expression becomes $\left(\frac{R_{i,q} \cdot (R_{i,q} - 1)}{2}\right) \cdot \Phi_{i,q}$. Similar to the above observation, the maximum blocking happens when all $R_{i,q}$ requests arrive at once, each from a core of $\tau_i$. The total blocking is then $(1 + 2 + 3 + \cdots + (R_{i,q} - 1)) \cdot \Phi_{i,q} = \frac{R_{i,q} \cdot (R_{i,q} - 1)}{2} \cdot \Phi_{i,q}$. □

*Intra-Task Critical-Path Blocking.* We now present the intra-task blocking component for the critical-path blocking $B_i^L$ of $\tau_i$. Consider a path $P$ of the DAG of $\tau_i$ which contains $Y$ critical sections of lock $l_q$. The blocking collected along $P$ due to contention for $l_q$ within the task $\tau_i$ is bounded as follows.

**Lemma 4.** *The intra-task blocking collected along a path $P$ of task $\tau_i$ which contains $Y$ critical sections of lock $l_q$ is bounded by: Intra_Cpb$(Y) = \min\{(n_i - 1) \cdot Y, R_{i,q} - Y\} \cdot \Phi_{i,q}$.*

**Proof.** Since at any time, only one processor executes nodes on the path $P$, each request from $P$ can be blocked by at most $n_i - 1$ requests from the other processors of $\tau_i$. Summing over all $Y$ requests of $P$ gives us the first term. Since the total number of requests of the other processors of $\tau_i$ to resource $l_q$ is $R_{i,q} - Y$, at most that many requests can block requests in $P$. Because both bounds must hold, the minimum of the two quantities is an upper bound on the intra-task blocking. □

This allows us to find the maximum intra-task blocking that can accumulate on path $P$ for lock $l_q$ by considering all possible values for $Y \in [1, R_{i,q}]$. Repeating the calculation for all locks that $\tau_i$ accesses and summing them up gives the upper-bound on the blocking that any single path of $\tau_i$ may incur due to contention within the same task.

## 5 Calculating Inter-Task Blocking

In this section, we present upper-bounds for the inter-task blocking component of the work-blocking $B_i^C$ and critical-path blocking $B_i^L$ of task $\tau_i$. Here, the bounds for inter-task blocking are different for FIFO-ordered and priority-ordered locks. We first show the bounds for FIFO-ordered locks.

### 5.1 Inter-Task Blocking for FIFO-Ordered Locks

*Inter-Task Work Blocking.* We first bound the inter-task component of the work blocking for FIFO-ordered locks.

**Lemma 5.** *For FIFO-ordered spin locks, the inter-task blocking of task $\tau_i$ with respect to resource $l_q$ caused by a task $\tau_j$ is upper bounded by: Inter_Wb_Fifo$(\tau_j) = \min\{R_{i,q} \cdot n_j, njobs(\tau_j, D_i) \cdot R_{j,q} \cdot n_i\} \cdot \Phi_{j,q}$.*

**Proof.** The first term is derived by calculating the maximum number of requests from $\tau_j$ that can block a particular request from $\tau_i$. Since requests are satisfied in FIFO order and $\tau_j$ has $n_j$ cores, each request from $\tau_i$ can wait for at most $n_j$ requests from $\tau_j$. Summing over $\tau_i$'s $R_{i,q}$ requests, at most $R_{i,q} \cdot n_j$ requests from $\tau_j$ can block $\tau_i$ overall. The second term is derived by calculating the maximum number of requests from $\tau_i$ that can be blocked by a particular request $r$ of $\tau_j$. Note that $r$ can block at most $n_i$ requests of $\tau_i$, i.e., one request per core. While a job of $\tau_i$ is pending, at most $njobs(\tau_j, D_i)$ jobs of $\tau_j$ can execute; therefore, at most $njobs(\tau_j, D_i) \cdot R_{j,q}$ requests of $\tau_j$ can arrive while $\tau_i$ is pending. Since both quantities are independently upper bounds, the blocking cannot exceed either of them. Therefore, the minimum of the two quantities is also an upper bound. □

*Inter-Task Critical-Path Blocking.* Again, we consider a path $P$ of $\tau_i$ that contains $Y$ critical sections of lock $l_q$. The blocking of $P$ with respect to resource $l_q$ caused by contention with other tasks is bounded as follows.

**Lemma 6.** *For FIFO-ordered spin locks, the inter-task blocking accumulated along a path $P$ of $\tau_i$ which contains $Y$ requests to $l_q$ is bounded by: Inter_Cpb_Fifo$(Y) = \sum_{\tau_j \in \tau \setminus \{\tau_i\}} \min\{n_j \cdot Y, njobs(\tau_j, D_i) \cdot R_{j,q}\} \cdot \Phi_{j,q}$.*

**Proof.** If task $\tau_j$ contends with $\tau_i$ for lock $l_q$, for each request $r$ to $l_q$ from $P$, at most $n_j$ requests of $\tau_j$ can block $r$. Since $P$ has $Y$ requests to $l_q$, it can be blocked by at most $n_j \cdot Y$ requests of $\tau_j$. The inter-task blocking on $P$ is also bounded by the maximum number of requests of $\tau_j$ that can interfere with requests from $P$. Because at most $njobs(\tau_j, D_i)$ jobs of $\tau_j$ can interfere with a job of $\tau_i$ and each job of $\tau_j$ has total $R_{j,q}$ requests, the inter-task blocking is bounded by $njobs(\tau_j, D_i) \cdot R_{j,q}$. Taking the minimum of the two quantities and summing over all tasks $\tau_j$ give us the bound on inter-task blocking. □

### 5.2 Inter-Task Blocking for Priority-Ordered Locks

For each shared resource, each task has a *distinct* locking-priority that is used to choose the next request to obtain the lock—a request with a higher locking-priority should acquire the lock before a request with a lower one. Note that locking-priorities are only used to schedule requests to shared resources, and not to schedule the tasks themselves since tasks do not share cores. In this section, we assume the locking-priorities have already been assigned for the tasks. Our evaluation in Section 7 explores three possible strategies for assigning locking-priorities. Also, as was mentioned in Section 4, all requests from the same task have the same priority.

In order to bound the inter-task blocking for priority-ordered locks, we must calculate a quantity called "delay-per-request", which bounds the maximum amount of time a request can be delayed due to contention with other requests for the same resource. This allows us to compute the maximum number of interfering jobs that can be pending while the request is delayed, which can then be used to bound the maximum blocking experienced by the task. For simplicity of exposition, in this section we assume that all requests have identical length $\Phi$. We generalize the blocking bounds for different request lengths in Appendix B.

*Delay Per Request Calculation.* Let $dpr(\tau_i, l_q)$ denote the maximum time a single request of task $\tau_i$ may have to wait until it acquires lock $l_q$. Specifically, a request from $\tau_i$ may have to wait for requests with higher, lower, and equal locking-priorities before it can get the lock and enter its critical section. Let $\tau_i^{LP}$ and $\tau_i^{HP}$ denote a set of tasks with lower and higher locking-priorities than $\tau_i$'s, respectively. The "delay-per-request" $dpr(\tau_i, l_q)$ is thus computed by: $dpr(\tau_i, l_q) = higher(\tau_i, l_q) + lower(\tau_i, l_q) + equal(\tau_i, l_q)$; where $higher(\tau_i, l_q)$, $lower(\tau_i, l_q)$, and $equal(\tau_i, l_q)$ are the delays caused by higher, lower, and equal locking-priority requests, respectively.

Since each request of $\tau_i$ can only be preceded by at most one lower locking-priority request, $lower(\tau_i, l_q) = \Phi$, which is the length of a single request. The third term, $equal(\tau_i, l_q)$ accounts for the delay caused by equal locking-priority requests that can only originate from the same task $\tau_i$, since each task has a distinct priority. Hence, $equal(\tau_i, l_q) = \min(n_i - 1, R_{i,q} - 1)\Phi$ due to the FIFO ordering between requests from the same task. Note that the calculation of $equal(\tau_i, l_q)$ is similar to that of intra-task critical-path blocking (Section 4), except that it only accounts for the delay incurred by a single request of $\tau_i$ to $l_q$.

The calculation of $higher(\tau_i, l_q)$ is more involved. Observe that a request of $\tau_i$ potentially can be delayed by requests from multiple jobs of a task with higher locking-priority since jobs may complete and new jobs arrive while this request is still waiting. Therefore,

$$higher(\tau_i, l_q) = \sum_{\tau_j \in \tau_i^{HP}} njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot \Phi.$$

This calculation is derived as follows. Since the number of requests of $\tau_j$ in the interval equal to $dpr(\tau_i, l_q)$ is bounded by $njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q}$. Summing over all higher locking-priority tasks gives us the equation. Since $higher(\tau_i, l_q)$ and $dpr(\tau_i, l_q)$ depend on each other, we use a fixed-point calculation to determine $dpr(\tau_i, l_q)$. We first set $dpr(\tau_i, l_q)$ to 0 and progressively recalculate it until it converges. However, if $dpr(\tau_i, l_q)$ is larger than $D_i$, the task is unschedulable.

*Inter-Task Work Blocking.* We first state a straightforward lemma for the blocking caused by lower locking-priority tasks.

**Lemma 7.** *The inter-task blocking of task $\tau_i$ caused by lower locking-priority tasks with respect to $l_q$ is at most $Inter\_Wb\_Lower = R_{i,q} \cdot \Phi$.*

**Proof.** Each request $r$ of $\tau_i$ can only be blocked by a single lower locking-priority request that is executing when $r$ arrives. Summing over all requests gives us the bound. □

Next, we bound the blocking caused by higher priority tasks.

**Lemma 8.** *The inter-task blocking of task $\tau_i$ caused by higher locking-priority tasks with respect to resource $l_q$ is bounded by: $Inter\_Wb\_Higher = \sum_{\tau_j \in \tau_i^{HP}} \min\{njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot R_{i,q}, njobs(\tau_j, D_i) \cdot R_{j,q} \cdot n_i\} \cdot \Phi$.*

**Proof.** $dpr(\tau_i, l_q)$ is the maximum delay a particular request of job $\tau_i$ can incur. During this time, there can be at most $njobs(\tau_j, dpr(\tau_i, l_q))$ jobs of $\tau_j$. Hence, the total number of requests from $\tau_j$ that can delay a particular request of $\tau_i$ is at most $njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q}$. Multiplying by the total number of requests of $\tau_i$ gives us the maximum inter-task blocking caused by higher locking-priority requests of $\tau_j$.

Recall that $njobs(\tau_j, D_i) \cdot R_{j,q}$ is an upper bound on the number of requests to $l_q$ sent by jobs of task $\tau_j$ when $J_i$ is pending. Each of these requests can block at most $n_i$ requests of $\tau_i$. Thus, the inter-task blocking of $\tau_i$ caused by $\tau_j$ is also bounded by $njobs(\tau_j, D_i) \cdot R_{j,q} \cdot n_i$. Hence, the inter-task blocking is bounded by the minimum of the two. Summing over all higher locking-priority tasks in $\tau_i^{HP}$ yields the inter-task blocking caused by higher locking-priority tasks. □

*Inter-Task Critical-Path Blocking.* Again, we consider a path $P$ of task $\tau_i$ which contains $Y$ critical sections of lock $l_q$.

**Lemma 9.** *For priority-ordered spin locks, the inter-task blocking accumulated on path $P$ of task $\tau_i$ that has $Y$ requests to $l_q$ due to contention for $l_q$ with lower locking-priority tasks is at most $Inter\_Cpb\_Lower(Y) = Y \cdot \Phi$.*

**Proof.** The proof is similar to Lemma 7, except that path $P$ only has $Y$ requests in total. □

**Lemma 10.** *For priority-ordered spin locks, the inter-task blocking accumulated on path $P$ of task $\tau_i$ that has $Y$ requests to $l_q$ due to contention for $l_q$ with higher locking-priority tasks is bounded by: $Inter\_Cpb\_Higher(Y) = \sum_{\tau_j \in \tau_i^{HP}} \min\{njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot Y, njobs(\tau_j, D_i) \cdot R_{j,q}\} \cdot \Phi$.*

**Proof.** Similar to Lemma 8, the number of requests of $\tau_j$ that can delay a particular request of $\tau_i$ is bounded by $njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q}$. Since path $P$ has $Y$ requests to $l_q$, the first term is derived directly.

The second term is derived by calculating the number of requests of $\tau_j$ that can interfere with a job of $\tau_i$. There are $njobs(\tau_j, D_i) \cdot R_{j,q}$ such requests. It is also an upper bound on the number of requests of $\tau_j$ that can contribute to the blocking accumulated on path $P$. Since both bounds must hold, we take the minimum of these two quantities. Summing over all higher locking-priority tasks in $\tau_i^{HP}$ gives us the bound. □

## 6 Overall Blocking Bounds

We now present the overall bounds for both FIFO-ordered and priority-ordered locks, by combining the results from Sections 4 and 5.

### 6.1 Bounds for FIFO-Ordered Locks

To calculate the total work blocking for FIFO-ordered locks, we simply combine the results from Lemmas 3 and 5 and take the sum for all resources that the task accesses.

**Theorem 11.** *The work blocking $B_i^C$ of task $\tau_i$ with FIFO-ordered spin locks is upper bounded by*

$$B_i^C \leq \sum_{l_q \in Q_i} \Big( Intra\_Wb + \sum_{\tau_j \in \tau \setminus \{\tau_i\}} Inter\_Wb\_Fifo(\tau_j) \Big).$$

Similarly, the critical-path blocking is calculated by combining the results of Lemmas 4, and 6.

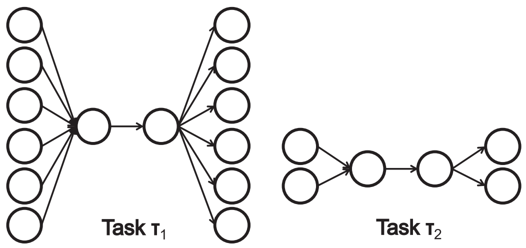


Fig. 2. Example task set with FIFO-ordered spin locks. Each node is a unit node with a work of 1 time unit.

**Theorem 12.** *The critical-path blocking $B_i^L$ of task $\tau_i$ with FIFO-ordered spin locks is upper bounded by*

$$B_i^L \leq \sum_{l_q \in Q_i} \max_{Y \in [1, R_{i,q}]} \Big( Intra\_Cpb(Y) + Inter\_Cpb\_Fifo(Y) \Big),$$

*where, for each resource $l_q$, $Y$ is the number of requests to $l_q$ in a single path of task $\tau_i$.*

**Proof.** By definition, the critical-path blocking of task $\tau_i$ is an upper bound on the blocking that can be collected along a single path $P$ of $\tau_i$. Note that in general, as $Y$ increases the intra-task blocking on $P$ reduces, while the inter-task blocking increases. Thus, for each resource $l_q$, we can maximize over $Y$ ranging from 1 to $R_{i,q}$, and apply Lemmas 4 and 6 to get the maximum blocking due to $l_q$. In the worst case, the same path experiences maximum blocking due to all resources, so summing over all shared resources that $\tau_i$ accesses gives an upper bound for critical-path blocking. □

*Example for the Analysis of FIFO-Ordered Locks.* As is usual for most blocking analyses, these bounds are not tight for all tasks. Here, we present an example task set where the bounds are tight. Consider a task set with two high-utilization tasks with implicit deadlines: $\tau_1(C_1 = 14, L_1 = 4, D_1 = 10)$ and $\tau_2(C_2 = 6, L_2 = 4, D_2 = 5)$. The DAGs of the tasks are shown in Fig. 2. Both tasks have two requests to a single resource protected by a FIFO-ordered lock; each has length of one time unit. Without blocking, the number of cores allocated to each task is computed by Lemma 1: $n_1 = 2$ and $n_2 = 2$.

We focus on the blocking in task $\tau_1$. For $n_1 = 2$, we can apply Theorem 11 and get $B_1^C \leq 5$. Fig. 3a shows an example schedule where $\tau_1$ does experience the total blocking of 5. Similarly, we can apply Theorem 12 to calculate $B_1^L \leq 4$. Fig. 3b shows an example schedule for the critical-path blocking. Therefore, both bounds are tight for this task set, albeit for different schedules. In both schedules, $\tau_1$ misses deadlines with $n_1 = 2$. The updated number of cores allocated to $\tau_1$, $n_1' = \lceil \frac{14+5-4-4}{10-4-4} \rceil = 6$ (Equation (1)), is sufficient to guarantee that $\tau_1$ meets its deadlines as shown in Fig. 4.

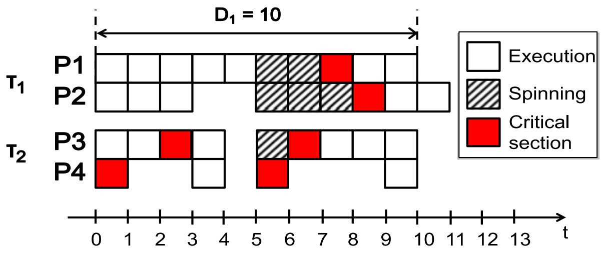
## 6.2 Bounds for Priority-Ordered Locks

We now state the bounds for work and critical-path blocking for priority-ordered locks, assuming that all requests have identical length $\Phi$. The bounds for when request lengths may differ are presented in Appendix B. The proofs are similar to the ones for FIFO locks—for work blocking, we simply sum the bounds from Lemmas 3, 7, and 8 over all shared resources. For critical-path blocking, we combine Lemmas 4, 9, and 10 and maximize over $Y$, the number of critical sections on a single path, for each resource and then sum over all resources.

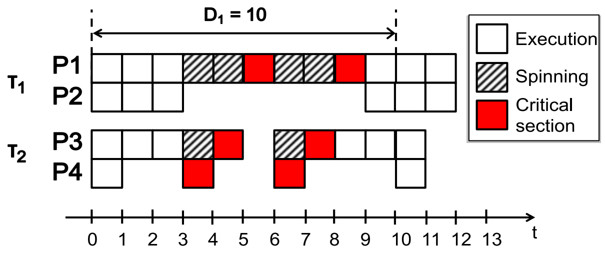
**Theorem 13.** *The work blocking $B_i^C$ of task $\tau_i$ with priority-ordered spin locks is upper bounded by*

$$B_i^C \leq \sum_{l_q \in Q_i} \Big( Intra\_Wb + Inter\_Wb\_Lower + Inter\_Wb\_Higher \Big).$$

**Theorem 14.** *The critical-path blocking of task $\tau_i$ with priority-ordered spin locks is upper bounded by*

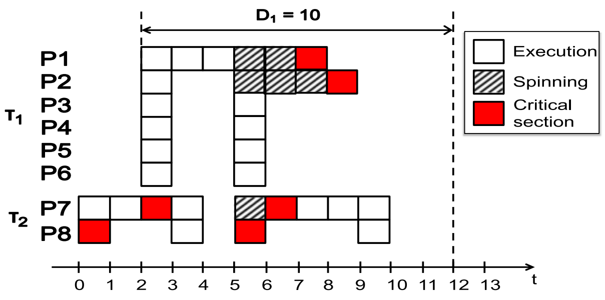


(a) $\tau_1$ misses deadline with the worst case work blocking

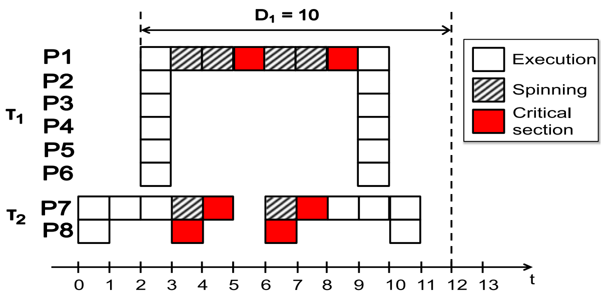


(b) $\tau_1$ misses deadline with the worst case critical-path blocking

Fig. 3. Example schedules that cause worst case work blocking and critical-path blocking for $\tau_1$ with FIFO-ordered spin locks.



(a) $\tau_1$ meets deadline with the worst case work blocking



(b) $\tau_1$ meets deadline with the worst case critical-path blocking

Fig. 4. Example schedules for the updated number of processors for $\tau_1$ with FIFO-ordered spin locks.

$$B_i^L \leq \sum_{l_q \in Q_i} \max_{Y \in [1, R_{i,q}]} \Big( Intra\_Cpb(Y) + Inter\_Cpb\_Lower(Y) + Inter\_Cpb\_Higher(Y) \Big),$$

*where, for each resource $l_q$, $Y$ is the number of critical sections of $l_q$ on a single path of task $\tau_i$.*

We provide an example in which bounds for priority-ordered locks are tight, in Appendix A.

### 6.3 Discussion

We now discuss some qualitative properties of our bounds and some possible avenues for improvement. Given our constraints (we don't know the graph structures of DAG tasks and don't control the internal schedules of their job releases), our individual bounds for work blocking and critical-path blocking are as tight as they can be, in the sense that there exist task sets and respective schedules in which the actual worst case work blocking (critical-path blocking, respectively) is exactly equal to the work blocking bound (critical-path blocking bound, respectively) calculated using our method. However, the worst cases for work blocking and critical-path blocking may not both happen in one schedule. For instance, Fig. 3a shows a schedule in which $\tau_1$ incurs the worst case work blocking of 5 (and critical-path blocking of 3), whereas Fig. 3b shows a different schedule in which $\tau_1$ incurs the worst case critical-path blocking of 4 (and work blocking of 4). The reason is that different arrangements of resource requests among processors of a task may cause different values for work blocking and critical-path blocking for each task, and the one that causes worst case work blocking may not cause worst case critical-path blocking, and vice versa. However, in general, in a schedule where a task incurs its worst case work blocking, its critical-path blocking in that schedule is also close to its worst case critical-path blocking, and vice versa. Because of this, even though the numbers of processors allocated to tasks using Lemma 2 may be more than the actual numbers of processors required by the tasks to be schedulable, the differences are small.

Second, we focus on analyzing tasks with high-utilizations, but our analyses can also be extended to work for task sets with both low and high-utilization tasks. For systems that have low-utilization tasks, if the low-utilization tasks do not share any resources with high-utilization tasks, then we can directly apply an existing synchronization protocol for sequential tasks (e.g., [15], [16]) and apply the respective analysis to low-utilization tasks, since federated scheduling executes them sequentially using existing multiprocessor schedulers (e.g., P-EDF) on the remaining set of cores. However, if the low-utilization tasks share resources with the high-utilization tasks, one possible method is to extend the *Multiprocessor Stack Resource Policy* (MSRP) [16] to resources shared by low- and high-utilization tasks. In particular, shared resources are categorized as *global resources*—resources accessed by tasks from different processors, and *local resources*—resources accessed only by tasks from the same processor. The low-utilization tasks are scheduled sequentially on the remaining set of processors using schedulers such as P-EDF. Local resources can only be shared between low-utilization tasks allocated to the same processor. Therefore, the *Stack Resource Policy* (SRP) [13] can be used to coordinate access to local resources on each processor running low-utilization tasks. For global resources, spin locks can be used to coordinate accesses from both high- and low-utilization tasks. The analysis for high-utilization tasks then also would need to account for blocking caused by low-utilization tasks, e.g., for FIFO-ordered spin locks, a request from a high-utilization task can be blocked by at most one request from each processor allocated to low-utilization tasks. Similarly, low-utilization tasks must account for blocking caused by contention for global resources.

## 7 Numerical Evaluation

In this section, we evaluate our schedulability analysis on randomly generated tasks to see which lock ordering offers better schedulability. We also evaluate the effect of the number of cores, number of tasks in the task set, total utilization, number of shared resources, and number of critical sections, on schedulability.

*Task Set Generation.* We randomly generated task sets for systems with varying numbers of cores $m \in \{12, 24, 36\}$. Periods of the parallel tasks were generated to be $2^\lambda \mu s$, where integer $\lambda$ was uniformly chosen from $[13, 20]$. The periods hence were in the range from approximately 8 milliseconds to 1 second, which covers a wide range of real-time applications. Critical-path lengths of the tasks were generated proportionally to the periods: the ratio between critical-path length and period for each task was chosen uniformly from $\{0.125, 0.125, 0.125, 0.125, 0.1625, 0.1625, 0.1625, 0.1875, 0.1875, 0.25\}$. For each value of $m$, tasks were generated with utilizations in the range $[1.25, \sqrt{m}]$, so that tasks can have higher utilizations on systems with larger numbers of cores, which reveals the scalability of the system. The total utilization of task sets $U$ was varied over $\{0.5m, 0.625m, 0.75m\}$. We used Stafford's RandFixedSum algorithm [25], [26] to generate tasks' utilizations. This algorithm guarantees that the generated utilizations are sampled uniformly in the range $[1.25, \sqrt{m}]$, and that they sum to the chosen total utilization $U$. The algorithm also allows us to control the number of tasks $n$ in the task sets.

The number of shared resources was varied among $\{1, 2, \ldots, n\}$. For each shared resource, we varied the total number of critical sections across all tasks of the task set and randomly assigned them to the tasks. We consider two types of critical section length: short and moderate. The lengths were picked uniformly between $[1\ \mu s, 15\ \mu s]$ for short critical sections, and $[1\ \mu s, 100\ \mu s]$ for moderate ones. In our numerical evaluation, for each parameter configuration we generated 1000 task sets and measured the percentage of schedulable task sets. Schedulability of each task set was also gauged using Algorithm 1.

*Locking-Priority Assignment.* While finding the best method to assign locking-priorities is beyond the scope of this paper, we are interested in the performance of priority-ordered spin locks with a reasonable locking-priority assignment. Hence, we consider three strategies to assign locking-priorities. In the first strategy, *DM*, we simply assign the locking-priorities based on the tasks' relative deadlines (i.e., in a <u>d</u>eadline-<u>m</u>onotonic manner). Therefore, tasks with tighter deadlines get higher locking-priority, which is reasonable since it reflects the urgency of each task. In the second strategy, *OPT*,

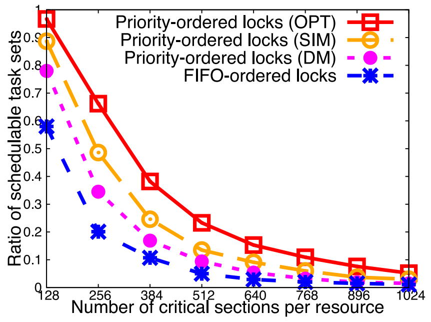


Fig. 5. Schedulability for $m = 36$, $n = 7$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and short critical sections.

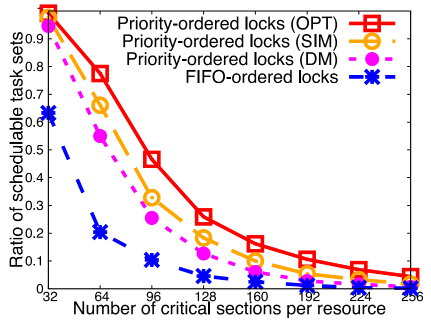


Fig. 6. Schedulability for $m = 36$, $n = 7$, $U = 0.75m$, 4 shared resources, varying number of critical sections per resource, and short critical sections.

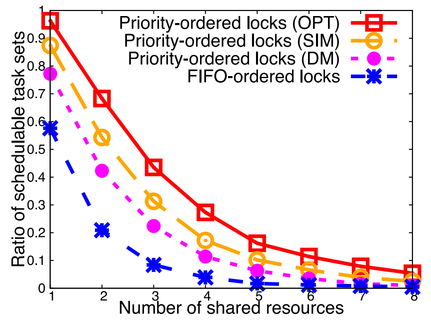


Fig. 7. Schedulability for $m = 36$, $n = 7$, $U = 0.75m$, varying number of resources, 128 requests per shared resource, and short critical sections.

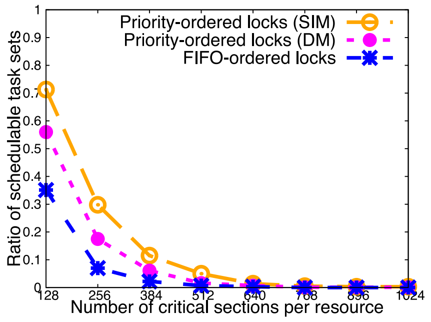


Fig. 8. Schedulability for $m = 36$, $n = 11$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and short critical sections.

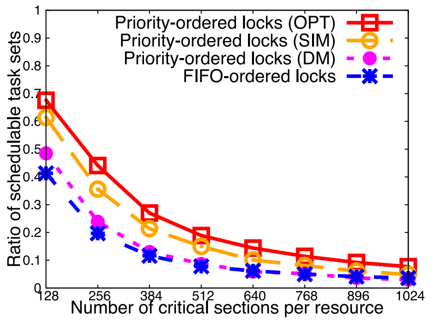


Fig. 9. Schedulability for $m = 12$, $n = 5$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and short critical sections.

we search for an optimal assignment by trying all permutations of priorities. For each permutation, we apply Algorithm 1 to test the schedulability with that assignment. The procedure stops when we find an assignment that renders the task set schedulable, or after all permutations have been checked. This gives us the best results for priority-ordered locks at the cost of increasing the running time of the test. The last strategy, *SIM*, implements simulated annealing to find an approximately optimal locking-priority assignment. We define the cost function for an assignment to be the total number of cores required for the task set to be schedulable. Starting from an original solution, which is the same locking-priority as in *DM*, the algorithm picks a random neighbor solution and goes to the neighbor with a probability returned from an acceptance probability function. It stops when it finds a solution that requires less than $m$ cores or after it finishes all iterations. This method improves the running time for *OPT* without sacrificing too much schedulability for priority-ordered locks.

*FIFO-Ordered versus Priority-Ordered.* Figs. 5, 6, 7, 8 and 9 show the schedulability results for representative settings (other settings have similar trends). Note that in Fig. 8 we truncate the curve for optimal locking-priority assignment since for 11 tasks, the exhaustive search is impractical in the worst case. With FIFO-ordered locks, each task experiences interference from requests by all the other tasks, while for priority-ordered locks most of the interference comes from tasks with higher locking-priorities. On the other hand, priority-ordered locks can cause a single request to be blocked by multiple jobs of the same task, which cannot happen for FIFO-ordered locks. Results indicate that the first effect exceeded the other in our experiments, and thus priority-ordered locks yielded higher schedulability. We also observe that while *DM* gave reasonable schedulability for priority-ordered locks (compared to FIFO-ordered locks), it was much worse than *OPT* (Figs. 5, 6, 7, and 9). *SIM* offers a trade-off between these two: the schedulability was closer to that of *OPT* while the running time improved significantly. For instance, on our machine (see Section 8 for description), the schedulability test for 9 tasks sharing a single resource, where the total number of requests by all tasks to the resource is 896, takes an hour for *OPT*, but only takes a couple of minutes for *SIM*, and a tenth of a second for *DM* to finish.

*Effect of the Number of Shared Resources.* Figs. 6 and 7 show the effect of the number of shared resources. In Fig. 6, the task sets shared 4 resources, but the total number of critical sections over all resources was kept the same as in Fig. 5. Similarly, in Fig. 7 we varied the number of shared resources and kept the number of critical sections per resource at 128. Compared to Fig. 5 we can see that having critical sections spread across multiple shared resources versus having them all on

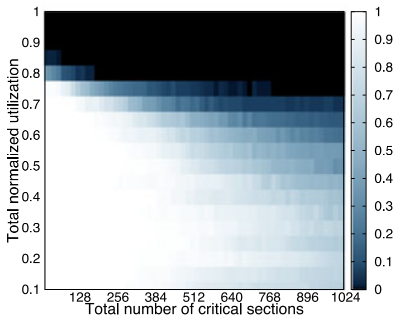


Fig. 10. Schedulability for $m = 36$, $n = 9$, 1 shared resource with a priority-ordered lock, short requests, varying total utilization and number of requests.

one resource doesn't affect the schedulability much. (Note that the corresponding points on these figures represent the *same* number of critical sections overall). This seems counter-intuitive since individual requests should experience smaller contention if the requests are spread out across resources. However, the sum of the worst case blocking times over multiple resources turns out to be similar to that of a single resource with the same total number of requests.

*Effect of $m$.* Comparing Figs. 5 and 9 reveals that the schedulability for 36 cores is higher when the number of requests is small but decreases more quickly when the number of requests increases, compared to 12 cores. In our analysis and experiment design, there are competing factors, some of which favor fewer cores and some more. For the former, $B_i^L$ and $B_i^C$ increase as $m$ increases (since they depend on $n_i$'s and the sum of $n_i$'s is bounded by $m$). For the latter, since the total utilization ($0.75m$) is 9 on 12 cores and is 27 on 36 cores, there is more "absolute slack" (9 versus 3) for the 36 core experiments. It appears that the latter factor may exceed the former when the number of critical sections is small; however, as the number of critical sections increases, the increase in contention decreases schedulability despite the extra slack.

*Effect of Utilization and the Number of Critical Sections.* Fig. 10 shows the schedulability results for $m = 36$ with $n = 9$ tasks and a single shared resource protected by a priority-ordered lock (the results for FIFO-ordered locks are similar). We varied the normalized total utilization in range $[0.1, 1.0]$ with the gap of 0.05 and the number of critical sections in range $[16, 1024]$ with the gap of 16. For each pair of the values for the two parameters, we recorded the percentage of the task sets that are schedulable. The lighter the data point, the greater the percentage of the task sets that are schedulable. Both parameters have a noticeable impact on schedulability. Also, for very high utilization (above $0.8m$), the task sets are unschedulable even for a small number of critical sections.

## 8 Empirical Evaluation

We incorporated an implementation of both FIFO-ordered and priority-ordered spin locks into a federated scheduling system for OpenMP programs [7]. The locks are implemented in user-space and provide `lock` and `unlock` interfaces to real-time tasks. Typically, spin locks require support for atomic operations; we specifically utilized atomic built-in extensions of GCC [27] to implement our spin locks.

*FIFO-Ordered Spin Locks.* For FIFO-ordered locks, we implemented an MCS lock, a scalable list-based FIFO spin lock [28]. In the MCS lock approach, each processor maintains a data structure called an MCS node: a structure containing a pointer `next` to the next MCS node in the waiting list, and a flag `spin` indicating whether the owner of the MCS node gets the lock. In this approach, requests to a lock form a linked list of MCS nodes, each from a processor. An MCS lock is a pointer to the MCS node at the tail of the list, or null if the list is empty. When a processor sends a request, it atomically obtains the current value of the lock, and sets the lock to point to the processor's MCS node. If the lock is null, the requesting processor is the first in the list and has successfully acquired the lock. Otherwise, it will append itself to the list by setting the `next` pointer of the preceding MCS node (i.e., the previous tail of the list) to point to its MCS node, and spin on its MCS node's `spin` flag. When a processor finishes its critical section, it passes ownership of the lock to the next processor in the list by flipping the succeeding MCS node's `spin` flag to allow it to break out of spinning. If there is no succeeding processor, it simply resets the lock to null and returns.

*Priority-Ordered Spin Locks.* Pseudocode for priority-ordered locks is shown in Algorithm 2. We extend the MCS lock approach for priority-ordered locks where each processor also maintains an MCS node and uses the node whenever it sends a request to a resource. A priority-ordered lock is a structure consisting of `owner`, a pointer pointing to the MCS node of the processor that currently owns the lock, and `task_heads`, an array of MCS locks, each for a task. Note that here as well, an MCS lock is a pointer to the tail MCS node in the list of requests for that lock. Each task is associated with an MCS lock in the array at an index calculated from the locking-priority of requests from the task. For instance, a task with the highest locking-priority uses the MCS lock at index 0, a task with the second highest locking-priority uses the MCS lock at index 1, and so on. Elements in the array are also aligned to the cache line size to avoid false sharing. Each MCS lock in the array is used to accommodate requests issued by processors of the corresponding task. When a processor of a task issues a request, it first calls the MCS's `lock` method to acquire the task's MCS lock (line 3). After the processor successfully acquires the task's MCS lock, it constantly checks for the existence of any higher locking-priority requests (line 5). It does this by checking the values of the MCS locks of the tasks with higher locking-priorities (recall that each MCS lock is a pointer to the tail of a list of MCS nodes from processors of the same task). If all MCS locks for higher locking-priority tasks are null, which means there are no requests from those tasks, the processor will try to acquire the priority-ordered lock by atomically comparing the lock's `owner` with null and if so setting `owner` to point to its MCS node (line 7). It returns if the compare-and-exchange instruction succeeded (line 9). Otherwise, the lock must have been acquired by some higher locking-priority processor. In this case, the processor spins again until there is no higher locking-priority request (line 4). When the processor finishes its critical section, it calls the MCS lock's `unlock` method to release its task's MCS lock and resets the priority-ordered lock's `owner` to null (lines 16, 17).

*Spin Locks Overhead.* Table 1 summarizes the average overhead per critical section of FIFO-ordered and priority-

TABLE 1
Overhead of Spin Locks

| Number of cores | FIFO-ordered ($ns$) | Priority-ordered ($ns$) |
|---|---|---|
| 1 | 32 | 58 |
| 2 | 461 | 143 |
| 4 | 500 | 287 |
| 8 | 550 | 526 |
| 12 | 540 | 765 |
| 16 | 609 | 891 |
| 24 | 625 | 1,575 |
| 32 | 676 | 2,185 |

ordered locks, recorded on the same machine that ran our empirical experiments (see below for description). We created $m$ threads ($m$ varies in the first column) and pinned one thread on each core. The threads were synchronized using a `pthread` barrier. Each thread repeatedly acquires and releases the lock and performs work inside each critical section. We measured the time between the earliest lock acquisition and the latest lock release among all threads. We also measured the time to perform the same total amount of work (by all threads) on a single thread. The difference between these two measurements was divided by the total number of acquire-release pairs to get the average overhead per critical section. We repeated the measurement 100 times and recorded the largest average overhead in Table 1. We notice that FIFO-ordered locks implemented by the MCS algorithm show good scalability when the number of cores increases, as expected. Whereas priority-ordered locks does not scale as well. This is due to the way each processor, after acquiring its task's MCS lock, checks for the existence of any higher locking-priority requests (Algorithm 2, line 5). Recall that it does this by constantly reading the MCS locks of all tasks with higher locking-priorities, and only executes the compare-and-exchange instruction when all these MCS locks are null. Thus, when any MCS lock of a higher locking-priority task changes, either by an arrival of a new request or by a departure of the last request, processors (who hold their tasks' MCS locks) of the lower locking-priority tasks will have to read the new value into their caches. This creates high contention for the shared bus, thus priority-ordered locks do not scale as well as MCS locks.

*Empirical Results.* We conducted experiments on our federated scheduling platform to observe the performance of the two types of spin locks in practice. Using the same task sets generated in Section 7, we constructed synthetic parallel task sets written in OpenMP. Critical sections were distributed randomly inside the structure of the tasks. We ran the experiments on a 48-core machine composed of four AMD Opteron 6168 processors, each with 12 cores. Linux 3.4.4 with RT-PREEMPT patch version 3.4.4-rt14 was used as the underlying RTOS. Since each task was allocated a set of dedicated cores, we used Linux's CPU mask and `sched_setaffinity` system call to bind each task to its cores. For each parameter configuration, we tested 100 task sets; each task set was run for 100 hyper-periods. After each experiment finished, we recorded the ratio of task sets in which no job of any task missed its deadline. For all of the settings, we observed a similar trend as in the schedulability analysis (Section 7)—priority-ordered locks have better results than

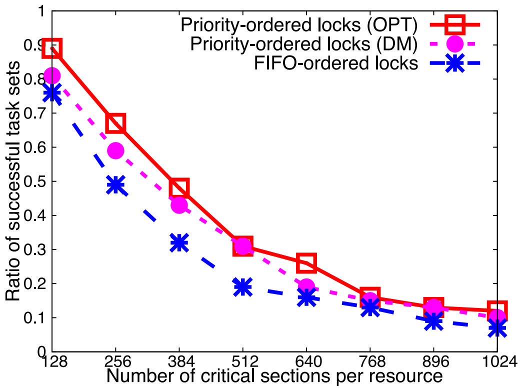


Fig. 11. Ratio of successful task sets for $m = 36$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and moderate critical sections.

FIFO-ordered locks. Fig. 11 shows a representative result for $m = 36$ cores, total utilization $U = 0.75m$, a single shared resources, and all requests have moderate lengths.

**Algorithm 2.** Priority-Ordered Spin Locks

```
1:  procedure LOCK(PriorityLock: lock, MCSNode: mynode)
2:      Get the pointer tail to my task's MCS lock
3:      lock_mcs(tail, mynode)
4:      while (1) do
5:          Check if any higher locking-priority requests exist
6:          if (No higher locking-priority requests exist) then
7:              cmpxchg(lock→owner, null, mynode)
8:              if (cmpxchg was successful) then
9:                  Return
10:             end if
11:         end if
12:     end while
13: end procedure
14: procedure UNLOCK(PriorityLock: lock, MCSNode:
    mynode)
15:     Get the pointer tail to my task's MCS lock
16:     unlock_mcs(tail, mynode)
17:     lock→ owner = null
18: end procedure
```

## 9 RELATED WORK

*Real-Time Scheduling for Parallel Tasks.* Lakshmanan et al. [2] provided a scheduling algorithm for strict parallel synchronous tasks. Later, Saifullah et al. [1] studied more general parallel synchronous tasks and scheduling for generalized directed acyclic graph (DAG) tasks [29]. Schedulers like G-EDF have been extensively studied [3], [5], [30], [31], [32]. Ferry et al. [33] proposed a real-time scheduling service for parallel synchronous tasks. Federated scheduling [7], [21], [22], [23], the underlying scheduling approach for this paper, is a generalization of partitioned scheduling algorithms that works with a general DAG task model.

*Real-Time Locking Protocols.* Real-time locking protocols for sequential tasks have been studied extensively for both uniprocessor and multiprocessors systems. For uniprocessor systems, the *Priority Ceiling Protocol* (PCP) [12], and the *Stack Resource Policy* [13] provide optimal priority-inversion blocking. Rajkumar et al. [34] extended PCP to work on distributed multiprocessors. The *Multiprocessor Priority Ceiling Protocol* (MPCP) [14] extended PCP to support shared memory multiprocessors. Gai et al. [16] proposed the *Multiprocessor Stack Resource Policy*, an extension of SRP. MSRP applies SRP for local resources and non-preemptable spin locks for global resources.

Block et al. [17] proposed the *Flexible Multiprocessor Locking Protocol* (FMLP) that works for both partitioned and global scheduling. Nested accesses to shared resources is also supported by using group locks at a cost of reducing concurrency. Brandenburg et al. [11] performed an empirical study of several options for synchronization: lock-free, wait-free, spin locks, and semaphores, on LITMUS$^{RT}$ [35]. For suspension-based locking protocols, Brandenburg et al. [19] clarified the definition of blocking for multiprocessor systems. They distinguished two blocking analysis techniques: suspension-oblivious and suspension-aware, and constructed asymptotically optimal blocking results a protocol can guarantee using either analysis method. Later, Ward et al. [36] introduced the *Real-time Nested Locking Protocol* (RNLP), which supports fine-grained nested resource accesses, and conducted suspension-oblivious and suspension-aware analysis for this protocol. For spin-based locks, Wieder et al. [37] analyzed various types of spin locks and proposed a new analysis technique for worst case spin delays for P-FP scheduling that improves on classic analysis techniques. Carminati et al. [38] explored the design space for multiprocessor synchronization protocols with partitioned, static priority scheduling. They considered a set of characteristics for protocols, including queuing order (e.g., FIFO versus priority order), preemptability of critical sections, type of blocking (e.g., suspension versus spin), and proposed several variations of MPCP. For GPU platforms, Elliott et al. [39] modified an OpenVX implementation to support real-time computer vision workloads, such as pedestrian detection. Their platform pipelines the graphs of computer vision applications and guarantees the end-to-end response times of their graphs are bounded. Most recently, Biondi et al. [40] proposed a novel fine-grained, non-asymptotic analysis for nested FIFO spin locks under partitioned fixed-priority scheduling, based on a graph abstraction that encodes all possible conflicts for shared resources among tasks.

## 10 Conclusions

We have presented the first blocking analysis and schedulability test for parallel real-time tasks that use spin locks to arbitrate access to shared resources. We analyze blocking times under federated scheduling, which assigns dedicated cores to each task and thus avoids CPU priority inversion. We also incorporated spin lock implementations on a federated scheduling platform, which allows us to run parallel real-time programs written in OpenMP with spin locks. Our numerical and empirical experiments using randomly generated task sets both indicate that priority-ordered locks outperform FIFO-ordered locks in term of schedulability. This work opens up new research problems for schedulability tests for parallel task sets which access shared resources: for instance, how to construct a schedulability test when using schedulers, such as G-EDF, where tasks can share cores and how to design good protocols for systems that have both high and low-utilization tasks.

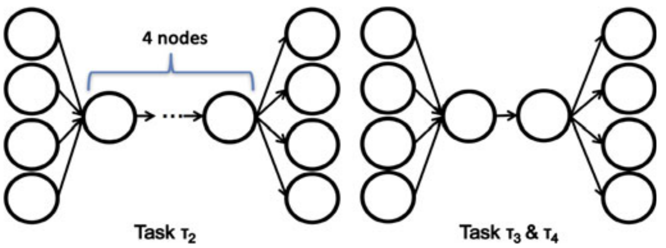


Fig. 12. Example task set with priority-ordered spin locks. Each node has work of 1 time unit.

## Appendix A Example for Priority-Ordered Spin Locks

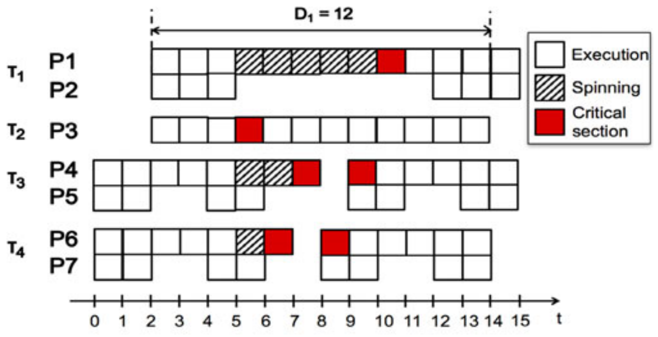
Consider a task set with four implicit deadline tasks: $\tau_1(C_1 = 14, L_1 = 4, D_1 = 12)$, $\tau_2(C_2 = 12, L_2 = 6, D_2 = 12)$, $\tau_3(C_3 = 10, L_3 = 4, D_3 = 8)$, and $\tau_4(C_4 = 10, L_4 = 4, D_4 = 8)$. Task $\tau_1$ has the same structure as task $\tau_1$ in Fig. 2. The DAGs for the other tasks are shown in Fig. 12 (Appendix A) where tasks $\tau_3$ and $\tau_4$ have the same DAG. All tasks access a single shared resource protected by a priority-ordered lock. Each task has one request to the shared resource; all requests have lengths of 1 time unit. Without blocking, the numbers of processors allocated to these tasks are $n_1 = 2, n_2 = 1, n_3 = 2, n_4 = 2$ respectively (using Lemma 1). Assume the tasks have the following order of locking-priorities, $\tau_2 < \tau_1 < \tau_3 < \tau_4$, with $\tau_4$ having the highest locking-priority.

We now calculate the blocking bounds for $\tau_1$ caused by contention with the other tasks. First the delay-per-request is: $dpr(\tau_1, l) = lower(\tau_1, l) + equal(\tau_1, l) + higher(\tau_1, l)$; where $lower(\tau_1, l) = 1$, $equal(\tau_1, l) = 0$, and $higher(\tau_1, l) = njobs(\tau_3, dpr(\tau_1, l)) + njobs(\tau_4, dpr(\tau_1, l))$. Thus, $dpr(\tau_1, l) = 1 + 2\cdot \left\lceil \frac{dpr(\tau_1,l)+8}{8} \right\rceil$. The value of $dpr(\tau_1, l)$ converges at 5 time units.

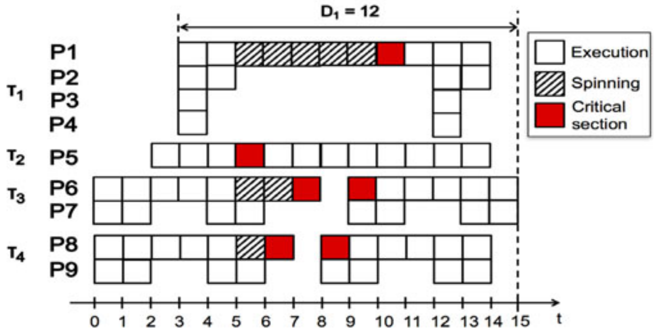
Applying Theorems 13 and 14, we can calculate the bounds for the work blocking and critical-path blocking for $\tau_1$: $B_1^C \le 5$ and $B_1^L \le 5$. Note that since $\tau_1$ has only one request to the shared resource, the work blocking bound is equal to the critical-path blocking bound. Also, there is no intra-task blocking within task $\tau_1$. Fig. 13a (Appendix A) shows a schedule that causes the worst case blocking for task $\tau_1$. It happens when the request from a job $J_1$ of $\tau_1$ is delayed by one from a job of $\tau_2$ (lower locking-priority) and requests from two consecutive jobs of task $\tau_3$ and two consecutive jobs of task $\tau_4$ (higher locking-priority) in a back-to-back manner. The schedule shows that the blocking bounds for priority-ordered locks are tight. Note that $\tau_1$ misses deadlines with the current number of processors allocated to it. The new number of processors of $\tau_1$ is computed using Equation (1): $n_1' = \left\lceil \frac{14+5-4-5}{12-4-5} \right\rceil = 4$. This guarantees that $\tau_1$ will meet all deadlines for the current blocking bounds, as depicted in Fig. 13b (Appendix A).

## Appendix B Generalized Bounds for Priority-Ordered Locks

The work blocking $B_i^C$ and critical-path blocking $B_i^L$ can be generalized easily for critical sections of different lengths. Note that task $\tau_i$ can be blocked by at most $R_{i,q}$ lower locking-priority requests (Lemma 7). Therefore, the blocking



(a) $\tau_1$ misses deadline with the worst case blocking on two cores



(b) $\tau_1$ meets deadline with the worst case blocking on four cores

Fig. 13. Example schedules with the current and updated number of processors for $\tau_1$ with priority-ordered spin locks.

caused by lower locking-priority tasks to $\tau_i$ with respect to resource $l_q$ is bounded by summing the lengths of $R_{i,q}$ longest lower locking-priority requests. We use $lcs(R_{i,q}, l_q)$ to denote this quantity. Thus, $B_i^C$ and $B_i^L$ are now bounded as follows:

$$B_i^C \leq \sum_{l_q \in Q_i} \Bigg( \bigg( \frac{\min(R_{i,q}, n_i) \cdot (\min(R_{i,q}, n_i) - 1)}{2} + (n_i - 1) \cdot \max(R_{i,q} - n_i, 0) \bigg) \cdot \Phi_{i,q} + lcs(R_{i,q}, l_q) + \sum_{\tau_j \in \tau_i^{HP}} \min\{njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot R_{i,q}, njobs(\tau_j, D_i) \cdot R_{j,q} \cdot n_i\} \cdot \Phi_{j,q} \Bigg)$$

$$B_i^L \leq \sum_{l_q \in Q_i} \max_{Y \in [1, R_{i,q}]} \Bigg( \min\{(n_i - 1) \cdot Y, R_{i,q} - Y\} \cdot \Phi_{i,q} + lcs(Y, l_q) + \sum_{\tau_j \in \tau_i^{HP}} \min\{njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot Y, njobs(\tau_j, D_i) \cdot R_{j,q}\} \cdot \Phi_{j,q} \Bigg).$$

## Acknowledgements

This research was supported by NSF grants CCF-1337218, CNS-1329861, and ONR grant N000141612108

## References

[1] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Syst.*, vol. 49, no. 4, pp. 404–435, 2013.
[2] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proc. IEEE Real-Time Syst. Symp.*, 2010, pp. 259–268.
[3] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic DAG task model," in *Proc. Euromicro Conf. Real-Time Syst.*, 2013, pp. 225–233.
[4] B. Andersson and D. de Niz, "Analyzing global-EDF for multiprocessor scheduling of parallel tasks," in *Proc. Int. Conf. Principles Distrib. Syst.*, 2012, pp. 16–30.
[5] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global EDF for parallel tasks," in *Proc. Euromicro Conf. Real-Time Syst.*, 2013, pp. 3–13.
[6] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic DAG task systems," in *Proc. Euromicro Conf. Real-Time Syst.*, 2014, pp. 97–105.
[7] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *Proc. Euromicro Conf. Real-Time Syst.*, 2014, pp. 85–96.
[8] J. Kim, H. Kim, K. Lakshmanan, and R. R. Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," in *Proc. Int. Conf. Cyber-Phys. Syst.*, 2013, pp. 31–40.
[9] D. Ferry, et al., "Real-time system support for hybrid structural simulation," in *Proc. 14th Int. Conf. Embedded Softw.*, 2014, pp. 1–10.
[10] B. Brandenburg and J. Anderson, "Feather-trace: A lightweight event tracing toolkit," in *Proc. Int. Workshop Operating Syst. Platforms Embedded Real-Time Appl.*, 2007, pp. 19–28.
[11] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2008, pp. 342–353.
[12] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.
[13] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, 1991.
[14] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 1990, pp. 116–123.
[15] C.-M. Chen and S. K. Tripathi, "Multiprocessor priority ceiling based protocols," *Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-94-42*, University of Maryland at College Park, College Park, MD, USA, 1994.
[16] P. Gai, G. Lipari, and M. D. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proc. Real-Time Syst. Symp.*, 2001, pp. 73–83.
[17] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2007, pp. 47–56.
[18] B. B. Brandenburg and J. H. Anderson, "Reader-writer synchronization for shared-memory multiprocessor real-time systems," in *Proc. Euromicro Conf. Real-Time Syst.*, 2009, pp. 184–193.
[19] B. B. Brandenburg and J. H. Anderson, "Optimality results for multiprocessor real-time locking," in *Proc. Real-Time Syst. Symp.*, 2010, pp. 49–60.
[20] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *Proc. Real-Time Syst. Symp.*, 2009, pp. 377–386.
[21] J. Li, K. Agrawal, C. Gill, and C. Lu, "Federated scheduling for stochastic parallel real-time tasks," in *Proc. IEEE 20th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2014, pp. 1–10.
[22] S. Baruah, "Federated scheduling of sporadic DAG task systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 179–186.
[23] S. Baruah, "The federated scheduling of constrained-deadline sporadic DAG task systems," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2015, pp. 1323–1328.
[24] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," PhD thesis, Dept. Comput. Sci., Univ. North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2011.
[25] R. Stafford, "Random vectors with fixed sum," 2006. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum
[26] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proc. 1st Int. Workshop Anal. Tools Methodologies Embedded Real-Time Syst.*, 2010, pp. 6–11.

[27] GCC atomic built-in functions. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Atomic-Builtins.html. Accessed on: Feb. 21, 2016.

[28] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[29] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. Gill, "Parallel real-time scheduling of DAGs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3242–3252, Dec. 2014.

[30] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *Proc. Real-Time Syst. Symp.*, 2012, pp. 63–72.

[31] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms," in *Proc. Euromicro Conf. Real-Time Syst.*, 2013, pp. 25–34.

[32] C. Liu and J. Anderson, "Supporting soft real-time parallel applications on multicore processors," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2012, pp. 114–123.

[33] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, "A real-time scheduling service for parallel tasks," in *Proc. IEEE 19th Real-Time Embedded Technol. Appl. Symp.*, 2013, pp. 261–272.

[34] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proc. Real-Time Syst. Symp.*, 1988, pp. 259–269.

[35] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS$^{RT}$: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proc. Real-Time Syst. Symp.*, 2006, pp. 111–126.

[36] B. C. Ward and J. H. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *Proc. 24th Euromicro Conf. Real-Time Syst.*, 2012, pp. 223–232.

[37] A. Wieder and B. B. Brandenburg, "On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks," in *Proc. Real-Time Syst. Symp.*, 2013, pp. 45–56.

[38] A. Carminati, R. S. De Oliveira, and L. F. Friedrich, "Exploring the design space of multiprocessor synchronization protocols for real-time systems," *J. Syst. Archit.*, vol. 60, no. 3, pp. 258–270, 2014.

[39] G. A. Elliott, K. Yang, and J. H. Anderson, "Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms," in *Proc. IEEE Real-Time Syst. Symp.*, 2015, pp. 273–284.

[40] A. Biondi, B. B. Brandenburg, and A. Wieder, "A blocking bound for nested FIFO spin locks," in *Proc. IEEE 37th Real-Time Syst. Symp.*, 2016, pp. 291–302.

**Son Dinh** is a Ph.D. student in the department of Computer Science and Engineering at Washington University in St. Louis. His research interests include real-time scheduling and synchronization protocols for parallel task systems, and cyber-physical systems.

**Jing Li** is an assistant professor in the Department of Computer Science at New Jersey Institute of Technology. She received her Ph.D. degree from Washington University in St. Louis in 2017 on the topic of Parallel Real-Time Scheduling for Latency-Critical Applications. Her research interests include real-time systems, parallel computing, and cyber-physical systems.

**Kunal Agrawal** is an Associate Professor of Computer Science and Engineering at Washington University in St. Louis. Her research interests include parallel algorithms and data structures; scheduling algorithms, runtime systems and tools for parallel programs; real-time scheduling; concurrency platforms for parallel and real-time systems. She has published regularly at top-tier conferences including SODA, PPoPP, SPAA, PLDI, IPDPS, RTSS, and RTAS. She has served in numerous program committees and served as the program committee chair at SPAA 2015.

**Chris Gill** is a Professor of Computer Science and Engineering at Washington University in St. Louis, MO, USA. His research interests include real-time, embedded, and cyber-physical systems. Results of his research have appeared regularly in top-tier conferences and journals, with more than 80 peer-reviewed papers and articles in press over the past two decades. He has an extensive record of service in review panels, standards bodies, workshops, and conferences for distributed, real-time, embedded, and cyber-physical systems, including serving as both Program Chair and General Chair for the ICCPS, RTAS and RTSS conferences. A Senior Member of both the ACM and the IEEE, he is currently a member of the Executive Committee for the IEEE Technical Committee on Real-Time Systems, and has served previously as Treasurer of IEEE TCRTS and as Vice Chair of ACM SIGBED.

**Chenyang Lu** is the Fullgraf Endowed Chair Professor in the Department of Computer Science and Engineering at Washington University in St. Louis. His research interests include Internet of Things, real-time systems, and cyber-physical systems. He is Editor-in-Chief of ACM Transactions on Sensor Networks, Area Editor of IEEE Internet of Things Journal and Associate Editor of ACM Transactions on Cyber-Physical Systems and Real-Time Systems Journal. He chaired premier conferences such as IEEE Real-Time Systems Symposium (RTSS), ACM Conference on Embedded Networked Sensor Systems (SenSys) and ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS). He is the author and co-author of over 150 research papers with over 17,000 citations and an h-index of 57. He received the Ph.D. degree from University of Virginia in 2001, the M.S. degree from Chinese Academy of Sciences in 1997, and the B.S. degree from University of Science and Technology of China in 1995. He is a Fellow of IEEE.