Resource-Oriented Partitioned Scheduling in Multiprocessor Systems: How to Partition and How to Share?

Wen-Hung Huang[1], Maolin Yang[2], and Jian-Jia Chen[1]
[1]Department of Computer Science, Technical University of Dortmund, Germany
[2]University of Electronic Science and Technology of China, China

ABSTRACT

When concurrent real-time tasks have to access shared resources, to prevent *race conditions*, the synchronization and resource access must ensure *mutual exclusion*, e.g., by using semaphores. That is, no two concurrent accesses to one shared resource are in their critical sections at the same time. For uniprocessor systems, the priority ceiling protocol (PCP) has been widely accepted and supported in real-time operating systems. However, it is still arguable whether there exists a preferable approach for resource sharing in multiprocessor systems. In this paper, we show that the proposed resource-oriented partitioned scheduling using PCP combined with a reasonable allocation algorithm can achieve a non-trivial speedup factor guarantee. Specifically, we prove that our task mapping and resource allocation algorithm has a speedup factor $11-6/(m+1)$ on a platform comprising m processors, where a task may request at most one shared resource and the number of requests on any resource by any single job is at most one. Our empirical investigations show that the proposed algorithm is highly effective in terms of task sets deemed schedulable.

1. INTRODUCTION

Real-time systems are designed for applications in which the response time is critical. In a real-time system with multi-tasking, a mechanism for tasks to communicate with each other or to ensure their synchronization must be furnished. In uniprocessor systems, locking protocols based on *priority inheritance*, such as the priority ceiling protocol (PCP) [28], the priority inheritance protocol (PIP) [28], and the stack resource policy (SRP) [3], have been shown reasonably good at managing synchronization and mutual exclusions.

To schedule real-time tasks on multiprocessor platforms, there have been three widely adopted paradigms: partitioned, global, and semi-partitioned scheduling. The partitioned scheduling approach partitions the tasks statically among the available processors. That is, each task is statically assigned to a processor to execute all its jobs on the processor. The global scheduling approach allows a job to be migrated from one processor to another. The semi-partitioned scheduling approach decides whether a task is divided into subtasks statically and each task/subtask is then assigned to a processor statically. A comprehensive survey of multiprocessor scheduling in real-time systems can be found in [18].

In addition to the above task scheduling paradigms, resource sharing and synchronization should also be considered for multiprocessor systems. Therefore, concerns among real-time system designs on multiprocessor platforms are not only resource sharing between the real-time tasks but also task-to-processor mapping. If there is no need for task synchronization, partitioned scheduling has low run-time overhead, although the performance (in terms of schedulability) can be worse than global scheduling. However, in need of synchronization, (*i*) *is partitioned scheduling still a good option? And,* (ii) *how should we derive good task partitions?*

The first question has been partially answered by a few researches in the literature. For example, Rajkumar [28] proposed Multiprocessor Priority Ceiling Protocol (MPCP) (based on suspension locks) and Burns and Wellings [14] proposed Multiprocessor resource sharing Protocol (MrsP) (with spin locks). However, it is known that the number of priority-inversion blockings (*pi*-blockings) can be lower bounded by the number of processors of the multiprocessor system in the worst case, for specific task partitions, as shown in [13]. That is, the elegance of partitioned scheduling to run a task all the time on one processor suffers from the synchronization between the real-time tasks if the tasks are not partitioned well.

Essentially, the performance of resource sharing protocols highly depends on how the tasks are partitioned. Therefore, the second question has been explored in [23, 25, 27, 33]. However, these results are heuristic algorithms, and there has been no algorithmic analysis provided in the literature to necessitate these approaches from the perspective of resource augmentation (or speedup) factors. Moreover, there is no clear evidence showing whether the above synchronization strategies and task partitioning approaches should be strictly designed to follow the traditional partitioned scheduling paradigm.

Resource-oriented partitioned scheduling: Alternatively, we can consider a **new** partitioned scheduling policy for the tasks that need synchronizations. The literature of real-time systems has been very bias towards the *computing tasks*. However, since the resources are the bottlenecks, it is sensible to change our view angle to focus on the shared resources. To this end, we can define the following *resource-oriented partitioned scheduling* policy used in this paper:

- Each shared resource should be assigned on one processor, and the critical sections guarded by this shared resource are executed on the designated processor.
- The non-critical sections of a task are executed on its designated processor (that can be different from the processors executing the critical sections of the task).

The spirit of resource-oriented partitioned scheduling is to position the resource accesses as the first-class citizens in the system when we consider task and resource partitioning so as to keep the response time of critical sections as short as possible. Note that, in general, we still follow the partitioned scheduling policy for the non-critical sections, but the critical sections are proposed to be executed on the designated synchronization processors. A similar concept to execute the critical sections guarded by one shared resource

Work supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/).

on a designated processor can also be found in Distributed Priority Ceiling Protocols (DPCP) developed by Rajkumar et al. [29]. But, there has been no further elaboration in the literature to provide evidence on how to assign tasks and resources among multiple processors.

The additional overheads in resource-oriented partitioned scheduling are just identical to semi-partitioned scheduling (which is less than global scheduling) due to the critical-section migration, since we only specifically migrate critical sections. The migration can be managed directly from the operating systems, in which a request to lock a shared resource can be handled directly (by the resource manager residing) on the designated processor. We will discuss this further in Section 3.2. The resource-oriented partitioned scheduling has more overhead than the original partitioned scheduling since the critical sections have to be executed on the remote processors. Even though this may seem to be a drawback, we should also keep in mind the implementations of MPCP and MrsP also require an arbiter to decide which critical section is granted to be executed, which is also not free.

In this paper, we provide fundamental explorations of the resource-oriented partitioned scheduling for *fixed-priority* scheduling, in which each task has a unique priority level. We show that the resource-oriented partitioned scheduling policy has good potential to maintain the low-overhead of partitioned scheduling (if there are only a few critical sections per task) and good acceptance ratios of schedulability. To the best of our knowledge, this is the first work studying this seemingly insurmountable problem involving task partitioning and the design of synchronization protocols at the same time.

Our technical contributions: Our contributions are:

- In Section 4, we first provide a schedulability test for a given resource-oriented task partition. Combined with the proposed schedulability test, we propose an algorithm for deriving resource-oriented task partitioning in Section 5.
- In Section 6, we show that our algorithm using PCP (after resource and task partitioning) can achieve a speedup factor $11 - 6/(m + 1)$ if a task may request at most one shared resource and each job of a task may request the resource at most once during its execution, where $m \geq 2$ is the number of processors. To the best of our knowledge, this is the best result studying the problem of resource sharing on multiprocessors in terms of non-optimality when partitioned scheduling schemes are considered. Earlier results by Andersson and Easwaran [1] achieved a speedup factor $12(1 + 3r/4m)$ by using G-EDF and virtualization under the same resource access model, where r is the number of shared resources. We note that the above speedup factor may not be small enough to have very practical implications, but the understanding of this constant speedup factor with simple scheduling algorithms implies the potential of resource-oriented partitioned algorithms.
- The effectiveness of our proposed algorithm is highly supported by the evaluation results in Section 8. Empirical results show that our algorithm along with either PCP or Non-preemptive protocol (NPP) for task sets with utilization below 50% is nearly optimal in the sense that task sets not deemed schedulable by our algorithm are also not schedulable using any scheduling policy.
- Further, we observe that while the additional overhead is incurred by the implementation of PCP in each single processor, the improvement over the simple non-preemptive scheduling by using PCP is very little: the long blocking incurred by the non-preemptive scheduling can be painlessly removed by finding a good task partitioning. Therefore, this enables us to use the simple non-preemptive scheduling for resource sharing on multiprocessor systems while keeping high schedulability.

2. SYSTEM MODEL

2.1 System and Task Model

We assume in this paper that we have a multiprocessor platform comprised of $m \geq 2$ identical multiprocessors $\wp = \{\wp_1, \wp_2, .., \wp_m\}$. A real-time system with shared resources is specified using r shared resources $\mathcal{RS} = (R_1, R_2, ..., R_r)$ and n sporadic task $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$.

Task model. Each sporadic task is characterized as $\tau_i = (C_i, A_i, T_i, D_i)$, where C_i the upper bound on the amount of *non-critical-section* execution time; A_i the upper bound on the amount of *critical-section* execution time; T_i denotes the minimum inter-arrival time; and D_i the relative deadline. We note that the total worst-case execution time (WCET) of task τ_i, including critical-sections and non-critical-sections, is equal to $WCET_i = C_i + A_i$. Each job of task τ_i requires $WCET_i$ units of processing capacity within D_i time units from its release, and this processing capacity must be supplied sequentially, i.e., the job cannot be scheduled on more than one processor at any given time instant. Further, any two successive jobs of this task must be released at least T_i time units apart.

Shared resources. A shared resource can be in-memory data, such as a set of variables, and external objects, such as files, database connections, and network connections. To prevent *race conditions*, resources shared between sets of tasks must be accessed under *mutual exclusion*: no two concurrent accesses to one shared resource are in their critical sections at the same time. We note that in this work we focus ourselves on *logical* shared resources: a piece of code to be executed on processors. Hence, no shared resources are processor-specific.

The jobs of any task can issue requests for exclusive access to shared resources $R_1, R_2, ..., R_r$. A job of task τ_i could request resource R_q on multiple occasions during its execution, and we denote as $N_{i,q} \leq N$ the maximum number of such requests by any single job of τ_i where N is an integer. Associated with these requests is the worst-case duration of time for which a job uses resource R_q. We do not put any assumption on the access patterns of resource requests, dependent on different execution paths with different execution times. Resource requests cannot be *nested*. We denote by $V_{i,q}$ the maximum (worst-case) resource usage time among all requests for resource R_q by jobs of τ_i. We denote by $A_{i,q}$ an upper bound on the total resource usage time for resource R_q by any single job of τ_i (sum of resource usage times over all requests for resource R_q). Clearly, $A_{i,q}/N_{i,q} \leq V_{i,q}$. Further, we denote by $\mathcal{RS}_i \subseteq \{R_1, R_2, ..., R_r\}$ the set of all resources accessed by jobs of τ_i, and the cardinality of this set is at most Q, i.e. $|\mathcal{RS}_i| \leq Q \leq r$.

We denote the total resource usage time of task τ_i as $A_i = \sum_{R_q \in \mathcal{RS}_i} A_{i,q}$. We assume that $C_i + A_i \leq D_i$ for any task $\tau_i \in \tau$. The utilization of resource R_q from task τ_i is denoted by $U_i^{R_q} = A_{i,q}/T_i$. The total resource utilization of task τ_i is denoted by $U_i^A = A_i/T_i$. We denote the utilization of task τ_i as $U_i = (C_i + A_i)/T_i$. The utilization of task τ_i with non-resource execution is defined as $U_i^C = C_i/T_i$. We further assume that $U_\Sigma = \sum_{i=1}^{n} U_i \leq m$. Otherwise, it cannot

be feasibly scheduled. The total utilization of resource R_q is denoted by $U^{R_q} = \sum_{\tau_i \in \tau} U_i^{R_q}$. The total utilization of non-critical-section is denoted by $U^C = \sum_{\tau_i \in \tau} U_i^C$; and the total utilization of shared resources is denoted by $U^{\mathcal{RS}} = \sum_{R_q \in \mathcal{RS}} U^{R_q}$.

Task system τ is said to be an *implicit-deadline* system if $D_i = T_i$ holds for each $\tau_i \in \tau$, and a *constrained-deadline* system if $D_i \leq T_i$ holds for each $\tau_i \in \tau$; otherwise, an *arbitrary-deadline* system.

In this paper, we focus on preemptive fixed-priority scheduling, in which each task τ_i is associated with a unique priority level, called *base priority* $\pi(\tau_i)$. In this paper, $\pi(\tau_i) > \pi(\tau_j)$ if task τ_i has a higher base priority than task τ_j. We restrict our attention to *implicit-deadline* task systems. A system τ is said to be *feasible* if there exists a scheduling algorithm that can schedule the system without any deadlines being missed. A *schedulability test* of a scheduling algorithm is to verify whether the task system is feasible under the given algorithm.

2.2 Resource-Oriented Partitioned Scheduling

As already defined in Section 1, we will adopt the resource-oriented partitioned scheduling in this paper: The resource-oriented task partitioned scheduling has the following characteristics:

- All the critical sections associated with the same resource must be bound to the same processor, called *synchronization* processor. It implies that access to shared resources must execute on the designated *synchronization* processor.
- Tasks are statically allocated onto processors for executing their non-critical sections. All *non-critical-section* codes generated by a task only execute on the processor on which the task is assigned, called *application* processor. Each time a job enters a critical section, unlike spin locks, it suspends itself on its application processor, yielding this processor to other tasks, and executes critical section codes on the synchronization processor on which the corresponding shared resource(s) is bound.
- A processor may still execute both critical and non-critical sections, depending on our resource and task partitioning algorithm, to be explained in Section 5.

Specifically, in this paper, we will explore

- how *many* processors should be designed as synchronization processors,
- how to *partition the shared resources* onto synchronization processors under the resource-oriented partitioned scheduling,
- how to *partition the sporadic real-time tasks* for their non-critical sections in application processors, and
- how to *assign the base priorities* of the sporadic tasks.

To this end, in Section 3 we will discuss how to handle the competition of shared resource access, and provide schedulability analyses in Section 4 and Section 7 to ensure the schedulability of the tasks under given resource-oriented partitioned scheduling and base priority assignments. Then, we will develop our resource-oriented partitioned scheduling algorithm in Section 5, whose non-optimality will be quantified in Section 6.

For the simplicity of presentation, we will assume that each task may request at most one shared resource, and further each job of that task may request the resource at most once during its execution, that is, $Q = 1$ and $N = 1$ (through Section 4 to Section 6). In Section 7, we will relax this assumption to include multiple resource accesses during a job's execution and multiple occasions on one resource request. To the best of our knowledge, even under this restrictive assumption, the problem of scheduling tasks with resource sharing on multiprocessors under partitioned scheduling is still an open question - no preferable scheme is known.

2.3 Speedup Factors

Ideally, an exact 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 formally defined as follows:

Definition 1 (Processor speedup factor). *A schedulability test has a processor 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.*

The concept of demand bound function (DBF) has been widely used in real-time schedulability analysis. The demand bound function (DBF) $dbf_i(t)$ bounds the maximum cumulative execution requirement by jobs of a sporadic task τ_i that both arrive in and have absolute deadlines within any interval of length t [4]. The demand bound function of task τ_i with an interval of length t is

$$dbf_i(t) = \max\left(0, (\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1) \times (A_i + C_i)\right) \quad (1)$$

Specifically, the demand bound functions of task τ_i for non-critical-section execution and for accesses to resource R_q, with an interval of length t are

$$dbf_i^C(t) = \max\left(0, (\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1) \times C_i\right) \quad (2)$$

and

$$dbf_i^{R_q}(t) = \max\left(0, (\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1) \times A_{i,q}\right) \quad (3)$$

3. RESOURCE SHARING PROTOCOLS

3.1 Single Processor Systems

To achieve mutual exclusion, the shared resources subject to mutual exclusion constraints must be serially executed. This inevitably causes some delays, namely *priority inversion*: a task is prevented from executing due to another lower-base-priority task with a current higher effective priority (e.g., when holding a resource).

NPP vs PCP. The non-preemptive protocol (NPP) is characterized by the fact that once a critical section has started to execute, it cannot be preempted until it finishes the section. This has several advantages:

- The implementation of a non-preemptive scheduling is simpler because the scheduler is inactive during the execution of a non-preemptiable section.
- A set of tasks may need to share resources that must be accessed under *mutual exclusion*. It implies that once a job enters a critical section, it cannot be preempted by any access to this critical section until it finishes the critical section. This condition is automatically satisfied under non-preemptive scheduling.

Let $\pi(\tau_i)$ denote the base priority of task τ_i. It is shown in [15] that under NPP a task may incur blocking due to

any lower-priority task that accesses resources. Under the NPP, the blocking for a task τ_k being analyzed is

$$B_k = \max_{i,q}\{V_{i,q} | \pi(\tau_i) < \pi(\tau_k)\} \quad (4)$$

Despite some advantages, NPP falls short of avoiding some unnecessary blocking incurred by lower-priority tasks [15]. The priority ceiling protocol (PCP) has thus been introduced by Sha, et. al [30] to remove the unnecessary blocking while preventing deadlocks and chained blocking. The idea underlying PCP is that the system maintains a system ceiling and disallows any other jobs with lower priorities than the system ceiling to access any shared resources. The blocking has been shown in [30] to be:

$$B_k = \max_{i,q}\{V_{i,q} | \pi(\tau_i) < \pi(\tau_k), \mathcal{C}(R_q) \geq \pi(\tau_k)\} \quad (5)$$

where $\mathcal{C}(R_q)$ is the *ceiling* priority of shared resource R_q under PCP.

In the following theorem, we show that PCP associated with the RTA test by [2] under RM scheduling offers non-trivial quantitative guarantees:

THEOREM 1. *The speedup factor of the priority ceiling protocol (PCP) associated with the RTA by [2] under RM scheduling in uniprocessor systems is* 2.

PROOF. The proof is in Appendix. □

3.2 Multiprocessor Systems

It is known that $\Omega(m)$ pi-blocking is unavoidable on multiprocessor systems, provided that any task partition is given, as shown in [13]. Therefore, as can be seen in the literature [14, 25, 28] in which task partitioning is given, a higher-priority job under MPCP or MrsP may suffer from the so-called *chained blocking*: a higher-priority job can be blocked for the duration of either m, for MrsP, or n, for MPCP, critical sections. The chained blocking has significant influence on the schedulability analyses.

A clear advantage of resource-oriented scheduling by bounding all accesses to each shared resource to the same processor is that it avoids chained blocking: each access to a shared resource of a task can be only blocked by those low-priority accesses that are assigned on the same processor. Therefore, higher schedulability is expected. This is also empirically confirmed in our experiments (see Section 8). As mentioned in Section 1, a similar concept can also be found in Distributed Priority Ceiling Protocols (DPCP) [29]. But, there has been no further elaboration in the literature to provide evidence on how to assign tasks and resources among multiple processors.

In addition, based on resource-oriented partitioned scheduling, we actually break down the problem of scheduling tasks with shared resources on multiprocessors into smaller uniprocessor sub-problems, on which standard consolidated techniques can still be applied (on each synchronization processor), e.g., NPP and PCP in uniprocessor systems, denoted by R-NPP and the R-PCP for the rest of this paper, respectively. Moreover, unlike uniprocessor systems, the unnecessary blocking incurred by the NPP (compared to PCP) could be removed by partitioning tasks properly on multiprocessor systems. This might in turns enable us to use the R-NPP due to its simplicity of implementations, where maintaining a list of currently locked semaphores and priority ceiling orders during runtime are not needed.

Compared to traditional partitioned scheduling, resource-oriented partitioned scheduling has additional overheads. In our approach, the execution of a task in the resource-oriented scheduling is split into more than one processor, similar to *semi-partitioned* scheduling. From the implementation's point of view, the resource-oriented partitioned scheduling can 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 [5]). This gives us several advantages :

- As which critical-section executions will migrate and also among which processors are known beforehand, cache-related preemption and migration delay (CPMD) accounting is task-specific and hence less pessimistic.
- Furthermore, since push-migrations can be implemented with mostly-local state, migrations of the resource-oriented scheduling entail less overhead and are easier to implement.

However, for example, MrsP requires a lock holder to progress within the critical section of a task waiting for a resource already locked by a preempted task executing on a different processor. Therefore, the migration of jobs across partitions must be furnished, and the next processor is dynamically determined at runtime, the so-called *pull-migrations*. Such migrations imply much higher overheads than push-migrations at runtime.

Inspired by these, in this work we aim at obtaining a better understanding of this seemingly promising scheduling along with task and resource partitioning.

4. SCHEDULABILITY ANALYSIS

In this section, we present the response time analysis and the schedulability test for a specific task τ_k. We assume that the priority assignment for fixed-priority preemptive scheduling is already specified and the tasks are already mapped onto the processors. Note that as mentioned in Section 3, we will focus on the case that $N = 1$ and $Q = 1$ in this section. In Section 4.1, we begin with a simpler case in which the synchronization processors are used only for executing critical sections and application processors are used only for executing non-critical sections. We will consider running non-critical sections on some synchronization processors in Section 4.2.

We implicitly assume that the higher-priority tasks already meet their deadlines while analyzing task τ_k. Since we use partitioned scheduling for the non-critical sections, we define $\mathbf{hpl}(k)$ as the set of the higher-priority tasks that are assigned on the same processor where task τ_k is assigned to run its non-critical sections. Suppose that a job of task τ_k arrives at time t_0 and has an absolute deadline at time $t_0 + D_k$. To analyze whether we can finish the job in time, we need to analyze the interference from the higher-priority tasks $\mathbf{hpl}(k)$ on the *local* application processor and that due to resource accesses on the *remote* synchronization processors in the time interval $[t_0, t_0 + D_k)$. Without loss of generality, we can set t_0 to 0.

4.1 Response Time Analysis

Here, in this subsection, suppose that the synchronization processors are used only for executing critical sections and application processors are used only for executing non-critical sections. Under resource-oriented partitioned scheduling, every time a task requests a shared resource mapped onto a remote processor, the task suspends itself on the local processor until the request is complete. With the suspension-based scheduling, no *critical instant theorem*, at which the worst-case behavior in analyzing a task is concretely captured, has been yet established. To cover the worst-case behavior for the non-existence of critical instant, accounting for the so-called *carry-in* jobs, that may

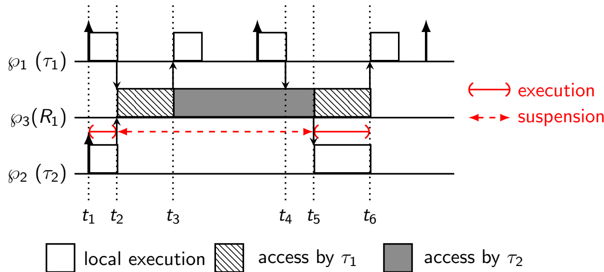

Figure 1: An example of accessing a shared resource R_1 by two tasks on a three-processor platform, on which τ_1, τ_2 are allocated on processor $\wp_1$ and $\wp_2$, respectively, and accesses to R_1 are bound to processor $\wp_3$.

be carried into the interval of our interest, i.e., $[t_0, t_0 + t)$, is commonly used in schedulability analysis, as shown in [7, 17, 24, 26]. Moreover, the interference due to such jobs can be more precisely quantified as *jitter*. Importantly, such jitter has to be carefully incorporated. Several misconceptions with incorrect quantifications of jitter for self-suspending task systems were reported in a recent survey paper by Chen et al. [16]. Nevertheless, it has been reported in [7,17] that using $RT_i - C_i$ as the jitter is safe, where RT_i is the worst-case response time of task τ_i, like global scheduling in multiprocessor systems [6].

LEMMA 1. *The cumulative non-critical-section execution time of a task τ_i in* $\mathbf{hpl}(k)$ *in the time interval* $[t_0, t_0 + t)$ *is upper bounded by:*

$$W_i(t) = \left\lceil \frac{t + RT_i - C_i}{T_i} \right\rceil C_i \tag{6}$$

LEMMA 2. *The cumulative execution time of a task τ_i for accesses to resource R_q in the time interval $[t_0, t_0 + t)$ is upper bounded by:*

$$E_{i,q}(t) = \left\lceil \frac{t + RT_i - A_{i,q}}{T_i} \right\rceil A_{i,q} \tag{7}$$

where RT_i (in the above two lemmas) is the worst-case response time of task τ_i and $RT_i \leq T_i$.

PROOF. The proofs are omitted since they are identical to the proofs in the literature. See [7,17] for details. □

Under multiprocessor partitioned scheduling for sporadic tasks without resource sharing, all the jobs generated by a task are constrained to execute only upon the processor to which the task is assigned. It thus follows that only those tasks that are executed on the same processor have to be taken into consideration when we analyze the schedulability of a task. However, in the presence of resource sharing, the time a task waits for executing itself is determined by not only the interference caused by local tasks but also resource accesses to the processors on which the task may request for shared resources.

Let $S(t)$ be the upper bound on the total time that task τ_k *executes* or *waits in the waiting queue to be granted to enter* the critical section in time interval $[t_0, t_0 + t)$. Let $X(t)$ be the upper bound on the total time that task τ_k *executes* or *waits in the waiting queue to execute* its non-critical-section in time interval $[t_0, t_0 + t)$. At any time $t_0 + t$, if the job of task τ_k released at time t_0 has not finished yet, it can only either (i) execute or wait on the local processor at time $t_0 + t$ or (ii) execute or wait on its remote synchronization processor(s).

For example, consider the simplest case in which two tasks τ_1 and τ_2 are assigned on processor $\wp_1$ and processor $\wp_2$, respectively. That is, there is no multitasking on each processor. We assume that τ_1 has higher priority than τ_2, and we are now analyzing task τ_2 in the following example. The schedule of accessing a shared resource by these two tasks is indicated in Figure 1:

- At time t_1, tasks τ_1 and τ_2 start their computation.
- At time t_2, tasks τ_1 and τ_2 both attempt to access the shared resource. The request from task τ_1 is granted, while task τ_2 suspends itself on $\wp_2$.
- At time t_3, task τ_1 finishes its access to the shared resource and resumes its local computation. At the same time, task τ_2 starts to access the shared resource.
- At time t_4, task τ_1 again attempts to access the shared resource. To maintain mutual exclusion, this request from task τ_1 is blocked by task τ_2.
- At time t_5, after leaving the critical section, task τ_2 unlocks the shared resource and resumes its local computation. At the same time, task τ_1 starts to access the shared resource.
- At time t_6. At the same time, task τ_2 finishes its execution.

As can be seen, Task τ_2 executes its computation on the application processor, in time intervals $[t_1, t_2)$ and $[t_5, t_6)$, and suspends itself to access shared resource R_1 in time intervals $[t_2, t_5)$. Therefore, if a task may request at most one resource, this task is awaiting either on the application (local) processor or on the remote synchronization processor before its completion. Suppose that $S(t)$ and $X(t)$ are safe. Then, if the job of task τ_k released at time t_0 cannot finish its execution at time t_0+t, it must be the case that $S(t)+X(t) > t$. Equivalently, the negation of this condition is sufficient to upper bound the response time of a task. As a result, the classic response-time analysis (RTA) can be extended as follows:

THEOREM 2. *Suppose that $S(t)$ and $X(t)$ are both safe upper bounds. The smallest t satisfying the following*

$$S(t) + X(t) \leq t \tag{8}$$

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

PROOF. From the above argument, task τ_k is awaiting/executing either on the local processor or on one of its remote synchronization processor before its completion. If the job of task τ_k released at time t_0 cannot finish its execution at time $t_0 + t$ and by the definition of $S(t)$ and $X(t)$, it must be the case that $S(t) + X(t) > t$. Equivalently, the negation of this condition is sufficient to upper bound the response time of a task. Hence, we here conclude this theorem. □

We now explain how to compute $X(t)$ and $S(t)$.

Computing $X(t)$. The waiting time on the local processor is only dependent on the execution of non-critical-section codes of tasks bound to the same processor, which can be elaborated as follows:

$$X(t) = I^{nRes}(t) + C_k \tag{9}$$

where the terms are as described below:

- C_k denotes an upper bound on the amount of *non-critical-section* execution from task τ_k itself.
- $I^{nRes}(t)$ denotes an upper bound on the amount of interferences from higher-priority non-critical-section execution with an interval of length t on the same processor.

From Lemma 1, it is safe to set

$$I^{nRes}(t) = \sum_{\tau_i \in \mathbf{hpl}(k)} W_i(t) \quad (10)$$

where $\mathbf{hpl}(k)$ is the set of the tasks with priority higher than τ_k on the local application processor where τ_k runs its non-critical sections.

Computing $S(t)$. The waiting time on the synchronization processor that runs the critical section of task τ_k can be elaborated as follows:

$$S(t) = A_k + B_k + I_k^{rRes}(t) \quad (11)$$

where the terms are as described below:

- A_k denotes an upper bound on the amount of *critical-section* execution from task τ_k itself.
- B_k denotes an upper bound on the amount of blocking from lower-priority critical-section accesses bound on the same synchronization processor.
- $I_k^{rRes}(t)$ denotes an upper bound on the amount of interferences from higher-priority synchronization execution with an interval of length t.

Under the resource-oriented scheduling, the total time that task τ_k executes or waits in the waiting queue to be granted to enter the critical sections in time interval $[t_0, t_0 + t)$ is contributed by all the resources that are bound to the same synchronization processor. Let Θ_r be the set of shared resources that are bound to the synchronization processor $\wp_r$ on which task τ_k executes its critical section.

The B_k parameter is similar to the one in uniprocessor systems, but dependent on only those low-priority accesses to the same synchronization processor: under the R-NPP

$$B_k = \max_{i,q}\{V_{i,q} | \pi(\tau_i) < \pi(\tau_k), R_q \in \Theta_r\} \quad (12)$$

and under the R-PCP

$$B_k = \max_{i,q}\{V_{i,q} | \pi(\tau_i) < \pi(\tau_k), \mathcal{C}(R_q) \geq \pi(k), R_q \in \Theta_r\} \quad (13)$$

From Lemma 2, it follows that

$$I_k^{rRes}(t) = \sum_{R_q \in \Theta_r} \sum_{\tau_i \in hp(k)} E_{i,q}(t) \quad (14)$$

4.2 Non-Critical Sections on Synchronization Processors

A synchronization processor can also execute non-critical sections. In our design, if there are non-critical sections assigned to be executed on a synchronization processor, the execution of the non-critical-sections has the priority *lower* than any of the critical-sections. Suppose that task τ_k is assigned to a synchronization processor. In this case, when calculating $X(t)$, we also need to consider the interference from critical-section executions running at higher priority:

$$X(t) = I^{nRes}(t) + I^{lRes}(t) + F_k \quad (15)$$

where $I_k^{lRes}(t)$ denotes an upper bound on the amount of interferences from local critical-section executions of tasks with their *base* priority higher than τ_k with an interval of length t, i.e. $\pi(\tau_i) > \pi(\tau_k)$; and F_k denotes an upper bound on the amount of interferences from local critical-section executions of tasks, denoted by a set $lp(k)$, with their *base* priority lower than τ_k, i.e. $\pi(\tau_i) < \pi(\tau_k)$, i.e., $lp(k) = \{\tau_i \mid \pi(\tau_i) < \pi(\tau_k)\}$.

Let Θ_l be the set of shared resources that are bound to the processor $\wp_l$ on which task τ_k executes its non-critical-section codes. We have

$$I^{lRes}(t) = \sum_{R_q \in \Theta_l} \sum_{\tau_i \in hp(k)} E_{i,q}(t) \quad (16)$$

To calculate F_k, we can again use Lemma 2 directly if we are aware of the worst-case response time of the lower-priority task under the assumption that $RT_i \leq T_i$ for each $\tau_i \in lp(k)$. Later in Section 5, we will not be able to know RT_i of a lower-priority task τ_i in $lp(k)$ when testing the schedulability of task τ_k. However, it can be safely assumed that $RT_i \leq T_i$ and this predicate $RT_i \leq T_i$ for $\tau_i \in lp(k)$ will be verified later when we test the schedulability of task τ_i. Therefore, we have

$$F_k = \sum_{R_q \in \Theta_l} \sum_{\tau_i \in lp(k)} \left\lceil \frac{t + T_i - A_{i,q}}{T_i} \right\rceil A_{i,q} \quad (17)$$

As $S(t)$ and $X(t)$ have been computed, we present our schedulability in the following theorem:

THEOREM 3. *For each task τ_k in τ, if there exists $0 \leq t \leq D_k$ s.t. Eq. (8) holds, then task τ_k with resource sharing is schedulable in fixed-priority partitioned scheduling under a given task partitioning.*

Algorithm 1: Linear Search

```
input  : A set of n tasks τ, m processors ℘, and r resources RS
output: Resources allocations Θ, task allocations Γ and the
        feasibility of system τ
for m^R = 1, ..., min(m, r) do
    if WFD (Q, m^R) returns "feasible allocation" then
        Θ ←WFD (Q, m^R);
    else
        continue;
    if FFRM(τ,Θ) returns "feasible allocation";
    then
        return "feasible system";
return "infeasible system";
```

5. TASK AND RESOURCE ALLOCATIONS

In this section we present our algorithm that determines a set of synchronization processors and that allocates both shared resources and tasks onto processors. The intuition underlying the proposed algorithm is that under the resource-oriented scheduling, once shared resources are mapped onto the synchronization processors, the blocking time incurred by lower-priority tasks can be determined, irrespective of their task allocations. Hence, by initially sorting the tasks in an order of decreasing priorities (non-decreasing order of relative deadlines), any task being assigned will not jeopardize the schedulability of the tasks that have been successfully assigned onto processors:

1. First, we iteratively determine a *configuration* of initializing a set of processors to be used as synchronization processors. From a schedulability point of view, the reduction in the number of the synchronization processors is a tradeoff between
   - an increase on the time spent on the execution of critical sections on the synchronization processors, and
   - a reduction on the time spent on the execution of non-critical sections on the application processors.

   As each resource is bound to one processor, at most $min(m, r)$ configurations of processors need to be checked until either a feasible system is found or does not exist. In each configuration of processors, resources and tasks are respectively allocated by *Worst-*

Algorithm 2: Worst-Fit Decreasing (WFD)

```
input  : A set resource RS and m^R identical processors for
         synchronization
output: Resources allocations Θ
Θ_j ← ∅, ∀j = 1, 2, ..., m;
sort the r shared resources RS with non-increasing U^{R_q};
for R_q ∈ RS do
    // put these processors furthest
    for h = m − m^R + 1, ..., m do
        calculating the load Σ_{R_j ∈ Θ_h} U^{R_j};
    // least utilization first
    assign R_q to the m^R processor ℘_h with the minimum load;
    if U^{R_q} + Σ_{R_j ∈ Θ_h} U^{R_j} > 1 then
        return "infeasible allocation";
    else
        Θ_h ← Θ_h ∪ {R_q};
return "feasible allocation",Θ;
```

Fit Decreasing (WFD) algorithm and *First-Fit Rate-Monotonic* (FFRM) algorithm, presented later.

2. The resources are ordered in a list in a non-increasing order of their utilization. The algorithm attempts to allocate each resource onto the synchronization processor with the *least* load, called WFD. We note that this is a well-known strategy for the bin-packing problem. The intuition underlying the WFD is that by distributing resources evenly, it is sensible to reduce the time spent by tasks waiting for resource accesses remotely. These synchronization processors are preferably put as far as possible from the processors on which tasks will be allocated later.
3. Tasks are prioritized and sorted in the order of non-decreasing relative deadlines, i.e., $D_1 \leq D_2 \leq ... \leq D_n$. That is, for implicit-deadline task systems, we use the well-known rate-monotonic (RM) policy for assigning the base priorities of the tasks. Then, the algorithm considers to assign (the non-critical sections of) the tasks to processors from the highest base priority to the lowest base priority. Our algorithm here, called First-Fit Rate-Monotonic (FFRM), places the task in the first processor that can accommodate the task according to Theorem 3. In addition, the algorithm heuristically places the task in the processors to which no shared resources are bound. If no such processors can accommodate this task, then the algorithm will also check the feasibility of putting it in those processors initialized to be synchronization processors.

We denote our algorithm as Algorithm ROP-PCP when PCP is used and Algorithm ROP-NPP when NPP is adopted.

Runtime complexity. In attempting to find a configuration for synchronization processors, we need at most m rounds. We note that the overall sorting time of Algorithm 2 and Algorithm 3 in all rounds can be amortized to $\mathcal{O}(r \log r + n \log n)$ by using appropriate data structures. In each round, Algorithm 2 runs in time complexity $\mathcal{O}(r \log m)$ by maintaining the processor utilization with a heap data structure, and Algorithm 3 requires $\mathcal{O}(mnD_n)$ for checking whether a task can fit into one processor according to Theorem 3, where D_n is the longest relative deadline among tasks. Overall, our algorithm runs in $\mathcal{O}(r \log r + n \log n + m(r \log m + mnD_n))$, which is in pseudo-polynomial time complexity.

6. SPEEDUP FACTOR UNDER RM

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

```
input  : A set τ of tasks, m^C identical processors for
         non-synchronization, resources allocation Θ
output: Task allocations Γ_j and the feasibility of system τ
sort the given n tasks in τ s.t. D_1 ≤ D_2 ≤ ··· ≤ D_n;
Γ_j ← ∅, ∀j = 1, 2, ..., m;
for k = 1, 2, ..., n do
    for p = 1, ..., ..., m do
        if task τ_k is schedulable according to Theorem 3 then
            Γ_p ← Γ_p ∪ {τ_k}; // assign τ_k to processor p
            break ;
    if τ_k cannot fit any processor in the above loop then
        return "infeasible allocation";
return "feasible allocation";
```

In this section, we obtain a speedup factor for Algorithm ROP-PCP under rate-monotonic (RM) scheduling, in which $D_i = T_i$ for every task $\tau_i \in \tau$. The approach is as follow. We identify the smallest value of $x \geq 1$ for which we can prove that any task set that is feasible upon a platform comprising m unispeed processors is deemed to be schedulable by the RMFF upon a platform in which each processor is at least x times as fast. Consequently, we can conclude that the value x is a processor speedup bound.

In the following lemma, we first provide necessary conditions for any optimal scheduling, which are based upon the concept of demand bound functions (dbf) as defined in Section 2.3.

LEMMA 3. *Any implicit-deadline task system τ that is feasible upon a platform comprised of m processors must satisfy*

$$U^C + U^{\mathcal{RS}} \leq m \text{ and } \forall \tau_i \in \tau, \quad U_i \leq 1 \tag{18}$$

and $\forall \tau_k \in \tau$, $\forall R_q \in \mathcal{RS}_k$

$$\frac{\max_{\tau_i : D_i > D_k} V_{i,q} + \sum_{\tau_i : D_i \leq D_k} dbf_i^{R^q}(D_k)}{D_k} \leq 1 \tag{19}$$

PROOF. The proof is in Appendix. □

We will derive a speedup factor for the RMFF. Before that, we first need the following two lemmas:

LEMMA 4. *For $t \geq D_i$,*

$$3dbf_i^C(t) \geq W_i(t) \tag{20}$$

and

$$3dbf_i^{R^q}(t) \geq E_{i,q}(t) \tag{21}$$

PROOF. The proof is in Appendix. □

LEMMA 5. *Suppose that $U_i \leq 1$ for every task $\tau_i \in \tau$. Given m^R synchronization processors, the utilization of shared resources on each processor under WFD is at most*

$$1 + \frac{U^{\mathcal{RS}} - 1}{m^R} \tag{22}$$

PROOF. The proof is similar to the scheduling algorithms of the *makespan* problem. Let L^* be the maximum utilization among the m^R processors to schedule shared resources under WFD. Let u_ℓ be the utilization of the last resource mapped onto L^*. The WFD algorithm assigns each resource onto the processor with the least load. It follows that

$$L^* - u_\ell \leq \frac{U^{\mathcal{RS}} - u_\ell}{m^R} \Rightarrow L^* \leq \frac{U^{\mathcal{RS}} - u_\ell}{m^R} + u_\ell \leq \frac{U^{\mathcal{RS}} - 1}{m^R} + 1 \tag{23}$$

where the last inequality is due to the condition $u_\ell \leq 1$. □

In the following theorem, we show that the speedup factor of our algorithm is $11 - 6/(m+1)$ when $N = 1$ and $Q = 1$, irrespective of how many shared resources, r, are present.

THEOREM 4. *The speedup factor of the proposed resource-oriented partitioned scheduling algorithm ROP-PCP is* $11 - \frac{6}{m+1}$ *when* $m \geq 2$, $N = 1$, *and* $Q = 1$ *under the resource-oriented scheduling using PCP in Section 3.2.*

PROOF. We prove this theorem by showing that any task set that is feasible upon a platform comprising m unispeed processors is deemed to be schedulable by Algorithm ROP-PCP upon a platform in which each processor is at least $11 - 6/(m+1)$ times as fast.

Suppose that Algorithm ROP-PCP fails to obtain an allocation for τ: there exists task τ_k which cannot be mapped on to any processor by Algorithm FFRM. Note that due to the sorting of the tasks in Algorithm ROP-PCP, all the tasks before task τ_k mapped onto processor have been ensured $RT_i \leq D_i = T_i$ for $i = 1, 2, \ldots, k-1$. Since τ_k fails the test of Theorem 3, it must be the case that for every processor

$$S(D_k) + X(D_k) > D_k \tag{24}$$

The failure of Algorithm ROP-PCP implies that τ_k also fails the test of Theorem 3 on each of the $m^C = m - m^R$ application processors.

Summing over all m^C such processors and after reformulation, we obtain

$$\frac{C_k + A_k}{D_k} + \frac{B_k}{D_k} + \frac{I_k^{rRes}(D_k)}{D_k} + \frac{\sum_{\tau_i \in hp(k)} W_i(D_k)}{m^C D_k} > 1 \tag{25}$$

Let $B_k \neq 0$ be the longest critical section of accessing resource R_b of tasks having relative deadline larger than D_k that blocks task τ_k's critical-section execution on the designated processor. We now consider two separate cases:

- τ_k may request on R_b. From Lemma 3, it is necessary for task τ_k that $\left(B_k + \sum_{\tau_i : D_i \leq D_k} dbf_i^{R_b}(D_k)\right)/D_k \leq 1$. Clearly, it follows that $B_k \leq D_k$.
- τ_k doesn't request on R_b. Recall that under PCP a task can only be blocked by lower-priory tasks' critical sections that are accessed by a task with an equal or higher priority than τ_k. Thus, under RM scheduling there must exist a task τ_a with a relative deadline $D_a \leq D_k$ that may request on R_b. As $D_a \leq D_k$, the execution time from tasks having relative deadlines larger than D_a that has to be serialized must be at least B_k. From Lemma 3, it is necessary for task τ_a being schedulable that $\left(B_k + \sum_{\tau_i : D_i \leq D_a} dbf_i^{R_b}(D_a)\right)/D_a \leq 1$. It then follows that $B_k \leq D_a \leq D_k$.

In either case, we can see that $B_k \leq D_k$.

If each processor is at least x times as fast, and by Lemma 3 and the above discussions, we know $(C_k + A_k)/D_k \leq 1/x$, $B_k/D_k \leq 1/x$, and

$$U^C \leq \frac{m - U^{\mathcal{RS}}}{x} \tag{26}$$

By Lemma 5, if each processor is at least x times as fast, it must be the case that

$$\frac{\sum_{R_q \in \Theta_h} \sum_{\tau_i \in hp(k)} dbf_i^{A_q}(D_k)}{D_k} \overset{\text{Eq. (22)}}{\leq} \frac{1 + \frac{U^{\mathcal{RS}} - 1}{m^R}}{x} \tag{27}$$

Putting the pieces together, at processors with speed x we have

$$\begin{aligned}\frac{I_k^{rRes}(D_k)}{D_k} &= \frac{\sum_{\tau_i \in hp(k)} \sum_{R_q \in \Theta_h} X_{i,q}(D_k)}{D_k} \\ &\overset{\text{Eq. (21)}}{\leq} \frac{3 \sum_{\tau_i \in hp(k)} \sum_{R_q \in \Theta_h} dbf_i^{A_q}(D_k)}{D_k} \\ &\overset{\text{Eq. (27)}}{\leq} \frac{3(1 + \frac{U^{\mathcal{RS}} - 1}{m^R})}{x}\end{aligned} \tag{28}$$

and

$$\begin{aligned}&\frac{\sum_{\tau_i \in hp(k)} W_i(D_k)}{m^C D_k} \\ \overset{\text{Eq. (20)}}{\leq}\ & \frac{3 \frac{\sum_{\tau_i \in \tau} dbf_i^C(D_k)}{D_k}}{m^C} \leq \frac{3U^C}{m^C} \overset{\text{Eq. (26)}}{\leq} \frac{3(m - U^{\mathcal{RS}})}{x(m - m^R)}\end{aligned} \tag{29}$$

Summing over the corresponding terms and to contradict to Eq. (25), we need to set

$$x \geq 5 + \frac{3(U^{\mathcal{RS}} - 1)}{m^R} + \frac{3(m - U^{\mathcal{RS}})}{m - m^R} \tag{30}$$

Let $f(m, U^{\mathcal{RS}}) = 5 + \frac{3(U^{\mathcal{RS}} - 1)}{m^R} + \frac{3(m - U^{\mathcal{RS}})}{m - m^R}$. In the following proof, we show that $f(m, U^{\mathcal{RS}})$ is upper bounded by $11 - 6/(m+1)$. We consider two separate cases:

- m is even. Let m^R be $\frac{m}{2}$. Thus, we have

$$\begin{aligned}f(m, U^{\mathcal{RS}}) &= 5 + \frac{3(U^{\mathcal{RS}} - 1)}{m/2} + \frac{3(m - U^{\mathcal{RS}})}{m/2} \\ &= 5 + 6\frac{m-1}{m} = 11 - \frac{6}{m}\end{aligned} \tag{31}$$

- m is odd. Due to our assumption, $m \geq 3$; therefore, $(m-1)/2 \geq 1$. In this case, we further consider two subcases:
  - $U^{\mathcal{RS}} \geq \frac{m+1}{2}$. Let m^R be $\frac{m+1}{2}$.

$$\begin{aligned}f(m, U^{\mathcal{RS}}) &= 5 + \frac{3(U^{\mathcal{RS}} - 1)}{(m+1)/2} + \frac{3(m - U^{\mathcal{RS}})}{(m-1)/2} \\ &= 5 + 6\left(\frac{(m^2 - 1) + 2 - 2U^{\mathcal{RS}}}{m^2 - 1}\right) \\ &\leq^1 11 - \frac{6}{m+1}\end{aligned} \tag{32}$$

    where $\leq^1$ is due to the assumption that $U^{\mathcal{RS}} \geq \frac{m+1}{2}$.
  - $U^{\mathcal{RS}} < \frac{m+1}{2}$. Let m^R be $\frac{m-1}{2}$.

$$\begin{aligned}f(m, U^{\mathcal{RS}}) &= 5 + \frac{3(U^{\mathcal{RS}} - 1)}{(m-1)/2} + \frac{3(m - U^{\mathcal{RS}})}{(m+1)/2} \\ &= 5 + 6\left(\frac{(m^2 - 1) - 2m + 2U^{\mathcal{RS}}}{m^2 - 1}\right) \\ &\leq^1 11 - \frac{6}{m+1}\end{aligned} \tag{33}$$

    where $\leq^1$ is due to the assumption that $U^{\mathcal{RS}} < \frac{m+1}{2}$.

In either case, we can see that $f(m, U^{\mathcal{RS}})$ is upper bounded by $11 - 6/(m+1)$.

Note that our analysis in this proof greedily sets m^R instead of searching m^R sequentially as in Algorithm 1. It is possible that m^R in our greedy setting is larger than r, the number of shared resources. However, this does not create any problem for the above analysis of the speedup factor. In this case, $m^R - r$ processors are completely unused and wasted in the above proof since they are not used under

WFD in Algorithm 2. Therefore, the above analysis is more pessimistic, and we here conclude this theorem. □

7. MULTIPLE RESOURCE ACCESSES

In this section, we extend the schedulability analyses presented in Section 4 to include multiple resource accesses during a job's execution and multiple occasions on one resource request, i.e., $N \geq 2$ or $Q \geq 2$.

7.1 Multiple Occasions on Each Resource

Eq. (5) is based on the assumption that once a job begins execution, it does not suspend itself until completion. However, the execution on the synchronization processor under the resource-oriented scheduling may suspend itself for executing the non-critical-section codes on the application processor; each access may suffer from the longest duration of one critical section of lower priority jobs. Hence, the number of blocking of a task having multiple occasions on one resource request can be up to the total number of accesses to critical sections:

$$B_k = \gamma_k \times B_k \tag{34}$$

where B_k is defined as in Eq. (5) and

$$\gamma_k = \sum_{R_q \in \mathcal{RS}_k} N_{k,q} \tag{35}$$

7.2 Multiple Resource Accesses

The resource-oriented scheduling assumes *partitioned fixed-priority* scheduling; the computation has to be executed locally in accordance with the designated processors. With multiple resource accesses, there may be more than one processor on which task τ_k may execute *remotely*, dependent of resource allocations. Similar to Section 4.1, at any time, a task being busy must execute/wait on one of the processors that the task may execute. Hence, the $S(t)$ on synchronization processors can be generalized as follows: Let Φ_k be the *set* of processors on which task τ_k may execute *remotely*.

$$S(t) = \sum_{\wp_h \in \Phi_k} S_h(t) \tag{36}$$

Elaborating the time spent on each processor gives us:

$$S_h(t) = A_{k,h} + B_{k,h} + I_{k,h}^{rRes}(t) \tag{37}$$

where the terms are as described below:

- $A_{k,h}$ denotes an upper bound on the amount of critical section execution from task τ_k itself on processor $\wp_h$.
- $B_{k,h}$ denotes an upper bound on the amount of blocking from lower-priority tasks on processor $\wp_h$.
- $I_{k,h}^{rRes}(t)$ denotes an upper bound on the amount of interferences from higher-priority synchronization execution on processor $\wp_h$ with an interval of length t.

With the above definition of $S(t)$ in Eq. (36) and B_k in Eq. (34), we can again apply Theorem 2 and Theorem 3 for schedulability test.

8. EXPERIMENTAL RESULTS

In this section, we conduct extensive experiments using synthesized task sets. Due to space limitations, only a subset of the results is presented. We evaluate these tests on m-processor platforms. We generate 100 task sets for each utilization level, from $0.05m$ to m, in steps of $0.05m$. The metric to compare the results is to measure the *acceptance ratio*. The acceptance ratio of a level is said to be the number of task sets that are deemed schedulable by the test divided by the number of task sets for this level, i.e., 100.

The cardinality of the task set is 10 times the number of processors, e.g. 40 tasks on 4 processors. We use the approach suggested by Emberson et al. [20] to generate the task periods according to the *exponential distribution*. The distribution of periods is within two orders of magnitude, i.e., 10ms-1000ms. Task relative deadlines are implicit, i.e., $D_i = T_i$.

We vary the ratio of non-critical-sections to critical-sections $\alpha \in \{20, 5\}$ to evaluate the effect of resource sharing: the smaller the α, the more the critical-sections. For example, if $\alpha = 5$ and the utilization level $U_\Sigma = 120\%$, we have $U^{\mathcal{RS}} = 120\% \times \frac{1}{5+1} = 20\%$ and $U^C = 120\% \times \frac{5}{5+1} = 100\%$ in the system. In each utilization step, the `Randfixedsum` method [20] is adopted twice to generate two sets of utilization values with the given goals of critical-sections and non-critical-sections. We ensure that for every task τ_i, $U_i^A + U_i^C \leq 1$. The worst-case execution time of a task for its non-critical-sections and critical-sections is set accordingly, i.e., $C_i = T_i U_i^C$ and $A_i = T_i U_i^A$.

We assume there are 5, 8, and 16 shared resources in multiprocessor systems comprised of 4, 8, and 16 processors, respectively. We vary the number of shared resources that a task requests $N \in \{1, 3\}$. For each task, we then again use the `Randfixedsum` method to generate a set of vectors which are evenly distributed in the Q resource accesses and the total access utilization sums to its critical-section utilization. The critical section of accessing resource R_q of task τ_i is set accordingly, i.e. $A_{i,q} = U_{i,q} T_i$. We set the number of requests on each resource during execution N to either 1 or 3. The maximum total resource usage time $V_{i,q}$ for resource R_q by any single job of τ_i is drawn uniformly from $[A_{i,q}/N, A_{i,q}]$.

The global/partitioned RM scheduling is applied by default in the system, unless otherwise stated. The evaluated tests are listed as follows:

- PIP: the Priority Inheritance Protocol (PIP) [19], which is a fixed-priority global scheduling algorithm.
- MPCP: the Multiprocessor Priority Ceiling Protocol (MPCP) [28] along with the Synchronization-Aware Partitioning Algorithm (SPA) [25]. Informally speaking, contrary to the proposed approach, the MPCP can be thought of as the conservative approach wherein task bodies are not split.
- MrsP: the Multiprocessor resource sharing Protocol (MrsP) [14] along with the SPA.
- R-NPP: the proposed resource-oriented partitioned scheduling using ROP-NPP by Algorithm 1 where Theorem 3 uses the blocking time Eq. (12).
- R-PCP: the proposed resource-oriented partitioned scheduling using ROP-PCP by Algorithm 1 where Theorem 3 uses the blocking time Eq. (13).
- NCDBF: the theoretical upper bound using necessary conditions stated in Lemma 3.

To make a fair comparison, all the implemented tests have pseudo-polynomial time complexity. We note that the accuracy of the sufficient schedulability tests listed above can be improved by formalizing the problem of finding a response-time bound as a linear optimization problem [35]. However, tests using the LP solver may suffer from their high time complexity, especially, in conjunction with partitioning resources and tasks. Besides, we here focus on showing the effectiveness of protocols themselves, and similar results can be seen under LP formalizations.

Results. Figure 2 shows the evaluation of the performance by the above scheduling algorithms in terms of task sets deemed schedulable. In the first two figures (Figures 2a and 2b), we vary the number of processors $m \in \{4, 8\}$,

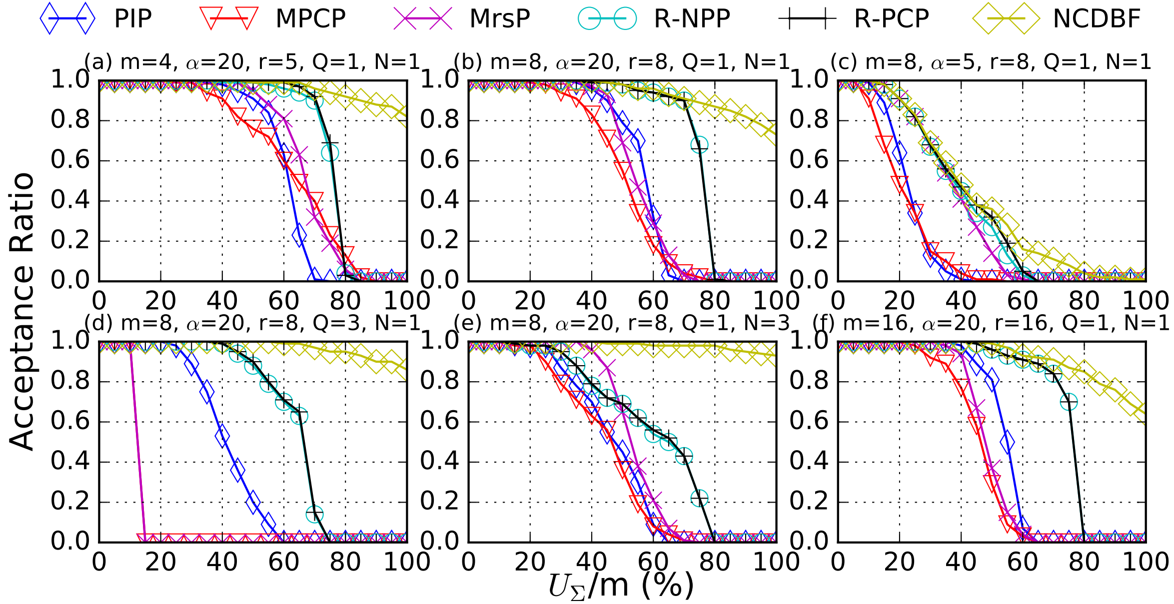

Figure 2: Effectiveness by different algorithms, varying m, α, r, Q, and N

assuming $\alpha = 20$, $Q = 1$, and $N = 1$. We first notice that both the R-NPP and the R-PCP dominate all the existing approaches, including the PIP, the MPCP, and the MrsP. The later three are neither better than another.

Although the PIP bounds the priority inversion and is shown to be relatively effective [35], the execution for a job still suffers from the chained blocking. MPCP and MrsP using the SPA attempt to assign each of the so-called *macrotasks* (defined in [25]) onto a processor. However, it is often the case that a macrotask is too heavy to fit into one processor and hence is forced to be split into several processors. After that, any task belonging to the split macrotask suffers from the excessive blocking time incurred by lower-priority tasks on other processors. On the other hand, even though the total utilization of a macrotask can fit into one processor, sometimes it is better to shift some non-critical-section execution to other processor as the utilization of a macrotask may become quite heavy. Hence, not surprisingly, PIP, MrsP, and MPCP perform worse than R-PCP and R-NPP, both minimizing the number of blocking with $Q = 1$ and $N = 1$.

It is remarkable that both R-PCP and R-NPP can still keep pace with the theoretical upper bound, the NCDBF, until utilizations are over 70%. A similar result can also be seen from Figure 2f where a 16-processor platform is assumed.

In Figures 2b and 2c, we vary α in $[20, 5]$ on a 8-processor platform with 8 shared resources where a task requests one resource out of the five, assigned randomly, and each job of a task requests the resource once during its execution. We first notice that NCDBF drops off as α decreases from 20 to 5. R-PCP and R-NPP can still keep pace with the theoretical upper bound, NCDBF, until utilizations are over 50%

In Figures 2d and 2e, we evaluate the effect of accessing multiple resource and requesting multiple times on one resource. Here we can see that despite that R-PCP and R-NPP are still superior to the others, there exists a large gap between the theoretical upper bound and the best available algorithm, dropping down from 40% with multiple resource accesses and from 25% with multiple requests on one resource. We also notice that in Figure 2d both MPCP and MrsP drop quickly from 15%. This is due to that the SPA algorithm only considers utilizations to allocate tasks onto processors. With multiple resource accesses, it is often the case that a *macrotask*, within which tasks directly or indirectly share resources are bundled, becomes extremely heavy. The algorithm allocates the tasks within the macrotask onto one processor as far as possible, so as to reduce the processors used (also remote blocking). As a result, some tasks on this processor can be punished by such excessive interferences: the effectiveness of partitioned protocols is highly dependent on task partitioning. Hence, both designing synchronization protocols and finding task mapping are needed to be considered.

From the scheduling's point of view, using PCP in each individual processor is as good as using the non-preemptive scheduling. Surprisingly, R-PCP and R-NPP have almost the same performance in our evaluations (although invisible, there still have some tasks not deemed schedulable by R-NPP but schedulable by R-PCP). This is because the long blocking incurred by the NPP might be painlessly removed by finding a good task and resource partitioning.

Maintaining a list of currently locked semaphores and priority ceiling orders during runtime may incur a noticeable computational overhead. In practice, one would expect R-NPP to be default, and R-PCP to be used only when the unnecessary blocking from the non-preemptive scheduling can be surely removed. Last but not least, the research result reported in this paper suggests that finding a good task mapping and resource allocation is as important as designing a good resource sharing protocol while ignoring task partitioning.

9. RELATED WORK

To handle the synchronization problem when tasks share resources in real-time systems, a wide variety of real-time locking protocols have been developed. Briefly, a real-time locking protocol is used to limit priority inversions [13, 30], such that higher-priority tasks incur priority inversions only if lower-priority tasks execute in critical sections. Recent analysis and comparisons of semaphore protocols may be found in [10] for partitioned scheduling (e.g., the DPCP [29] and the MPCP [28], etc.) and in [35] for global scheduling

(e.g., the global PIP, etc.). To support nested critical sections, Ward and Anderson [31, 32] recently introduced the Real-time Nested Locking Protocol (RNLP) [31], which adds supports for fine-grained nested locking on top of non-nested protocols.

Further, to ensure mutual exclusive access to shared resources, tasks either self-suspend (under semaphore protocols such as the DPCP [29] and the MPCP [28]) or busy wait (under spin-based protocols such as the MSRP [21]) when blocked on shared resources. Busy waiting has been shown to be efficient when critical sections are short [9, 22], but the resulting loss of processor service must be accounted for. More recently, Burns and Wellings [14] proposed a variant of the MSRP, the Multiprocessor resource sharing Protocol (MrsP) [14], to exploit the spinning cycles of one task to help other tasks make progress. Wieder and Brandenburg [34] presented a fine-grained analysis for several types of spin locks. In contrast, self-suspensions are more efficient for long critical sections [9, 25]. Moreover, in some distributed-configured scheduling systems, such as the designated [29] or dedicated [23] synchronization frameworks, jobs self-suspend on host processors, waiting for resource service on remote processors, is a natural fit for the scheduling strategy. In this work, we adopt the suspension-based methodology for resource sharing.

With regard to task partitioning, Lakshmanan et al. [25] presented a synchronization-aware partitioned heuristic tailored to the MPCP [28], 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. [27] presented a blocking-aware partitioning method that uses advanced cost heuristic to split task group when an entire group fails to be assigned on one processor. In subsequent work, Hsiu et al. [23] proposed a dedicated-core framework to separate the execution of critical sections and normal sections, and employed an priority-based RPC-like mechanism for resource sharing, such that each higher-priority request can be blocked by at most one lower-priority request. More recently, Wieder and Brandenburg [33] use integer linear programming to solve the partitioning problem in the presence of spin locks.

From an algorithmic optimality point of view, Brandenburg and Anderson [13] are first to study the multiprocessor real-time locking problem. It is shown that FIFO-based locking protocols, such as the Flexible Multiprocessor Locking Protocol (FMLP) [8], the Generalized FIFO Multiprocessor Locking Protocol (FMLP$^+$) [12], and the Distributed FIFO Locking Protocol (DFLP) [11], are asymptotically optimal in terms of maximum priority-inversion blocking. Andersson and Easwaran [1] presented an virtualization-based G-EDF scheduling with shared resources, which guarantees a $12(1 + 3r/(4m))$ competitive ratio on a platform comprising m identical processors, where each task uses at most one of the r shared resources and each job may request the resource at most once, whereas the presented algorithm in this work achieves a constant speedup factor 11, without using any virtualization.

10. CONCLUSIONS

In this paper, we show that the resource-oriented scheduling using PCP combined with a reasonable allocation algorithm can achieve a non-trivial speedup factor guarantee. Our empirical investigations show that the proposed algorithm is highly effective in terms of task sets deemed schedulable. Specifically, empirical results show that the proposed algorithm is very close to the optimal scheduling for task sets with utilization below 50% in the sense that task sets not deemed schedulable by our algorithm are also not schedulable using any scheduling policy. More importantly, empirical results suggest that by partitioning tasks properly, a simple non-preemptive protocol can be directly applied in each processor under the proposed resource-oriented scheduling, without punishing tasks having short relative deadlines by long critical-sections.

Nevertheless, our results also indicate that the existing algorithms, including the algorithm in this paper, do not scale very well in the presence of multiple resource accesses and multiple requests on one resource. In future work, we aim at bridging this gap. Further, it is also interesting in providing quality-guaranteed algorithms for resource sharing with nested requests.

11. REFERENCES

[1] B. Andersson and A. Easwaran. Provably good multiprocessor scheduling with resource sharing. *Real-Time Systems*, 46(2):153–159, 2010.

[2] N. C. Audsley, A. Burns, M. M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

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

[4] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[5] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Is semi-partitioned scheduling practical? In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 125–135, 2011.

[6] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Real-Time Systems Symposium (RTSS)*, pages 149–160, 2007.

[7] 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. Technical Report CISTER-TR-150713, CISTER, July 2015.

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

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

[10] B. B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 141–152, 2013.

[11] B. B. Brandenburg. Blocking optimality in distributed real-time locking protocols. *LITES*, 1(2):01:1–01:22, 2014.

[12] B. B. Brandenburg. The FMLP+: an asymptotically optimal real-time locking protocol for suspension-aware analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 61–71, 2014.

[13] B. B. Brandenburg and J. H. Anderson. Optimality results for multiprocessor real-time locking. In *Real-Time Systems Symposium (RTSS)*, pages 49–60, 2010.

[14] A. Burns and A. J. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 282–291, 2013.

[15] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*, volume 24 of *Real-Time Systems Series*. Springer, 2011.

[16] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, Neil, Audsley, R. Rajkumar, and D. de Niz. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, Faculty of Informatik, TU Dortmund, 2016. http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2016-chen-techreport-854.pdf.

[17] J.-J. Chen, G. Nelissen, and W.-H. K. Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[18] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011.

[19] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Real-Time Systems Symposium (RTSS)*, pages 377–386, 2009.

[20] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *International*

Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010), pages 6–11, 2010.
[21] 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 *Real-Time Systems Symposium (RTSS)*, pages 73–83, 2001.
[22] 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. Industrial Informatics*, 10(2):903–918, 2014.
[23] P. Hsiu, D. Lee, and T. Kuo. Task synchronization and allocation for many-core real-time systems. In *International Conference on Embedded Software, (EMSOFT)*, pages 79–88, 2011.
[24] 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 *DAC*, 2015.
[25] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Real-Time Systems Symposium, (RTSS)*, pages 469–478, 2009.
[26] C. Liu and J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *RTSS*, pages 173–183, 2014.
[27] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *Principles of Distributed Systems - International Conference, OPODIS*, pages 253–269, 2010.
[28] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems (ICDCS)*, pages 116–123, 1990.
[29] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium (RTSS)*, pages 259–269, 1988.
[30] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.
[31] B. C. Ward and J. H. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Euromicro Conference on Real-Time Systems ECRTS*, pages 223–232, 2012.
[32] B. C. Ward and J. H. Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In *International Conference on Real-Time Networks and Systems, RTNS*, pages 67–76, 2013.
[33] A. Wieder and B. B. Brandenburg. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In *International Symposium on Industrial Embedded Systems, (SIES)*, pages 49–58, 2013.
[34] A. Wieder and B. B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of fifo, unordered, and priority-ordered spin locks. In *Real-Time Systems Symposium*, pages 45–56, 2013.
[35] M. Yang, A. Wieder, and B. B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *Real-Time Systems Symposium (RTSS)*, pages 1–12, 2015.

APPENDIX

Proof of Theorem 1. We prove this theorem by showing that a task set deemed schedulable by the priority ceiling protocol (PCP) using TDA by [2] under RM scheduling is also not schedulable by any scheduling polices on a speed-1/2 uniprocessor.

Suppose task τ_k is the task that misses its relative deadline. Let $B_k \neq 0$ be the longest critical section of accessing resource R_b of lower-priority tasks τ_{lp} that blocks task τ_k. Note that if $B = 0$, we can conclude the utilization is at least 69% by the Liu and Layland bound, which by taking its multiplicative inverse implies a speedup factor 1.44. Recall that under the PCP a task can only be blocked by lower-priority tasks' critical sections that are accessed by a task with higher priority than τ_k. Thus, under RM scheduling there must exist a task τ_a with a relative deadline $D_a \leq D_k$ that may request on R_b.

We now release task τ_{lp} alone, at time $-\epsilon$, right before time 0, starting to execute the critical section B_k of accessing R_b. We then release all the other tasks at time 0 and let the following jobs execute in their worst case and release as soon as possible. To complete τ_a's execution under mutual exclusion, it is necessary for the task set being schedulable that B_k together with all the necessary demands are finished at τ_a's relative deadline D_a and afterwards. This means that by Eq. (1), it must be

$$\forall t \geq D_a, \quad B_k + \sum_{\tau_i \in \tau \setminus \{\tau_{lp}\}} dbf_i(t) \leq t \tag{38}$$

The failure of RTA by [2] implies that

$$\forall 0 < t \leq D_k, \qquad B_k + \sum_{\tau_i \in hep(k)} \left\lceil \frac{t}{T_i} \right\rceil (C_i + A_i) > t. \tag{39}$$

where $hep(k)$ is the set of task τ_k together with tasks having higher-priority than τ_k. Let us instantiate this inequality for $t \leftarrow D_k$:

$$\begin{aligned}
&B_k + C_k + A_k + \sum_{\tau_i \in hep(k)} \left\lceil \frac{D_k}{T_i} \right\rceil (C_i + A_i) > D_k \\
&\quad \{\text{Thanks to RM scheduling, } hep(k) = \{\tau_i | D_i \leq D_k\}\} \\
\Rightarrow &B_k + \sum_{\tau_i : D_i \leq D_k} \left\lceil \frac{D_k}{T_i} \right\rceil (C_i + A_i) > D_k \\
&\quad \{\forall \tau_i\colon D_i \leq D_k, 2dbf_i(D_k) \geq \left\lceil \frac{D_k}{T_i} \right\rceil (C_i + A_i)\} \\
\Rightarrow &B_k + 2 \sum_{\tau_i : D_i \leq D_k} dbf_i(D_k) > D_k \\
\Rightarrow &2(B_k + \sum_{\tau_i \in \tau \setminus \{\tau_{lp}\}} dbf_i(D_k)) > D_k
\end{aligned}$$

which implies that by Eq. (38) and $D_k \geq D_a$ this task set is not schedulable by any scheduling policies on a speed-1/2 uniprocessor. Hence, we here conclude this theorem. □

Proof of Lemma 3. If τ_i generates jobs with execution time exactly $C_i + A_i$, it is necessary for meeting all the deadlines of task τ_i that $(C_i + A_i)/D_i \leq 1$. Moreover, in order for the task system to be schedulable by any algorithm upon a platform comprised of m processors, it is necessary that

$$U^C + U^{\mathcal{RS}} \leq m \tag{40}$$

Suppose that task τ_{lp} is the task having a relative deadline larger than D_k with the longest critical section of accessing R_q. Let task τ_{lp} be released at time $-\epsilon$, right before time 0, starting to execute this longest critical section of accessing R_q. Suppose that all the tasks having relative deadlines $D_i \leq D_k$ that may request on R_q along with τ_k are released at time 0, and each task τ_i generates jobs *only* requesting on R_q and as soon as possible. Recall these executions are subject to exclusion constraints, and therefore must be serialized. To complete τ_k's execution under mutual exclusion feasibly, it is necessary to finish τ_{lp}'s critical section together with all the necessary demands at relative deadline D_k:

$$\max_{\tau_i : D_i > D_k} V_{i,q} + \sum_{\tau_i : D_i \leq D_k} dbf_i^{R^q}(D_k) \leq D_k \tag{41}$$

Hence, we here conclude this lemma. □

Proof of Lemma 4. The proofs for both cases are essentially identical; hence, we only detail either of them. Let $\hat{W}_i(t)$ be $(\lceil (t - D_i)/T_i \rceil + 2) \times C_i$. By definition of constrained-deadline tasks ($D_i \leq T_i$), we have

$$\hat{W}_i(t) \geq \left\lceil \frac{t + D_i}{T_i} \right\rceil C_i \geq \left\lceil \frac{t + D_i - C_i}{T_i} \right\rceil C_i \geq W_i(t) \tag{42}$$

Due to the fact that $\forall b \in \mathbb{N}$, $\lfloor b \rfloor + 1 \geq \lceil b \rceil$ and noting

$\left\lceil \frac{t-D_i}{T_i} \right\rceil \geq 1$, $\forall t \geq D_i$, we obtain $\forall t \geq D_i$

$$3dbf_i^C(t) = 3C_i(\left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1) \geq 3C_i \left\lceil \frac{t-D_i}{T_i} \right\rceil \tag{43}$$

$$\geq C_i \left\lceil \frac{t-D_i}{T_i} \right\rceil + 2C_i = \hat{W}_i(t) \tag{44}$$

Observing the above inequalities, we can conclude the proof. □