Multiprocessor Synchronization of Periodic Real-Time Tasks Using Dependency Graphs

Junjie Shi, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen
TU Dortmund University, Germany

Abstract—**When considering recurrent real-time tasks in multiprocessor systems, access to shared resources, via so-called critical sections, can jeopardize the schedulability of the system. The reason is that resource access is mutual exclusive and a task must finish its execution of the critical section before another task can access the same resource. Therefore, the problem of multiprocessor synchronization has been extensively studied since the 1990s, and a large number of multiprocessor resource sharing protocols have been developed and analyzed. Most protocols assume *work-conserving* scheduling algorithms which make it impossible to schedule task sets where a critical section of one task is longer than the relative deadline of another task that accesses the same resource. The only known exception to the *work-conserving* paradigm is the recently presented Dependency Graph Approach where the order in which tasks access a shared resource is not determined online, but based on a pre-computed dependency graph. Since the initial work only considers frame-based task systems, this paper extends the Dependency Graph Approach to periodic task systems. We point out the connection to the uniprocessor non-preemptive scheduling problem and exploit the related algorithms to construct dependency graphs for each resource. To schedule the derived dependency graphs, List scheduling is combined with an earliest-deadline-first heuristic. We evaluated the performance considering synthesized task sets under different configurations, where a significant improvement of the acceptance ratio compared to other resource sharing protocols is observed. Furthermore, to show the applicability in real-world systems, we detail the implementation in LITMUS^{RT} and report the resulting scheduling overheads.**

I. Introduction

Multicore and multiprocessor systems have become standard commercial off-the-shelf computing platforms that enabled massive parallel and concurrent computing. To schedule real-time tasks on multiprocessor platforms, three paradigms are widely adopted: partitioned, global, and semi-partitioned scheduling. The *partitioned* scheduling approach assigns the tasks statically among the available processors, i.e., a task is always executed on the assigned processor. The *global* scheduling approach allows a task to migrate from one processor to another at any time. The *semi-partitioned* scheduling approach decides whether a task is divided into subtasks statically and each task or subtask is then assigned to a processor. A comprehensive survey of multiprocessor scheduling in real-time systems can be found in [19].

When tasks have to synchronize or access mutually exclusive shared resources, synchronization protocols ensure a correct behaviour. Synchronization and mutual exclusion can be achieved by using *binary semaphores* or *mutex locks*, in which the protected sections are called *critical sections*. Alternatively, one can apply transactional memory (TM), e.g., [45], or lock-free or wait-free algorithms, e.g., [2], which allow accesses to a shared resource using retry loops. The critical-section-based approaches ensure the execution correctness whenever a critical section is started, whilst the lock-free or wait-free algorithms ensure the correctness whenever a shared object is successfully updated or a transaction is successfully committed. We focus on approaches with critical sections.

Since the execution of a critical section must be finished before another task can access the same resource, mutual-exclusive execution can result in *priority inversion*, where a higher-priority task is blocked by a lower-priority task. Direct blocking, i.e., a higher-priority task cannot access a resource that was previously locked by a lower-priority task, cannot be avoided. However, high-priority tasks may also suffer from blocking by lower-priority tasks that access critical sections for other resources, or by medium-priority tasks that prevent a lower-priority task that locked the resource beforehand from executing. To prevent such unnecessary blocking in uniprocessor systems, resource sharing protocols have been proposed, i.e., the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) by Sha et al. [48], and the Stack Resource Policy (SRP) by Baker [8]. The Immediate PCP, a variant of PCP, is implemented in Ada (called Ceiling locking) and POSIX (called Priority Protect Protocol).

Due to the development of multiprocessor platforms, multiprocessor resource sharing protocols are of significant practical interest, and multiple protocols have been designed and analyzed in the past decades. Examples are the Distributed Priority Ceiling Protocol (DPCP) [43], the Multiprocessor Priority Ceiling Protocol (MPCP) [42], the Multiprocessor Stack Resource Policy (MSRP) [25], the Flexible Multiprocessor Locking Protocol (FMLP) [10], the Multiprocessor PIP [22], the $O(m)$ Locking Protocol (OMLP) [13], the Multiprocessor Bandwidth Inheritance (M-BWI) [24], gEDF-vpr [3], LP-EE-vpr [4], and the Multiprocessor resource sharing Protocol (MrsP) [15]. Some of these protocols have been implemented in the real-time operating systems LITMUS^{RT} [11], [16] and RTEMS [1]. In addition, worst-case response time analysis and schedulability tests for multiprocessor locking and synchronization protocols have been studied in the literature, e.g., under multiprocessor partitioned scheduling the linear-programming (LP) based analysis [12], the suspension-aware response time analysis [17] for suspension-based protocols, and the blocking analysis [55] for spin-based protocols.

However, the above multiprocessor locking or synchronization protocols usually assume that the tasks are already reasonably prioritized (under global scheduling) or reasonably partitioned (under partitioned scheduling). The performance of these protocols highly depends on such assumptions, since a poor task priority assignment or partitioning may lead to a significant performance drawback. Towards this, task partitioning algorithms for different protocols have been developed, i.e., for MPCP by Lakshmanan et al. [35] and Nemati et al. [40], for MSRP by Wieder and Brandenburg [56], and for DPCP by Hsiu et al. [30], Huang et. al [31], and von der Brüggen et al. [52]. Specifically, the approaches by Huang et. al [31] and von der Brüggen et al. [52], so called *resource-oriented partitioned* (ROP) scheduling, focus on the shared resources by first assigning each resource to a designated processor where then uniprocessor protocols like PCP can be applied, and afterwards partitioning the non-critical sections on the remaining processors.

Regardless, the primary focus has been the design and analysis of resource sharing protocols. Most of these protocols assume that the critical sections must be executed in a *work-conserving* manner, i.e., whenever there is a critical section that is ready to be executed and the corresponding processor that is assigned to execute the critical section is idle, the critical section has to be executed. To the best of our knowledge, the only exception is the Dependency Graph Approaches by Chen et al. [18] that consists of the following two steps:

- First, a dependency graph is constructed that determines the execution order of the critical sections guarded by one binary semaphore.
- Then, multiprocessor partitioned, semi-partitioned, or global scheduling algorithms can be applied to schedule the tasks by respecting the constructed dependency graph.

Due to the first step, the schedule is not necessarily work-conserving since it is possible that a critical section is ready to be executed but one of its predecessors in the dependency graph is not ready yet.

However, the approaches presented in [18] can only be applied for frame-based real-time task systems, i.e., all tasks have the same period and release their jobs always at the same time, and a special case of periodic real-time task systems when a binary semaphore or a mutex lock is only shared by tasks with the same period. Under these assumptions, Chen et al. [18] showed that the dependency graph approach is both theoretically and practically sound, with significant improvement against traditional multiprocessor synchronization and locking protocols. Furthermore, they presented a series of analyses regarding the computational complexity and approximation ratio for frame-based real-time task systems when there is at most one non-nested critical section per task. Specifically, they showed that finding a schedule to meet the given common deadline is $\mathcal{NP}$-hard in the strong sense, regardless of the number of processors in the system.

In Sections III and IV, we will detail the connection between uniprocessor non-preemptive scheduling and multiprocessor synchronization for periodic real-time tasks. The problem to derive good non-preemptive schedules for a set of periodic real-time tasks (or equivalently a set of jobs unrolled up to the hyper-period when the release pattern repeats after the hyper-period) is a classical scheduling problem. Existing algorithms are, for example, the extended Jackson's rule by Jackson [32] (i.e., non-preemptive earliest-deadline-first (EDF)), the iterative improvement algorithm by Potts [41], an iterative improvement algorithm by Hall and Shmoys [28], the Precautious-RM by Nasri et al. [37], [39], and the critical time window-based EDF scheduling policy (CW-EDF) by Nasri and Fohler [38]. There are also scheduling algorithms and schedulability analyses, e.g., [20], [21], [26], [33], [51], for uniprocessor non-preemptive scheduling for *sporadic* real-time task systems, in which a sporadic task does not have to release its jobs periodically but under a bounded minimum inter-arrival time. Since the tasks are allowed to release their jobs sporadically, such results are pessimistic in a strictly periodic setting and, therefore, not applied here.

Contributions: In this paper, we increase the applicability of the dependency graph approaches by handling multiprocessor synchronization for *periodic* real-time task systems. We focus on a fundamental setting, in which each job has only one non-nested critical section, motivated in Section II. In Section IX, we explain possible solutions for jobs with multiple critical sections. Our contributions are as follows:

- We motivate the use of the dependency graph approach in [18] to construct non-work-conserving schedules for periodic tasks. Specifically, in Section III, we present a task set that is not schedulable by any work-conserving algorithm but for a non-work-conserving schedule.
- We develop a framework to construct independent dependency graphs for periodic tasks that access the same shared resource by reducing the problem to a uniprocessor non-preemptive scheduling problem over one hyper-period in Section IV. In this framework, any algorithm for uniprocessor non-preemptive scheduling problem is applicable, including [28], [32], [37]–[39], [41]. In our implementation, we adopted the method by Potts [41] and the extended Jackson's rule [32].
- To schedule a set of dependency graphs for the given binary semaphores on M processors, we combine List scheduling with an earliest-deadline-first (EDF) heuristic, denoted as List-EDF, in Section V. A detailed example is provided in Section VI to illustrate the procedure.
- Section VII explains how we implemented the List-EDF algorithm in LITMUS$^{\text{RT}}$ and reports the overheads.
- We further evaluate the performance by applying numerical evaluations under different configurations in Section VIII and observe significant improvement of the acceptance ratios compared to state-of-the-art techniques for multiprocessor resource sharing.

To the best of our knowledge, we present the first non-work-conserving approach for multiprocessor resource synchronization of periodic real-time tasks.

II. System Model

We consider a set of n real-time tasks $\mathbf{T} = \{\tau_1, \ \ldots, \ \tau_n\}$ that is scheduled on M identical (homogeneous) processors. Each task is described by $\tau_i = ((C_{i,1}, \ A_{i,1}, \ C_{i,2}), \ T_i, \ D_i)$. We assume that all tasks release an infinite number of task instances, called jobs, strictly periodically, i.e., if a job of τ_i is released at time t the subsequent job is released exactly at time $t + T_i$, and that the first instance of all tasks is released at time 0. The relative deadline of the task is denoted by D_i and to fulfill its timing requirements a job of τ_i released at time t must finish its execution before its absolute deadline $t + D_i$. We consider constrained-deadline task systems which means that $D_i \leq T_i$ for every task $\tau_i \in \mathbf{T}$.

The hyper-period H of the task set $\mathbf{T}$ is defined as the least common multiple (LCM) of the periods of the tasks in $\mathbf{T}$. In our approach, we unroll the jobs in one hyper-period and design a schedule for all of them. To make sure that the time and space complexity is affordable, we assume that the task set has one of the following properties:

- *Harmonic Periods*: T_i is an integer multiple of T_j if $T_i \geq T_j$ for any two tasks τ_i and τ_j in $\mathbf{T}$.
- *Semi-Harmonic Periods*: We call a task set semi-harmonic if $T_i \cdot n_i = H \; \forall \tau_i \in \mathbf{T}$ where n_i is a small integer value.

One prime example for task sets with semi-harmonic periods are automotive applications where the periods of the tasks are in $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ ms [29], [34], [44], [50], [54]. Note that our methods can still be applied to any periodic real-time task systems at the cost of higher complexity if the hyper-period is large compared to the task periods.

Each task τ_i is assumed to have exactly one (non-nested) critical section, which leads to three subtasks with the following properties:

- $C_{i,1} \geq 0$ is the worst-case execution time (WCET) of the first non-critical section.
- $A_{i,1} \geq 0$ is the WCET of the critical section, where a binary semaphore $\sigma(\tau_i)$ is applied to prevent the concurrent accesses of the shared resource.
- $C_{i,2} \geq 0$ is the WCET of the second non-critical section.

For the rest of this paper, we will use a binary semaphore for a mutex lock. In this context, we will implicitly use synchronization (which also implies locking/unlocking) in this paper. In addition, we assume that there are in total z binary semaphores (shared resources), and that the critical sections guarded by one binary semaphore s must be sequentially executed and is mutually exclusive, i.e., if two jobs share the same resource, their critical sections have to be executed without any overlap. However, a critical section guarded by a binary semaphore can be preempted by another critical section guarded by another binary semaphore or by a non-critical section arbitrarily.

The problem studied in this paper is to find a schedule for a given task set $\mathbf{T}$ on M homogeneous processors that can meet all timing constraints. A schedule $S_m(t)$ on a processor m is a function of time that determines for each time t which task is executed on processor m, i.e., if $S_m(t)$ is i, then task τ_i is executed on processor m at time t and if $S_m(t)$ is Ω, then processor m idles at time t. For a partitioned schedule, the execution of each task is tied to one processor, i.e., all executions related to a task are done on the same processor, while under global or semi-partitioned scheduling the processor where a task (or job) is executed may change over time. We assume that the execution of a task cannot be parallelized, i.e., $S_m(t) = i$ for at most one processor m at any time t. Suppose that a job of task τ_i starts to hold (successfully locks) the binary semaphore s at time t. Then any other critical section that is guarded by s cannot be executed until the job of task τ_i releases (unlocks) the semaphore s.

When analyzing the schedulability, we assume that all subjobs of all tasks always are executed according to their worst-case execution time. To be schedulable, the ℓ-th job of task τ_i must finish not later than $(\ell - 1)T_i + D_i$, i.e., $\int_{(\ell-1)T_i}^{(\ell-1)T_i+D_i} \left(\sum_{m=1}^{M} [S_m(t) \text{ is } i]\right) dt = C_{i,1} + A_{i,1} + C_{i,2}$ where $[condition]$ is the Iverson bracket, i.e., it is 1 if the condition holds, and 0 otherwise. A task is schedulable if all jobs are schedulable, and a task set is schedulable if all tasks are schedulable.

The studied task and system model is applicable if a shared resource is used at most once during the execution of a job. For example, when a *shared* GPU is used for accelerating the execution of a job, the execution behavior is usually divided into three segments: pre-processing, GPU execution, and post-processing. The pre-processing and post-processing phases do not access the shared GPU, whilst the GPU execution needs to lock the GPU exclusively. Similarly, shared memory access should be aggregated instead of having multiple individual access to the memory variables. Such a concept is also used in the AER superblock model by Schranzhofer et al. [46], [47], the PREM model by Bak et al. [6], and the memory-processor co-scheduling by Melani et al. [36].

Since this setting is the most fundamental problem in multiprocessor resource sharing, basic understanding and good solutions for this model will be a cornerstone in future design of multiprocessor resource sharing protocols, and we focus on it in this paper from Section III to Section VIII. However, since creating only one critical section per job requires engineering efforts and may not always be achievable, we provide some ideas how the proposed methods can be extended to deal with periodic real-time tasks where a job may have multiple non-nested critical sections in Section IX.

III. Motivational Example and Preliminaries

As described in Section I, existing multiprocessor resource sharing protocols are work-conserving for the critical sections. In this section, we will demonstrate the benefits of non-work-conserving synchronization mechanisms when periodic real-time tasks are considered and explain how dependency graphs can be applied.

A. Motivational Example

The computational complexity analysis by Chen et al. [18] concludes that the multiprocessor synchronization problem is $\mathcal{NP}$-hard in the strong sense even for frame-based tasks systems with one shared resource, independent from the number of processors in the system. The proof was based on a reduction from a strongly $\mathcal{NP}$-hard uniprocessor non-preemptive scheduling problem to the multiprocessor resource sharing problem. Since the case that all tasks have the same period is a special case of periodic task systems, the result still holds if tasks with different periods are allowed.

We now explain how the multiprocessor resource sharing problem is connected to the uniprocessor non-preemptive scheduling problem. Consider a set of periodic real-time tasks in a uniprocessor system, in which each task τ_i is described by its worst-case execution time C_i, its period T_i, its relative deadline $D_i = T_i$, and its phase ϕ_i, i.e., ϕ_i is the release time of the first job of the task. It has been demonstrated by Nasri and Fohler [38] that work-conserving uniprocessor schedules are not optimal based on the following example:

- $\tau_1 = \{C_1 = 1, T_1 = D_1 = 5, \phi_1 = 0\}$,
- $\tau_2 = \{C_2 = 1, T_2 = D_2 = 10, \phi_2 = 0\}$, and
- $\tau_3 = \{C_3 = 8, T_3 = D_3 = 20, \phi_3 = 0\}$.

The analysis in Theorem 4 in [38] shows that (at least) one of the three tasks misses its deadline under any work-conserving schedule. On the other hand, a feasible non-work-conserving uniprocessor schedule is to run the first job of τ_1 from 0 to 1, the first job of τ_2 from 1 to 2, the second job of τ_1 from 5 to 6, and let the first job of task τ_3 start at time 6, i.e., by inserting an idle interval from 2 to 5 instead of running the available job of τ_3. Details can be found in Section 3 in [38].

We extend the above task set to the studied multiprocessor synchronization problem. Specifically, we consider the three tasks listed in Table I which access the same resource in their critical section, where $0 < \epsilon < 0.5$. Fig. 1 presents two multiprocessor partitioned schedules, which execute tasks τ_1 and τ_2 on one processor, denoted as Proc. 1, and task τ_3 on another processor, denoted as Proc. 2. Here, and for the rest of this paper, whenever we refer to the above example, ϵ is set to 0.2. The schedule in Fig. 1(a) is a work-conserving schedule with respect to the critical sections, which leads to a deadline miss. However, the schedule in Fig. 1(b) is a non-work-conserving schedule with respect to the critical sections, which meets all the deadlines.

If the execution of the critical sections is work-conserving, the above task set cannot be feasibly scheduled, since, when ϵ is arbitrarily small, the execution of the critical sections of the three tasks is identical to the uniprocessor non-preemptive scheduling problem described by Nasri and Fohler [38]. This immediately results in a deadline miss of task τ_1 in any multiprocessor synchronization protocols in the literature because $A_{3,1} = 8$ must be considered as blocking time of task τ_1.

Moreover, the gap between work-conserving and non-work-conserving schedules already exists in "uniprocessor scenarios". That is, if the non-critical sections are negligible, then the task set in the above example provided by Nasri and Fohler [38] demonstrates the reasons why non-work-conserving schedules can be beneficial.

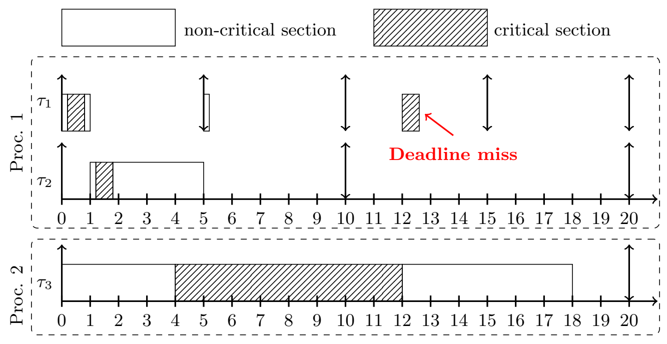
B. The Dependency Graph Approach

Therefore, there is a need to examine how non-work-conserving synchronization schedules can be designed. One possible solution is the Dependency Graph Approach by Chen et al. [18] that first constructs a dependency graph for each resource and afterwards schedules these graphs based on any scheduler for directed acyclic graphs. However, their approach is limited to scenarios when the real-time tasks that share the same resource have the same period.

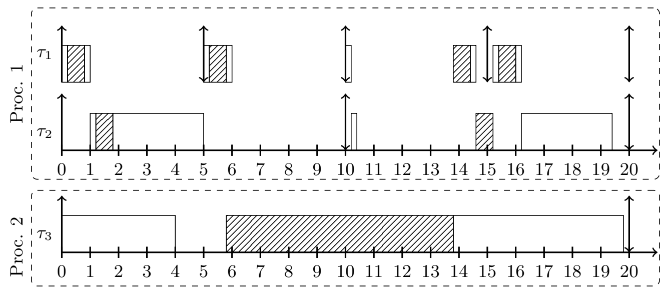
Specifically, a dependency graph in [18] is a directed acyclic graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, where a vertex $v_{i,j} \in \mathbf{V}$ represents either a critical section or a non-critical section of a job and a directed edge $e \in \mathbf{E}$ represents the precedence constraint between two vertices. By definition, for each job of τ_i, the vertex representing $C_{i,1}$ is an immediate predecessor of $A_{i,1}$ and $A_{i,1}$ is an immediate predecessor of $C_{i,2}$. The critical sections of one shared resource are *chained* as they have to be executed sequentially. To construct the graph, existing algorithms for uniprocessor non-preemptive scheduling for

Task	$C_{i,1}$	$A_{i,1}$	$C_{i,2}$	$T_i = D_i$	ϕ_i	$\sigma(\tau_i)$
τ_1	ϵ	$1-2\epsilon$	ϵ	5	0	1
τ_2	ϵ	$1-2\epsilon$	$3+\epsilon$	10	0	1
τ_3	4	8	6	20	0	1

TABLE I. An example task set that will be used in this paper for demonstration, where $0 < \epsilon < 0.5$.



(a) A work-conserving schedule.



(b) A non-work-conserving schedule.

Fig. 1. Work-conserving multiprocessor synchronization versus non-work-conserving multiprocessor synchronization for the tasks defined in Table I.

aperiodic real-time tasks were applied in [18], i.e., Potts algorithm [41] and the Extended-Jackson's Rule [32].

One possible way to extend their approach is to build a dependency graph for the tasks with the same period. If a resource is shared by tasks with ℓ different periods, then ℓ dependency graphs are constructed individually, and, afterwards, the ℓ dependency graphs are scheduled. However, the scheduler has to decide which dependency graph has a higher priority than other dependency graphs. Moreover, the scheduler has to decide whether the dependency graphs should be scheduled in the work-conserving manner. However, such an extension does not necessarily result in any benefits. For instance, when considering the example in Table I no additional information is gained, since the three tasks are represented by three dependency graphs.

Another possibility is to build a dependency graph for each shared resource s over one hyper-period. The dependency graph for the shared resource s is denoted as $\mathbf{G}_s$, in which the vertices in $\mathbf{G}_s$ represent all the critical sections and non-critical sections of the jobs that are released in the hyper-period of the corresponding tasks. The edges in graph $\mathbf{G}_s$ represent the precedence constraints of these subjobs.

Fig. 2 shows a possible dependency graph for the three

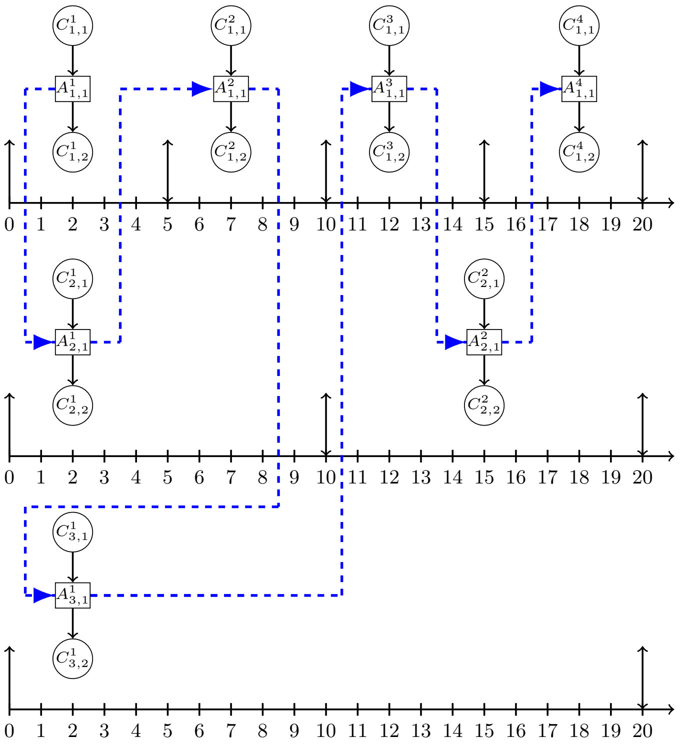


Fig. 2. The dependency graph resulting from Fig. 1(b) for the three tasks in Table 1 over one hyper-period. The precedence constraints for the critical sections are detailed by the blue dashed line.

tasks used in Table I over one hyper-period. For notational brevity, we denote the ℓ-th job of task τ_i in as J_i^ℓ and the vertices of the three subjobs of J_i^ℓ are $J_{i,1}^\ell, J_{i,2}^\ell, J_{i,3}^\ell$, representing $C_{i,1}^\ell, A_{i,1}^\ell$, and $C_{i,2}^\ell$. In the detailed graph, $J_{1,2}^2$ is an immediate predecessor of $J_{3,2}^1$. Hence, it forces $J_{3,2}^1$ to start after $J_{1,2}^2$ finishes, independent from the time $J_{3,2}^1$ is ready to be executed.

IV. Dependency Graph for Periodic Tasks

The idea of the dependency graph approach for periodic tasks is to reduce the multiprocessor resource sharing problem to the uniprocessor non-preemptive scheduling problem. We extend this approach by unrolling all jobs in one hyper-period and create the related dependency graphs. Afterwards, the graphs can be used for the scheduling of each hyper-period.

Suppose that H_s is the least common multiple (LCM) of the periods of the tasks that share semaphore s. For each task τ_i that requests semaphore s, we create H_s/T_i jobs of task τ_i.[1] For the ℓ-th job J_i^ℓ of task τ_i, the earliest time that the critical section can be executed is $r_i^\ell = (\ell-1)T_i + C_{i,1}$ and the absolute deadline to finish the critical section must be no later than $d_i^\ell = (\ell-1)T_i + D_i - C_{i,2}$. Furthermore, the processing time of the job is p_i^ℓ is $A_{i,1}$. The set of the jobs that have to be scheduled over one hyper-period H_s for a semaphore s after the above reduction is hence defined as $\mathbf{J}_s = \left\{ J_i^\ell \mid \sigma(\tau_i) \text{ is } s \text{ and } 1 \leq \ell \leq H_s/T_i \right\}$.

[1]Since H is an integer multiple of H_s by definition, one can also create a dependency graph for H/T_i jobs of task τ_i at cost of higher complexity.

Algorithm 1 Dependency graph construction for periodic tasks

```
Input: Union of task graphs that share a common semaphore $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_z$;
Output: Dependency graphs for all jobs in a hyper-period for each semaphore $\mathbf{G}'_1, \mathbf{G}'_2, \ldots, \mathbf{G}'_z$;
 1: for each $\mathbf{G_s} \leftarrow \mathbf{G}_1, \mathbf{G_s} \leftarrow \mathbf{G}_2, \ldots, \mathbf{G_s} \leftarrow \mathbf{G}_z$ do
 2:     $H_s \leftarrow LCM(\mathbf{G}_s)$
 3:     unroll all jobs that are released in the hyper-period for the tasks in $\mathbf{G}_s$;
 4:     for the $\ell$-th job of task $\tau_i$ in $\mathbf{G}_s$ with $1 \leq \ell \leq \lceil H_s/T_i \rceil$ do
 5:         $r_i^\ell \leftarrow (\ell-1) \cdot T_i + C_{i,1}$;
 6:         $p_i^\ell \leftarrow A_{j,1}$;
 7:         $d_i^\ell \leftarrow (\ell-1) \cdot T_i + D_i - C_{i,2}$;
 8:     end for
 9:     $\mathbf{G}'_s \leftarrow$ calculate the precedence constraints for the critical sections;
10:     for the $\ell$-th job of task $\tau_i$ in $\mathbf{G}_s$ with $1 \leq \ell \leq \lceil H_s/T_i \rceil$ do
11:         add the precedence constraints $C_{i,1}^\ell \rightarrow A_{i,1}^\ell \rightarrow C_{i,2}^\ell$ to $\mathbf{G}'_s$;
12:     end for
13: end for
14: return $\mathbf{G}'_1, \mathbf{G}'_2, \ldots, \mathbf{G}'_z$;
```

Note, that we only consider the construction of the dependency graph for the critical sections at this moment and assume sufficient resources for the non-critical sections, i.e., all critical sections are executed sequentially on one processor and each task has an exclusively assigned processor for the execution of non-critical sections. Therefore, the execution of the non-critical sections is not considered but the related WCET is used for setting up the release time and the absolute deadline.

After this reduction, we can apply any existing algorithm for non-preemptive uniprocessor scheduling to construct the precedence constraints for the critical sections in $\mathbf{J}_s$, i.e., the execution order for the critical sections. Moreover, since the execution of a job has to follow the execution order of $C_{i,1}$, $A_{i,1}$, and $C_{i,2}$, we include these precedence constrains to get the complete dependency graph for semaphore s.

Algo. 1 shows the pseudo-code of our approach and Fig. 2 displays an example for a resulting dependency graph. When constructing the precedence constraints for $\mathbf{J}_s$ (Line 9 in Algo. 1), any algorithm for non-preemptive uniprocessor scheduling can be exploited. For example, when considering the literature of the real-time systems community, Precautious-RM by Nasri et al. [37], [39] and the critical time window-based EDF scheduling policy (CW-EDF) by Nasri and Fohler [38] can be applied.

Alternatively, classical results for the machine scheduling problem of independent jobs under non-preemptive scheduling can be considered, e.g., the algorithms by Hall and Shmoys [28], Potts [41], and Jackson [32]. These algorithms assume knowledge about the release times, processing times, and deadlines of all jobs that have to be considered in the schedule. Note, that the classical scheduling problem assumes a delivery time that a job needs after it finishes its execution instead of a deadline and minimizes the length of the schedule. However, our studied problem can directly be transferred by setting the delivery time as $H_s - d_i^\ell$ for each job and checking whether the schedule is shorter than H_s. In this work, we consider Potts algorithm [41] and the extended Jackson's rule [32] for the construction. While the extended

Jackson's rule [32] is similar to non-preemptive EDF, Potts algorithm [41] starts with a non-preemptive EDF schedule that is updated over a (fixed) number of iterations by defining a critical job J_c, i.e., a job that misses the deadline, and forcing a job with a later absolute deadline than J_c that is executed before J_c (do to an earlier release time than J_c) to execute after J_c. This procedure may lead to idle time and therefore to a non-work-conserving schedule.

V. List-EDF Scheduling

Here, we show how to schedule the unrolled dependency graphs over the hyper-period $H = LCM(H_1, H_2, \dots, H_z)$. According to our notation, each job J_i^ℓ has three subjobs $J_{i,1}^\ell, J_{i,2}^\ell, J_{i,3}^\ell$ that represent the related subjobs $C_{i,1}, A_{i,1}, C_{i,2}$, respectively, the release time of the first subjob is $J_{i,1}^\ell$ is $(\ell-1)T_i$, and the absolute deadline of the last subjob $J_{i,3}^\ell$ is $(\ell-1)T_i + D_i$. Regarding the release times of the second and third subjob, we initially set the earliest possible time the job may be released based on the WCETs of the other subjobs, and regarding the deadline of the first and second subjob, we initially assign the latest possible time the subjob can finish while still allowing schedulability. To be precise, the release time of $J_{i,2}^\ell$ is set to $(\ell-1)T_i + C_{i,1}$, the release time of $J_{i,3}^\ell$ is set to $(\ell-1)T_i + C_{i,1} + A_{i,1}$, the absolute deadline of $J_{i,2}^\ell$ is set to $(\ell-1)T_i + D_i - C_{i,2}$, and the absolute time of $J_{i,1}^\ell$ is set to $(\ell-1)T_i + D_i - C_{i,2} - A_{i,1}$.

We assume that each dependency graph $\mathbf{G}_s$ for a binary semaphore s is constructed for the corresponding jobs released (strictly) within one hyper-period H. If $H_s < H$, then $\frac{H}{H_s}$ copies of $\mathbf{G}_s$ are applied in a consecutive order to represent the precedence constraints of the critical sections. For notational brevity, we denote $r_{i,j}^\ell$ as the release time of the subjob $J_{i,j}^\ell$ and $d_{i,j}^\ell$ as the absolute deadline of $J_{i,j}^\ell$. If the absolute deadline of an immediate predecessor of $J_{i,j}^\ell$, denoted as $IPre(J_{i,j}^\ell)$, is larger than $d_{i,j}^\ell$, the absolute deadline of the immediate predecessor should be reassigned to $d_{i,j}^\ell$ minus the WCET of $J_{i,j}^\ell$. This is a standard procedure for scheduling jobs subject to release dates and precedence constraints. Details can be found in [7] and an example is provided in Section VI for illustration.

For the rest of this paper, we assume that the absolute deadline assignment is adjusted accordingly so that $d_{i,j}^\ell$ for the subjob $J_{i,j}^\ell$ is always greater than the absolute deadline of $IPre(J_{i,j}^\ell)$.

After the construction of $\mathbf{G}_1, \mathbf{G}_2, \dots, \mathbf{G}_z$, the scheduling problem becomes a classical multiprocessor scheduling problem. In scheduling theory, a scheduling problem is defined by a triple notation $\alpha|\beta|\gamma$, where

- α: describes the machine environment,
- β: specifies the processing characteristics and constraints,
- γ: presents the objective to be optimized.

Scheduling $\mathbf{G}_1, \mathbf{G}_2, \dots, \mathbf{G}_z$ on M homogeneous (identical) processors is a special case of the classical scheduling problem $P|prec; r_j|L_{\max}$, i.e., scheduling a set of jobs with specified release times and precedence constraints on M identical processors, minimizing the maximum lateness. Our goal here is not to have any deadline misses. Therefore, a schedule is feasible if $L_{\max} \leq 0$.[2]

One possible scheduling strategy is to use the List scheduling developed by Graham [27] in combination with earliest-deadline-first scheduling (EDF). A List schedule works as follows: Whenever a processor idles and there are subjobs eligible to be executed (i.e., all of their predecessors in the dependency graph have finished), one of the eligible subjobs is executed on the processor. If more subjobs than processors are available, we prioritize the subjobs that have the earlier absolute deadlines, and if two subjobs have the same absolute deadline, the one with the larger remaining workload has a higher priority. We denote this scheduling algorithm List-EDF.

We note that List-EDF in our setting is a preemptive algorithm. Whenever a new (eligible) subjob has an earlier absolute deadline than an executing subjob on a processor m, this new subjob can preempt the one that is executing on processor m. Such flexibility to allow preemption does not create any problem for the mutual-exclusive constraint of the critical sections guarded by one binary semaphore s because their execution order has been predefined in the dependency graph $\mathbf{G}_s$. Therefore, a critical section guarded by a semaphore s can only be preempted by either non-critical sections or by critical sections guarded by other semaphores.

Moreover, Graham [27] showed that the list scheduling can suffer from multiprocessor timing anomalies. Specifically, the reduction of the execution time of a subjob can lead to longer response times of other subjobs. We do not prove that no multiprocessor timing anomaly exists in List-EDF. Hence, after deriving a schedule based on List-EDF, the schedule has to be applied *statically*, i.e., in an offline fashion. One option is to apply table-driving scheduling to ensure the repetitive schedule in every hyper-period. Another is to enforce the actual execution time of each subjob to be the same as its worst-case execution time. Since the scheduling algorithm is deterministic, the schedule is always repeated. We will discuss both options in Section VII.

VI. An Illustrative Example for List-EDF

In this section, we provide an example to demonstrate how our algorithm works. We consider a task set consisting of the three tasks defined in Table II, which are identical to the tasks in Table I when $\epsilon = 0.2$, all requesting shared resource 1, and two additional tasks defined in Table III which request shared resource 2. These five tasks are scheduled on $M = 2$ processors by using List-EDF.

We begin with the tasks requesting resource 1 (Table I), provided that the chain of the critical sections is defined by the dependency graph shown in Fig. 2. This dependency graph results in an order of $J_{1,2}^1, J_{2,2}^1, J_{1,2}^2, J_{3,2}^1, J_{1,2}^3, J_{2,2}^2, J_{1,2}^4$. Based on this, the release times and deadlines for the individual subjobs can be determined for all releases of all subjobs of all tasks as displayed in Table II. Regarding the release times, we will only consider the earliest possible release time of the critical sections and of the second non-critical

[2]$L_{\max}$ is $\max_j\{f_j - d_j\}$, where f_j is the finishing time of job J_j and d_j is the absolute deadline of job J_j. Therefore, if $L_{\max} \leq 0$, then all jobs meet their deadlines.

Task	WCETs			Other Parameters				Release Times			Deadlines		
	$C_{i,1}$	$A_{i,1}$	$C_{i,2}$	$T_i = D_i$	ϕ_i	$\sigma(\tau_i)$	ℓ	$J^\ell_{i,1}$	$J^\ell_{i,2}$	$J^\ell_{i,3}$	$J^\ell_{i,1}$	$J^\ell_{i,2}$	$J^\ell_{i,3}$
τ_1	0.2	0.6	0.2	5	0	1	1	0	0.2	0.8	4.2	4.8	5
							2	5	5.2	5.8	5.4	6	10
							3	10	13.8	14.4	14.2	14.8	15
							4	15	15.2	15.8	19.2	19.8	20
τ_2	0.2	0.6	3.2	10	0	1	1	0	0.8	1.4	4.8	5.4	10
							2	10	14.4	15	16.2	16.8	20
τ_3	4	8	6	20	0	1	1	0	5.8	13.8	6	14	20

TABLE II. The (earliest possible) release times and the deadlines for the task set presented in Table I (with $\epsilon = 0.2$), based on the dependency graph in Fig. 2. The release times and deadlines of the critical sections are changed based on the order and the resulting restrictions in the dependency graph (colored red), which propagates to later releases of the second non-critical section or an earlier deadline of the first non-critical section (colored blue).

Task	WCETs			Other Parameters				Release Times			Deadlines		
	$C_{i,1}$	$A_{i,1}$	$C_{i,2}$	$T_i = D_i$	ϕ_i	$\sigma(\tau_i)$	ℓ	$J^\ell_{i,1}$	$J^\ell_{i,2}$	$J^\ell_{i,3}$	$J^\ell_{i,1}$	$J^\ell_{i,2}$	$J^\ell_{i,3}$
τ_4	0.2	0.2	0.2	10	0	2	1	0	0.2	0.4	9.6	9.8	10
							2	10	10.2	10.4	19.6	19.8	20
τ_5	2	3	2	20	0	2	1	0	2	5	15	18	20

TABLE III. The tasks requesting the shared resource 2 (including release times and deadlines).

sections resulting from the dependency graph, assuming no early completion of jobs. Note that the actual release times may be later, depending on the actual schedule.

All first subjobs are released exactly periodically. For all other subjobs, the release times may be adjusted based on the earliest times their predecessors may finish. This is done in a forward manner, i.e., starting from time 0 up until time H, i.e., 20 in Table II and Table III. Hence, $J^1_{2,2}$ is released at time 0.8 (marked red in Table II) due to the earliest possible finishing time of $J^1_{1,2}$ at 0.8. Therefore, the release of $J^1_{2,3}$ is postponed as well (marked blue here and for all other third subjobs that are postponed). The second subjob of the second release of τ_1 can finish no earlier than at time 5.8, hence the release of $J^1_{3,2}$ is postponed accordingly. Due to the long critical section of task τ_3, the releases of $J^3_{1,2}$ and $J^2_{2,2}$ are postponed to time 13.8 and 14.4 as well.

The deadlines in Table II are constructed in a backward manner, i.e., from the end of the hyper-period to the beginning. Here, all third subjobs have a deadline identical to the end of the related period. The deadlines of the second subjobs are based on the dependency graph. While for $J^4_{1,2}, J^2_{2,2}, J^3_{1,2}$, and $J^1_{3,2}$ the deadlines directly result from that job's third subjob the large duration of $J^1_{3,2}$ leads to an early deadline of $J^2_{1,2}$ (6 instead of 9.8) which again leads to the deadline of 5.4 (instead of 6.8) for $J^1_{2,2}$. Again, the changed deadlines for the second subjobs are marked red while the resulting adjustment for the first subjobs is marked blue in Table II.

Regarding the tasks τ_4 and τ_5 (that access resource 2) shown in Table III, we assume the order $J^1_{4,2}, J^1_{5,2}, J^2_{4,2}$ for the access to resource 2. Since no deadlines or release times are adjusted in Table III, details are omitted.

The schedule based on global EDF is displayed in Fig. 3 and considers the deadlines provided in Table II and Table III. Execution on processor 1 is marked blue while execution on processor 2 is marked red. In addition, the access to the critical sections related to resource 1 and resource 2 are shown with different hatching patterns as detailed in Fig. 3. We assume that if two tasks with the same deadline compete for a processor, then the subjob with the larger remaining workload is preferred in the scheduling decision. At time 0 the subjobs $J^1_{1,1}$ and $J^1_{2,1}$ are scheduled since their deadlines are 4.2 and 4.8. As soon as $J^1_{2,1}$ is finished at time 0.2, $J^1_{2,2}$ cannot be scheduled since $J^1_{1,2}$ is its predecessor and has not finished yet. Therefore, $J^1_{3,1}$ gets the processor due to its deadline at time 6. At time 0.8, $J^1_{3,1}$ is preempted by $J^1_{2,2}$ which has a shorter deadline, namely 5.4, but $J^1_{3,1}$ is assigned to the other processor at time 1.0. Subsequently, $J^1_{3,1}$ finishes its execution at time 4.4, and $J^1_{4,1}$ is assigned to the processor. After $J^1_{2,3}$ finishes executing at time 4.6, we observe the non-work-conversing behaviour of our approach. Here, $J^1_{3,2}$ has an absolute deadline of 14 which is earlier than the absolute deadline of $J^1_{5,1}$ at time 15. Nevertheless, due to the precedence constrains resulting from the dependency graph, i.e., that $J^2_{1,2}$ executes before $J^1_{3,2}$ in the total order, $J^1_{3,2}$ is not eligible and processor 2 is assigned to $J^1_{5,1}$. Note that if processor 2 would be assigned to $J^1_{3,2}$ at this point in time, then $J^2_{1,2}$ would miss its deadline since $J^1_{3,2}$ would not finish its execution before time 12.6. Under the dependency graph approach this is prevented by postponing $J^1_{3,2}$ until $J^2_{1,2}$ is finished at time 6.

We point out that the release times displayed in Table II and Table III are only considered when constructing the dependency graph but do not have any impact on the scheduling of the jobs afterwards. In the actual schedule, the subjobs are released based on the actual finishing time of all predecessors which may be much later than the earliest possible release times considered during construction. The reason is that during construction we assume that all non-critical sections can be executed as soon as they are released without considering if sufficient processors are available to schedule all available subjobs. For instance, in the actual schedule in Fig. 3, the second subjobs of the first job of τ_4 and τ_5 are released much later than at time 0.2 and 2, which are the release times in Table III. However, not considering the combined workload of the non-critical sections when constructing the dependency graph allows us to construct the graphs for different resources individually and to use well-known algorithms for uniprocessor non-preemptive scheduling.

VII. Implementation and Overheads

This section details our implementation in LITMUS$^{\text{RT}}$ and compares the implementation overheads of our approach and FMLP provided by LITMUS$^{\text{RT}}$ for both partitioned and global scheduling. Our implementation has been released in [49].

We considered two possibilities for the implementation in LITMUS$^{\text{RT}}$: 1) applying the table-driven scheduling that LITMUS$^{\text{RT}}$ provides, or 2) implementing a new binary semaphore which can enforce the execution order of subjobs.

A table-driven scheduling is defined by a static scheduling table that repeat periodically, i.e., for each point in time in the hyper-period the table determines which entity each processor should be executing (if any). However, the large number of subjobs and the fact that jobs may be migrated under global List-EDF may lead to a very large table size. While table driven scheduling directly prevents the multiprocessor timing anomalies (detailed at the end of Sec. V), this can also be avoided if early completion of jobs is not allowed, i.e., each subjob is forced to execute the related WCET. Therefore, we implemented a new binary semaphore that supports the properties of our approach.

The new approach is implemented under the plug-in Partitioned EDF with synchronization support (PSN-EDF) and the plug-in Global EDF with synchronization support (GSN-EDF). The EDF feature of *List-EDF* is guaranteed by the original design of these two plug-ins. Therefore, we only have to provide the relative deadlines for the three subjobs of each task, and let the system update the absolute deadline for each subjob in different periods accordingly during the run time. Hence, we only explain how the subjob-order resulting from the dependency graph is enforced. On the one hand, we must ensure that jobs access the shared resource according to the predefined graph order. On the other hand, the inner execution order of the jobs must be ensured, i.e., the subjobs are always executed in the order: first non-critical section, critical section, second non-critical section.

The order of the three subjobs within one job is directly provided by *liblitmus*, the user-space library of LITMUS$^{\text{RT}}$. The task deploy tool `rtspin` is used to emulate purely CPU-bound workload and to define the structure of tasks can directly, e.g., the number of non-critical sections and critical sections, the order of these non-critical sections and critical sections, the related execution times, and the number of the resources that the task will access. Each job is executed in `rtspin` by using Algo. 2 where `execution_for` is a simple spin loop function that emulates purely CPU-bound workloads. The function `execution_for(a, b, c)` has three inputs: `a` is the release time of the subjob, `b` is the execution time of the subjob, and `c` is the deadline of the

Algorithm 2 Inner order enforcement in `rtspin`

```
Input: Execution times for three subjobs: $C_{i,1}$, $A_{i,1}$, and $C_{i,2}$,
    deadlines for each subjob: $d^{\ell}_{i,j}$;
1: execution_for($r^{\ell}_{i,1}$, $C_{i,1}$, $d^{\ell}_{i,1}$);
2: semaphore_lock();
3: execution_for($r^{\ell}_{i,2}$, $A_{i,1}$, $d^{\ell}_{i,2}$);
4: semaphore_unlock();
5: execution_for($r^{\ell}_{i,3}$, $C_{i,2}$, $d^{\ell}_{i,3}$);
```

subjob. Note, that the release times of the subjobs are not predetermined but result from the moment the job is released (for the first subjob), and the moments where the first non-critical section and the critical section are finished (for the second and third subjob). The deadlines of the subjobs however are calculated beforehand and result from the dependency graph and the intra job dependencies. All the commands in Algo. 2 are executed sequentially, which directly ensures that the critical section and the second non-critical section are only released after their predecessor in the job has finished.

For a set of periodic real-time tasks that release the first job at the same time, the schedule of the given task set will be repeated for each hyper-period if

- the scheduler is deterministic, i.e., it always makes the same scheduling decision for a given situation,
- the WCRT of all jobs is smaller than the period of the tasks, i.e., it is ensured that at any point in time at most one job of each task is in the system, and
- no early completion of (sub)jobs is allowed.

In one hyper-period, the order of all jobs' critical sections which request the same resource is defined according to the dependency graph constructed by Algo. 1. We enforce that jobs are executed in this order by extending the data structures that describes tasks and implementing a new binary semaphore:

- The data structure `rt_params` that defines the properties of each task, e.g., priority, period, and execution time, is extended with the following three parameters:
  - `rt_order` defines the positions of the jobs of one task in the total order of the critical sections that access the same shared resource over one hyper-period.
  - `rt_jobs` is the number of jobs of the related task in one hyper-period.
  - `total_jobs` is the number of all jobs that share the related resource (identical for all tasks accessing the same resource).
- The binary semaphore for the dependency graph approach with List-EDF is named `list-edf_semaphore`. Besides the common components for semaphore, i.e., `semaphore_owner`, `wait_queue`, and `litmus_lock`, an additional parameter named `current_serving_ticket` is defined that controls the execution order of the critical sections.

The pseudo code provided in Algo. 3 shows three main functions in our implementation, details are as follows:

The function **`get_job_order`** returns the position of the job in the execution order for jobs that access the same shared resource among different hyper-periods. In LITMUS$^{\text{RT}}$, `job_no` by default counts the number of jobs that one

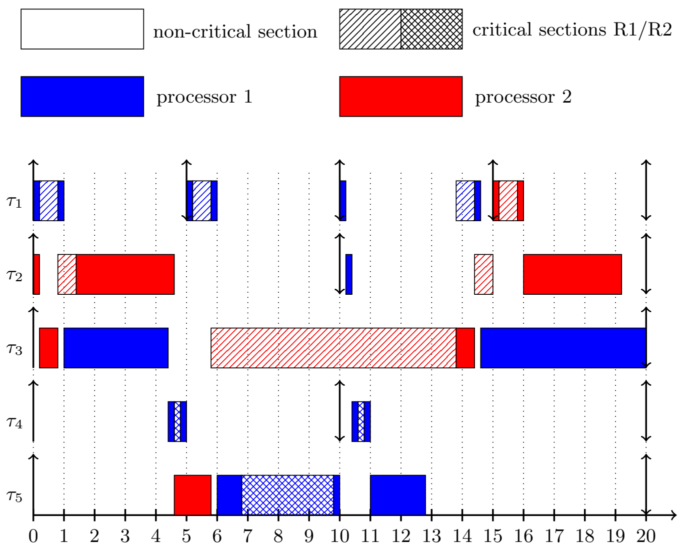


Fig. 3. An Example of List-EDF with two shared resources.

Task	`rt_order`	`rt_jobs`	`total_jobs`
τ_1	$[0, 2, 4, 6]$	4	7
τ_2	$[1, 5]$	2	7
τ_3	$[3]$	1	7

TABLE IV. An example of the data structure for tasks.

Algorithm 3 List-EDF implementation

```
Input: Task $\tau_i$, total_jobs for the resource accessed by $\tau_i$ within
       one hyper-period, and current_serving_ticket for the
       related semaphore;

    Function get_job_order():
1:  index ← $\tau_i$.job_no mod $\tau_i$.rt_jobs;
2:  $\tau_i$.job_order ← rt_order[index];

    Function list-edf_lock():
3:  if semaphore_owner is NULL and
    current_serving_ticket equals to $\tau_i$.job_order then
4:      semaphore_owner ← $\tau_i$;
5:      $\tau_i$ starts the execution of its critical section;
6:  else
7:      Add $\tau_i$ to the corresponding wait_queue;
8:  end if

    Function list-edf_unlock():
9:  $\tau_i$ releases the semaphore lock;
10: current_serving_ticket++;
11: if current_serving_ticket = total_jobs then
12:     Set current_serving_ticket ← 0;
13: end if
14: Next task $\tau_{next}$ ← the head of the wait_queue;
15: if current_serving_ticket equals to $\tau_{next}$.job_order
    then
16:     semaphore_owner ← $\tau_{next}$;
17:     $\tau_{next}$ starts the execution of its critical section;
18: else
19:     semaphore_owner ← NULL;
20:     Add $\tau_{next}$ to the corresponding wait_queue;
21: end if
```

task releases. We apply a modulo operation on `job_no` and `rt_jobs` to find out the exact position of this job within one hyper-period. After that, the value of `job_order` is searched from `rt_order` based on the obtained index.

We take the first three tasks in Fig. 3 which share the same resource as an example and present the related data structure in Table IV. Assume that the `job_no` for τ_1 is 17. Line 1 in Algo. 3 returns the corresponding relative position in the hyper-period, i.e., the 17th job of τ_1 is the job of τ_1 with index 1 in the current hyper-period. Therefore, the corresponding element in `rt_order` is 2 (Line 2 in Algo. 3). Namely, the 17th job of task τ_1 is the second job of τ_1 that is granted access to the resource within the corresponding hyper-period.

The function **`list-edf_lock`** tries to lock the semaphore to obtain the related resource. After getting the correct position in the execution order according to the dependency graph, the job's `job_order` and the semaphore's `current_serving_ticket` are compared, even if the semaphore is not occupied at that point in time. If they are equal, the job will be granted access to the resource and start its critical section; otherwise, the job will be added to the `wait_queue`, which is sorted by the jobs' parameter `job_order`, i.e., the job with the smallest ticket number is the head of the waiting queue. This ensures that the predefined execution order is kept since a job can only lock the semaphore after all its predecessors in the dependency graph finished executing their critical sections in this hyper-period.

The function **`list-edf_unlock`** unlocks the semaphore once a job has finished its critical section. The semaphore's `current_serving_ticket` is increased by one, i.e., it can now be locked by the next job in the order. If `current_serving_ticket` reaches `total_jobs` the dependency graph is traversed completely, i.e., all jobs that access the related resource finished their executions of the critical sections in the current hyper-period, the parameter `current_serving_ticket` is reset to 0 to start the next iteration. After that, the first job (if any) in the `wait_queue`, called τ_{next}, is checked. If τ_{next} has the `job_order` which is the same as the semaphore's `current_serving_ticket`, τ_{next} is set as the owner of the semaphore, and can start the execution of its critical section. Otherwise, the semaphore owner is set as NULL, and the task τ_{next} is added into the corresponding `wait_queue`.

The implementations for the global and the partitioned plug-in are similar. The only difference is that for global scheduling preemptions are very frequent. Thus, in order to protect the executions of our aforementioned functions, the preemption should be disabled inside the semaphore related functions.[3] In addition, the over-run situation has not been treated carefully in our implementation. Once a task misses the deadline, it may destroy the ticket system.

To evaluate the applicability of our approach, we tracked multiple overheads under LITMUS^{RT}:

[3]If a job cannot obtain the semaphore and sleeps in the corresponding wait queue until the end of the simulation, it will remain in the system with the *uninterruptable sleep* status. To avoid such a situation, the run-time of the simulation should be a multiple of the hyper-period to ensure all the tasks can finish their periods. Otherwise, the system should be restarted before the next run to avoid complications.

- **CXS:** context-switch overhead.
- **RELEASE:** time spent to enqueue a newly released job into a ready queue.
- **SCHED:** time spent to make a scheduling decision, i.e., to find the next job that is executed.
- **SCHED2:** time spent to perform post context switch and management activities.
- **SEND-RESCHED:** inter-processor interrupt latency, including migrations.

The hardware platform used in our experiments is a cache-coherent SMP, consisting of two 64-bit Intel Xeon Processor E5-2650Lv4 running at 1.7 GHz, with 35 MB cache and 64 GB of main memory. Table V reports the overheads for protocols supported by plug-in PSN-EDF and GSN-EDF in LITMUS$^{\text{RT}}$, namely of the existing implementation of the Flexible Multiprocessor Locking Protocol (FMLP) [10] and our implementations of List-EDF. The partitioned version of List-EDF (which maps each task statically to a processor) and the global version of List-EDF presented in Section V are both included. Table V shows that the overheads of our approach and of FMLP are comparable in LITMUS$^{\text{RT}}$.

We note that we only analyzed the resulting overheads of partitioned List-EDF here. The actual performance of partitioned List-EDF may strongly depend on a good task partitioning algorithm which is out of the scope of this work.

VIII. Numerical Performance Evaluation

We conducted evaluations for M = 4, 8 and 16 processors while the number of shared resources (binary semaphores) was $z \in \{4, 8, 16\}$. For each M, we generated 100 task sets with $10 \cdot M$ tasks. For each setting, we considered total utilization levels from $30\% \cdot M$ to $100\% \cdot M$ in steps 5%. The task periods T_i are selected randomly from a set of semi-harmonic periods, i.e., $T_i \in \{1, 2, 5, 10\}$, that is a subset of the periods used in automotive systems [29], [34], [44], [50], [54]. For each task set $\mathbf{T}$, tasks with the target total utilization are generated randomly by applying the RandomFixedSum method [23], i.e., $\sum_{\tau_i \in \mathbf{T}} \frac{C_{i,1}+C_{i,2}+A_{i,1}}{T_i}$ is the target utilization. We enforced for each task τ_i that $U_i \leq 0.5$. To determine $U^C_{i,1}$, $U^C_{i,2}$, and $U^A_{i,1}$ (the utilization for the first non-critical section, the second non-critical section, and the critical section), first $U^A_{i,1}$ was drawn randomly as a percentage β of the total utilization U_i from a given range, i.e., β was either in $[5\% - 10\%]$, $[10\% - 40\%]$, or $[40\% - 50\%]$, depended on the actual setting. Afterwards, the remaining utilization U^C_i was split by drawing $U^C_{i,1}$ randomly uniform from $(0, U^C_i)$ and setting $U^C_{i,2}$ to $U^C_i - U^C_{i,1}$. The execution time equals to the utilization multiplied with the period, e.g., $C_{i,1} = U^C_{i,1} \cdot T_i$.

We compare our method with state-of-the-art multiprocessor resource sharing protocols and scheduling approaches:

- LIST-EDF-JKS: Our approach by applying the algorithm by Jackson [32] for generating the dependency graphs and List-EDF scheduling.
- LIST-EDF-POTTS: Our approach by applying Potts algorithm [41] for generating the dependency graphs and List-EDF scheduling.
- ROP-FP [52]: The Resource Oriented Partitioned (ROP) scheduling in [52] under fixed-priority scheduling and release enforcement.
- ROP-EDF [52]: ROP under dynamic-priority scheduling and release enforcement.
- LP-GFP-FMLP [10]: a linear-programming-based (LP) analysis for global FP scheduling using the FMLP [10].
- GS-MSRP [56]: the Greedy Slacker (GS) partitioning heuristic with the spin-based locking protocol MSRP [25] under Audsley's Optimal Priority Assignment [5].
- LP-GFP-PIP: LP-based global FP scheduling using the Priority Inheritance Protocol (PIP) [22].
- LP-PFP-DPCP [12]: LP-based analysis for partitioned FP and DPCP [43]. Tasks are assigned using Worst-Fit-Decreasing (WFD) as proposed in [12].
- LP-PFP-MPCP [12]: LP-based analysis for partitioned FP using MPCP [42]. Tasks are partitioned according to WFD as proposed in [12].

To the best of our knowledge, our method and ROP scheduling with release enforcement [52] are the only approaches that primarily focus on the critical sections and actively consider them during the design process. On the other hand, resource sharing protocols primarily try to handle the occurring resource requests in a reasonable way. However, for some resource sharing protocols, processor assignment strategies have been developed later.

We compare the approaches using the metric of *acceptance ratios*. A subset of the results is presented in Fig. 4.[4] In all considered configurations, the acceptance ratios of LP-PFP-DPCP and LP-PFP-MPCP are zero. Hence, both are not shown in our evaluation results. In general, our proposed method when applying Potts algorithm for generating the dependency graphs clearly outperforms other approaches. The results also show that work-conserving resource synchronization protocols do not perform very well in the evaluated settings. Since the extended Jackson's rule (JKS) generates the dependency graph in a work-conserving manner as well, LIST-EDF-JKS performs worse than LIST-EDF-POTTS. In addition, the ROP-based methods work comparably well due to the release enforcement feature, which enforces the releases of the critical sections to be periodic, thus removing the possible higher interference due to release jitter. The enforcement also, as a side product, leads to non-work-conserving schedules for the critical sections. That is, a critical section can be delayed even when it is ready to be executed to reduce the jitter.[5]

In the evaluation, we also specifically analyzed the effect of the three parameters individually by changing:

1) $\boldsymbol{M = z \in \{4, 8, 16\}}$ (Fig. 4(a) to Fig. 4(c)): Increasing z and M at the same time does not have significant impact on the LIST-EDF-POTTS but the other approaches perform slightly worse.
2) $\boldsymbol{z}$ **for a fixed** $\boldsymbol{M}$, i.e., $z \in \{4, 8, 16\}$ and $M = 8$ (Fig. 4(d) to Fig. 4(f)): When the number of resources is increased, compared to the number of processors, the performance gap between the LIST-EDF-POTTS and the ROP approaches increases. This indicates that when the number of resources is large the benefit of our non-work-conserving method becomes more significant.

[4]Fig. 4 shows 7 different settings. Fig. 4(b), Fig. 4(e), and Fig. 4(h) are the same but repeated for the clarity of the comparisons and discussions.

[5]Throughout this paper, we consider such non-work-conserving behavior as a side product and not as creating non-work-conserving schedules on purpose.

Max. (Avg.) in μs	Partitioned FMLP	Partitioned List-EDF	Global FMLP	(Global) List-EDF
CXS	34.45 (0.71)	29.93 (0.81)	42.25 (1.14)	30.87 (1.79)
RELEASE	30.01 (0.63)	25.37 (0.75)	65.44 (2.98)	61.63 (12.06)
SCHED	63.91 (0.92)	32.3 (1.03)	80.81 (1.77)	59.05 (4.46)
SCHED2	33.23 (0.13)	25.24 (0.15)	31.43 (0.19)	27.17 (0.25)
SEND-RESCHED	65.81 (11.38)	20.71 (1.44)	92.78 (17.2)	72.09 (20.77)

TABLE V. Overheads of different protocols in LITMUS^{RT}.

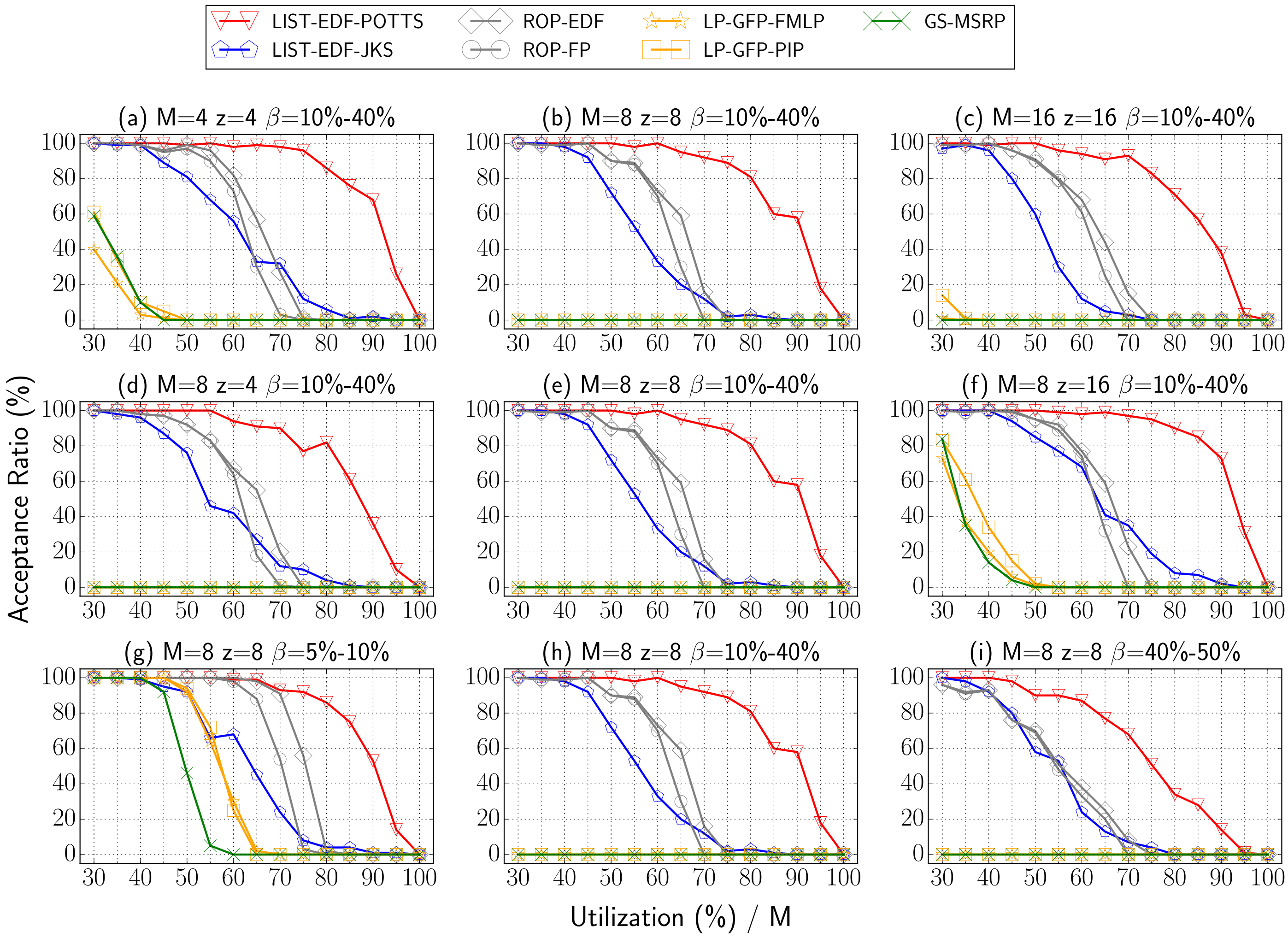


Fig. 4. Comparison of different approaches for semi-harmonic periodic task sets

3) **Workload of Shared Resources, i.e., $\beta \in \{[5\% - 10\%], [10\% - 40\%], [40\% - 50\%]\}$** (Fig. 4(g) to Fig. 4(i)): If the workload of the critical sections is increased, the performance of all methods is reduced. The reason is that, when $\beta = [40\% - 50\%]$, the execution time of the critical section for tasks with period 10 can be large, i.e., longer than 1 ms. Therefore, tasks with period 1 directly miss the deadline for all other approaches and, hence, the performance drops down quickly when the utilization is increased and the critical section workload is large as shown in Fig. 4(i). Nevertheless, due to the non-work-conserving property, the LIST-EDF-POTTS is still able to schedule some of these task sets, i.e., some of the sets where the longest critical sections are larger than 1 ms but shorter than 2 ms.

IX. Tasks with Multiple Critical Sections

In this work, we consider tasks that access exactly one shared resource exactly once. Therefore, an important question is how our approach could be extended to tasks that access the same shared resource multiple times or multiple shared resources (in a non-nested way). For the former case, it can be considered to directly used the provided approach with a slight extension, since the construction of the dependency graph computes the order of the critical sections based on the earliest release times and the absolute deadlines. Therefore, as long as it is ensured that the release time of a critical section is after all previous critical sections of the same job are finished according to the dependency graph, the proposed algorithm can be applied directly.

In the latter case, a direct extension seems not possible anymore. The reason is that the dependency graphs are constructed individually for each resource and therefore it is not trivial to guarantee that previous critical sections of the same task that accessed a different resource are finished at the point in time a critical section is released. However, it is possible to use our approach after modifying tasks that access more than one shared resource in a non-nested way. To achieve independence of the graph constructions, such tasks can be decomposed into two (or more) subtasks that do not overlap during their execution and each task handles one of the resource requests. Consider a task τ_i with $k+1$ non-critical and k critical sections, i.e., τ_i has the execution order $C_{i,1}, A_{i,1}, C_{i,2}, A_{i,2}, ..., A_{i,k}, C_{i,k+1}$. We decompose such a task into k subtasks where the j-th subtask consists of $C_{i,j}$ and $A_{i,j}$, i.e., a task with just one non-critical and one critical section. The only exception is the last subtask that contains of three sections, i.e., $C_{i,k}, A_{i,k}$, and $C_{i,k+1}$. Each subtask is assigned a chunk of the relative deadline D_i such that the j-th subtask has a relative deadline of $D_{i,j}$ and $\sum_{j=1}^{k} D_{i,j} = D_i$. Furthermore, each subtask is assigned a phase $\phi_{i,j}$ based on the summation of the previous relative deadlines to ensure that execution of subtasks does not overlap (as long as all subtasks meet their deadline). To be precise, $\phi_{i,1} = 0$ and for $1 \leq j \leq k$ we set $\phi_{i,j} = \sum_{g=1}^{j-1} D_{i,g}$. While this violates our assumption that all tasks have an offset of 0, it has no negative impact for our algorithm. The reason is that this assumption is only used to ensure that it is sufficient to unroll the periods to one hyper-period since at that point in time no leftover workload of any task remains in the system if all tasks meet their deadline and, therefore, the schedule is ensured to repeat itself. Since we assume that $\sum_{j=1}^{k} D_{i,j} = D_i$ for all tasks that are decomposed, this property still holds. Such decomposition strategies can also be applied when the same shared resource is accessed multiple times.

Note that how the tasks and deadlines are decomposed are open problems and, therefore, left for future research. The problem is similar to the deadline assignment problem for Fixed-Relative-Deadline strategies used for scheduling self-suspending task sets [53]. Common strategies are to divide the deadlines equally or proportionally to the workload in the related interval.

X. Conclusion

We study multiprocessor resource sharing of periodic real-time tasks, develop a framework to construct a dependency graphs for the critical sections, and present the List-EDF scheduling algorithm. To the best of our knowledge, this is the first non-work-conserving approach for multiprocessor resource sharing for periodic real-time tasks without period restrictions. We explain the implementation of List-EDF in LITMUS$^{\text{RT}}$, and show that the overheads are similar to the FMLP implementation in LITMUS$^{\text{RT}}$. Furthermore, we evaluated the performance of List-EDF applying numerical evaluations, observing that our approach outperforms the state-of-the-art in all considered settings.

We believe that this paper presents an initial step rather than a conclusive approach towards multiprocessor synchronization. We show the potential power of non-work-conserving multiprocessor resource sharing. However, further explorations are needed since multiple open problems remain.

- As the schedule derived from the presented List-EDF algorithm has to be applied statically to avoid the multiprocessor timing anomaly, it would be interesting to analyze the schedulability of the online version of List-EDF. Time efficient schedulability tests, like utilization-based analyses, could be helpful for an online version.
- In this work we only evaluated the schedulability of global List-EDF. While we implemented partitioned List-EDF in LITMUS$^{\text{RT}}$ and compared the resulting overheads, how to efficiently partition tasks under List-EDF remains an open problem. Recent research shows that the assumption that global scheduling leads to a higher acceptance rate than partitioned and semi-partitioned algorithms not always holds [9], [14]. However, an improper task partition can result in a poor performance.
- The dependency graph approach does not actually need the locking/unlocking mechanism anymore since the dependency graph handles the mutual exclusion of the critical sections. However, our current implementation in LITMUS$^{\text{RT}}$ is still based on the existing locking mechanism. How to implement the dependency graph approach in modern real-time operating systems with low overhead is an interesting research direction.
- We limit our study to real-time task systems with one non-nested critical section per task. While we discuss some ideas in the previous section, how to deal with multiple non-nested critical sections per task is a challenging open problem.

Acknowledgment

This paper is supported by DFG, as part of the Collaborative Research Center SFB876, project A1 and A3 (http://sfb876.tu-dortmund.de/).

References

[1] RTEMs. http://www.rtems.org/.
[2] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, 1997.
[3] B. Andersson and A. Easwaran. Provably good multiprocessor scheduling with resource sharing. *Real-Time Systems*, 46(2):153–159, 2010.
[4] B. Andersson and G. Raravi. Real-time scheduling with resource sharing on heterogeneous multiprocessors. *Real-Time Systems*, 50(2):270–314, 2014.
[5] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS-164, Department of Computer Science, University of York, 1991.
[6] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 300–309, 2012.

[7] K. R. Baker, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Preemptive scheduling of a single machine to minimize maximum cost subject to release dates and precedence constraints. *Operations Research*, 31(2):381–386, 1983.

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

[9] A. Biondi and Y. Sun. On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling. *Real-Time Systems*, 54(3):515–536, 2018.

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

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

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

[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] B. B. Brandenburg and M. Gul. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*, pages 99–110. IEEE Computer Society, 2016.

[15] 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.

[16] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium (RTSS)*, pages 111–126. IEEE, 2006.

[17] 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. *Real-Time Systems*, 2018. https://link.springer.com/article/10.1007/s11241-018-9316-9.

[18] J.-J. Chen, G. von der Brueggen, J. Shi, and N. Ueter. Dependency graph approach for multiprocessor real-time synchronization. In *Real-Time Systems Symposium (RTSS)*, 2018. Preprint: https://arxiv.org/abs/1809.02892.

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

[20] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[21] R. I. Davis, A. Thekkilakattil, O. Gettings, R. Dobrin, S. Punnekkat, and J.-J. Chen. Exact speedup factors and sub-optimality for non-preemptive scheduling. *Real-Time Systems*, Oct 2017.

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

[23] 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.

[24] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 90–99, 2010.

[25] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Real-Time Systems Symposium (RTSS)*, pages 73–83, 2001.

[26] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Research Report RR-2966, Projet REFLECS, 1996.

[27] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[28] L. A. Hall and D. B. Shmoys. Jackson's rule for single-machine scheduling: Making a good heuristic better. *Math. Oper. Res.*, 17(1):22–35, 1992.

[29] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst. Communication centric design in complex automotive embedded systems. In *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, pages 10:1–10:20, 2017.

[30] P.-C. Hsiu, D.-N. Lee, and T.-W. Kuo. Task synchronization and allocation for many-core real-time systems. In *International Conference on Embedded Software, (EMSOFT)*, pages 79–88, 2011.

[31] 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.

[32] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. Technical report, University of California, Los Angeles, 1955.

[33] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings of the Real-Time Systems Symposium - 1991*, pages 129–139, 1991.

[34] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmark for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[35] 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.

[36] A. Melani, M. Bertogna, R. I. Davis, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. Exact response time analysis for fixed priority memory-processor co-scheduling. *IEEE Trans. Computers*, 66(4):631–646, 2017.

[37] M. Nasri, S. K. Baruah, G. Fohler, and M. Kargahi. On the optimality of RM and EDF for non-preemptive real-time harmonic tasks. In *22nd International Conference on Real-Time Networks and Systems, RTNS*, page 331, 2014.

[38] M. Nasri and G. Fohler. Non-work-conserving non-preemptive scheduling: Motivations, challenges, and potential solutions. In *28th Euromicro Conference on Real-Time Systems, ECRTS*, pages 165–175, 2016.

[39] M. Nasri and M. Kargahi. Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks. *Real-Time Systems*, 50(4):548–584, 2014.

[40] 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.

[41] C. N. Potts. Technical note—analysis of a heuristic for one machine sequencing with release dates and delivery times. *Operations Research*, 28(6):1436–1441, 1980.

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

[43] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88)*, pages 259–269, 1988.

[44] A. Sailer, S. Schmidhuber, M. Deubzer, M. Alfranseder, M. Mucha, and J. Mottok. Optimizing the task allocation step for multi-core processors within autosar. In *2013 International Conference on Applied Electronics*, pages 1–6, Sept 2013.

[45] M. Schoeberl, F. Brandner, and J. Vitek. RTTM: real-time transactional memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 326–333, 2010.

[46] A. Schranzhofer, R. Pellizzoni, J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *Proceedings of the 47th Design Automation Conference, DAC*, pages 332–337, 2010.

[47] A. Schranzhofer, R. Pellizzoni, J. Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 213–222, 2011.

[48] 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.

[49] J. Shi. HDGA-LITMUS-RT. https://github.com/Strange369/Dependency-Graph-Approach-for-Periodic-Tasks, 2019.

[50] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein. System-level timing feasibility test for cyber-physical automotive systems. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, May 2016.

[51] G. von der Bruggen, 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 *Euromicro Conference on Real-Time Systems, ECRTS*, pages 90–101, 2015.

[52] G. von der Brüggen, J.-J. Chen, W.-H. Huang, and M. Yang. Release enforcement in resource-oriented partitioned scheduling for multiprocessor systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, pages 287–296, 2017.

[53] 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.

[54] G. von der Brüggen, N. Ueter, J. Chen, and M. Freier. Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS 2017, Grenoble, France, October 04 - 06, 2017*, pages 108–117, 2017.

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

[56] 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.