Release Enforcement in Resource-Oriented Partitioned Scheduling for Multiprocessor Systems

Georg von der Brüggen[1], Jian-Jia Chen[1], Wen-Hung Huang[1], and Maolin Yang[2]
[1]Department of Informatics, TU Dortmund University, Germany
[2]University of Electronic Science and Technology of China, China

Abstract

When partitioned scheduling is used in real-time multiprocessor systems, access to shared resources can jeopardize the schedulability if the task partition is not done carefully. To tackle this problem we change our view angle from focusing on the computing tasks to focusing on the shared resources by applying *resource-oriented partitioned scheduling*. We use a release enforcement technique to shape the interference from the higher-priority jobs to be sporadic, analyze the schedulability, and provide strategies for partitioning both the critical and the non-critical sections of tasks onto processors individually. Our approaches are shown to be effective, both in the evaluations and from a theoretical point of view by providing a speedup factor of 6, improving previously known results.

1 Introduction

In real-time systems timeliness has to be achieved in addition to functional correctness, i.e., calculations must not only be done correctly but the results must be delivered within a certain amount of time. In uniprocessor systems mutual exclusion and synchronization based on *priority inheritance* techniques have been well studied. Examples are the priority ceiling protocol (PCP) [27], the priority inheritance protocol (PIP) [27], and the stack resource policy (SRP) [3].

When real-time tasks are scheduled on multiprocessor platforms three paradigms are widely adopted: partitioned, global, and semi-partitioned scheduling. While partitioned scheduling partitions the tasks statically among the available processors the global scheduling approach allows a job to be migrated freely. Under the semi-partitioned scheduling approach each task may be statically divided into subtasks and each (sub)task is then assigned to a processor statically. A comprehensive survey of multiprocessor scheduling in real-time systems can be found in [18].

Resource sharing and synchronization in multiprocessor systems lead to additional synchronization overhead. The existing multiprocessor resource sharing protocols like the Multiprocessor Priority Ceiling Protocol (MPCP) (based on suspension locks) by Rajkumar [27] and the Multiprocessor resource sharing Protocol (MrsP) (with spin locks) by Burns and Wellings [12] ensure mutual exclusion, assuming a given task partition. However, it was shown in [10] that the number of priority-inversion blockings (*pi*-blockings) can be lower bounded by the number of processors in the worst case. This means that the advantages of reducing the multiprocessor scheduling problem to uniprocessor subproblems under partitioned scheduling can be outweighed by the synchronization overhead if the partition is not done carefully. Some heuristics to find good task partitions have been proposed [24, 25, 37] but without theoretical analysis with regards to speedup factors.

An alternative approach, namely *resource-oriented partitioned scheduling* (ROP), was proposed by Huang et. al [23] in 2016 and is adopted here. As the shared resources are usually the bottlenecks, ROP changes the view angle and focus on the *shared resources* instead of the *computing tasks*. The spirit behind ROP is to first assign each shared resource to one designated *synchronization* processor, and the non-critical sections will be executed on other *application* processors, decoupled from the critical sections. By focusing on the resource access, we can try to keep the response times of the critical sections as short as possible. When considering the non-critical sections, the worst-case response time of those critical sections can be seen as suspension time of *self-suspending tasks* on a uniprocessor due to the use of partitioned scheduling. ROP focuses on the strategy of task and resource partitioning and is in general compatible with any suspension-based locking protocols extended from the uniprocessor PCP (i.e., DPCP in [29]), SRP, FIFO (i.e., DFLP in [7]), and priority-based non-preemptive scheduling.

Huang et. al [23] provided a general exploration of ROP under *fixed-priority* scheduling. To keep their model general, they assumed that the execution pattern of a task is not known and can change from one job to another, i.e., similar to the *dynamic self-suspension task model.*

Contributions: We analyze how the schedulability under ROP can be improved by using release enforcement for the task migration, assuming that the tasks can be modelled according to the *segmented self-suspension model.* Our work is restricted to a very fundamental and yet challenging special case where each task has one non-nested critical section and the self-suspension time is reduced by the introduced release enforcement in both critical and non-critical sections.

- Due to the recent development of scheduling and analyzing segmented self-suspending real-time tasks,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
RTNS '17, October 4–6, 2017, Grenoble, France,
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5286-4/17/10…$15.00
DOI: https://doi.org/10.1145/3139258.3139287

we explain how these scheduling algorithms and analyses, e.g., [22, 26, 35], can be jointly applied with ROP in Section 4. This results in a family of possible algorithms, which adopt *fixed-priority* scheduling for the processors that execute *critical sections*, and either *fixed-priority* or *dynamic-priority* scheduling for the processors that execute only *non-critical sections*.

- In Section 5, we show that the rate-monotonic priority assignment together with PCP under release enforcement and ROP has a speedup factor of 6 with respect to a necessary scheduling condition, improving the best previously known result $11 - 6/(m+1)$ by Huang et. al [23].
- Based on release enforcement under ROP, we explore 8 different algorithms, combining four different approaches for scheduling non-critical sections and two approaches for scheduling critical sections. These are shown effective against the state-of-the-art multiprocessor synchronization scheduling algorithms and their analyses in the evaluations in Section 6.

Note that we do not intend to compare the performance of the locking protocols in this paper. We refer the readers to [39] for a survey and detailed comparisons of the global scheduling protocols that will be shortly summarized in Section 3.2. We show that a simple combination of release enforcement and ROP can take care of task partitioning and priority assignment directly and yield good performance both *theoretically and empirically*. For comparing with the other lock-based protocols that are shortly summarized in Section 3.2 for partitioned or semi-partitioned scheduling, reasonable task partitioning or priority assignments have to be provided. We will detail the settings in Section 6.

2 System Model and Notation

In this section we will introduce the basic task model and general notation we use in this paper. We consider a set of n independent sporadic tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ in a multiprocessor platform, consisting of $m \geq 2$ identical processors $\wp = \{\wp_1, \wp_2, .., \wp_m\}$ and r mutually exclusive shared resources $\mathcal{RS} = \{R_1, R_2, ..., R_r\}$.

Task model: Each sporadic task is characterized as $\tau_i = (C_i, A_i, T_i, D_i)$, where C_i is the upper bound on the amount of execution time without resource access, called *non-critical-section* execution time, and A_i is the upper bound on the amount of execution time during resource access, called *critical-section* execution time. The worst-case execution time (WCET) of task τ_i is $WCET_i = C_i + A_i$, i.e., it includes critical-sections and non-critical-sections. Each task releases an infinite number of task instances (also called jobs) where two job releases are separated at least by the given minimum inter-arrival time constraint T_i, i.e., if a job of task τ_i arrives at time θ_a, the next instance of the task must arrive at $\theta_a + T_i$ or later. The relative deadline of task τ_i is denoted with D_i, i.e., a job at θ_a must finish up to $C_i + A_i$ units of execution time before $\theta_a + D_i$. Furthermore, we assume all tasks have to be executed sequentially, i.e., a task instance can only run at one processor at any time, and that when a job requests a shared resource it can not continue its execution until the shared resource is granted. We consider implicit-deadline task sets in this paper, i.e., $D_i = T_i \; \forall \tau_i$.

Shared resources: In general, shared resources can be in-memory data, e.g., a set of variables, or external objects, like files, database connections, and network connections. To prevent *race conditions*, shared resources are accessed mutually exclusively, i.e., for each shared resource R_j it is not possible that two jobs are both in a critical section that accesses the same resource R_j at the same time. We focus on *logical* shared resources, i.e., a piece of code executed on processors. Hence, we assume that shared resources are not processor-specific. The jobs of any task may request exclusive access to any of the shared resources $R_1, R_2, ..., R_r$. Furthermore, we assume the access to a shared resource to be non-preemptive from other accesses to the same resource, i.e., once a task τ_i accesses a shared resource R_j no other task is allowed to access R_j until τ_i finishes the execution on R_j.

Execution pattern: We restrict ourselves to the case where each job of each task accesses only one shared resource at most once. Therefore, the critical sections are not nested by definition. We assume a given execution pattern. Specifically, for each τ_i, we assume to know the maximum amount of execution time before the critical section, denoted as $C_{i,1}$, the maximum amount of execution time inside the critical section, denoted as A_i, and the maximum amount of execution time after the critical section, denoted as $C_{i,2}$. We assume this pattern is precisely known, i.e., $C_{i,1} + C_{i,2} = C_i$. This assumption can be relaxed by applying Hybrid Self-Suspension Models proposed by von der Brüggen et al. [34].

Further notation: The utilization of task τ_i regarding execution of non-critical sections is defined as $U_i^C = C_i/T_i$ while the total critical section utilization of task τ_i is denoted by $U_i^A = A_i/T_i$. Therefore, the utilization of task τ_i is denoted as $U_i = (C_i + A_i)/T_i$, assuming that $U_\Sigma = \sum_{i=1}^{n} U_i \leq m$ as otherwise a feasible schedule is not possible. The utilization of τ_i on resource R_q is $U_i^{R_q}$, i.e., $U_i^{R_q} = U_i^A$ if τ_i accesses R_q and 0 otherwise. The total utilization of R_q is $U^{R_q} = \sum_{\tau_i \in \tau} U_i^{R_q}$; the total utilization of non-critical-sections is $U^C = \sum_{\tau_i \in \tau} U_i^C$; the total utilization of shared resources is $U^{\mathcal{RS}} = \sum_{R_q \in \mathcal{RS}} U^{R_q}$. $WCRT(C_{i,1})$, $WCRT(C_{i,2})$, and $WCRT(A_i)$ denote the worst-case response time of the related subtask (under the considered scheduling algorithm).

Schedulability and Priorities: A system τ is called *feasible* under a scheduling algorithm A, if the schedule produced by A ensures that the task set is scheduled without any deadlines misses. A *schedulability test* of a scheduling algorithm verifies whether the task system is feasible under the given algorithm. A schedulability test is sufficient, if all the task sets it deems schedulable are in fact schedulable. A necessary schedulability test deems all the task sets unschedulable that are unschedulable; and an exact test is both necessary and sufficient. We consider both fixed-priority scheduling (FP) and dynamic-priority scheduling (DP) in this paper. If FP is used, we assume that each task has a unique priority, and

$hp(\tau_k)$ and $lp(\tau_k)$ are the sets of higher-priority and lower-priority tasks than task τ_k, respectively. Specifically, we will always use FP for scheduling the critical sections.

Speedup Factors: We use the *speedup factor* to quantify the sub-optimality of our scheduling algorithm: A scheduling algorithm (schedulability test, respectively) has a speedup factor $\rho \geq 1$, if it is guaranteed that any task system that is feasible upon a specified platform can be scheduled (deemed schedulable by the test, respectively) upon a platform in which each processor is speed up by at least ρ. Chen et al. [15] have recently presented the potential pitfalls of arguments based on the speedup factors. We note that our scheduling algorithm and statements do not have any of those pitfalls.

3 Background and Related Work

3.1 Single Processor Systems

As shared resources must be serially executed to achieve mutual exclusion, the execution of critical sections inevitably causes some delay due to *priority inversion*, i.e., a task is prevented from executing due to another task with a lower priority that holds a shared resource, also called *pi-blocking*. Three approaches are considered here:

Non-Preemptive Protocol (NPP): A critical section that has started to be executed cannot be preempted by any other job until the critical section is finished. Under fixed-priority NPP the maximum blocking time B_k for a task τ_k is

$$B_k = \max_{\tau_i \in lp(\tau_k)} \{A_i\} \tag{1}$$

NPP can also be applied by using an FIFO queue or non-preemptive EDF with a different response time analysis.

Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP): To avoid unnecessary blocking of high-priority tasks due to unrelated shared resources, the PIP and PCP were introduced by Sha et. al [30]. The PIP allows a lower-priority task to temporally inherit the priority of a higher-priority task that it blocks. In the PCP, each resource R_q is assigned a priority ceiling $C(R_q)$ that is equivalent to the base priority of task τ_j of the highest-priority task that accesses R_q. A job can only allocate a resource, if its priority is higher than the highest priority ceiling among the currently allocated resources. Suppose that $\mathbf{L}_k$ is a subset of $lp(\tau_k)$, in which the resource ceiling of the shared resource requested by a task τ_i in $\mathbf{L}_k$ is higher than or equal to the priority of τ_k. Under PCP, as shown in [30],

$$B_k = \max_{\tau_i \in \mathbf{L}_k} \{A_i\}. \tag{2}$$

Stack Resource Policy (SRP): The SRP is also a classical uniprocessor locking protocol proposed by Baker [3]. The worst-case response time of a job can incur under the SRP is the same as under the PCP. Therefore, the results in this paper based on the PCP can also be applied to the SRP.

3.2 Multiprocessor Systems

Multiprocessor real-time locking protocols can be classified into suspension-based protocols [9, 10, 27, 29] and spin-based protocols [12, 21, 36]. From an algorithmic optimality point of view, there are two categories for quantifying multiprocessor real-time locking protocols. One is from the speedup factor perspective. For multiprocessor scheduling with resource sharing the first algorithm with a speedup factor, i.e., $12(1 + 3r/4m)$, was *gEDF-vpr* by Andersson and Easwaran [1]. This bound was improved by Andersson and Raravi [2], who proposed *LP-EE-vpr* which has a speedup factor of $4 \cdot (1 + \lceil \frac{r}{m} \rceil) \geq 8$. Note that we simplified the bound for *LP-EE-vpr* to match the case we analyze here while the bound in [2] is more general. The best known general bound was presented by Huang et al. [23] for their algorithm *ROP-PCP*, i.e., $11 - \frac{6}{m+1}$. These speedup factors are only valid when there is at most one (non-nested) critical section per task.

Another theoretical perspective is to analyze the pi-blocking. Brandenburg and Anderson [10] showed that $\Omega(m)$ pi-blocking is unavoidable under suspension-oblivious schedulability analysis. To this end, the Flexible Multiprocessor Locking Protocol (FMLP) [6], the Generalized FIFO Multiprocessor Locking Protocol (FMLP$^+$) [9], $O(m)$ multiprocessor locking protocol (OMLP) [11], and the Distributed FIFO Locking Protocol (DFLP) [8] are proved to be asymptotically optimal for minimizing the pi-blocking by using FIFO-waiting queues. However, the empirical results in [39] showed that asymptotically optimal protocols do not necessarily perform well. Yang et al. [39] have summarized and compared the protocols using global scheduling, and concluded that the FMLP and the Priority Inheritance Protocol (PIP) are the best of the existing protocols under global rate-monotonic scheduling when the linear-programming (LP) based schedulability tests in [7] are used.

For partitioned and semi-partitioned scheduling, there are several real-time locking protocols such as the Distributed PCP (DPCP) [29], the Multiprocessor PCP (MPCP) [27], the Multiprocessor Resource Stack Policy (MSRP) [21], the Flexible Multiprocessor Locking Protocol (FMLP) [6], and the Multiprocessor resource sharing Protocol (MrsP) [12]. Essentially, the performance of resource sharing protocols highly depends on how the tasks are partitioned. With regard to task partitioning, the following results have been reported: 1) a synchronization-aware partitioned heuristic tailored to the MPCP in [24], and 2) a blocking-aware partitioning method in [25]. Unfortunately, unsafe schedulability tests were adopted in the above results. Please refer to [38] for details. Wieder and Brandenburg [37] used integer linear programming and developed a Greedy Slacker (GS) algorithm to partition the tasks under the MRSP protocol.

4 Our ROP Scheduling

This section first explains the general concept of resource-oriented partitioned scheduling (ROP) and then presents our extension of ROP for the special case when each task has at most one non-nested critical section.

4.1 ROP by Huang et al. [23]

The general ROP approach is: 1) The m processors are partitioned into m^R *synchronization* processors for critical

sections and m^C *application* processors for non-critical sections, 2) The related critical sections of each shared resource are assigned to one designated *synchronization* processor, 3) The *non-critical section* of each task is statically allocated onto a designated *application* processor. When a job enters a critical section, it suspends itself on its application processor and returns to the ready queue of the application processor after its critical section is executed on the synchronization processor. Note that both critical and non-critical sections of a task may still be executed on the same processor as the remaining capacity on the synchronization processors can be used to execute non-critical sections. As a task is possibly executed on more than one processor, resource-oriented partitioned scheduling has additional overheads, similar to *semi-partitioned* scheduling, when compared to partitioned scheduling. The critical points for designing a good algorithm based on ROP are:

1. the *number of synchronization processors*,
2. regarding the critical sections, the *partition of the shared resources* on those synchronization processors,
3. regarding the non-critical sections, the *partition of the sporadic real-time tasks* onto application processors,
4. the *assigned base priorities* for the sporadic tasks, and
5. the *moment* on the synchronization processor to *request* the critical section of a task.

Huang et. al [23] only considered the first four points and assumed that task migration is possible at any time, due to the use of the more general dynamic self-suspension model.

4.2 Release Enforcement

The main idea of ROP is to separate the critical and non-critical sections by migrating all critical sections for the same resource to one processor. After that the schedulability of the critical sections and the non-critical sections can be analyzed individually. Therefore, executing the critical section of a task on the synchronization processor can be considered as if the task suspends itself from its application processor. Please note that the literature regarding self-suspension has been seriously flawed as reported in [14]. However, the techniques used in this paper have none of the flaws reported in [14]. A current review of the state-of-the-art regarding self-suspension can be found in [16].

From the perspective of the shared resources, the critical sections migrating to the synchronization processor can be modeled as incoming sporadic tasks. However, if the task migration happens directly when $C_{k,1}$ finishes its execution, the time difference between the best-case and the worst-case response time of $C_{k,1}$ has to be taken into account as release jitter, e.g., detailed in [38] for the original DPCP. This jitter introduces pessimism when analysing the WCRT of the critical section and can be removed by enforcing the critical sections to be periodic, i.e., migrating the task τ_k to the synchronization processor at the fixed time $t_{k,1}^{migr}$ after the task is released on the application processor. This does not mean that the task migration is enforced to be periodic but that the critical section is only considered by the scheduler at time $t_{k,1}^{migr}$ plus the arrival time of the job. The same holds true for the non-critical sections if the task is migrated back according to the WCRT of the critical section. We use $WCRT(A_k)$ on the synchronization processor as suspension time directly, i.e., the task is migrated back after exactly that amount of time. Therefore, we have release enforcement for both migrations and there is no release jitter for $C_{k,2}$ as well.

This approach is also called *phase modification* (PM) in [5, 31] and *static offset* in [26] in the literature. Due to the strict release enforcement of the release times of computation demands $C_{k,1}$, A_k, and $C_{k,2}$, we do not need to consider any release jitter. This enforcement is different from other enforcement strategies with similar names, i.e., the period enforcer in [28] and the release guard in [31], in which the release time of A_k ($C_{k,2}$, respectively) of task τ_k should be at least T_k apart from A_k ($C_{k,2}$, respectively) of the previous job of τ_k. The period enforcer [28] has recently been shown incompatible with all existing analysis regarding suspension-based locking protocols by Chen and Brandenburg in [13].

Under release enforcement, the timing analysis is equivalent to *end-to-end* deadline analysis, e.g., [5], or the *static-offset* FP uniprocessor analysis, e.g., [26]. However, our focus here is to partition and schedule those tasks. We will apply existing safe timing analysis and scheduling algorithms based on self-suspension, detailed in Section 4.3. Other approaches that can be used to assign relative deadlines and validate $WCRT(C_{k,1}) + WCRT(A_k) + WCRT(C_{k,2}) \leq T_k$ under release enforcement can be applied.

4.3 Schedulability Tests under Release Enforcement

Under ROP, the schedulability on each processor can be analyzed individually. While the actual mappings of tasks and resources will be described in Section 4.4, we focus on the the scheduling decisions for the individual processors and the schedulability tests in this subsection. Therefore, we will assume a mapping of shared resources and tasks onto processors to be given. First, we consider the scheduling regarding critical sections and the resulting response time on a synchronization processor, determining the maximum suspension time S_k for τ_k regarding the application processors. After that, we use this suspension time to analyze the schedulability on the application processor. We set individual deadlines $D_{k,1}$ and $D_{k,2}$ for the first and second computation segments, respectively. Under the release enforcement, a task τ_k is schedulable by a scheduling algorithm if 1) $D_{k,1} + S_k + D_{k,2} \leq T_k$, 2) $WCRT(C_{k,1}) \leq D_{k,1}$ and $WCRT(C_{k,2}) \leq D_{k,2}$, and 3) the suspension time is bounded by S_k, i.e., $WCRT(A_k) \leq S_k$. As we use them for the preplanned migration, $D_{i,1}$, S_i, and $D_{i,2}$ will also be set for FP. *For the simplicity of presentation, we will say that task τ_i migrates to its synchronization processor $D_{i,1}$ time units after a job of task τ_i arrives and task τ_i migrates to its application processor $S_i + D_{i,1}$ time units after a job of task τ_i arrives.*

We first assume that each processor is either a synchronization processor or an application processor and consider the case that the critical and the non-critical sections are

scheduled on the same processor afterwards. We will always consider the schedulability of task τ_k, assuming that the schedulability of the tasks that are previously assigned on the same processor is already assured. The task set is deemed schedulable if all tasks are schedulable.

For τ_k the set of tasks placed on the same synchronization processor as the critical section and the set of tasks placed on the same application processor as the non-critical section are not necessarily identical. In addition, the priority ordering of tasks on a synchronization processor is not necessarily the same as on an application processors. Therefore, we introduce the following notation:

- regarding critical sections: $hps(\tau_k)$ denotes the tasks with higher priority than τ_k on the synchronization processor, i.e., fixed-priority scheduling (FP) is used.
- regarding non-critical sections: $hpa(\tau_k)$ denotes the tasks with higher priority than τ_k on the application processor if FP is used.

4.3.1 Critical Section Response Time Analysis

For each synchronization processor, the task priorities are assigned according to RM. Due to release enforcement, the inter-arrival time of a task τ_i on the synchronization processor is at least T_i. Therefore, the following time-demand analysis (TDA) in [17, 33] can be safely used to test whether the response time of A_k is no more than T_k:

$$\exists t,\ \ 0 < t \leq T_k \text{ and } B_k + A_k + \sum_{\tau_i \in hps(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil A_i \leq t \qquad (3)$$

If Eq. (3) holds for some values of t, we can take the minimum $t^*_{k,s}$ among those values as the maximum suspension time S_k. Note that B_k is calculated depending on the resource sharing policy, i.e., Eq. (1) for NPP and Eq. (2) for PCP.

4.3.2 Scheduling Analysis for Non-Critical Sections

We need to validate if $WCRT(C_{k,1}) + S_k + WCRT(C_{k,2}) \leq T_k$ to determine whether τ_k is deemed to be schedulable under the scheduling policy on the application processor. Note that $S_k = WCRT(A_k)$ was determined beforehand (see Section 4.3.1). We set individual deadlines $D_{k,1} \geq WCRT(C_{k,1})$ and $D_{k,2} \geq WCRT(C_{k,2})$ for $C_{k,1}$ and $C_{k,2}$, respectively, with $D_{k,1}+S_k+D_{k,2} = T_k$. While those deadlines are not necessary when scheduling tasks with FP, they are used for both FP and DP to determine the point in time where the migration takes place, i.e., the migration to the synchronization processor happens at $\theta_a+D_{k,1}$ and the migration back to the application processor at $\theta_a + D_{k,1} + S_k$ where θ_a is the jobs arrival time.

When considering τ_k, we assume that $D_{i,1}$ and $D_{i,2}$ are already assigned for all tasks τ_i that are already allocated to the processor. As recently shown in [22, 35], those tasks can be modeled as general multiframe (GMF) tasks [4] with two frames, i.e., τ_i is represented by two 3-tuples $\tau_i = \{(C_i^1, D_i^1, T_i^1), (C_i^2, D_i^2, T_i^2)\}$, representing two alternately released subtasks. The computation time for the GMF subtasks is the same as for the computation segments, i.e., $C_i^1 = C_{i,1}$ and $C_i^2 = C_{i,2}$. As the second computation segment is released after the suspension interval we know that $D_i^1 = D_{i,1}$ and $T_i^1 = D_{i,1} + S_i$. Moreover, $T_i^2 = T_i - T_i^1 = T_i - D_{i,1} - S_i$ and $D_i^2 = D_{i,2} = T_i - D_{i,1} - S_i$.

Fixed-priority - FRD - Rate Monotonic (FP-RM): The task priorities on each application processor are assigned in RM order. We determine $WCRT(C_{k,1}) = t^*_{k,1}$ and $WCRT(C_{k,2}) = t^*_{k,2}$ by looking for the minimum t such that

$$0 < t \leq T_i \ \text{ and } \ C_{k,1} + \sum_{\tau_i \in hpa(\tau_k)} W_i(t) \leq t \qquad (4)$$

holds, where $W_i(t)$ is the maximum interference of τ_i over the interval $[0, t)$. It was shown in [32] that $W_i(t)$ can be calculated as the maximum $\max\left\{E_i^h(t)\right\}$ of the interference patterns $E_i^h(t)$ for $h \in \{1, 2\}$ where $E_i^h(t) = \sum_{j=h}^{h+l+1} C_i^{\{j \bmod 2\}}$ and l is the mimimum integer with $\sum_{j=h}^{h+l+1} T_i^{\{j \bmod 2\}} \geq t$. Note that the interfernce from $hpa(\tau_k)$ calculated above is identical to that based on the static offset analysis in [26]. Similarly, $t^*_{k,2}$ can be determined using $C_{k,2}$ in Eq. (4).

If $t^*_{k,1} + t^*_{k,2} < T_k - S_k$, the slack can be freely distributed when setting the deadlines $D_{k,1}$ and $D_{k,2}$. We used an equal density assignment, i.e., $\frac{C_{k,1}}{D_{k,1}+S_k} = \frac{C_{k,2}}{D_{k,2}}$, with respect to $t^*_{k,1}$ and $t^*_{k,2}$. This means, if $D_{k,1}$ would be less than $t^*_{k,1}$ we set $D_{k,1} = t^*_{k,1}$ and adjust $D_{k,2}$ accordingly (similar for $D_{k,2}$).

Fixed-priority - Execution Interval Monotonic (FP-EIM): The only difference to FP-RM is that tasks are considered and prioritized in increasing order according to their execution interval $T_i - S_i$.

Earliest Deadline First - Fixed Relative Deadline Assignment - EIM (EDF-EIM): The first DP approach we consider is to use SEIFDA by von der Brüggen et al. [35] on the individual processors. The renaming to EDF-EIM is to match the terminology in this paper. For each task individual relative deadlines are assigned for the two computation segments using the demand-bound functions presented in [35]. Afterwards the subjobs are scheduled accordingly using EDF. As the deadline assignment strategy plays a big role for EDF-EIM we use the strategy that performs the best according to [35], i.e., Proportionally-Bounded-Min. Note that when trying to assign the deadlines for τ_k we also need to check weather the previously assigned tasks are still schedulable under that assignment, as one of the segments of τ_k could be assigned to a shorter deadline than one of the segments of the previously assigned tasks. To reduce the computational complexity we use approximated demand-bound functions for the GMF tasks as proposed in [35].

EDF - FRD - RM (EDF-RM): The only difference to EDF-EIM is that tasks are assigned in RM instead of EIM order.

4.3.3 Improvements for Non-Critical Sections

Allowing non-critical sections to be executed on the synchronization processors may increase the schedulability as it uses otherwise unused capacities. We let the critical sections have higher priority than all non-critical sections and use FP for the non-critical sections as well, even if DP is used on the processors that are only used for non-critical sections.

4.4 Resource and Task Allocation

Suppose we use a given number of $m^R \geq 1$ synchronization processors and $m^C = m - m^R$ application processors. Then, we proceed with the following three steps, which are a revised version of the algorithm by Huang et al. [23] as different tests and scheduling algorithms are applied:

(1) **Assign tasks to synchronization processors**: We assign the resources to the given number of synchronization processors using the *Worst-Fit Decreasing* (WFD) algorithm based on resource utilization, i.e., the resources are ordered non-increasingly according to U^{R_q}, for $q = 1, 2, \ldots, r$. WFD assigns the resource onto the synchronization processor with the *least* resource utilization before assigning the resource.

(2) **Calculate WCRT on synchronization processors**: For each task τ_k, we calculate $WCRT(A_k)$ by using Eq. (3), considering the m^R processors individually. The blocking time is calculated with either PCP (Eq. (2)) or NPP (Eq. (1)), depending on the synchronization protocol. This is treated as the suspension time $S_k = WCRT(A_k)$ for each task τ_k.

(3) **Assign tasks to application processors**: The tasks are sorted increasingly according to the scheduling approach for the application processors, i.e., either according to their execution intervals $T_i - S_i$ if EDF-EIM or FP-EIM is used, or according to their periods if EDF-RM or FP-RM is used. The non-critical sections of the tasks are then assigned to the application processors according to the first-fit approach in this order, using the related schedulability test. If the related schedulability condition in Section 4.3.2 holds, the deadlines are set accordingly and the non-critical sections of the task are assigned to the processor. If a task is not schedulable on any application processor we try to assign it to the synchronizations processors.

The key question in this algorithm is the setting of m^R. In general, there is a trade-off between 1) longer suspension times if m^R is small, and 2) longer worst-case response times of the non-critical sections if m^R is big. Since the above assignment algorithm works for any $m^R \leq m$, trying all possible settings of m^R in the above algorithm only increases the time complexity by a factor of m. This procedure is, hence, done for all sensible numbers of synchronization processors, i.e., $m^R \in \{1, ..., min(m, r)\}$. We show in the next section that setting $m^R = \max\left\{\left\lfloor 6 \sum_{\tau_i \in \tau} U_i^A \right\rfloor, 1\right\}$ can lead to a speedup factor of 6 when FP-RM is used together with PCP and release enforcement. However, we would like to note that an algorithm that sets $m^R = \max\left\{\left\lfloor 6 \sum_{\tau_i \in \tau} U_i^A \right\rfloor, 1\right\}$ greedily results in a *potential pitfall* of significant performance loss since such a setting of m^R (i.e., as an enforcement technique) is *too strong and applied at an early stage*, as recently pointed out by Chen et al. [15].

5 Speedup Factors

In this section we show that using release enforcement together with resource-oriented partitioned scheduling leads to a speedup factor of 6. We start with the necessary scheduling conditions for a task set to be feasible by any multiprocessor scheduling algorithm shown in Lemma 3 in [23].

Lemma 5.1 (Necessary Condition, Lemma 3 in [23])**.** *Any implicit-deadline task system τ that is feasible upon a platform comprised of m processors must satisfy the following conditions*

$$U^C + U^{\mathcal{RS}} \leq m \tag{5}$$

$$\forall \tau_i \in \tau, \quad U_i \leq 1 \tag{6}$$

$$\forall \tau_k \in \tau \qquad \max_{\tau_i \in ld_{R_q}(\tau_k)} A_i + A_k + \sum_{\tau_j \in sd_{R_q}(\tau_k)} \left\lfloor \frac{T_k}{T_j} \right\rfloor A_j \leq T_k \tag{7}$$

where $ld_{R_q}(\tau_k)$ and $sd_{R_q}(\tau_k)$ are the sets of tasks that access the same shared resource R as τ_k but with longer ($T_i > T_k$) periods and shorter or the same ($T_i \leq T_k$) periods, respectively.

Lemma 5.2. *Following the necessary condition in Eq.* (7) *the blocking time B_k of a task τ_k derived under resource-oriented partitioned scheduling with PCP must be upper bounded by its period T_k when the tasks are prioritized by using RM.*

Due to Eq. (7), $U^{R_q} \leq 1$ must hold $\forall\ R_q \in \mathcal{RS}$. The proof of Lemma 5.2 was provided by Huang et. al in [23] as a part of the proof of their Lemma 5. For the speedup analysis, we first provide the following lemma regarding the worst-case response time of a computation segment E_k if the utilization on the related processor is low enough.

Lemma 5.3. *Let $hp^*(E_k)$ be the periodic computation segments with higher priority than E_k on the same processor. Suppose the worst-case response time (WCRT) analysis for a computation segment E_k is to find the minimum $t > 0$ where*

$$E_k + \sum_{\tau_i \in hp^*(E_k)} \left\lceil \frac{t}{T_i} \right\rceil E_i = t \tag{8}$$

If $T_i \leq T_k \forall \tau_i \in hp^*(E_k)$ and

$$\left(\sum_{\tau_i \in hp^*(\tau_k)} E_i/T_i \right) + E_k/T_k = Y \leq 0.5 \tag{9}$$

then $WCRT(E_k)$ under Eq. (8) *is at most $T_k \cdot Y$.*

Proof. Suppose V_i is $\frac{E_i}{T_i}$. We consider two cases:

$$1) T_i \leq \frac{T_k}{2} \qquad \Rightarrow \left\lceil \frac{t}{T_i} \right\rceil E_i \leq T_k V_i \qquad \forall 0 < t \leq \frac{T_k}{2} \tag{10}$$

$$2) \frac{T_k}{2} < T_i \leq T_k \quad \Rightarrow \left\lceil \frac{t}{T_i} \right\rceil E_i = E_i \leq T_k V_i \quad \forall 0 < t \leq \frac{T_k}{2} \tag{11}$$

Therefore, we know that for all t with $0 < t \leq \frac{T_k}{2}$:

$$E_k + \sum_{\tau_i \in hp^*(E_k)} \left\lceil \frac{t}{T_i} \right\rceil E_i \leq E_k + \sum_{\tau_i \in hp^*(E_k)} V_i T_k = Y T_k \tag{12}$$

This means that Eq (8) holds when $t = YT_k$. □

Note that we will apply Lemma 5.3 for analyzing $WCRT(C_{k,1})$, $S_k = WCRT(A_k)$, and $WCRT(C_{k,2})$ by putting different formula in Eqs. (8) and (9). For the simplicity of presentation in the following statements, we will implicitly assume that the task set can be feasibly scheduled on the

original platform and therefore the necessary conditions in Lemmas 5.1 and 5.2 hold. Moreover, our goal here is to prove that a specific setting when the platform speed is 6. *For the rest of the proofs in this section, all the execution times, blocking times, utilization values, and analyses are based on the platform after speeding up by* 6.

Lemma 5.4. *If $S_k + T_k(U_k^C + 2\sum_{\tau_i \in hpa(\tau_k)} U_i^C) \leq T_k$ and $U_k^C + \sum_{\tau_i \in hpa(\tau_k)} U_i^C \leq 0.5$, the worst-case response time of task τ_k (under release enforcement, RM preemptive scheduling, and resource-oriented partitioned scheduling) is*

$$WCRT(\tau_k) \leq S_k + T_k\Big(U_k^C + 2\sum_{\tau_i \in hpa(\tau_k)} U_i^C\Big) \qquad (13)$$

Proof. Under RM preemptive scheduling and release enforcement, $WCRT(C_{k,1})$, i.e., the *offset* to release the critical section to its synchronization processor, is to find the minimum $t > 0$ such that Eq. (4) holds. Since $W_i(t)$ defined for Eq. (4) is $\leq \left\lceil \frac{t}{T_i} \right\rceil (C_{i,1} + C_{i,2})$ for task τ_i, we can safely approximate $WCRT(C_{k,1})$ by finding the minimum $t > 0$ with $C_{k,1} + \sum_{\tau_i \in hpa(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i = t$. Due to the assumption $\frac{C_{k,1}}{T_k} + \sum_{\tau_i \in hpa(\tau_k)} U_i^C \leq U_k^C + \sum_{\tau_i \in hpa(\tau_k)} U_i^C \leq 0.5$, we know that the condition in Eq. (9) is met with $E_k = C_{k,1}$ for τ_k and $E_i = C_i$ for $\tau_i \in hpa(\tau_k)$. Therefore, by applying Lemma 5.3,

$$WCRT(C_{k,1}) \leq C_{k,1} + T_k \sum_{\tau_i \in hpa(\tau_k)} U_i^C = D_{k,1} \qquad (14)$$

Similarly, $WCRT(C_{k,2}) \leq C_{k,2} + T_k \sum_{\tau_i \in hpa(\tau_k)} U_i^C = D_{k,2}$. Therefore, $WCRT(\tau_k) \leq S_k + T_k(U_k^C + 2\sum_{\tau_i \in hpa(\tau_k)} U_i^C)$ if $S_k + T_k(U_k^C + 2\sum_{\tau_i \in hpa(\tau_k)} U_i^C) \leq T_k$. □

Lemma 5.5. *Under FP-RM-PCP with release enforcement on a platform with m homogeneous processors of speed 6 and the number of synchronization processors set to $m^R = \max\left\{\left\lfloor 6\sum_{\tau_i \in \tau} U_i^A \right\rfloor, 1\right\}$, the maximum response time S_k of a task on a synchronization processor is at most*

$$S_k \leq \begin{cases} (1/6 + \sum_{\tau_i \in \tau} U_i^A)T_k & \text{if } m^R = 1 \\ 0.5T_k & \text{if } m^R \geq 2 \end{cases} \qquad (15)$$

when the resources are packed according to the worst fit.

Proof. By Lemma 5.2, when RM is used together with PCP for scheduling we know that $B_k/T_k \leq 1/6$ after speeding up.

When m^R is 1, we know that $\sum_{\tau_i \in \tau} U_i^A < 1/3$ and all critical sections in τ are assigned to one processor. Due to the release enforcement, the worst-case response time S_k of the critical section of task τ_k is to find the minimum t such that Eq. (3) holds, i.e., $B_k + A_k + \sum_{\tau_i \in hps(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil A_i = t$. Since $B_k/T_k \leq 1/6$ and $A_k/T_k + \sum_{\tau_i \in hps(\tau_k)} U_i^A \leq \sum_{\tau_i \in \tau} U_i^A \leq 1/3$, we know that $Y = \frac{B_k + A_k}{T_k} + \sum_{\tau_i \in hps(\tau_k)} U_i^A \leq 0.5$, i.e., the condition in Eq. (9) holds when RM is used for prioritizing the critical sections, $E_k = A_k + B_k$ for task τ_k, and $E_i = A_i$ for task τ_i in $hps(\tau_k)$. Therefore, when m^R is 1, we know that $S_k \leq YT_k \leq (1/6 + \sum_{\tau_i \in \tau} U_i^A)T_k$ due to Lemma 5.3.

For the rest of the proof, we focus on $m^R \geq 2$. We only need to prove the total resource utilization of the critical sections on any of the m^R synchronization processors is $\leq 1/3$. Then, since $\frac{B_k + A_k}{T_k} + \sum_{\tau_i \in hps(\tau_k)} U_i^A \leq 1/6 + 1/3 = 0.5$ the same response time analysis used above when m^R is 1 can be directly applied to conclude $S_k \leq 0.5T_k$. We first consider how the resources are packed to the m^R synchronization processors at the platform with a speed of 6. Suppose that we are now assigning the q-th resource R_q. By definition, $q \leq r$. Let the resource utilization on a synchronization processor $\wp_\ell$ be denoted as $U^{\wp_\ell}$. Before assigning R^q to any of the m^R synchronization processors, there is one synchronization processor with the minimum utilization so far. We denote this processor as $\wp_j$. Due to the *worst-fit* strategy, $U^{\wp_\ell} \geq U^{\wp_j}$ for any synchronization processor $\wp_\ell$.

We show that the utilization of the resources assigned to $\wp_j$ (after assigning R_q to $\wp_j$) is always $\leq \frac{2}{6} = \frac{1}{3}$, i.e., $U^{\wp_j} + U^{R_q} \leq \frac{1}{3}$, at a platform with a speed 6. Assume for contradiction that $U^{\wp_j} + U^{R_q} > \frac{1}{3}$. Therefore, $U^{\wp_\ell} + U^{R_q} > \frac{1}{3}$, i.e., $U^{\wp_\ell} > \frac{1}{3} - U^{R_q}$ for any $\wp_\ell$ in the m^R synchronization processors. Putting the above information together, we have

$$\begin{aligned} \sum_{i=1}^{q} U^{R_i} &= U^{R_q} + \sum_{i=1}^{q-1} U^{R_i} = U^{R_q} + \sum_{\ell} U^{\wp_\ell} \\ &> U^{R_q} + m^R\left(\frac{1}{3} - U^{R_q}\right) = \frac{1}{3} + (m^R - 1)\left(\frac{1}{3} - U^{R_q}\right) \\ &\geq^{\dagger} \frac{1}{3} + (m^R - 1)\frac{1}{6} = \frac{m^R + 1}{6} \\ &= \frac{1}{6} \times \left(\left\lfloor 6\sum_{\tau_i \in \tau} U_i^A \right\rfloor + 1\right) >^{*} \sum_{\tau_i \in \tau} U_i^A = \sum_{i=1}^{r} U^{R_i} \geq \sum_{i=1}^{q} U^{R_i} \end{aligned}$$

where $\geq^{\dagger}$ is due to $m^R \geq 2$ and $U^{R_q} \leq 1/6$ as the platform speed is 6, and $>^{*}$ is due to the fact $\lfloor x \rfloor > x - 1$. Therefore, we reach a contradiction. As a result, the total resource utilization of the critical sections on any of the m^R synchronization processors is $\leq 1/3$, and $S_k \leq 0.5T_k$ for any task τ_k when $m^R \geq 2$. □

Theorem 5.6. *The speedup factor of the proposed resource-oriented partitioned scheduling algorithm is 6 if PCP is used to schedule the critical sections on the synchronization processors when $m \geq 2$, and the worst-fit approach is used to assign the critical sections to the synchronization processors and the non-critical sections are assigned in rate-monotonic order.*

Proof. Suppose that the input task set τ can be feasibly scheduled on m uni-speed processors. We need to show that in this case the task set is also schedulable by the *resource-oriented partitioned scheduling* on m processors with speed $s = 6$. For the analysis, we consider a special setting of m^R with $m^R = \max\left\{\left\lfloor 6\sum_{\tau_i \in \tau} U_i^A \right\rfloor, 1\right\}$ and $m^C = m - m^R$. We have to show that $WCRT(\tau_k) \leq T_k$ for any task τ_k in τ. Since RM is used for the priority assignment on the synchronization processors and the tasks are assigned to the application processors in RM order, Lemmas 5.4 and 5.5 can be implicitly applied if the required utilization condition can be satisfied. Two cases are considered:

Case 1: $m^R \geq 2$**.** That is, $\sum_{\tau_i \in \tau} U_i^A \geq \frac{2}{6}$. Moreover, the necessary condition in Eq. (5) leads to the following inequality after speeding up with a factor of 6:

$$\sum_{\tau_i \in \tau} (6U_i^C + 6U_i^A) \leq m \Rightarrow m^R + \sum_{\tau_i \in \tau} 6U_i^C \leq m^C + m^R \Rightarrow \sum_{\tau_i \in \tau} U_i^C \leq \frac{m^C}{6}$$

Therefore, when we consider to assign task τ_k to an application processor, there must be an application processor with utilization $\leq \frac{1}{6}$ due to the pigeon hole principle. Let this processor be $\wp_j$ and the set of the tasks that are already assigned on this processor be $hpa^j(\tau_k)$. Therefore, we know that $\sum_{\tau_i \in hpa^j(\tau_k)} U_i^C \leq 1/6$ and $U_k^C + \sum_{\tau_i \in hpa^j(\tau_k)} U_i^C \leq 1/3$. By Lemma 5.5, $S_k \leq 0.5T_k$, and by Lemma 5.4, we know that

$$WCRT(\tau_k) \leq S_k + \Big(U_k^C + 2 \sum_{\tau \in hpa^j(\tau_k)} U_i^C\Big)T_k \leq T_k \quad (16)$$

Case 2: $m^R = 1$. That is, $\sum_{\tau_i \in \tau} U_i^A < \frac{2}{6}$. By Lemma 5.5, $S_k \leq (1/6 + \sum_{\tau_i \in \tau} U_i^A)T_k$. We consider two subcases 1) $m = 2$ and 2) $m \geq 3$. When m **is 2**, we know that one processor is used for synchronization and another processor is used for non-critical sections. The necessary condition in Eq. (5) after speeding up with a factor of 6 for $m = 2$ leads to:

$$U_k^C + \sum_{\tau_i \in hpa(\tau_k)} U_i^C \leq \sum_{\tau_i \in \tau} U_i^C \leq \frac{m}{6} - \sum_{\tau_i \in \tau} U_i^A = \frac{1}{3} - \sum_{\tau_i \in \tau} U_i^A \quad (17)$$

Therefore, using Lemma 5.4 due to $U_k^C + \sum_{\tau_i \in hpa(\tau_k)} U_i^C \leq 1/3 < 0.5$ when $m = 2$, results in

$$\begin{aligned} WCRT(\tau_k) &\leq S_k + \Big(U_k^C + 2 \sum_{\tau \in hpa(\tau_k)} U_i^C\Big)T_k \\ &\leq \Big(\frac{1}{6} + \sum_{\tau_i \in \tau} U_i^A + \frac{2}{3} - 2 \sum_{\tau_i \in \tau} U_i^A\Big)T_k \leq T_k \end{aligned} \quad (18)$$

When m **is at least** 3, due to the pigeon hole principle, before assigning τ_k, there exists an application processor $\wp_j$ in the $m^C = m - 1$ application processors with utilization $\leq (\sum_{i=1}^{k-1} U_i^C)/(m-1) \leq (-U_k^C + \sum_{\tau_i \in \tau} U_i^C)/(m-1)$. Let such a processor be $\wp_j$ and the set of the tasks that are already assigned on this processor be $hpa^j(\tau_k)$. Hence,

$$\begin{aligned} &U_k^C + 2 \sum_{\tau_i \in hpa^j(\tau_k)} U_i^C \leq U_k^C + \frac{-2U_k^C + \sum_{\tau_i \in \tau} 2U_i^C}{m-1} \\ &\overset{*}{\leq} U_k^C\Big(1 - \frac{2}{m-1}\Big) + \frac{\frac{2m}{6} - 2\sum_{\tau_i \in \tau} U_i^A}{m-1} \overset{\dagger}{\leq} \frac{1}{2} - \frac{2\sum_{\tau_i \in \tau} U_i^A}{m-1} \end{aligned} \quad (19)$$

where $\overset{*}{\leq}$ is due to $\sum_{\tau_i \in \tau} U_i^C + U_i^A \leq \frac{m}{6}$ after speeding up and $\overset{\dagger}{\leq}$ is due to $0 < U_k^C \leq \frac{1}{6}$ and $m \geq 3$. Similarly,

$$\begin{aligned} &U_k^C + \sum_{\tau_i \in hpa^j(\tau_k)} U_i^C \leq U_k^C + \frac{-U_k^C + \sum_{\tau_i \in \tau} U_i^C}{m-1} \\ &\leq U_k^C\Big(1 - \frac{1}{m-1}\Big) + \frac{\frac{m}{6} - \sum_{\tau_i \in \tau} U_i^A}{m-1} \leq_1 \frac{4}{12} = \frac{1}{3} < 0.5 \end{aligned} \quad (20)$$

where $\leq_1$ is due to the fact that the function is monotonically decreasing with respect to m. Therefore, using Lemma 5.4 due to $U_k^C + \sum_{\tau_i \in hpa^j(\tau_k)} U_i^C < 0.5$ in Eq. (20), when $m \geq 3$, the condition in Eq. (19) leads to

$$\begin{aligned} WCRT(\tau_k) &\leq S_k + \Big(U_k^C + 2 \sum_{\tau \in hpa^j(\tau_k)} U_i^C\Big)T_k \\ &\leq \Big(\frac{1}{6} + \sum_{\tau_i \in \tau} U_i^A + \frac{1}{2} - \frac{2\sum_{\tau_i \in \tau} U_i^A}{m-1}\Big)T_k \\ &\leq \Big(\frac{2}{3} + \frac{1}{m-1}\Big((m-3) \sum_{\tau_i \in \tau} U_i^A\Big)\Big)T_k \leq \Big(\frac{2}{3} + \frac{(m-3)\frac{2}{6}}{m-1}\Big)T_k \leq T_k \end{aligned} \quad (21)$$

Therefore, we can always find an application processor to assign task τ_k to meet its deadline at a speed of 6. □

6 Evaluations

We conduct evaluations with m = 4, 8, and 16 processors. Depending on m, we generate 100 task sets for each utilization level, from $5\% \cdot m$ to $100\% \cdot m$, in steps of $5\% \cdot m$. The cardinality of each task set is $10 \times m$. We use the approach suggested by Emberson et al. [20] to generate the task periods according to a *log-uniform distribution*. The distribution of periods is within one order of magnitude, i.e., from 1ms to 10ms. All tasks have implicit deadlines, i.e., $D_i = T_i$. The overall ratio of non-critical to critical-sections depends on $\alpha \in \{5, 10, 20\}$. For example, if $\alpha = 5$ and $U_\Sigma = 120\%$, we get $U^{\mathcal{RS}} = 120\% \times \frac{1}{5+1} = 20\%$ and $U^C = 120\% \times \frac{5}{5+1} = 100\%$, i.e., the larger α is, the smaller is the critical section. 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 utilization. Those values are combined ensuring that $U_i^A + U_i^C \leq 1$ for every task τ_i. The WCETs of the non-critical-sections and critical-section of task τ_i are set accordingly, i.e., $C_i = T_i U_i^C$ and $A_i = T_i U_i^A$, and $C_{i,1}$ is drawn uniformly from $[0, C_i]$, setting $C_{i,2} = C_i - C_{i,1}$. Each critical section was assigned to one of the r resources according to a uniform distribution.

We compare the protocols by the *acceptance ratio*. Due to the space limitation, only a subset of the results is presented. We evaluated the following approaches, using the RM order and priority assignment if not mentioned otherwise, where the color and linestyle are related to the curve in Figure 1.

- LP-GFP-FMLP [6] (black, dashed): a linear-programming-based (LP) analysis for global FP scheduling using the Flexible Multiprocessor Locking Protocol (FMLP) [6].
- LP-PFP-DPCP [7] (red, dashed): LP based analysis for partitioned FP and DPCP [29]. Tasks are assigned using WFD as proposed in [7].
- LP-PFP-MPCP [7] (magenta, dashed): LP based analysis for partitioned FP using MPCP [27]. Tasks are partitioned according to WFD as proposed in [7].
- GS-MSRP (blue, dashed) [36]: the Greedy Slacker (GS) partitioning heuristic with the spin-based locking protocol MSRP [21] under Audsley's Optimal Priority Assignment.
- LP-EE-vpr (NC) [2] (cyan, dashed): A necessary scheduling condition for LP-EE-vpr.
- gEDF-vpr (NC) [1] (green, dashed): A necessary scheduling condition for gEDF-vpr.
- LP-GFP-PIP (cyan, solid): LP based global FP scheduling using the Priority Inheritance Protocol (PIP) [19].
- MrsP (magenta, solid): the Multiprocessor resource sharing Protocol (MrsP) [12] with the Synchronization-Aware Partitioning Algorithm [24].
- ROP-PCP (black, solid): the ROP in [23] using PCP.
- FP-RM-PCP (blue, solid): this paper.
- FP-EIM-PCP (red, solid): this paper.
- EDF-EIM-PCP (green, solid): this paper.

We evaluated our proposed approaches in all 8 combinations, i.e., FP or EDF, RM or EIM, and PCP or NPP. We only present the FP and the EDF approach that (in general) leads to the best performance, i.e., FP-EIM-PCP and EDF-EIM-PCP, together with the approach that provides a speedup factor of 6, i.e., FP-RM-PCP. In all cases the approaches using the PCP

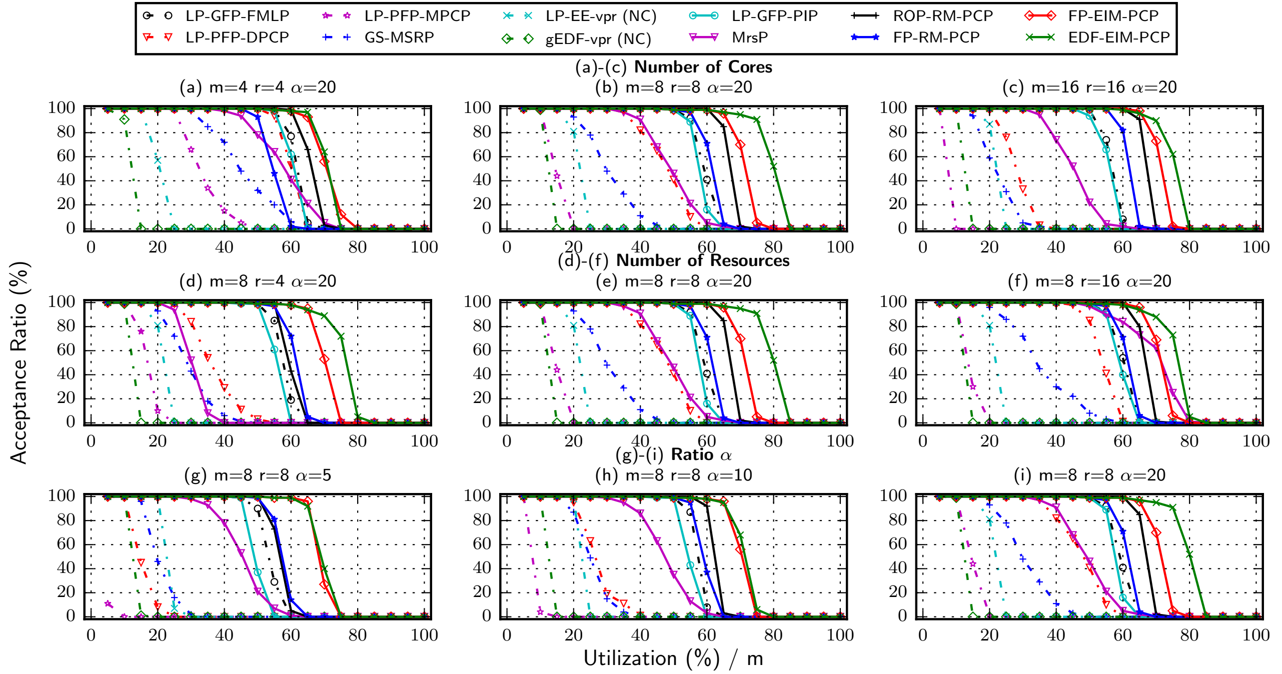

Figure 1. Comparison of different approaches under different parameter settings.

and NPP performed similarly. For our approaches and ROP-PCP we used approximated demand bound functions, where the linear approximation starts from the third period [35].

The results of our evaluations are shown in Figure 1. We analyzed the effect of the three parameters individually by changing: 1) $m = r \in \{4, 8, 16\}$ in Fig. 1 (a)-(c), 2) r for a fixed m, i.e., $r \in \{4, 8, 16\}$ and $m = 8$, in Fig. 1 (d)-(f), and 3) $\alpha \in \{5, 10, 20\}$ in Fig. 1 (g)-(i). In general, we can see that ROP-RM-PCP (black, solid) is outperformed by both FP-EIM-PCP (red, solid) and EDF-EIM-PCP (green, solid). While for most settings EDF-EIM-PCP clearly outperforms FP-EIM-PCP, there are some settings where FP-EIM-PCP and EDF-EIM-PCP are really close and there are even cases where RM-EIM-PCP deems more task sets schedulable than EDF-EIM-PCP. In general LP-GFP-PIP (cyan, solid), FP-RM-PCP (blue, solid), and LP-GFP-FMLP (black, dashed) behave similarly and mostly outperform all other approaches beside ROP-RM-PCP, FP-EIM-PCP and EDF-EIM-PCP. This behaviour was expected, as the empirical results in [39] showed that LP-GFP-FMLP and LP-GFP-PIP are the best locking protocols under global scheduling. RM-EIM-PCP performs much better than FP-RM-PCP because tasks with shorter execution intervals (EIs) are normally harder to schedule than a task with a longer EI. MrsP (magenta, solid) and LP-PFP-DPCP (red, dashed) have a very wide range regarding their acceptance ratio and no general trend can be determined. LP-PFP-MPCP (magenta, dashed), LP-EE-vpr (cyan, dashed), gEDF-vpr (green, dashed), and GS-MSRP (blue, dashed) are clearly outperformed and therefore not further discussed.

Fig. 1 (a)-(c), $\mathbf{m = r} \in \{4, 8, 16\}$: For LP-GFP-PIP, LP-GPF-FMLP, ROP-RM-PCP, and FP-EIM-PCP m does not have much impact. MrsP and LP-PFP-DPCP perform better if $m = 4$. While MrsP has similar performance for 8 and 16 cores, the acceptance ratio of LP-PFP-DPCP drops significantly for $m = 16$. EDF-EIM-PCP performs compatible with FP-EIM-PCP for $m = 4$ but has a better acceptance ratio for $m = 16$ while the gap is even larger for $m = 8$.

Fig. 1 (d)-(f), $\mathbf{r} \in \{4, 8, 16\}$ and $\mathbf{m} = 8$: The ratio of r to m seems to not have much effect on LP-GFP-PIP, LP-GFP-FMLP, and FP-EIM-PCP while ROP-RM-PCP performs worse for $r = 4$. For EDF-EIM-PCP the acceptance ratio is similar for $r = 4$ and $r = 16$ and better for $r = 8$. The most interesting observation here is the acceptance ratio of MrsP, which is worse than LP-GFP-PIP for $r = 4$ and $r = 8$ but for $r = 16$ performs nearly as good as EDF-EIM-PCP. LP-PFP-DPCP performs better if the number of resources is larger.

Fig. 1 (g)-(i), $\alpha \in \{5, 10, 20\}$: A higher value of α, and therefore a smaller percentage of critical section utilization, leads to a larger acceptance ratio. For $\alpha = 5$ and $\alpha = 10$ FP-EIM-PCP and EDF-EIM-PCP perform similar, for $\alpha = 20$ EDF-EIM-PCP clearly outperforms FP-EIM-PCP. This is most likely due to the fact that EDF-EIM-PCP only performs better than FP-EIM-PCP on the application processors and that when the critical section utilization is high the critical section has an even higher impact on the schedulability.

7 Conclusion

We provide several resource-oriented partitioned (ROP) scheduling strategies to tackle the problem of resource sharing for multiprocessor partitioned scheduling, using both fixed-priority and dynamic-priority scheduling. Compared to the initial work by Huang et al. [23], our approaches use release enforcement to ensure that release jitter does not have to be considered when analyzing the schedulability. We model the non-critical sections as segmented self-suspending tasks and apply the related state-of-the-art uniprocessor techniques to find a feasible partition. We show that one of our approaches, namely FP-RM-PCP, has a speedup factor of 6 compared to the optimal schedule, improving previously known results with respect to the speedup factor.

In the evaluations, two of our approaches, namely FP-EIM-PCP and EDF-EIM-PCP, are shown to outperform all previously known approaches, showing the effectiveness of our approaches and the resource-oriented partitioned scheduling approach in general. As both FP-EIM-PCP and EDF-EIM-PCP clearly outperform FP-RM-PCP we hope to prove a speedup factor of 6 for those approaches in the future.

Acknowledgement: This paper is supported by DFG, as part of the Collaborative Research Center SFB876, project B2 (http://sfb876.tu-dortmund.de/).

References

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

[2] B. Andersson and G. Raravi. Real-time scheduling with resource sharing on heterogeneous multiprocessors. *Real-Time Systems*, 50(2):270–314, 2014.

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

[4] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17:5–22, 1999.

[5] R. Bettati and J. W. Liu. End-to-end scheduling to meet deadlines in distributed systems. In *ICDCS*, pages 452–459, 1992.

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

[7] B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS*, 2013.

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

[9] 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.

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

[11] B. B. Brandenburg and J. H. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Autom. for Emb. Sys.*, 17(2):277–342, 2013.

[12] 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.

[13] J.-J. Chen and B. B. Brandenburg. A note on the period enforcer algorithm for self-suspending tasks. *LITES*, 4(1):01:1–01:22, 2017.

[14] 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. Technical Report 854, 2nd version, Faculty of Informatik, TU Dortmund, 2017. http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2017-chen-techreport-854-v2.pdf.

[15] 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 *Euromicro Conference on Real-Time Systems, ECRTS*, pages 9:1–9:25, 2017.

[16] J.-J. Chen, G. von der Brüggen, W.-H. Huang, and C. Liu. State of the art for scheduling and analyzing self-suspending sporadic real-time tasks. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, 2017.

[17] R. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Real-Time Systems, Volume 47, Issue 1*, pages 1–40, 2010.

[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] W.-H. Huang and J.-J. Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In *Design, Automation, and Test in Europe (DATE)*, pages 1078–1083, 2016.

[23] W.-H. Huang, M. Yang, and J.-J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium, RTSS*, pages 111–122, 2016.

[24] 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.

[25] 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.

[26] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Real-Time Systems Symposium (RTSS)*, pages 26–37, 1998.

[27] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems (ICDCS)*, pages 116–123, 1990.

[28] R. Rajkumar. Dealing with Suspending Periodic Tasks. Technical report, IBM T. J. Watson Research Center, 1991. http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/period-enforcer.ps.

[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] J. Sun and J. W.-S. Liu. Synchronization protocols in distributed real-time systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 38–45, 1996.

[32] H. Takada and K. Sakamura. Schedulability of generalized multiframe task sets under static priority assignment. In *Real-Time Computing Systems and Applications (RTCSA)*, pages 80–86, 1997.

[33] G. von der Brüggen, J.-J. Chen, and W.-H. Huang. Schedulability and optimization analysis for non-preemptive static priority scheduling based on task utilization and blocking factors. In *27th Euromicro Conference on Real-Time Systems, ECRTS*, pages 90–101, 2015.

[34] G. von der Brüggen, W.-H. Huang, and J.-J. Chen. Hybrid self-suspension models in real-time embedded systems. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, 2017.

[35] G. von der Brüggen, W.-H. Huang, J.-J. Chen, and C. Liu. Uniprocessor scheduling strategies for self-suspending task systems. In *International Conference on Real-Time Networks and Systems*, RTNS '16, pages 119–128, 2016.

[36] A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *RTSS*, 2013.

[37] 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.

[38] M. Yang, J.-J. Chen, and W.-H. Huang. A misconception in blocking time analyses under multiprocessor synchronization protocols. *Real-Time Systems*, 53(2):187–195, 2017.

[39] 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.