Resource-Oriented Partitioning for Multiprocessor Systems with Shared Resources

Maolin Yang, *Member, IEEE*, Wen-Hung Huang, *Member, IEEE*, and Jian-Jia Chen, *Member, IEEE*

Abstract—Predictable scheduling and resource sharing primitives are fundamental aspects of real-time systems. To prevent race conditions, access to shared resources must ensure *mutual exclusion*, e.g., using semaphores. Further, real-time locking protocols are required to avoid un-controlled priority inversions. For uniprocessor systems, the Priority Ceiling Protocol (PCP) has been widely accepted and supported in real-time operating systems. However, it remains arguable as to whether there exists a preferable approach for resource sharing in multiprocessor systems. In this paper, we show that the proposed Resource-Oriented Partitioned (ROP) scheduling with a distributed resource sharing policy, originating from the concept of the Distributed Priority Ceiling Protocol (DPCP), can achieve a non-trivial speedup factor guarantee. Specifically, we prove that the proposed R-PCP-rm-rm algorithm achieves a speedup factor of $11 - 6/(m+1)$ on a platform consisting of m processors, where each job of a task may request at most one shared resource at most one time. Our empirical evaluations show that the proposed algorithm is highly effective in terms of task sets deemed schedulable.

Index Terms—Multiprocessor real-time systems, partitioned scheduling, fixed-priority scheduling, locking protocols, shared resources, worst-case response time, speedup factor

1 Introduction

Real-time systems are designed for applications in which the response time is critical. To guarantee real-time performance while making the most effective use of the available computing resources, co-optimized scheduling and resource sharing policies are required. However, the question of how to co-optimize scheduling and resource sharing in real-time multiprocessor systems has not been completely answered.

To schedule real-time tasks on multiprocessor platforms, there have been four widely studied paradigms: partitioned, global, clustered, and semi-partitioned scheduling. The partitioned scheduling approach assigns the tasks among the available processors, where each task is allowed to execute only on the processor for which it is assigned. The global scheduling approach allows a job to be migrated from one processor to another. The clustered scheduling approach assigns the tasks onto clusters of processors, and the tasks on each cluster are scheduled by global scheduling. The semi-partitioned scheduling approach decides whether a task is divided into subtasks, and each task/subtask is then assigned to a processor. A comprehensive survey of real-time scheduling for multiprocessor systems can be found in [23].

To prevent unpredictable *priority inversions* when tasks require mutually exclusive access to shared resources, real-time locking protocols have been widely studied in recent years. The Priority Ceiling Protocol (PCP) [40] and the Stack Resource Policy (SRP) [7] have been shown to perform reasonably well in uniprocessor systems. However, challenges among real-time system designs on multiprocessor platforms include resource sharing between the real-time tasks as well as task-to-processor mapping. If task synchronization is unnecessary, partitioned and semi-partitioned scheduling, with their relatively low run-time overheads, are usually preferable [9], [17]. However, when synchronization is required, fundamental concerns arise: (*i*) *is partitioned scheduling still a good option? If so,* (ii) *how does one derive good partitions?*

Regarding question (i), real-time locking protocols, such as the Multiprocessor Priority Ceiling Protocol (MPCP) [38], the Multiprocessor Resource Stack Policy (MSRP) [25], and the Flexible Multiprocessor Locking Protocol (FMLP) [11], have been proposed to handle resource sharing for partitioned scheduling. However, it has been shown in [16] that the number of priority-inversion blockings (*pi*-blockings) is lower bounded by the number of processors of the multiprocessor system in the worst case. Specifically, the elegance of partitioned scheduling for running a task all the time on one processor suffers from issues related to the synchronization between the tasks if the tasks are not partitioned well.

To address question (ii), several task partitioning heuristics, such as in [29], [33], [37], [43], have been explored. However, there has been no algorithmic analysis provided in the literature to necessitate these approaches from the perspective of resource augmentation. Resource augmentation is also

- *M. Yang is with the School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu 610015, China. E-mail: maolyang@126.com.*
- *W.-H. Huang is with the Corporate ePF Division, DENSO Corporation, Kariya, Aichi 448-8661, Japan. E-mail: wenhung_huang@denso.co.jp.*
- *J.-J. Chen is with the Department of Computer Science, Technical University of Dortmund, Dortmund 44227, Germany. E-mail: jian-jia.chen@cs.uni-dortmund.de.*

Manuscript received 15 Oct. 2017; revised 13 Nov. 2018; accepted 18 Dec. 2018. Date of publication 26 Dec. 2018; date of current version 16 May 2019.
(Corresponding author: Maolin Yang.)
Recommended for acceptance by M. Caccamo.
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/TC.2018.2889985

referred to as speedup factor [32], which is a well-studied theoretical metric for measuring the sub-optimality of a scheduling algorithm, as we will discuss in Section 3.4. Moreover, there is no clear evidence on whether the above synchronization strategies and task partitioning approaches should be strictly designed to follow the traditional partitioned scheduling paradigm.

In the past decades, the literature on real-time scheduling has been very biased toward scheduling computational tasks. When resource sharing becomes the bottleneck, it is sensible to change perspectives to explore collaborative optimization methods for resource sharing. To this end, a new scheduling policy, *Resource-Oriented Partitioned (ROP)* scheduling, is proposed. This scheduling policy follows the following principles:

- Each shared resource is bound to a processor, and all the critical sections guarded by this shared resource are executed only on the processor.
- The non-critical sections of a task are executed on the processor where the task is assigned (which can be different from the processors executing the critical sections of the task).
- Once a task requests a shared resource, the effective priority of the corresponding critical sections is elevated to be higher than any non-critical section of any task on the same processor.

The spirit of ROP scheduling is to restrict resource contentions on designated processors and serve resource requests in priority. In addition, it follows the principle of partitioned scheduling and thus has good potential for maintaining the low-overhead of partitioned scheduling. It is noted that, this distributed synchronization framework was originally proposed by Rajkumar et al. in the Distributed Priority Ceiling Protocols (DPCP) [39]. However, there has been no further elaboration in the literature toward providing evidence on how to assign tasks and resources among multiple processors. ROP scheduling provides reasonable task and resource partitioning algorithms.

The fundamental validation of ROP scheduling is that the ROP heuristic with the priority ceiling mechanism guarantees a *speedup factor* of $11 - 6/(m+1)$, irrespective of the number of shared resources, if each job of a task requests at most one resource for once, where $m \geq 2$ is the number of processors. The understanding of this constant speedup factor with a simple scheduling algorithm implies the potential of ROP scheduling. The effectiveness of ROP scheduling is further supported by the empirical results, even when each job has more than one critical section.

The original version of our ROP scheduling was presented at the 2016 IEEE Real-Time Systems Symposium (RTSS 2016) [30]. Significant extensions are made in the current version, including

- Extensive analysis of the resource sharing methods. The priority ceiling mechanism [40] and the non-preemptive scheduling for shared resources are analyzed and compared under ROP scheduling.
- Extensive blocking time analysis. The original version focused on the cases in which each job of a task accesses at most one shared resource at most one time. In this paper, improved analysis for multiple accesses to shared resources is presented.
- Further discussions and comparisons on task ordering and priority assignment under ROP scheduling are given.
- More comprehensive comparisons and observations based on large-scale schedulability experiments are performed. Both tasks with at most one request per job and tasks with multiple requests per job are considered in the experiments.

The remainder of the paper is organized as follows: Related work is discussed in Section 2. The system model and definitions are introduced in Section 3. The scheduling framework and the resource sharing methods are discussed in Section 4. The proposed ROP scheduling algorithm is presented in Section 5. Schedulability analysis is derived in Section 6. The speedup factor of the proposed algorithm is derived in Section 3.4. Further discussions on task ordering and priority assignment are given in Section 8. Empirical results are analyzed in Section 9. Conclusions are drawn in Section 10.

2 Related Work

Several mutual exclusion locking protocols have been proposed to handle the synchronization problem in multiprocessor real-time systems. Recent analysis and comparisons of real-time locking protocols can be found in [13], [44] for partitioned scheduling and in [46] for global scheduling. In addition, several protocols for hybrid scheduling approaches, such as clustered [15] and semi-partitioned [1] scheduling, reservation-based scheduling [24], and open real-time systems [36], have been proposed in recent years. To support nested critical sections, Ward and Anderson [41], [42] have introduced the Real-time Nested Locking Protocol (RNLP), which employs a token lock and a request satisfaction mechanism to support fine-grained nested locking.

Multiprocessor real-time locking protocols can be classified into suspension-based (or semaphore) protocols [15], [16], [38], [39] and spin-based protocols [18], [25], [44]. Intuitively, tasks self-suspend when blocked on shared resources in semaphore protocols, while they perform a busy wait in spin-based protocols. In general, busy waiting requires fewer context switches; thus, it is more efficient, especially when critical sections are short [12], [26]. However, the resulting loss of processor service (due to spin) must be accounted for. In contrast, self-suspension allows waiting tasks to relinquish processors to other tasks and thus is preferable especially when critical sections are long [12], [33]. However, self-suspension results in jitter effects, which leads to additional schedulability losses that must be carefully quantified [20], [45]. Moreover, in some distributed-configured scheduling systems, such as designated [39] and dedicated [29] synchronization frameworks, jobs self-suspend on host processors; waiting for resource services on remote processors is thus a natural fit for the scheduling strategy. However, in this work, the suspension-based methodology is used for resource sharing.

Concerning partitioning, Lakshmanan et al. [33] presented a synchronization-aware partitioned heuristic [38], which organizes tasks sharing common resources into groups and attempts to assign each group of tasks to the same processor. Following the same principle, Nemati et al. [37] presented a blocking-aware partitioning method that uses an advanced cost heuristic to split the task group when an entire group fails to be assigned on one processor. In subsequent work, Hsiu et al. [29] proposed a dedicated-core framework to separate the execution of critical sections and normal sections, and they employed a priority-based mechanism for resource sharing such that each request can be blocked by at most one lower priority request. More recently, Wieder and Brandenburg [43] proposed a greedy slacker partitioning heuristic with the MSRP. Han et al. [27], [28] addressed the problem of utilization bounds for P-EDF with the MSRP and proposed the synchronization-cognizant task mapping algorithms [28]. However, no algorithmic evidence was provided to necessitate these approaches from the perspective of resource augmentation.

From an algorithmic optimality point of view, Brandenburg and Anderson [16] were the first to study the multiprocessor real-time locking problem. It has been shown that $\Omega(m)$ pi-blocking is unavoidable under suspension-oblivious schedulability analysis, and $\Omega(n)$ pi-blocking is unavoidable under suspension-aware schedulability analysis [16]. To this end, some locking protocols that adopt FIFO-waiting queues, such as the Generalized FIFO Multiprocessor Locking Protocol (FMLP$^+$) [15] and the Distributed FIFO Locking Protocol (DFLP) [14], have been proved to be asymptotically optimal. Notably, the empirical results in [46] showed that asymptotically optimal protocols do not necessarily perform well in terms of schedulability. This work shows that the proposed ROP scheduling with simple locking rules improves the schedulability significantly.

Andersson and Easwaran [4] presented a Global Earliest Deadline First (G-EDF) scheduling based on virtualization scheduling, which guarantees a speedup factor of $12(1+3r/(4m))$ on a platform consisting of m identical processors, where each job of a task issues at most one request to one of the r shared resources. Later, Andersson and Raravi [5] proposed another virtualization-based scheduling with guaranteed speedup factor for heterogeneous systems. In contrast, the algorithm presented in this work achieves a speedup factor of $11-\frac{6}{m+1}$ without using any virtualization.

3 System Model and Definitions

A set $\tau=\{\tau_1,\tau_2,\ldots,\tau_n\}$ of n sporadic real-time tasks are considered to execute upon a multiprocessor platform consisting of $m\geq 2$ identical processors $\wp=\{\wp_1,\wp_2,\ldots,\wp_m\}$ with n_r shared resources $\mathcal{RS}=\{\ell_1,\ell_2,\ldots,\ell_{n_r}\}$.

3.1 Task Model

Without loss of generality, we use τ_k as the task of interest. Each sporadic task is characterized by a 4-tuple $\tau_k=(C_k,A_k,T_k,D_k)$. C_k is the worst-case execution time on *non-critical sections*, A_k is worst-case execution time on *critical sections*, T_k is the minimum inter-arrival time, and D_k is the relative deadline. In this paper, *implicit-deadline* tasks are considered, i.e., $D_k=T_k$ holds for each $\tau_k\in\tau$.

Each task generates a potentially infinite sequence of jobs, and two successive jobs of a task are released at least T_k time units apart. At any time, a job can be scheduled on only one processor. Let J_k be an arbitrary job of τ_k. The response time of J_k is given by the length between its arrival time and finishing time. The worst-case response time of τ_k, denoted by R_k, is an upper bound on the response time of any job of τ_k. For simplicity, we assume discrete time.

3.2 Shared Resources

A shared resource can be in-memory data, such as a set of variables, or external objects such as files, database connections, and network connections. To prevent *race conditions*, these shared resources must be accessed with mutual exclusion: conflicting concurrent requests must be serialized. In this paper, shared resources are logically represented by pieces of codes (or *critical sections*) to be executed on processors. Hence, no shared resource is considered processor specific. We further assume that resource requests are non-nested. Systems with nested critical sections remain as an open and challenging problem that cannot be handled by the strategies presented in this paper.

A job J_k could request resource ℓ_q on multiple occasions during its execution. The maximum number of such requests by J_k is denoted by $N_{k,q}$. The maximum (worst-case) resource usage time among all requests for resource ℓ_q by J_k is denoted by $L_{k,q}$. For each resource ℓ_q, an upper bound on the total resource usage time by a job J_k is denoted by $A_{k,q}$. Clearly, $A_{k,q}/N_{k,q}\leq L_{k,q}$. Further, for each task τ_k, the set of all resources accessed by jobs of τ_k is denoted by $\mathcal{RS}(\tau_k)\subseteq\mathcal{RS}$, and the total resource usage time is denoted as $A_k=\sum_{R_q\in\mathcal{RS}(\tau_k)}A_{k,q}$.

3.3 Utilization

The utilization of resource ℓ_q from task τ_k is defined as $U_k^{\ell_q}=A_{k,q}/T_k$. The total utilization of resource ℓ_q is denoted by $U^{\ell_q}=\sum_{\tau_k\in\tau}U_k^{\ell_q}$, and the cumulative total utilization of all shared resources is denoted by $U^{\mathcal{RS}}=\sum_{\ell_q\in\mathcal{RS}}U^{\ell_q}$. Similarly, the utilization of task τ_k with non-critical sections is defined as $U_k^C=C_k/T_k$, and the total utilization of non-critical sections for all tasks is denoted by $U^C=\sum_{\tau_k\in\tau}U_k^C$. Further, the overall utilization of task τ_k is defined as $U_k=(C_k+A_k)/T_k$. It is assumed that the utilization of the task set $U_{\sum}=\sum_{k=1}^{n}U_k\leq m$. Otherwise, the task set cannot be feasibly scheduled.

3.4 Speedup Factors

Ideally, an exact schedulability test associated with an optimal scheduling algorithm is preferred. However, it is often the case that an optimal scheduling is unavailable and/or the sufficient test associated with some scheduling algorithm is computationally intractable. The *speedup factor* is one metric that may be used to quantify the quality of sufficient schedulability tests. It can be defined as follows:

Definition 1. *A schedulability test has a speedup factor x, $x\geq 1$, if it is guaranteed that any task system that is feasible upon a specified platform is deemed to be schedulable by the test upon a platform in which each processor is at least x times as fast.*

Speedup factors are widely used to quantify the approximation of the scheduling algorithms or schedulability tests. Potential pitfalls of speedup factors can be found in [21].

3.5 Demand Bound Functions

The concept of a demand bound function has been widely used in real-time schedulability analysis. The demand bound function $dbf_k(t)$ bounds the maximum cumulative execution requirement by jobs of a sporadic task τ_k that both arrive in and have absolute deadlines within any interval of length t [8]. The demand bound function of task τ_k with an interval of length t is defined as follows.

$$dbf_k(t) = \max\left(0, \left(\left\lfloor \frac{t - D_k}{T_k} \right\rfloor + 1\right) \times (A_k + C_k)\right). \quad (1)$$

In particular, the demand bound functions of task τ_k for non-critical sections and for critical sections of resource ℓ_q, with an interval of length t, are

$$dbf_k^C(t) = \max\left(0, \left(\left\lfloor \frac{t - D_k}{T_k} \right\rfloor + 1\right) \times C_k\right), \quad (2)$$

and

$$dbf_k^{\ell_q}(t) = \max\left(0, \left(\left\lfloor \frac{t - D_k}{T_k} \right\rfloor + 1\right) \times A_{k,q}\right), \quad (3)$$

respectively.

4 Scheduling and Resource Sharing

The ROP scheduling framework follows the principle of Partitioned Fixed-Priority (P-FP) scheduling and uses a distributed synchronization framework based on the DPCP [39]. In addition to the priority ceiling mechanism used in the DPCP, we will also study a simplified variant using non-preemptive scheduling of critical sections. The empirical results in Section 9.2 show that this simplified variant is in some cases comparable with the priority ceiling mechanism in terms of the task sets deemed schedulable. The detailed scheduling and resource sharing mechanisms are discussed in the following sections.

4.1 Scheduling Framework

Tasks and resources are statically assigned among the available processors. Specifically, each task is allowed to execute its non-critical sections only on the processor to which it is assigned, and all requests from all tasks to a resource ℓ_q are allowed to execute only on the processor where ℓ_q is assigned.

A processor is called an *application processor* if it executes non-critical sections only and is called *synchronization processor* if it executes critical sections. Synchronization processors may also execute non-critical sections, depending on the task and resource allocations.

Priority Assignment. Each task τ_k is assigned a unique *base priority* π_k, and $\pi_k > \pi_l$ if task τ_k has a base priority higher than task τ_l. The RM policy [35] is used for base-priority assignment (alternative priority assignment policies will be discussed in Section 8). Specifically, for implicit-deadline task systems, $T_k < T_l$ implies $\pi_k > \pi_l$. In the following, the base priority is also called priority for short.

A job of τ_k is assigned an *effective priority* equal to its base priority π_k when it is ready to execute non-critical sections. Further, if a job of τ_k is holding a resource and ready to execute critical sections, it is assigned an effective priority equal

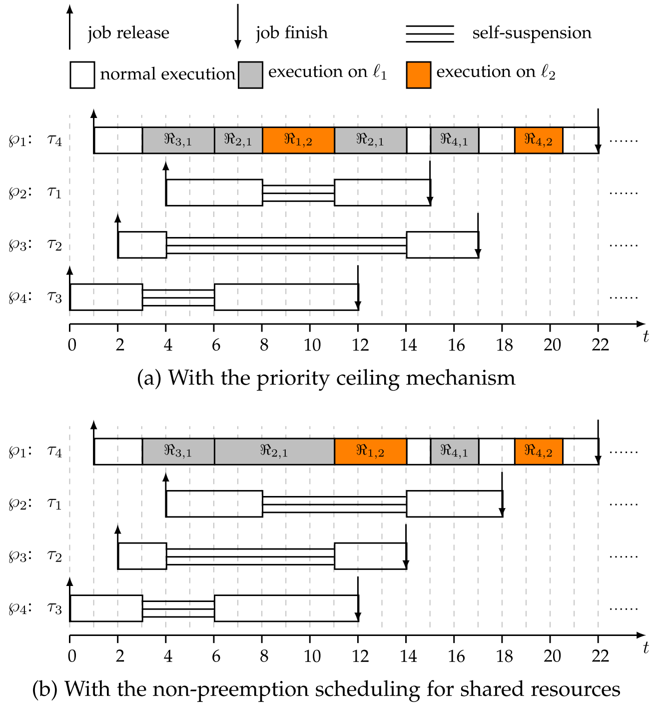


Fig. 1. Example schedules under ROP scheduling.

to $\pi^H + \pi_k$, where π^H is a priority level higher than any task in the systems, i.e., $\pi^H > \max\{\pi_k | \tau_k \in \tau\}$. At any point in time and on each processor, the job (or request) with the highest effective priority is dispatched.

Under the priority ceiling mechanism, each resource ℓ_q is associated with a *ceiling priority* Ω_q that is higher than the base priority of any task in the system. The ceiling priority of a resource ℓ_q is defined as follows:

$$\Omega_q = \pi^H + \max\{\pi_j | \exists q : \ell_q \in \mathcal{RS}(\tau_j)\}. \quad (4)$$

Locking Rules. At runtime, when a job J_k issues a request to resource ℓ_q that is assigned to a remote processor, it self-suspends on the local processor until the request is completed. If ℓ_q is currently held by another task, the request is inserted into a priority queue of ℓ_q. The priority used for queue insertion is the base priority of task τ_k. If ℓ_q is not held by any task, the following resource sharing mechanism determines whether access to resource ℓ_q is granted.

Let $\Re_{k,q}$ denote a request issued by a job J_k to resource ℓ_q. Suppose that job J_k issues a request $\Re_{k,q}$ that is bound to a processor $\wp_s$ at time t_1, and resource ℓ_q is not held by any task at time t_1.

- Under the priority ceiling mechanism, $\Re_{k,q}$ is granted at time t_1 if (i) there is no request holding a resource on processor $\wp_s$ or (ii) the highest ceiling priority of the resources that are currently locked is smaller than $\pi^H + \pi_k$ on processor $\wp_s$ at time t_1.
- Under the non-preemptive scheduling, $\Re_{k,q}$ is granted at time t_1 if (i) no request is holding a resource on processor $\wp_s$ and if (ii) $\Re_{k,q}$ is the highest-priority request on processor $\wp_s$ at time t_1.

As an example, Fig. 1 shows an ROP schedule on a 4-core processor, where $\wp_1$ is the synchronization processor and the other processors are application processors. Task τ_2 is suspended on $\wp_3$ after it requests ℓ_1 at time $t = 4$, and it is granted access to ℓ_1 on $\wp_1$ until $t = 6$ when τ_4 finishes its execution on ℓ_1. Task τ_1 requests ℓ_2 at time $t = 8$, and its request $\Re_{1,2}$ is allowed to preempt $\Re_{2,1}$ under the priority ceiling mechanism, as shown in Fig. 1a. That is because the priority ceiling of the resource that has been used at time $t = 8$ is $\pi^H + \pi_2$, while the request $\Re_{1,2}$ has a higher effective priority of $\pi^H + \pi_1$. In contrast, $\Re_{1,2}$ has to be delayed until $\Re_{2,1}$ finishes at time $t = 11$ under non-preemptive scheduling, as shown in Fig. 1b.

The above scheduling framework together with the locking rules ensures (as we will prove with Lemma 1) that each request can be blocked at most once. For clarity, a job J_k is said to be blocked at time t if J_k has issued a request $\Re_{k,q}$ that is bound to processor $\wp_s$ such that (i) $\Re_{k,q}$ is not scheduled at time t and (ii) another request issued by a lower-priority task is scheduled on processor $\wp_s$ at time t.

Lemma 1. *Under ROP scheduling with the priority ceiling mechanism or non-preemptive scheduling, a request can be blocked by at most one low-priority request.*

Proof. If we suppose that this is not the case, then there is a request $\Re_{k,q}$ that is blocked at least twice. Hence, at least two lower priority requests are scheduled on the same processor before $\Re_{k,q}$ completes. Suppose that the second such request, denoted by $\Re_{l,v}$ ($\pi_l < \pi_k$), is granted to lock resource R_v at time t_1 and starts executing at time t_2. Without loss of generality, we assume that $\Re_{k,q}$ is bound to processor $\wp_s$. According to the scheduling framework, $\Re_{l,v}$ has an effective priority $\pi^H + \pi_l$ at time t_2. Since $\Re_{k,q}$ is not scheduled at time t_2, it is either (i) holding resource ℓ_q but not scheduled or (ii) waiting to lock ℓ_q.

Suppose that (i) is true; then, $\Re_{k,q}$ has an effective priority $\pi^H + \pi_k$ at time t_2. By hypothesis, $\pi_l < \pi_k$, and thus, $\Re_{l,v}$ cannot be scheduled at time t_2 if the higher effective-priority request $\Re_{k,q}$ is not scheduled at time t_2. Thus, (i) is not true.

Suppose that (ii) is true; then, $\Re_{k,q}$ is not scheduled at time t_1. Two cases are considered: (a) ℓ_q is locked at time t_1, and (b) ℓ_q is not locked at time t_1.

If (a) holds, then $\Re_{l,v}$ cannot be scheduled at time t_1 under the non-preemptive scheduling. If (a) holds and $\Re_{l,v}$ is scheduled under the priority ceiling mechanism, then $\pi^H + \pi_l > \Omega_q \geq \pi^H + \pi_k$. However, this contradicts the assumption that $\pi_l < \pi_k$. If (b) holds and $\Re_{l,v}$ is scheduled under the non-preemptive scheduling, then $\Re_{l,v}$ is the highest-priority request at time t_1, which contradicts the assumption that $\pi_l < \pi_k$. If (b) holds and $\Re_{l,v}$ is scheduled under the priority ceiling mechanism, then $\pi^H + \pi_l$ must be greater than the ceiling priority of any resource that is locked on processor $\wp_s$ at time t_1. By hypothesis, if $\Re_{k,q}$ is not scheduled and ℓ_q is not held by any task at time t_1, then there is a resource with ceiling priority greater than $\pi^H + \pi_k$ that is locked at time t_1. This implies that $\pi^H + \pi_l > \pi^H + \pi_k$, which contradicts the assumption that $\pi_l < \pi_k$.

Thus, (ii) is not true under either the priority ceiling mechanism or the non-preemptive scheduling.

Therefore, neither (i) nor (ii) holds, which implies that no request can be blocked more than once. □

4.2 Tradeoff

In ROP scheduling, the critical sections of a task may be pushed to execute on different processors. This inevitably incurs some migration-related overhead. However, this resource sharing framework has the potential to break down the multiprocessor synchronization problem into uniprocessor sub-problems, which in turn improves the schedulability. There is also a tradeoff between performance and overhead for the considered resource-sharing mechanisms.

Scheduling. ROP scheduling is similar to *semi-partitioned* scheduling, where the execution of a task might be split among more than one processor. Therefore, this scheduling approach has additional overhead compared to traditional partitioned scheduling.

Nonetheless, from the implementation's point of view, ROP scheduling could benefit from the pre-planned nature of *push-migrations*: the jobs to be scheduled on the next processor are statically determined (more details can be found in [9]). Correspondingly, migrations are less pessimistic, as Cache-related Preemption and Migration Delay (CPMD) accounting is task specific. Further, since push-migrations can be implemented with mostly local states, migrations in ROP scheduling entail less overhead and are easier to implement.

Essentially, ROP scheduling breaks down the problem of scheduling tasks with shared resources on multiple processors into uniprocessor sub-problems, on which standard and consolidated techniques can still be applied. In particular, each request of a task can be blocked *only* by the requests that are bound to the *same* processor. Thus, higher schedulability is expected, as empirically confirmed in Section 9.

Resource Sharing. To ensure mutual exclusion, accesses to shared resources must be serialized. As a result, a request may be blocked, directly or indirectly, due to another request issued by a lower-priority job.

Let $b_{k,q}$ denote the maximum blocking time that a job of τ_k may experience each time it requests resource ℓ_q. Under the priority ceiling mechanism, a request can be blocked only if another job is holding a resource with a higher or equal ceiling priority. Thus, $b_{k,q}$ is upper bounded by

$$b_{k,q} = \max_{l,v}\{L_{l,v} | \pi_l < \pi_k, \Omega_v \geq \Omega_q\}. \tag{5}$$

Under the non-preemptive scheduling of shared resources, resource-holding requests cannot be preempted. Thus, a request can be block by *any* lower priority job that is holding a resource. Hence, $b_{k,q}$ is upper bounded by

$$b_{k,q} = \max_{l,v}\{L_{l,v} | \pi_l < \pi_k\}. \tag{6}$$

Clearly, $b_{k,q}$ in Eq. (5) might be smaller than in Eq. (6). Thus, the priority ceiling mechanism is in general preferable in terms of schedulability. Further, the priority ceiling mechanism together with the proposed ROP scheduling achieves a bounded speedup factor,[1] as shown in Section 5.

On the other hand, the implementation of non-preemptive scheduling is simpler: the scheduler is inactive, and context

1. It has been shown in [30] that the priority ceiling protocol [40] associated with the RM scheduling [35] achieves a constant speedup factor of 2 in uniprocessor scenarios.

switches can be avoided during the execution of critical sections. Conversely, the runtime system needs to maintain a list of priority ceiling orders under the priority ceiling mechanism, and several context switches might be incurred. Moreover, non-preemptive scheduling is in some cases empirically shown to have comparable performance with the priority ceiling mechanism, as discussed in Section 9.

To better understand this seemingly promising scheduling approach, the remainder of this paper will explore (i) how to partition tasks and shared resources in Section 5, (ii) how to perform schedulability analysis in Section 6, and (iii) how to justify ROP scheduling in Section 7 and Section 9 in terms of the speedup factor and empirical analysis, respectively.

5 Resource and Task Allocations

Since requests for shared resources are bound to designated processors, the blocking time of a request can be determined once the shared resources are partitioned. Further, since requests have higher priority than normal executions, the interference from any requests on a task can also be determined after shared resources are partitioned. In addition, the interference from non-critical sections can be determined once all higher priority tasks are partitioned. As a result, by partitioning shared resources in the first place and then assigning tasks in order of decreasing priority, any task being assigned will not jeopardize the schedulability of the tasks that have been successfully assigned. Following this principle, the algorithms that initially determine a set of synchronization processors and that allocate both shared resources and tasks are proposed.

Algorithm 1. Linear Search

Input: A set of n tasks τ, m processors $\wp$, and n_r resources $\mathcal{RS}$
Output: the feasibility of the system

```
for m^R = 1, ..., min(m, n_r) do
    if WFD (RS, m^R) then
        if FFRM (τ, m^R, m) then
            return feasible allocation
        end if
    end if
end for
return infeasible allocation
```

A *configuration* for initializing a set of processors as synchronization processors is determined iteratively, as shown in Algorithm 1. From a schedulability point of view, the reduction in the number of synchronization processors is a tradeoff between (i) an increase in the time spent on the execution of critical sections on the synchronization processors and (ii) a reduction in the time spent on the execution of non-critical sections on the application processors.

Since each resource is bound to one processor, there are at most $min(n_r, m)$ synchronization processors in the system. In each configuration, resources and tasks are respectively allocated according to the *Worst-Fit Decreasing* (WFD) algorithm and the *First-Fit Rate-Monotonic* (FFRM) algorithm.

Algorithm 2. Worst-Fit-Decreasing (WFD)

Input: A set of n_r resources $\mathcal{RS}$ and m^R synchronization processors
Output: resource allocations

```
sort the resources in a non-decreasing order of utilization
for q = 1, ..., n_r do
    Assign-Min(ℓ_q, m^R) //assign ℓ_q to the processor with minimum utilization among the m^R synchronization processors
    if U^{ℓ_q} + Σ_{R_v ∈ RS(℘_s)} U^{ℓ_v} > 1 then
        return infeasible allocation
    else
        assign ℓ_q to processor ℘_s
    end if
end for
return feasible allocation
```

In Algorithm 2, the resources are ordered in non-increasing order of utilization. The algorithm attempts to assign each resource to the synchronization processor with the *smallest* load. Notably, this is a well-known strategy for the bin-packing problem. The intuition underlying WFD is that by distributing resources evenly, it is sensible to reduce the time that tasks spend waiting for shared resources.

In Algorithm 3, tasks are sorted and prioritized in order of non-decreasing relative deadlines, i.e., $D_1 \leq D_2 \leq \cdots \leq D_n$. Then, the algorithm assigns (the non-critical sections of) the tasks to processors from the highest base priority to the lowest base priority. Each task is assigned to the first processor, starting from the application processors, that can accommodate the task according to the schedulability tests. Specifically, the algorithm attempts to assign each task to an application processor first, and if no such processor can accommodate this task, then the algorithm will attempt to assign the task to a synchronization processor.

In Algorithm 3, tasks are essentially sorted and prioritized by the RM policy [35]. In addition to the RM policy, other policies for task ordering and priority assignment can also be used under the proposed ROP scheduling framework, which we will further discuss in Sections 8 and 9.

For brevity, the algorithm is denoted by R-PCP-rm-rm when the priority ceiling mechanism and the RM policy are used (both for task ordering and priority assignment) and by R-NP-rm-rm when the non-preemptive scheduling and the RM policy are adopted.

Runtime Complexity. Algorithm 1 needs at most m rounds to find a feasible configuration if one exists. Further, the total sorting time of Algorithms 2 and 3 in all rounds can be amortized to $\mathcal{O}(n_r \log n_r + n \log n)$. In each round of Algorithm 1, Algorithm 2 runs with time complexity $\mathcal{O}(n_r \log m)$ by maintaining the processor utilization with a heap data structure, and Algorithm 3 requires $\mathcal{O}(mn)$ for assigning tasks. Overall, the algorithm runs in $\mathcal{O}(n_r \log n_r + n \log n + m(n_r \log m + mn))$, which is in pseudo-polynomial time complexity.

It is noted that in each round of Algorithm 1, a schedulability test is required in FFRM (in line 6 of Algorithm 3). In general, schedulability analysis is a tradeoff between complexity and accuracy. For the efficient analysis, a sufficient schedulability test is presented in the following.

Algorithm 3. First-Fit Rate-Monotonic (FFRM)

Input: A set of n tasks τ, m^R synchronization processors, and m processors
Output: task allocations
sort the tasks in non-decreasing order of relative deadlines
for $k = 1, \dots, n$ **do**
  for $c = m^R, \dots, m^R + m - 1$ **do**
    //suppose processors are indexed from 0
    $y = c$ **mod** m *// starts from application processors*
    if τ_k is schedulable on processor $\wp_y$ **then**
      assign τ_k on processor $\wp_y$
      Break
    end if
  end for
  if τ_k has not yet been assigned **then**
    return infeasible allocation
  end if
end for
return feasible allocation

6 Schedulability Analysis

To determine the speedup factor for R-PCP-rm-rm in Section 3.4, the standard fixed-priority response-time analysis is used to derive a schedulability analysis. Unlike the standard analysis framework for partitioned scheduling, we perform schedulability tests on synchronization processors and application processors respectively. Further, the analysis applies to both R-PCP- and R-NP- scheduling algorithms.

It is assumed in this section that each job of a task issues at most one request. To the best of our knowledge, even under this restrictive assumption, the problem of scheduling tasks with resource sharing under partitioned scheduling remains an open question—no preferable scheme is known in terms of speedup factors. The analysis for multiple resource accesses is given in the Appendix.

To analyze whether task τ_k is schedulable, the total delay that any job of τ_k suffers should be upper bounded. Therefore, the following definitions are introduced.

Definition 2. *Let $I_k(t)$ be the upper bound on the time whereby a job of τ_k is ready to execute its non-critical sections but is not scheduled in a time interval of length t.*

Definition 3. *Let $S_k(t)$ be the upper bound on the time whereby a job of τ_k is (i) suspended or (ii) ready to execute its critical sections but is not scheduled in a time interval of length t.*

Based on Definitions 2 and 3, the worst-case response time of a task τ_k is derived.

Lemma 2. *The smallest t satisfying*

$$C_k + A_k + I_k(t) + S_k(t) \leq t. \tag{7}$$

is a safe upper bound on the response time of task τ_k.

Proof. Proof by contradiction. Suppose that there is a $t \leq D_k$ such that $C_k + A_k + I_k(t) + S_k(t) \leq t$, and a job J_k does not finish by t. At any point in time while J_k is released but not finished, J_k is either (i) executing non-critical sections, (ii) executing critical sections, (ii) ready to execute non-critical sections but is not scheduled to do so, (iv) suspended or ready to execute critical section(s) but is not scheduled to do so.

According to Definitions 2 and 3, the cumulative delays due to (iii) and (iv) are upper bounded by $I_k(t)$ and $S_k(t)$, respectively. If J_k does not finish by t, it must be the case that J_k requires an execution time of greater than $t - I_k(t) - S_k(t)$. By definition, any job of τ_k executes non-critical sections and critical sections for a time of at most C_k and A_k, respectively. Thus, $C_k + A_k > t - I_k(t) - S_k(t)$. This contradicts the assumption that $C_k + A_k + I_k(t) + S_k(t) \leq t$. □

Further, a schedulability condition is obtained in the following theorem.

Theorem 1. *A set of tasks τ is schedulable if for each task $\tau_k \in \tau$ such that*

$$\exists t \in [0, D_k] : C_k + A_k + I_k(t) + S_k(t) \leq t. \tag{8}$$

Proof. The proof is essentially analogous to that of Lemma 2. The execution demand of a job J_k is upper bounded by $C_k + A_k$, and the total cumulative delay in a time interval of length t is upper bounded by $I_k(t) + S_k(t)$. Thus, if there is a $t \in [0, D_k]$ such that $C_k + A_k + I_k(t) + S_k(t) \leq t$, then the job will finish by D_k. The proof follows. □

To bound $I_k(t)$ and $S_k(t)$, the interfering workloads of other tasks are required. Under ROP scheduling, each time that a job requests a resource on a remote processor, it is suspended on the local processor until the request is complete. With the suspension-based scheduling, no *critical instant theory* that concretely captures the worst-case behavior in analyzing a task has been established. To cover the worst-case behavior, an accounting of the *carry-in* jobs is commonly used in schedulability analysis [10], [19], [31], [34]. Correspondingly, the workload can be bounded as below.

Lemma 3. *(From [10], [19]) For a sporadic self-suspending task system, the workload of a task τ_{ss} in a time interval of length t is upper bounded by*

$$w_{ss}(t) = \left\lceil \frac{t + R_{ss} - c_{ss}}{T_{ss}} \right\rceil c_{ss}. \tag{9}$$

where c_{ss} is the worst-case execution time of τ_{ss}.

This workload function is adopted in the remainder of the paper. In particular, we use the workload function for non-critical sections and critical sections. First, a task executes the non-critical sections on its local processor and self-suspends when it requests a resource on a remote processor. The non-critical sections of a task can be modeled as a self-suspending task on its local processor. According to Lemma 3, the workload of non-critical sections of any task τ_j in a time interval of length t is upper bounded by

$$W_j(t) = \left\lceil \frac{t + R_j - C_j}{T_j} \right\rceil C_j. \tag{10}$$

Further, all requests of task τ_j to a resource ℓ_q are bound to a designated synchronization processor. From the perspective of critical sections, the requests of task τ_j arrive sporadically with jitter (such jitter is due to the executions of non-critical sections on local processors). Thus, the execution time of the critical sections of τ_j for resource ℓ_q in a time interval of length t is upper bounded by

$$E_{j,q}(t) = \left\lceil \frac{t + R_j - A_{j,q}}{T_j} \right\rceil A_{j,q}. \tag{11}$$

With the workload bounds in place, the upper bounds on $I_k(t)$ and $S_k(t)$ are derived in the following.

6.1 Bounding $I_k(t)$

In ROP scheduling, task τ_k could be assigned to an application processor or a synchronization processor. If τ_k is assigned to an application processor, it can only be preempted by the non-critical sections of other tasks assigned to the same processor. Otherwise, if it is assigned to a synchronization processor, it could suffer additional interference from executions of critical sections on the processor. Therefore, the maximum interference on the executions of non-critical sections of task τ_k , i.e., $I_k(t)$, differs from the processor to which τ_k is assigned.

In preparation, let $\tau(\wp_c)$ and $\mathcal{RS}(\wp_c)$ denote the tasks and resources assigned to processor $\wp_c$, respectively.

Lemma 4. *If task τ_k is assigned to an application processor $\wp_a$, then $I_k(t)$ is upper bounded by*

$$I_k(t) = \sum_{\tau_h \in \tau(\wp_a), \pi_h > \pi_k} W_h(t). \tag{12}$$

Proof. Since non-critical sections are scheduled based on base priorities, at any point in time while a job J_k is ready but not scheduled in an application processor $\wp_a$, there must be a higher priority task, e.g., τ_h, executing non-critical sections on processor $\wp_a$. According to the workload bound (as shown in Eq. (10)), task τ_h executes its non-critical sections for at most $W_h(t)$ in a time interval of length t. Thus, job J_k can be delayed by all higher priority tasks for a total of at most $\sum_{\tau_h \in \tau(\wp_a), \pi_h > \pi_k} W_h(t)$. □

Lemma 5. *If task τ_k is assigned to a synchronization processor $\wp_s$, then $I_k(t)$ is upper bounded by*

$$I_k(t) = \sum_{\tau_h \in \tau(\wp_s), \pi_h > \pi_k} W_h(t) + \sum_{\tau_j \neq \tau_k, \ell_q \in \mathcal{RS}(\wp_s)} E_{j,q}(t). \tag{13}$$

Proof. In ROP scheduling, requests have higher effective priority compared to non-critical sections. Thus, at any point in time while a job J_k is ready to execute its non-critical sections but is not scheduled on processor $\wp_s$, there is either (i) a higher priority task executing non-critical sections on processor $\wp_s$ or (ii) a task other than τ_k executing critical sections on processor $\wp_s$.

It has been proven in Lemma 4 that the cumulative delay incurred by (i) can be upper bounded by $\sum_{\tau_h \in \tau(\wp_s), \pi_h > \pi_k} W_h(t)$. For (ii), only the requests to the resources assigned on processor $\wp_s$ can interfere with τ_k. Since the execution time of a task τ_j ($j \neq k$) on a resource ℓ_q in a time interval of length t is at most $E_{j,q}(t)$, job J_k could be delayed due to the executions of other tasks' critical sections for a total of at most $\sum_{\tau_j \neq \tau_k, \ell_q \in \mathcal{RS}(\wp_s)} E_{j,q}(t)$. Summing up the delays from (i) and (ii), the proof follows. □

6.2 Bounding $S_k(t)$

Since each shared resource is statically assigned to a processor, only the requests bound to the same processor can interfere with each other. Meanwhile, requests are assigned higher effective priority compared to non-critical sections; thus, a request can only be delayed by other requests.

First, a request $\Re_{k,q}$ can be delayed by all higher priority requests and blocked by a lower priority request bound to the same processor.

Lemma 6. *Suppose that a job J_k requests a resource ℓ_q on a synchronization processor $\wp_s$ and $\sum_{\ell_v \in \mathcal{RS}} N_{k,v} = N_{k,q} = 1$; then, $S_k(t)$ is upper bounded by*

$$S_k(t) = \sum_{\pi_h > \pi_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{h,v}(t) + b_{k,q}. \tag{14}$$

Proof. Since requests have higher effective priority than non-critical sections, at any point in time while a request of τ_k is not scheduled, there must be another request scheduled. Since resource ℓ_q is bound to processor $\wp_s$, the request of τ_k can only be delayed by the requests bound to processor $\wp_s$.

In a time interval of length t, each higher priority task τ_h ($\pi_h > \pi_k$) requires an execution on resource ℓ_v for at most $E_{h,v}(t)$; thus, each request of τ_k incurs a delay due to higher priority tasks for at most $\sum_{\pi_h > \pi_k, R_v \in \mathcal{RS}(\wp_s)} E_{h,v}(t)$. By Lemma 1, each request of task τ_k can be blocked by at most one request from lower priority tasks. Thus, by definition, each request of τ_k to resource ℓ_q can be blocked for a duration of at most $b_{k,q}$. Summing up $\sum_{\pi_h > \pi_k, R_v \in \mathcal{RS}(\wp_s)} E_{h,v}(t)$ and $b_{k,q}$, the proof follows. □

Further, if task τ_k and the resource it requests are assigned to the same processor $\wp_s$, then the cumulative delay $I_k(t) + S_k(t)$ is upper bounded by all higher priority workloads on processor $\wp_s$. Specifically, the maximum workload of the non-critical sections of the higher priority tasks on processor $\wp_s$, as well as the maximum workload of all requests, from other tasks, that are bound to processor $\wp_s$.

Lemma 7. *Suppose that task τ_k and the resource ℓ_q that it requests are assigned to the same processor $\wp_s$, and $\sum_{R_v \in \mathcal{RS}} N_{k,v} = N_{k,q} = 1$; then, $I_k(t) + S_k(t)$ is upper bounded by*

$$\begin{aligned} I_k(t) + S_k(t) = &\sum_{\tau_h \in \tau(\wp_s), \pi_h > \pi_k} W_h(t) \\ &+ \sum_{\tau_j \neq \tau_k, R_v \in \mathcal{RS}(\wp_s)} E_{j,v}(t). \end{aligned} \tag{15}$$

Proof. The proof is similar to that of Lemma 6. At any point in time while a job J_k is not scheduled before it finishes, there is either (i) a higher priority task executing non-critical sections or (ii) some other task executing critical sections on processor $\wp_s$. As already proved in Lemma 6, (i) can be upper bounded by $\sum_{\tau_h \in \tau(\wp_s), \pi_h > \pi_k} W_h(t)$, and (ii) can be upper bounded by $\sum_{\tau_j \neq \tau_k, R_v \in \mathcal{RS}(\wp_s)} E_{j,v}(t)$. □

According to the workload bound in Eq. (11), the value of $E_{j,q}(t)$ depends on the worst-case response time R_j. Since the tasks are assigned in order of base priority under FFRM, when analyzing task τ_k, the worst-case response times of higher priority tasks are known. However, the worst-case response times of lower priority tasks are unknown. To derive a safe upper bound on $\sum_{\tau_j \neq \tau_k, \ell_q \in \mathcal{RS}(\wp_s)} E_{j,q}(t)$ in Eqs. (13) and (15), R_j is temporarily set to D_j for each lower priority task τ_j ($\pi_j < \pi_k$). This is because for any given t, a larger R_j implies a larger $E_{j,q}(t)$. The actual value of R_j is derived when analyzing task τ_j.

7 Speedup Factor

In this section, we prove that any implicit-deadline task set that is feasible upon a platform consisting of m unispeed processors is deemed to be schedulable by R-PCP-rm-rm if the platform is at least x times as fast. To reiterate, each job of a task is assumed to issue at most one request.

In the following lemma, the necessary conditions for any feasible scheduling is derived, being based on the concept of demand bound functions as defined in Section 3.4.

Lemma 8. *Any implicit-deadline task system τ that is feasible upon a platform consisting of m unispeed processors must satisfy*

$$\forall \tau_k \in \tau, \quad U_k \leq 1, \tag{16}$$

$$\forall \ell_q \in \mathcal{RS}, \quad U_{\sum}^{\ell_q} \leq 1, \tag{17}$$

$$U_{\sum} \leq m, \tag{18}$$

and $\forall \tau_k \in \tau, \forall \ell_q \in \mathcal{RS}(\tau_k)$:

$$\max\{L_{l,q} | D_l > D_k\} + \sum_{D_h \leq D_k} dbf_h^{\ell_q}(D_k) \leq D_k. \tag{19}$$

Proof. By definition, each job of task τ_k requires an execution time of at most $C_k + A_k$. Thus, it is necessary for meeting all deadlines of task τ_k that $C_k + A_k \leq D_k$. Since $D_k = T_k$ and $U_k = (C_k + A_k)/T_k$, $U_k \leq 1$. Since accesses to shared resources are serialized, $U^{\ell_q} \leq 1$ for each resource ℓ_q in the system. Moreover, an implicit-deadline task set is feasible on a platform of m unispeed processors only if the cumulative utilization of the tasks is bounded by m, i.e., $U_{\sum} = \sum_{\tau_k \in \tau} U_k \leq m$.

Due to mutual exclusion, contested requests are serialized. Suppose an arbitrary task τ_k that issues a request $\Re_{k,q}$ at time t_1. Consider that task τ_l, with the longest critical section to resource ℓ_q among all tasks with relative deadlines larger than D_k, issues the longest request $\Re_{l,q}$ right before t_1, and all tasks with relative deadlines smaller than D_k issue requests to resource ℓ_q at time t_1 and generate jobs that only request ℓ_q as soon as possible. In a time interval $[t_1, t_1 + t]$, the cumulative request demand is at most $L_{l,q} + \sum_{D_h \leq D_k} dbf_h^{\ell q}(t)$. Therefore, if all requests are feasibly schedulable, for each task τ_k and each resource ℓ_q, $\max\{L_{l,q} | D_l > D_k\} + \sum_{D_h \leq D_k} dbf_h^{\ell q}(D_k) \leq D_k$. The proof follows. □

In multiprocessor systems with shared resources, the optimal scheduling algorithm is in general unknown. For algorithmic analysis, the necessary schedulability conditions are used as baselines. To obtain a speedup factor, the schedulability conditions for R-PCP-rm-rm are compared with the necessary conditions in Lemma 8. To begin with, the mathematical relations between the workload bound functions and the demand bound functions are established.

Lemma 9. *For any constrained-deadline task τ_j, if $R_j \leq D_j$, then for any $t \geq D_j$*

$$3dbf_j^C(t) \geq W_j(t), \tag{20}$$

and

$$3dbf_j^{\ell_q}(t) \geq E_{j,q}(t). \tag{21}$$

Proof. Since $R_j \leq D_j$

$$\left\lceil \frac{t + R_j - C_j}{T_j} \right\rceil \leq \left\lceil \frac{t + D_j}{T_j} \right\rceil \leq \left\lceil \frac{t}{T_j} \right\rceil + 1.$$

Further, since $D_j \leq T_j$

$$\left\lfloor \frac{t - D_j}{T_j} \right\rfloor \geq \left\lfloor \frac{t}{T_j} \right\rfloor - 1.$$

Thus,

$$\left\lceil \frac{t + R_j - C_j}{T_j} \right\rceil - \left\lfloor \frac{t - D_j}{T_j} \right\rfloor \leq \left\lceil \frac{t}{T_j} \right\rceil - \left\lfloor \frac{t}{T_j} \right\rfloor + 2 \leq 3.$$

The last inequality holds because for any $b > 0$, $\lceil b \rceil - \lfloor b \rfloor \leq 1$. Hence

$$\left\lceil \frac{t + R_j - C_j}{T_j} \right\rceil C_j \leq \left(\left\lfloor \frac{t - D_j}{T_j} \right\rfloor + 1 \right) C_j + 2C_j.$$

According to the definitions of $W_j(t)$ (in Eq. (10)) and $dbf_j^C(t)$ (in Eq. (2)) and by the assumption that $t \geq D_j$, $W_j(t) \leq dbf_j^C(t) + 2C_j$. Further, $dbf_j^C(t) \geq C_j$ when $t \geq D_j$ according to Eq. (2). Thus, $W_j(t) \leq 3dbf_j^C(t)$. This proves inequality (20). Inequality (21) is proved analogously. □

Next, an upper bound on the cumulative utilization of the shared resources of each synchronization processor is derived. Let $U^\ell(\wp_s)$ be the total utilization of the shared resources assigned to synchronization processor $\wp_s$, i.e., $U^\ell(\wp_s) = \sum_{\ell_q \in \mathcal{RS}(\wp_s)} U^{\ell_q}$. Since shared resources are assigned according to the WFD algorithm, bounding $U^\ell(\wp_s)$ is analogous to the classic makespan problem.

Lemma 10. *Given a feasible task system τ with m^R synchronization processors, for any $\wp_s \in \wp$, $U^\ell(\wp_s)$ is upper bounded under the WFD algorithm by*

$$U^\ell(\wp_s) \leq 1 + \frac{U^{\mathcal{RS}} - 1}{m^R}. \tag{22}$$

Proof. Two cases are considered: (i) $n_r > m^R$ and (ii) $n_r \le m^R$.

For (i), let ℓ_u be the last resource assigned on $\wp_s$. Since the WFD algorithm assigns each resource to the processor with the least load, it follows that

$$U^{\ell}(\wp_s) - U^{\ell_u} \le \frac{U^{\mathcal{RS}} - U^{\ell_u}}{m^R}$$
$$\Rightarrow U^{\ell}(\wp_s) \le U^{\ell_u} + \frac{U^{\mathcal{RS}} - U^{\ell_u}}{m^R}.$$

For (ii), there is at most one resource on each processor under the WFD algorithm. Let ℓ_v be the resource with the largest utilization in the system. Then,

$$U^{\ell}(\wp_s) \le U^{\ell_v} \le U^{\ell_v} + \frac{U^{\mathcal{RS}} - U^{\ell_v}}{m^R}.$$

Since the task system is by hypothesis feasible, according to the second condition in Lemma 8, $U^{\ell_q} \le 1$ for $\forall \ell_q$. Hence,

$$\forall \ell_q, \quad U^{\ell_q} + \frac{U^{\mathcal{RS}} - U^{\ell_q}}{m^R} \le 1 + \frac{U^{\mathcal{RS}} - 1}{m^R}.$$

Therefore, inequality (22) holds for both cases. □

The following theorem shows that the speedup factor of R-PCP-rm-rm is $11 - 6/(m+1)$ when each job of each task issues at most one request irrespective of how many shared resources are present.

Theorem 2. *Suppose that $m \ge 2$ and that each job of each task issues at most one request; then, the proposed algorithm R-PCP-rm-rm guarantees a speedup factor of $11 - \frac{6}{m+1}$.*

Proof. If a feasible task set τ is not schedulable under R-PCP-rm-rm, then there is a task τ_k that is not schedulable on any processor under R-PCP-rm-rm. Under ROP scheduling, all the tasks assigned before τ_k have been ensured schedulable, i.e., $R_h \le D_h$ for $h = 1, 2, \ldots, k-1$. From Theorem 1, it must be the case that for every application processor,

$$C_k + A_k + I_k(D_k) + S_k(D_k) > D_k. \tag{23}$$

Without loss of generality, suppose that task τ_k requests a resource ℓ_q that is bound to processor $\wp_s$. Then, summing over all $m^C = m - m^R$ application processors and after reformulation, we obtain

$$\frac{C_k + A_k}{D_k} + \frac{b_{k,q}}{D_k} + \frac{\sum_{\pi_h > \pi_k} W_h(D_k)}{m^C D_k} + \frac{\sum_{\pi_h > \pi_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{h,v}(D_k)}{D_k} > 1. \tag{24}$$

Since task set τ is by hypothesis feasible, it must be the case that $U_k \le 1$ according to Lemma 8. That is

$$\frac{C_k + A_k}{T_k} = \frac{C_k + A_k}{D_k} \le 1. \tag{25}$$

Let $\Re_{l,v}$ be the longest request from a task with $D_l \ge D_k$ that blocks task τ_k, i.e., $b_{k,q} = L_{l,v}$. By hypothesis, task set τ is feasible. Two cases are considered:

- τ_k requests ℓ_v, i.e., $\ell_q = \ell_v$. From Lemma 8, $\max\{L_{l,v} | D_l > D_k\} + \sum_{D_h \le D_k} dbf_h^{\ell_q}(D_k) \le D_k$. Thus, $b_{k,q} = L_{l,v} = \max\{L_{l,v} | D_l > D_k\} \le D_k$.
- τ_k does not request ℓ_v, i.e., $\ell_q \ne \ell_v$. Since the priority ceiling mechanism is used, there must be a higher priority task, τ_h ($\pi_h > \pi_k$), that requests ℓ_v. According to Eq. (5), $b_{h,v} = L_{l,v}$. From Lemma 8, $\max\{L_{l,v} | D_l > D_h\} + \sum_{D_y \le D_h} dbf_y^{\ell_q}(D_h) \le D_h$. Hence, $b_{k,q} = b_{h,v} = \max\{L_{l,v} | D_l > D_h\} \le D_h$. Further, since base priorities are assigned according to the RM algorithm, it follows that $D_h < D_k$. Therefore, $b_{k,q} < D_k$.

In either case, $b_{k,q} \le D_k$. Specifically

$$\frac{b_{k,q}}{D_k} \le 1. \tag{26}$$

By Lemma 8, task set τ is feasible only if $U \le m$. By definition, $U_{\sum} = U^C + U^{\mathcal{RS}}$; thus, $U^C \le m - U_{\mathcal{RS}}$. Therefore,

$$\frac{\sum_{\pi_h > \pi_k} W_h(D_k)}{m^C D_k} \overset{\text{Eq. (20)}}{\le} \frac{\frac{3\sum_{\tau_x \in \tau} dbf_x^C(D_k)}{D_k}}{m^C} \le \frac{3U^C}{m^C} \le \frac{3(m - U^{\mathcal{RS}})}{m - m^R}. \tag{27}$$

Further

$$\frac{\sum_{\pi_h > \pi_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{h,v}(D_k)}{D_k} \overset{\text{Eq. (21)}}{\le} \frac{3\sum_{\pi_h > \pi_k, \ell_v \in \mathcal{RS}(\wp_s)} dbf_h^{\ell_v}(D_k)}{D_k} \le 3U^{\ell}(\wp_s) \overset{\text{Eq. (22)}}{\le} 3\left(1 + \frac{U^{\mathcal{RS}} - 1}{m^R}\right). \tag{28}$$

Suppose that task set τ is schedulable under R-PCP-rm-rm when each processor is at least x times as fast. Then, summing over the corresponding terms in inequalities (25), (26), (27), and (28) and to contradict to inequality (24)

$$\frac{1}{x} + \frac{1}{x} + \frac{3(m - U^{\mathcal{RS}})}{x(m - m^R)} + \frac{3}{x} + \frac{3(U^{\mathcal{RS}} - 1)}{x \cdot m^R} \le 1$$
$$\Rightarrow x \ge 5 + \frac{3(U^{\mathcal{RS}} - 1)}{m^R} + \frac{3(m - U^{\mathcal{RS}})}{m - m^R}.$$

Let $f(m^R) = 5 + \frac{3(U^{\mathcal{RS}} - 1)}{m^R} + \frac{3(m - U^{\mathcal{RS}})}{m - m^R}$. Two cases are considered.

- m is even. Set $m^R = m/2$, then

$$f(m^R) = 5 + \frac{6(m-1)}{m} = 11 - 6/m.$$

- m is odd, i.e, $m \geq 3$, due to the assumption that $m \geq 2$. Then, $(m-1)/2 \geq 1$. Two subcases are further considered:
  If $U^{\mathcal{RS}} \geq \frac{m+1}{2}$, then set $m^R = \frac{m+1}{2}$. Thus
  $$f(m^R) = 5 + 6\left(\frac{(m^2-1)+2-2U^{\mathcal{RS}}}{m^2-1}\right) \leq 11 - \frac{6}{m+1}.$$
  If $U^{\mathcal{RS}} < \frac{m+1}{2}$, then set $m^R = \frac{m-1}{2}$. Thus
  $$f(m^R) = 5 + 6\left(\frac{(m^2-1)-2m+2U^{\mathcal{RS}}}{m^2-1}\right) \leq 11 - \frac{6}{m+1}.$$

In either case, $f(m^R)$ is upper bounded by $11 - \frac{6}{m+1}$. This prove that if a task set τ is feasible (on a platform of m unispeed processors), then τ is schedulable under R-PCP-rm-rm when the processors are $11 - \frac{6}{m+1}$ times as fast by setting m^R to $\frac{m}{2}$, $\frac{m+1}{2}$, or $\frac{m-1}{2}$, depending on the values of m and $U^{\mathcal{RS}}$.

It is noted that m^R is greedily set in the analysis; thus, it is possible that $m^R > n_r$, i.e., the number of synchronization processors might be greater than the number of shared resources. However, this does not jeopardize the proof of the speedup factor. In this case, inequality (22) still holds even though no shared resource is bound to the remaining $m^R - n_r$ "*synchronization*" processors, as proved in Lemma 10. Therefore, inequality (28) and the upper bounds of $f(m^R)$ still hold. By Definition 1, the R-PCP-rm-rm algorithm has a speedup factor of $11 - \frac{6}{m+1}$. □

8 Task Ordering and Priority Assignment

The proposed ROP scheduling framework does not place any restriction on task ordering, priority assignment, or any specific task or resource partitioning. In this work, we restrict ourselves to using the WFD algorithm for resource assignment (as in Algorithm 2) and the First-Fit (FF) algorithm for task assignment (as in Algorithm 3). Sophisticated task and resource partitioning are future work.

In the previous sections, the tasks were sorted and prioritized according to the RM policy. In this section, alternative approaches for task ordering and priority assignment are further discussed.

8.1 Slack Monotonic (SM) Scheme

SM is a priority assignment scheme that assigns higher priorities to tasks with less slack, where the slack of a task τ_i was originally defined as $T_i - C_i$. It has been shown in [2], [3] that SM performs reasonably well in multiprocessor static-priority scheduling in principle (i.e., guarantees a utilization bound lower than that of RM).

Intuitively, a larger slack infers a greater tolerance of deadline misses. Therefore, higher priorities are assigned to the tasks with less slack. In addition to the local executions, tasks may self-suspend while waiting for the response of the requested resources. Thus, the slack of a task τ_k is defined as $\theta_k = T_k - C_k - s'_k$, where s'_k is an upper bound on the total self-suspensions of a job J_k. Suppose that the time J_k spends to finish its requests on processor $\wp_c$ is upper bounded by $\mu_k^c(D_k)$ (as we will prove in the Appendix). Thus, s'_k can be safely upper bounded by summing over the respective terms, i.e., $s'_k \leq \sum_{\wp_c \in \wp} \mu_k^c(D_k)$. Accordingly, the slack of a task τ_k is calculated by

$$\theta_k = T_k - C_k - \sum_{\wp_c \in \wp} \mu_k^c(D_k). \tag{29}$$

Tasks are then sorted and prioritized in non-increasing order of slack and with ties broken arbitrarily. Thus, for any two tasks τ_i and τ_j, the following holds.

$$\theta_i \leq \theta_j \Leftrightarrow i < j \Leftrightarrow \pi_i > \pi_j. \tag{30}$$

The algorithm is denoted by R-PCP-sm-sm when the priority ceiling mechanism is used and by R-NP-sm-sm if the non-preemptive scheduling is used.

8.2 Modified Optimal Priority Assignment

The optimal priority assignment (OPA) algorithm [6] was originally derived to find an optimal priority ordering for uniprocessor fixed-priority scheduling in polynomial time. It assigns each priority level, starting from the lowest one to the highest one, to tasks iteratively. In the schedulability test in each round, the tasks that have not been assigned priorities are considered as higher priority tasks. The priority assignment process terminates as soon as no task can be assigned at some priority level or all priority levels are successfully assigned.

Davis and Burns [22] further studied the optimal priority assignment for multiprocessor systems. It is shown that the OPA approach is applicable for globally scheduled multiprocessor systems if the schedulability test meets the following conditions [22]. First, the schedulability of a task τ_i may depend on any independent properties of higher priority tasks but not on any properties related to their relative priority ordering. Second, the schedulability of a task τ_i may depend on any independent properties of lower priority tasks but not on any properties related to their relative priority ordering. Third, when the priority levels of two tasks of adjacent priority are swapped, the task with higher priority cannot become unschedulable if it was schedulable at the lower priority level.

In this work, the request delays as well as the workloads (Eqs. (10) and (11)) used in the response-time analysis depend on the response time of the higher priority tasks, which is in turn related to the relative priority ordering of these tasks. Thus, the first condition does not hold in the tests. As a result, the schedulability test is incompatible with the OPA algorithm. However, the OPA scheme under ROP scheduling remains worth studying since the OPA scheme may empirically perform well under OPA-incompatible analysis, as evidenced by the modified OPA in [43].

In ROP scheduling, the schedulability test must be performed each time a task is partitioned, and the worst-case response time of *each* higher priority task is required in the response-time analysis. Therefore, we apply a modified OPA together with our analysis for the schedulability test in each round. First, tasks are sorted in non-decreasing order of period or slack (according to Eq. (30)). Then, the tasks are partitioned to processors according to the First-Fit algorithm (e.g., Algorithm 3). Each time a task is partitioned, we iteratively assign priorities, starting from the lowest one (e.g., priority level n), to the tasks that have been partitioned until each such task has been successfully assigned a priority or some priority level cannot been assigned to any such task. In the response-time analysis, the worst-case response time of a task that has been partitioned but not assigned a priority is greedily set to its relative deadline, and the priority of such task is assumed to be the highest priority level. For instance, for any such task τ_j, $R_j = D_j$ and $\pi_j = \pi_0$. A task set is schedulable if each task is successfully assigned a priority and is unschedulable otherwise.

When the priority ceiling mechanism is used, the algorithm is denoted by R-PCP-rm-opa if tasks are sorted according to the RM policy and by R-PCP-sm-opa if tasks are sorted by the SM scheme (according to Eq. (30)). Analogously, when the non-preemptive scheduling is used, the corresponding algorithm is denoted by R-NP-rm-opa if tasks are sorted by the RM policy and by R-NP-sm-opa if tasks are sorted by the SM scheme.

9 Empirical Comparison

In this section, ROP scheduling is compared with existing methods using synthesized task sets. The presented analyses are implemented in SET-MRTS,[2] and the metric used to compare the results is to measure the *acceptance ratio*, that is, the number of task sets that are deemed schedulable divided by the number of task sets tested.

9.1 Experimental Setup

Parameter Generation. The experimental setup resembles the design of prior schedulability experiments for locking protocols [13], [15], [33], [46]. Two multiprocessor platforms with $m \in \{4, 8\}$ unispeed processors are considered. Task sets are generated according to the total utilization, i.e., U, from $0.05m$ to m in steps of $0.05m$.

For a given $U_{\sum}$, at least 1,000 task sets are tested. The task set characteristics varied by per-task utilization U_k, per-task period T_k, the maximum critical section length $L^{\max}$, the number of shared resources n_r, the probability p^r that each task requires each resource, and the maximum number of times that each job uses each resource $N^{\max}$. In the experiments, U_k is randomly chosen in $[0, 1]$ using an exponential distribution with a mean value of 0.1 (*light*) or 0.25 (*medium*), and $C_k + A_k$ is set to $T_k U_k$. T_k is randomly chosen from log-uniform distributions ranging over $[10\ ms, 100\ ms]$ (*homogeneous*) or $[1\ ms, 1000\ ms]$ (*heterogeneous*). $L^{\max}$ is chosen uniformly from $[1\ us, 50\ us]$ (*short*), $[50\ us, 150\ us]$ (*medium*), or $[150\ us, 300\ us]$ (*long*). n_r is varied across $\{1, 2, 4, 8\}$, p^r is varied across $\{0.1, 0.25\}$, and $N^{\max}$ is varied across $\{1, 3, 5\}$. These parameter configurations cover a total of 576 combinations, including both high- and low-contention scenarios.

In the experiments, $A_{k,q}$ is set to $N_{k,q} L_{k,q}$, and A_k is set to $\sum_{\ell_q} N_{k,q} L_{k,q}$. Further, by definition, $C_k = U_k T_k - A_k$. In case $C_k < 0$, C_k is mandatorily set to 1, and T_k is updated to be $(1 + \sum_{\ell_q} N_{k,q} L_{k,q}) / U_k$.

Algorithms Compared. In each experimental scenario, the schedulability of the proposed ROP scheduling is tested together with the following scheduling algorithms.

LP-FMLP [46]: the linear-programming-based (LP-based) analysis for the Global Fixed-Priority (GFP) scheduling with the FMLP [46].

LP-DPCP [13]: the LP-based analysis for the DPCP, in which the resources are partitioned evenly among m processors and the tasks are assigned using the WFD algorithm as in [13].

LP-R-DPCP: a combination of the ROP heuristic based on Algorithm 1 and the LP-based blocking analysis as used for DPCP in [13]. The LP-based analysis is performed each time a task is assigned.

GS-MSRP [43]: the Greedy Slacker (GS) partitioning heuristic with the MSRP, using the original blocking anaysis as presented in [25]. It has been shown in [43] that GS-MSRP outperforms previously developed synchronization-aware [33] and blocking-aware [37] partitioning algorithms.

LP-EE-vpr [5]: a virtualization-based dynamic scheduling algorithm with the assumption that each job requests at most one shared resource. LP-EE-vpr guarantees a speedup factor of 8 if $n_r \leq m$, which is the best result in terms of speedup factor under this restrictive assumption. Since there is no sufficient schedulability analysis for LP-EE-vpr, the necessary condition is compared instead.

There are also LP-based analyses [13], [46] for the other multiprocessor real-time locking protocols such as the MPCP [38], the DFLP [14], the refined FMLP [12], the Priority Inheritance Protocol (PIP) [40], and the FMLP$^+$ [15], etc. Empirical studies with the LP-based analyses [13] have shown that the DPCP and the DFLP performed better than the refined FMLP and the MPCP for partitioned scheduling, and that the DPCP often performed better than the DFLP. Therefore, only LP-DPCP is compared among the LP-based analyses for partitioned scheduling, due to space limits. Similarly, only LP-FMLP is compared among the LP-based analyses for global scheduling.

To compare with LP-EE-vpr, we also generate task sets such that each task has at most one critical section. This results in 192 additional experiment combinations, totaling 768 experiment scenarios.

To further study the impact of priority assignment under ROP scheduling, the algorithms discussed in Section 8, i.e., R-PCP-sm-sm, R-PCP-rm-opa, R-PCP-sm-opa, R-NP-sm-sm, R-NP-rm-opa, and R-NP-sm-opa, are tested as well.

9.2 Results

We report the major trends characterizing the experimental results from the 768 configurations as follows.

Partitioned Scheduling Performs Well. Our results clearly show that partitioned scheduling is highly effective for multiprocessor real-time scheduling with shared resources.

First, partitioned scheduling outperforms global scheduling. In the LP-based analyses, higher schedulability is obtained under partitioned scheduling, although global scheduling may perform better when critical sections are long, as shown in Fig. 3 (e.g., the acceptance ratio of

2. SET-MRTS is a schedulability experiment toolkit for multiprocessor real-time systems. The source codes are available online https://github.com/ChenZewei/SET-MRTS.

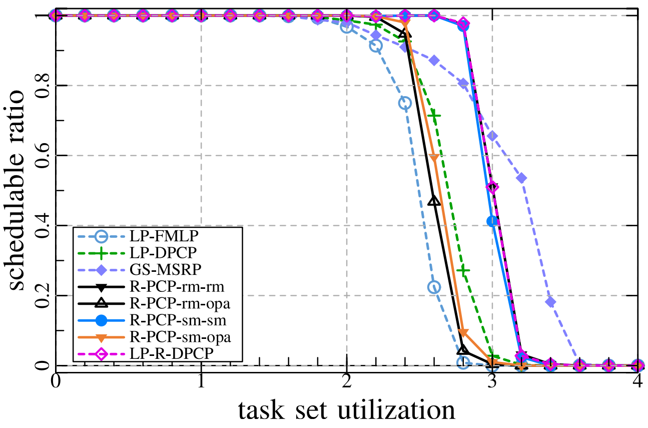


Fig. 2. $m = 4$, homogeneous periods, light utilization, short critical sections, $n_r = 4$, $p^r = 0.25$, and $N = 3$.

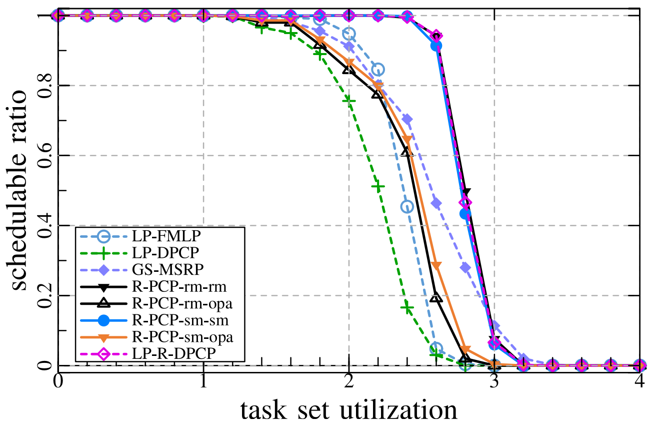


Fig. 3. $m = 4$, heterogeneous periods, light utilization, long critical sections, $n_r = 4$, $p^r = 0.25$, and $N = 3$.

LP-FMLP is higher than that of LP-DPCP). However, LP-DPCP *outperforms*[3] LP-FMLP in 89 percent of tested scenarios (i.e., 683 out of 786 experimental scenarios), and it *dominates*[4] LP-FMLP in 34 percent of tested scenarios. Moreover, our R-PCP-rm-rm outperforms LP-FMLP in 99 percent of scenarios, and it dominates LP-FMLP in 96 percent of scenarios.

Further, the resource-aware partitioning algorithms (e.g., GS-MSRP and R-PCP-rm-rm) considerably improve the resource-oblivious partitioning algorithms with LP-based analysis (e.g., LP-DPCP). For example, R-PCP-rm-rm dominates LP-DPCP in all tested scenarios. That is because although the LP-based analysis can obtain more accurate blocking bounds for a given partitioning, the resource-aware partitioning has the potential to further remove resource-induced delays through reasonable partitioning.

To the best of our knowledge, this is the first work to compare the performances of state-of-the-art locking analyses with resource-aware partitioning algorithms. The results are clear: there is a large gap between good partitioning algorithms and the sophisticated locking analyses when ignoring task partitioning.

ROP Scheduling is Empirically Verified. Prior results in [30] have shown that R-PCP-rm-rm is better than the synchronization-aware partitioning algorithm [33]. We further compare our ROP scheduling with GS-MSRP, which has been shown in [43] to outperform the synchronization-aware partitioning algorithm. Due to the impacts of spin-based locks, the performance of GS-MSRP is highly dependent on the critical section length. GS-MSRP tends to perform better when the critical sections are short, as shown in Fig. 2. Conversely, the performance of GS-MSRP degrades when the critical sections are long, as evident in Figs. 3 and 4. In our experiments, GS-MSRP outperforms R-PCP-rm-rm more often when the critical section length is short, while R-PCP-rm-rm dominates GS-MSRP more often when the critical section length is long. Further, the performance of GS-MSRP decreases earlier compared to R-PCP-rm-rm. For instance, in Fig. 4, the acceptance ratio of GS-MSRP drops at $U = 4$, while that of R-PCP-rm-rm drops until $U = 6$.

The performance of ROP scheduling can be further improved by using more sophisticated blocking analysis, e.g., the LP-based analysis as used for the DPCP [13] (i.e., LP-R-DPCP), at the expense of increased time complexity. In our experiments, the performance of LP-R-DPCP is often marginally better than or identical to R-PCP-rm-rm, as shown in Figs. 2, 3, 4, and 5. However, LP-R-DPCP does not maintain pseudo-polynomial time complexity as R-PCP-rm-rm. Similarly, the GS algorithm with the MSRP can potentially perform better when using the recent LP-based MSRP analysis [44]. However, due to the excessive time complexity, the combination of the GS algorithm and the LP-based MSRP analysis has not been considered in our experiments.

The Modified OPA does not Improve. The seemingly promising priority assignment methods, i.e., the SM and the modified OPA scheme, discussed in Secttion 8 do not necessarily improve the schedulability under ROP scheduling. In the experiments, R-PCP-sm-sm is shown to be comparable with R-PCP-rm-rm. R-PCP-sm-sm outperforms R-PCP-rm-rm in 53 percent of scenarios, and it dominates R-PCP-rm-rm in 19 percent of scenarios, while R-PCP-rm-rm outperforms R-PCP-sm-sm in 49 percent of scenarios and dominates

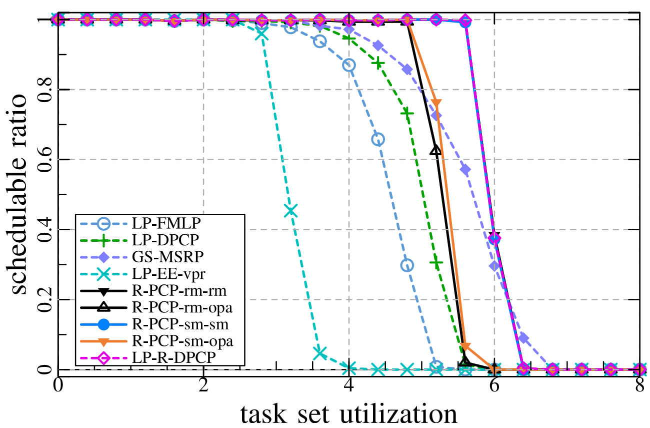


Fig. 4. $m = 8$, homogeneous periods, light utilization, medium critical sections, $n_r = 4$, $p^r = 0.25$, and each job has at most one request.

3. In this section, Algorithm A is said to outperform Algorithm B in an experimental scenario if Algorithm A scheduled more task sets than Algorithm B.

4. In this section, Algorithm A is said to dominate Algorithm B in some experimental scenario if its acceptance ratio is higher than that of Algorithm B at some tested points and never lower than that of Algorithm B at any tested point.

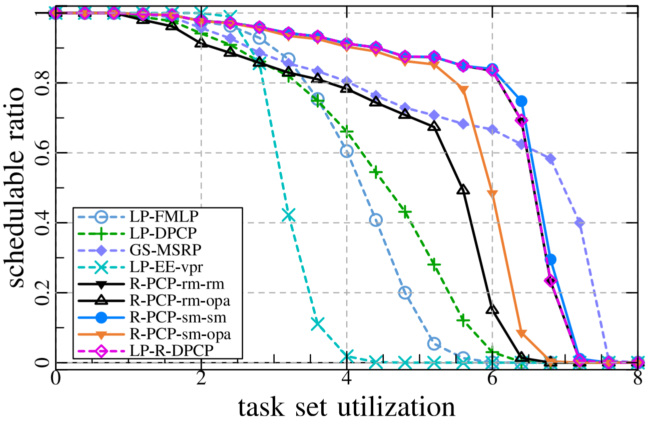


Fig. 5. $m = 8$, heterogeneous periods, medium utilization, long critical sections, $n_r = 8$, $p^r = 0.1$, and each job has at most one request.

R-PCP-sm-sm in 25 percent of scenarios. In contrast, our results show that R-PCP-rm-opa and R-PCP-sm-opa never outperform R-PCP-rm-rm in any scenario.

Given that the modified OPA performs well under GS-MSRP [43], we expect the OPA scheme to perform better under ROP scheduling. We attribute the relative weakness of R-PCP-rm-opa and R-PCP-sm-opa to the OPA-incompatibility of the analysis. Unlike the spin-based MSRP in [43], the priorities of all tasks, not simply the tasks on the same processor, under R-PCP- are related since the priority ceiling mechanism is used. Thus, when the priority of a task changes, the priority ceilings of the resources that it requests may also change. This may further impact the request delays of the tasks on the other processors.

R-PCP-rm-rm Outperforms LP-EE-vpr. LP-EE-vpr and R-PCP-rm-rm are compared in the scenarios that each task contains at most one critical section. In such scenarios, LP-EE-vpr guarantees a speedup factor of 8 if $n_r \leq m$, whereas R-PCP-rm-rm has a speedup factor of at least 9 (i.e., $11 - \frac{6}{2+1}$) on $m \geq 2$ processors. While LP-EE-vpr seems to be better in terms of speedup factor when $n_r \leq m$, the acceptance ratio of LP-EE-vpr drops at low system load even under a necessary schedulability condition. As shown in Fig. 4, the acceptance ratio of LP-EE-vpr drops from U = 2.8 and is zero by U = 4 in a 8-core system, whereas that of R-PCP-rm-rm does not drop until U = 6.

In our experiments, R-PCP-rm-rm outperforms LP-EE-vpr in all scenarios, and it dominates LP-EE-vpr in 73 percent

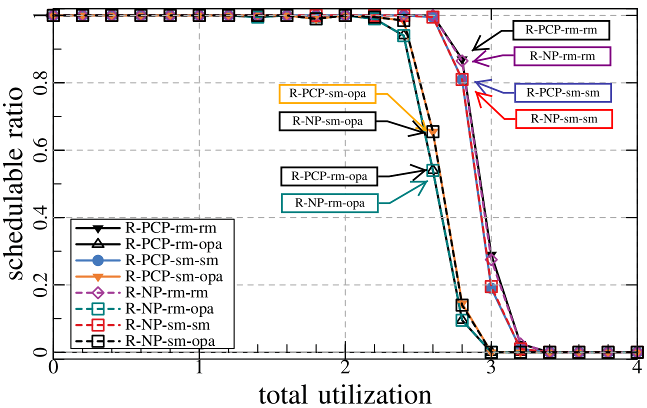


Fig. 6. $m = 4$, homogeneous periods, light utilization, medium critical sections, $n_r = 8$, $p^r = 0.1$, and $N = 5$.

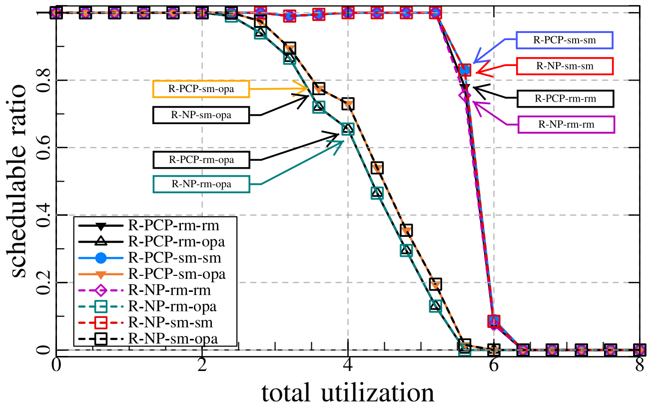


Fig. 7. $m = 8$, homogeneous periods, light utilization, medium critical sections, $n_r = 8$, $p^r = 0.1$, and $N = 5$.

of scenarios. This is mainly because the stringent relative deadlines and slower virtual processors enforced in LP-EE-vpr result in signicant schedulability losses.

R-NP- is in Some Cases Comparable with R-PCP-. From the perspective of schedulability analysis (Eqs. (5) and (6)), the priority ceiling mechanism used in R-PCP-rm-rm has the potential to reduce blocking time compared to the non-preemptive scheduling as used in R-NP-rm-rm. However, the empirical results from the considered experimental setup show that R-NP-rm-rm and R-PCP-rm-rm achieve almost the same performance, as evident in Figs. 6 and 7.

In all tested scenarios, the performance of R-NP-rm-rm is identical to that of R-PCP-rm-rm in 52 percent of scenarios. Similarly, the performance of R-PCP-sm-sm is identical to that of R-NP-sm-sm in 67 percent of scenarios. The performance of R-PCP-sm-opa is identical to that of R-NP-sm-opa in 76 percent of scenarios, and the performance of R-PCP-rm-opa is identical to that of R-NP-rm-opa in 90 percent of scenarios. The performance of the priority ceiling mechanism depends on the ceiling priorities of the resources, which is in turn determined by the tasks-to-resources relationship. While the priority ceiling mechanism achieves a lower bound on the blocking time, the improvement may be insignicant, especially in heavy-contention scenarios. For instance, R-PCP-rm-rm and R-NP-rm-rm perform the same in the analysis if the highest-priority tasks use all the resources.

While the experiments show that both R-NP- and R-PCP- are competitive, such observation is only valid for the considered workload and should not be understood as an absolute ranking. Given that the non-preemptive scheduling incurs less runtime overhead than the priority ceiling mechanism, our results also evidence the potential of the non-preemptive scheduling of shared resources under ROP scheduling.

10 Conclusions

For multiprocessor real-time systems with shared resources, we address the problem of task partitioning and resource sharing for fixed-priority scheduling. A novel ROP scheduling and the associated schedulability analysis are proposed. We prove that the ROP scheduling algorithm R-PCP-rm-rm achieves a non-trivial speedup factor. Large-scale schedulability experiments show that the proposed ROP scheduling is highly effective in terms of task sets deemed schedulable.

It is also noted that while ROP scheduling achieves good performance with regard to worst-case behavior, the intrinsic self-suspension and migration properties lead to additional overhead in average-case behavior, compared to the MSRP and the MPCP, especially when there is minimal contention. Further, the presented analysis does not support nested critical sections. Therefore, intensive studies on improving runtime behavior and supporting nested critical sections remain as future work.

APPENDIX

For the sake of self-containment, we present the schedulability analysis for multiple resource accesses (i.e., a job may request more than one resource and may further request each resource more than once), based on the existing fixed-priority response-time analysis. The current state-of-the-art analysis for partitioned semaphore protocols, including the LP-based analysis for the DPCP, can be found in [13].

Lemma 11. *Suppose that a request $\Re_{k,q}$ is bound to a synchronization processor $\wp_s$, then $\Re_{k,q}$ takes at most $H_{k,q}$ to finish, such that*

$$H_{k,q} = L_{k,q} + b_{k,q} + \sum_{\pi_h > \pi_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{h,v}(H_{k,q}). \quad (31)$$

Proof. Without loss of generality, suppose $\Re_{k,q}$ releases at time t_1 and finishes at time t_2. If $\Re_{k,q}$ is not scheduled at time point $t \in [t_1, t_2]$, then $\Re_{k,q}$ is either (i) blocked by a lower priority request, or (ii) delayed due to higher priority requests at time t. By definition, (i) is upper bounded by $b_{k,q}$. Further, each higher priority task τ_h ($\pi_h > \pi_k$) requires a total of at most $E_{h,v}(t_2 - t_1)$ to execute on resource ℓ_v. Thus, (ii) is upper bounded by $\sum_{\pi_h > \pi_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{h,v}(t_2 - t_1)$. Since $\Re_{k,q}$ finishes at time t_2, it is scheduled for a duration of $t_2 - t_1 - b_{k,q} - \sum_{\pi_h > \pi_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{h,v}(t_2 - t_1)$. By definition, $\Re_{k,q}$ executes at most $L_{k,q}$, thus

$$t_2 - t_1 - b_{k,q} - \sum_{\pi_h > \pi_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{h,v}(t_2 - t_1) \leq L_{k,q}.$$

Maximizing $t_2 - t_1$ and by reformulation

$$H_{k,q} = L_{k,q} + b_{k,q} + \sum_{\pi_h > \pi_k, R_v \in \mathcal{RS}(\wp_s)} E_{h,v}(H_{k,q}).$$

□

Eq. (31) can be solved according to the classic iterative techniques [6]. Next, we bound the cumulative requset delays.

Definition 4. *Let $\Theta_k^s(t)$ be the upper bound on the time that a job of τ_k spends to finish all its requests on processor $\wp_s$ in a time interval of length t.*

According to Definition 4, $\Theta_k^s(t) = 0$ if task τ_k does not request any resource on processor $\wp_s$. Otherwise, $\Theta_k^s(t)$ depends on the requests that are bound to processor $\wp_s$.

Lemma 12. $\Theta_k^s(t) = \min(\lambda_k^s, \mu_k^s(t))$, *wherein*

$$\lambda_k^s = \sum_{\ell_q \in \mathcal{RS}(\wp_s) \cap \mathcal{RS}(\tau_k)} N_{k,q} \cdot H_{k,q}, \quad (32)$$

$$\mu_k^s(t) = \sum_{\ell_q \in \mathcal{RS}(\wp_s)} A_{k,q} + \sum_{\tau_j \neq \tau_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{j,v}(t). \quad (33)$$

Proof. By Lemma 11, it takes at most $H_{k,q}$ to finish a request of task τ_k for resource ℓ_q. By definition, a job J_k issues at most $N_{k,q}$ requests to resource ℓ_q. Thus, it takes a total of at most λ_k^s (as shown in Eq. (32)) to finish all the requests of τ_k to the resources bound to processor $\wp_s$. According to Definition 4, $\Theta_k^s(t)$ is then upper bounded by λ_k^s.

At any point in time while a request of τ_k is released but not finished, either (i) the request of τ_k is executing, or (ii) another request on the same processor is executing. For (i), a job J_k executes on the resources bound to processor $\wp_s$ for at most $\sum_{\ell_q \in \mathcal{RS}(\wp_s)} A_{k,q}$. For (ii), any other task τ_j may execute on resource ℓ_v for a time of at most $E_{j,v}(t)$. Thus, all requests from all other tasks can execute for a total of at most $\sum_{\tau_j \neq \tau_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{j,v}(t)$ on processor $\wp_s$. Summing up both terms, $\Theta_k^s(t)$ is then upper bounded by $\mu_k^s(t)$.

Therefore, $\Theta_k^s(t)$ can be upper bounded by the minimum of λ_k^s and $\mu_k^s(t)$. □

With the requests delays being upper bounded, the worst-case response time analysis can be derived as follows.

Theorem 3. *If task τ_k is assigned on an application processor $\wp_a$, then the smallest t satisfying*

$$C_k + \sum_{\tau_h \in \tau(\wp_a), \pi_h > \pi_k} W_h(t) + \sum_{\wp_c \neq \wp_a} \Theta_k^c(t) \leq t. \quad (34)$$

is a safe upper bound on the response time of task τ_k, given that $t \leq T_k$.

Proof. Proof by contradiction. Suppose there is a $t \leq T_k$ such that inequality (34) holds but there is a job J_k that does not finish within a time interval of length t.

Since τ_k is assigned to an application processor $\wp_a$, at any point in time while a job of τ_k is released but not finished, it is either (i) ready to execute non-critical sections but not scheduled, (ii) waiting a request to finish on another processor $\wp_c$ ($c \neq a$), or (iii) executing non-critical sections on processor $\wp_a$.

Since non-critical sections are scheduled according to base priority, (i) takes place only if a higher-priority task is executing non-critical sections. Thus, by Lemma 4, the cumulative delay of (i) is upper bounded by $\sum_{\tau_h \in \tau(\wp_a), \pi_h > \pi_k} W_h(t)$. By Definition 4, the time that a job of J_k spends to finish its requests on processor $\wp_c$ in a time interval of length t is upper bounded by $\Theta_k^c(t)$. Thus, the cumulative time of case (ii) is upper bounded by $\sum_{\wp_c \neq \wp_a} \Theta_k^c(t)$.

By hypothesis, a job of τ_k does not finish within a time interval of length t. It must be the case that, the j ob executes its non-critical sections on application processor $\wp_a$ for more than $t - \sum_{\tau_h \in \tau(\wp_a), \pi_h > \pi_k} W_h(t) - \sum_{\wp_c \neq \wp_a} \Theta_k^c(t)$. Thus

$$C_k > t - \sum_{\tau_h \in \tau(\wp_a), \pi_h > \pi_k} W_h(t) - \sum_{\wp_c \neq \wp_a} \Theta_k^c(t).$$

Therefore, inequality (34) does not hold. Contradiction. □

Analogously, an upper bound on the response time of a task assigned on a synchronization processor can be derived according to the following theorem.

Theorem 4. *If task τ_k is assigned on a synchronization processor $\wp_s$, then the smallest t satisfying*

$$\begin{aligned} C_k + \sum_{\ell_q \in \mathcal{RS}(\wp_s)} A_{k,q} + \sum_{\tau_h \in \tau(\wp_s), \pi_h > \pi_k} W_h(t) \\ + \sum_{\tau_j \neq \tau_k, \ell_v \in \mathcal{RS}(\wp_s)} E_{j,v}(t) + \sum_{\wp_c \neq \wp_s} \Theta_k^c \leq t. \end{aligned} \tag{35}$$

is a safe upper bound on the response time of task τ_k, given that $t \leq T_k$.

ACKNOWLEDGMENTS

We would like to thank Zewei Chen for conducting the experiments on SET-MARTS, and we appreciate the anonymous reviewers for their valuable comments and suggestions, which significantly improved the quality of this paper. Work supported by the National Natural Science Fundation of China (Grant No. 61802052), the China Postdoctoral Science Fundation Funded Project (Grant No. 2017M612947), and DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/).

REFERENCES

[1] S. Afshar, F. Nemati, and T. Nolte, "Resource sharing under multiprocessor semi-partitioned scheduling," in *Proc. Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2012, pp. 290–299.

[2] B. Andersson, "Global static-priority preemptive multiprocessor scheduling with utilization bound 38%," in *Proc. Int. Conf. Principles Distrib. Syst.*, 2008, pp. 73–88.

[3] B. Andersson, "The utilization bound of uniprocessor preemptive slack-monotonic scheduling is 50%," in *Proc. ACM Symp. Appl. Comput.*, 2008, pp. 281–283.

[4] B. Andersson and A. Easwaran, "Provably good multiprocessor scheduling with resource sharing," *Real-Time Syst.*, vol. 46, no. 2, pp. 153–159, 2010.

[5] B. Andersson and G. Raravi, "Real-time scheduling with resource sharing on heterogeneous multiprocessors," *Real-Time Syst.*, vol. 50, no. 2, pp. 270–314, 2014.

[6] N. C. Audsley, "On priority assignment in fixed priority scheduling," *Inf. Process Lett.*, vol. 79, no. 1, pp. 39–44, 2001.

[7] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, 1991.

[8] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proc. Real-Time Syst. Symp.*, 1990, pp. 182–190.

[9] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "Is semi-partitioned scheduling practical?" in *Proc. Euromicro Conf. Real-Time Syst.*, 2011, pp. 125–135.

[10] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen, "Errata for three papers (2004–05) on fixed-priority scheduling with self-suspensions," CISTER Res. Centre, Porto, Portugal, Tech. Rep. CISTER-TR-150713, Jul. 2015.

[11] 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.

[12] 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.

[13] B. B. Brandenburg, "Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2013, pp. 141–152.

[14] B. B. Brandenburg, "Blocking optimality in distributed real-time locking protocols," *Leibniz Trans. Embedded Syst.*, vol. 1, no. 2, pp. 01:1–01:22, 2014.

[15] B. B. Brandenburg, "The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis," in *Proc. Euromicro Conf. Real-Time Syst.*, 2014, pp. 61–71.

[16] B. B. Brandenburg and J. H. Anderson, "Optimality results for multiprocessor real-time locking," in *Proc. Real-Time Syst. Symp.*, 2010, pp. 49–60.

[17] B. B. Brandenburg and M. Gul, "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," in *Proc. Real-Time Syst. Symp.*, 2016, pp. 99–110.

[18] A. Burns and A. J. Wellings, "A schedulability compatible multiprocessor resource sharing protocol - MrsP," in *Proc. Euromicro Conf. Real-Time Syst.*, 2013, pp. 282–291.

[19] J.-J. Chen, G. Nelissen, and W.-H. Huang, "A unifying response time analysis framework for dynamic self-suspending tasks," in *Proc. Euromicro Conf. Real-Time Syst.*, 2016, pp. 61–71.

[20] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, Neil, Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," 2nd version, Faculty of Informatik, TU Dortmund, Dortmund, Germany, Tech. Rep. 854, 2017. [Online]. Available: http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2017-chen-techreport-854-v2.pdf

[21] J.-J. Chen, G. von der Brüggen, W.-H. Huang, and R. I. Davis, "On the pitfalls of resource augmentation factors and utilization bounds in real-time scheduling," in *Proc. 29th Euromicro Conf. Real-Time Syst.*, 2017, pp. 9:1–9:25.

[22] R. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," *Real-Time Syst.*, vol. 47, pp. 1–40, 2010.

[23] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surveys*, vol. 43, no. 4, 2011, Art. no. 35.

[24] D. Faggioli, G. Lipari, and T. Cucinotta, "The multiprocessor bandwidth inheritance protocol," in *Proc. Euromicro Conf. Real-Time Syst.*, 2010, pp. 90–99.

[25] 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.

[26] G. Han, H. Zeng, M. D. Natale, X. Liu, and W. Dou, "Experimental evaluation and selection of data consistency mechanisms for hard real-time applications on multicore platforms," *IEEE Trans. Ind. Informat.*, vol. 10, no. 2, pp. 903–918, May 2014.

[27] J. Han, X. Tao, D. Zhu, and L. T. Yang, "Resource sharing in multicore mixed-criticality systems: Utilization bound and blocking overhead," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3626–3641, Dec. 2017.

[28] J.-J. Han, D. Zhu, X. Wu, L. T. Yang, and H. Jin, "Multiprocessor real-time systems with shared resources: Utilization bound and mapping," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 11, pp. 2981–2991, Nov. 2014.

[29] P. Hsiu, D. Lee, and T. Kuo, "Task synchronization and allocation for many-core real-time systems," in *Proc. Int. Conf. Embedded Softw.*, 2011, pp. 79–88.

[30] W.-H. Huang, M. Yang, and J.-J. Chen, "Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share?" in *Proc. Real-Time Syst. Symp.*, 2016, pp. 111–122.

[31] W.-H. Hung, J.-J. Chen, H. Zhou, and C. Liu, "PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling," in *Proc. 52nd Annu. Des. Autom. Conf.*, 2015, Art. no. 154.

[32] B. Kalyanasundaram and K. Pruhs, "Speed is as powerful as clairvoyance," *J. ACM*, vol. 47, no. 4, pp. 617–643, 2000.

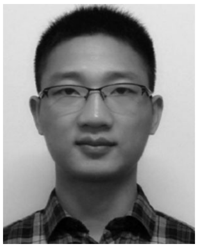
[33] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proc. Real-Time Syst. Symp.*, 2009, pp. 469–478.

[34] C. Liu and J. Chen, "Bursty-interference analysis techniques for analyzing complex real-time task models," in *Proc. Real-Time Syst. Symp.*, 2014, pp. 173–183.

[35] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[36] F. Nemati, M. Behnam, and T. Nolte, "Independently-developed real-time systems on multi-cores with shared resources," in *Proc. Euromicro Conf. Real-Time Syst.*, 2011, pp. 251–261.

[37] F. Nemati, T. Nolte, and M. Behnam, "Partitioning real-time systems on multiprocessors with shared resources," in *Proc. Int. Conf. Principles Distrib. Syst.*, 2010, pp. 253–269.
[38] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 1990, pp. 116–123.
[39] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proc. Real-Time Syst. Symp.*, 1988, pp. 259–269.
[40] 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.
[41] B. C. Ward and J. H. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *Proc. Euromicro Conf. Real-Time Syst*., 2012, pp. 223–232.
[42] B. C. Ward and J. H. Anderson, "Fine-grained multiprocessor real-time locking with improved blocking," in *Proc. Int. Conf. Real-Time Netw. Syst.*, 2013, pp. 67–76.
[43] A. Wieder and B. B. Brandenburg, "Efficient partitioning of sporadic real-time tasks with shared resources and spin locks," in *Proc. Int. Symp. Ind. Embedded Syst.*, 2013, pp. 49–58.
[44] 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.
[45] M. Yang, J.-J. Chen, and W.-H. Huang, "A misconception in blocking time analyses under multiprocessor synchronization protocols," *Real-Time Syst.*, vol. 53, no. 2, pp. 187–195, 2017.
[46] M. Yang, A. Wieder, and B. B. Brandenburg, "Global real-time semaphore protocols: A survey, unified analysis, and comparison," in *Proc. Real-Time Syst. Symp.*, 2015, pp. 1–12.



Maolin Yang received the MS degree in software engineering from Zhejiang University, China, in 2011, and the PhD degree in software engineering from the University of Electronic Science and Technology of China (UESTC), China, in 2016. He is a postdoctor with the School of Information and Software Engineering, University of Electronic Science and Technology of China (UESTC), China. He has been a visiting scholar with the Max-Planck Institute for Sortware Systems (MPI-SWS), Germany, during Oct. 2014 and May 2015, and with the Department of Informatics, Technical University of Dortmund, Germany, during Jun. 2015 and Jan. 2016, respectively. His research interests include real-time scheduling algorithms, real-time locking protocols, and cyber-physical systems. He is a member of the IEEE.

Wen-Hung Huang received the bachelor's degree in computer science and the master's degree in electrical engineering from the National Cheng Kung University, in 2009 and 2011. He is working toward the PhD degree in computer science at the Technical University of Dortmund, Germany. His research interests include the areas of real-time systems, timing analysis, and embedded and distributed systems. He is a member of the IEEE.

Jian-Jia Chen received the BS degree from the Department of Chemistry, National Taiwan University, in 2001, and the PhD degree from the Department of Computer Science and Information Engineering, National Taiwan University, in 2006. He is a professor with the Department of Informatics, TU Dortmund University, Germany. He was junior professor with the Department of Informatics, Karlsruhe Institute of Technology, Germany from May 2010 to Mar. 2014. Between Jan. 2008 and Apr. 2010, he was a postdoc researcher with ETH Zurich, Switzerland. His research interests include real-time systems, embedded systems, energy-efficient scheduling, power-aware designs, temperature-aware scheduling, and distributed computing. He received Best Paper Awards from International Conference on Hardware/Software Codesign and System Synthesis (CODES +ISSS) 2014, IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) in 2005, 2013 and 2015, and ACM Symposium on Applied Computing (SAC) in 2009. He has been involved in Technical Committees in many international conferences. He is a member of the IEEE.