Dependency Graph Approach for Multiprocessor Real-Time Synchronization

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

***Abstract*—Over the years, many multiprocessor locking protocols have been designed and analyzed. However, the performance of these protocols highly depends on how the tasks are partitioned and prioritized, and how the resources are shared locally and globally. This paper answers a few fundamental questions when real-time tasks share resources in multiprocessor systems. We explore the fundamental difficulty of the multiprocessor synchronization problem and show that a very simplified version of this problem is $\mathcal{NP}$-hard in the strong sense regardless of the number of processors and the underlying scheduling paradigm. Therefore, the allowance of preemption or migration does not reduce the computational complexity. On the positive side, we develop a dependency-graph approach that is specifically useful for frame-based real-time tasks, i.e., when all tasks have the same period and release their jobs always at the same time. We present a series of algorithms with speedup factors between 2 and 3 under semi-partitioned scheduling. We further explore methodologies for and tradeoffs between preemptive and non-preemptive scheduling algorithms, and partitioned and semi-partitioned scheduling algorithms. Our approach is extended to periodic tasks under certain conditions.**

1 Introduction

In a multi-tasking system, mutual exclusion for the accesses to shared resources, e.g., data structures, files, etc., has to be guaranteed to ensure the correctness of these operations. Such accesses to shared resources are typically done within the so-called *critical sections*, which can be protected by using *binary semaphores* or *mutex locks*. Therefore, at any point in time no two task instances are in their critical sections that access the same shared recourse. Moreover, advanced embedded computing systems heavily interact with the physical world, and *timeliness* of computation is an essential requirement of correctness. To ensure safe operations of such embedded systems, the satisfaction of the real-time requirements, i.e., worst-case timeliness, needs to be verified.

If aborting or restarting a critical section is not allowed, due to mutual exclusion, a higher-priority job may have to be stopped until a lower-priority job unlocks the requested shared resource that was already locked earlier, a so-called priority inversion. The study of mutual exclusion in uniprocessor real-time systems can be traced back to the priority inheritance protocol (PIP) and priority ceiling protocol (PCP) by Sha et al. [41] in 1990 and the stack resource policy (SRP) by Baker [5] in 1991. The Immediate PCP, a variant of the PCP, has been implemented in Ada (called Ceiling locking) and POSIX (called Priority Protect Protocol).

To schedule real-time tasks on multiprocessor platforms, there have been three widely adopted paradigms: partitioned, global, and semi-partitioned scheduling. The *partitioned* scheduling approach partitions the tasks statically among the available processors, 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 how each task/subtask is then assigned to a processor. A comprehensive survey of multiprocessor scheduling in real-time systems can be found in [17].

The design of synchronization protocols for real-time tasks on multiprocessor platforms started with the distributed priority ceiling protocol (DPCP) [40], followed by the multiprocessor priority ceiling protocol (MPCP) [39].[1] The MPCP is based on partitioned fixed-priority scheduling and adopts the PCP for local resources. When requesting global resources that are shared by several tasks on different processors, the MPCP executes the corresponding critical sections with priority boosting. By contrast, under the DPCP, the sporadic/periodic real-time tasks are scheduled based on partitioned fixed-priority scheduling, except when accessing resources that are bound to a different processor. That is, the DPCP is semi-partitioned scheduling that allows migration at the boundary of critical and non-critical sections.

Over the years, many locking protocols have been designed and analyzed, including the multiprocessor stack resource policy (MSRP) [21], the flexible multiprocessor locking protocol (FMLP) [7], the multiprocessor PIP [18], the $O(m)$ locking protocol (OMLP) [11], the Multiprocessor Bandwidth Inheritance (M-BWI) [20], gEDF-vpr [2], LP-EE-vpr [3], and the Multiprocessor resource sharing Protocol (MrsP) [12]. Also, several protocols for hybrid scheduling approaches such as clustered scheduling [10], reservation-based scheduling [20], and open real-time systems [34] have been proposed in recent years. To support nested critical sections, Ward and Anderson [47], [48] introduced the Real-time Nested Locking Protocol (RNLP) [47], which adds supports for fine-grained nested locking on top of non-nested protocols.

However, the performance of these protocols highly depends on 1) how the tasks are partitioned and prioritized, 2) how the resources are shared locally and globally, and 3) whether a job/task being blocked should spin or suspend itself.

Regarding task partitioning, Lakshmanan et al. [29] presented a synchronization-aware partitioned heuristic for the MPCP, which organizes the tasks that share common resources into groups and attempts to assign each group of tasks to the same processor. Following the same principle, Nemati et al. [35] presented a blocking-aware partitioning method

[1]Neither of these two protocols had a concrete name in the original papers. In the literature, most authors referred to the protocols in [40] as DPCP and [39] as MPCP, respectively.

that uses an advanced cost heuristic algorithm to split a task group when the entire group fails to be assigned on one processor. In subsequent work, Hsiu et al. [24] proposed a dedicated-core framework to separate the execution of critical sections and normal sections, and employed a priority-based mechanism for resource sharing, such that each request can be blocked by at most one lower-priority request. Wieder and Brandenburg [50] proposed a greedy slacker partitioning heuristic in the presence of spin locks. The *resource-oriented partitioned* (ROP) scheduling was proposed by Huang et. al [25] in 2016 and later refined by von der Brüggen et al. [45] with release enforcement for a special case.

For priority assignment, most of the results in the literature use rate-monotonic (RM) or earliest-deadline-first (EDF) scheduling. To the best of our knowledge, the priority assignment for systems with shared resources has only been seriously explored in a small numbers of papers, e.g., relative deadline assignment under release enforcement in [45], priority assignment for spinning [1], reasonable priority assignments under global scheduling [18], and the optimal priority assignment used in the greedy slack algorithm in [50]. However, no theoretical evidence has been provided to quantify the non-optimality of the above heuristics.

Although many multiprocessor locking protocols have been proposed in the literature, there are a few unsolved fundamental questions when real-time tasks share resources (via locking mechanisms) in multiprocessor systems:

- *What is the fundamental difficulty?*
- *What is the performance gap of partitioned, semi-partitioned, and global scheduling?*
- *Is it always beneficial to prioritize critical sections?*

To answer the above questions, we focus on the simplest and the most basic setting: all tasks have the same period and release their jobs always at the same time, so-called *frame-based real-time task systems*, and are scheduled on M identical (homogeneous) processors. Specifically, we assume that each critical section is non-nested and is guarded by only one binary semaphore or one mutex lock.

Contribution: Our contributions are as follows:

- We show that finding a schedule of the tasks to meet the given common deadline is $\mathcal{NP}$-hard in the strong sense *regardless of the number of processors M in the system*. Therefore, there is no polynomial-time approximation algorithm that can bound the allocated number of processors to meet the given deadline. Moreover, the $\mathcal{NP}$-hardness holds under any scheduling paradigm. Therefore, the allowance of preemption or migration does not reduce the computational complexity.
- We propose a dependency graph approach for multiprocessor synchronization, which consists of two steps: 1) the construction of a directed acyclic graph (DAG), and 2) the scheduling of this DAG. We prove that, for minimizing the makespan, the approximation ratio of such an approach is lower bounded by at least $2 - \frac{2}{M} + \frac{1}{M^2}$ under any scheduling paradigm and $2 - \frac{1}{M}$ under partitioned or semi-partitioned scheduling.
- We demonstrate how existing results in the literature of uniprocessor non-preemptive scheduling can be adopted to construct the DAG in the first step of the dependency graph approach when each task has only one critical section. This results in several polynomial-time scheduling algorithms with different constant approximation bounds for minimizing the makespan. Specifically, the best approximation developed is a polynomial-time approximation scheme with an approximation ratio of $2 + \epsilon - \frac{1+\epsilon}{M}$ for any $\epsilon > 0$ under semi-partitioned scheduling strategies. We further discuss methodologies for and tradeoffs between preemptive and non-preemptive scheduling algorithms, and partitioned and semi-partitioned scheduling algorithms.
- We also implemented the dependency graph approach as a prototype in LITMUS$^{\text{RT}}$ [8], [13]. The experimental results show that the overhead is almost the same as for state-of-the-art multiprocessor locking protocols. Moreover, we provide extensive numerical evaluations, which demonstrate the performance of the proposed approach under different scheduling constraints. Comparing to the state-of-the-art resource-oriented partitioned (ROP) scheduling, our approach shows significant improvement.

2 System Model

2.1 Task Model

In this paper, we will implicitly consider *frame-based real-time task systems* to be scheduled on M identical (homogeneous) processors. The given tasks release their jobs at the same time and have the same period and relative deadline. Our studied problem is the task synchronization problem where all tasks have exactly one (not nested) critical section, denoted as *TS-OCS*. Specifically, each task τ_i releases a job (at time 0 for notational brevity) with the following properties:

- $C_{i,1}$ is the execution time of the first non-critical section of the job.
- $A_{i,1}$ is the execution time of the (first) critical section of the job, in which a binary semaphore or a mutex $\sigma(\tau_{i,1})$ is used to control the access to the critical section.
- $C_{i,2}$ is the execution time of the second non-critical section of the job.

A subjob is a critical section or a non-critical section. Therefore, each job of task τ_i has three subjobs. We assume the task set $\mathbf{T}$ is given and that the deadline is either implicit, i.e., identical to the period, or constrained, i.e., smaller than the period. The cardinality of a set $\mathbf{X}$ is denoted as $|\mathbf{X}|$. We also make the following assumptions:

- For each task τ_i in $\mathbf{T}$, $C_{i,1} \geq 0$, $C_{i,2} \geq 0$, and $A_{i,1} \geq 0$.
- The execution of the critical sections guarded by one binary semaphore s must be sequentially executed under a total order. That is, if two tasks share the same semaphore, their critical sections must be executed one after another without any interleaving.
- The execution of a job cannot be parallelized, i.e., a job must be sequentially executed in the order of $C_{i,1}, A_{i,1}, C_{i,2}$.
- There are in total z binary semaphores.

This paper will implicitly focus on the above task model. In Section 8, we will explain how the algorithms in this paper can be extended to periodic task systems under certain conditions.

2.2 Scheduling Strategies

Here, we define scheduling strategies and the properties of a schedule for a frame-based real-time task system. Note that the terminology used here is limited to the scenario where each task in **T** releases *only one* job at time 0. Therefore, we will use the term jobs and tasks interchangeable.

A schedule is an assignment of the given jobs (tasks) to one of the M identical processors, such that each job is executed (not necessarily consecutively) until completion. A schedule for **T** can be defined as a function $\rho : \mathbb{R} \times M \rightarrow \mathbf{T} \cup \{\perp\}$, where $\rho(t, m) = \tau_j$ denotes that the job of task τ_j is executed at time t on processor m, and $\rho(t, m) = \perp$ denotes that processor m is idle at time t. We assume that a job has to be sequentially executed, i.e., intra-task parallelism is not possible. Therefore, it is not feasible to run a job in parallel on two processors, i.e., $\rho(t, m) \neq \rho(t, m')$ for any $m \neq m'$ if $\rho(t, m) \neq \perp$.

Some other constraints may also be introduced. A schedule is *non-preemptive* if a job cannot be preempted by any other job, i.e., there is only one interval with $\rho(t, m) = \tau_j$ on processor m for each task τ_j in **T**. A schedule is *preemptive* if a job can be preempted, i.e., more than one interval with $\rho(t, m) = \tau_j$ for any task τ_j in **T** on processor m is allowed.

For a *partitioned* schedule, a job has to be executed on one processor. That is, there is exactly one processor m with $\rho(t, m) = \tau_j$ for every task τ_j in **T**. Such a schedule can be preemptive or non-preemptive. For a *global schedule*, a job can be arbitrarily executed on any of the M processors at any time point. That is, it is possible that $\rho(t, m) = \tau_j$ and $\rho(t', m') = \tau_j$ for $m \neq m'$ and $t \neq t'$. By definition, a global schedule is preemptive (for frame-based real-time task systems) in our model. For a *semi-partitioned* schedule, a subjob (either a critical section or a non-critical section) has to be executed on one processor. Such a semi-partitioned schedule can be preemptive or non-preemptive.

Based on the above definitions, a partitioned schedule is also a semi-partitioned schedule, and a semi-partitioned schedule is also a global schedule.

2.3 Scheduling Theory

In the rich literature of scheduling theory, one specific objective is to minimize the completion time of the jobs, called **makespan**. For frame-based real-time task systems, if the makespan of the jobs released at time 0 is no more than the relative deadline, then the task set can be feasibly scheduled to meet the deadline.[2] We state the makespan problem for *TS-OCS* that is studied here as follows:

Definition 1: **The *TS-OCS* Makespan Problem:** We are given M identical (homogeneous) processors. There are N tasks arriving at time 0. Each task is given by $\{C_{i,1}, A_{i,1}, C_{i,2}\}$ and has at most one critical section, guarded by one binary semaphore. The objective is to find a schedule that minimizes the makespan.

Alternatively, we can also investigate the **bin packing** version of the problem, i.e., minimizing the number of allocated processors to meet a given common deadline D.

[2]Note that the deadline is never larger than the period in our setting.

Definition 2: **The *TS-OCS* Bin Packing Problem:** We are given identical (homogeneous) processors. There are N tasks arriving at time 0 with a common deadline D. Each task is given by $\{C_{i,1}, A_{i,1}, C_{i,2}\}$ and has at most one critical section, guarded by one binary semaphore. The objective is to find a schedule to meet the deadline with the minimum number of allocated processors.

Essentially, the decision versions of the makespan and the bin packing problems are identical:

Definition 3: **The *TS-OCS* Schedulability Problem:** We are given M identical (homogeneous) processors. There are N tasks arriving at time 0 with a common deadline D. Each task is given by $\{C_{i,1}, A_{i,1}, C_{i,2}\}$ and has at most one critical section, guarded by one binary semaphore. The objective is to find a schedule to meet the deadline by using the M processors.

In the domain of scheduling theory, a scheduling problem is described by a triplet $\text{Field}_1|\text{Field}_2|\text{Field}_3$.

- Field_1: describes the machine environment.
- Field_2: specifies the processing characteristics and constraints.
- Field_3: presents the objective to be optimized.

For example, the scheduling problem $1|r_j|L_{\max}$ deals with a uniprocessor system, in which the input is a set of jobs with different release times and different absolute deadlines, and the objective is derive a non-preemptive schedule which minimizes the maximum lateness. The scheduling problem $P||C_{\max}$ deals with a homogeneous multiprocessor system, in which the input is a set of jobs with the same release time, and the objective is to derive a *partitioned* schedule which minimizes the makespan. The scheduling problem $P|prec|C_{\max}$ is an extension of $P||C_{\max}$ by further considering the precedence constraints of the jobs. The scheduling problem $P|prec, prmp|C_{\max}$ further allows preemption. Note that in classical scheduling theory, preemption in parallel machines implies the possibility of job migration from one machine to another.[3] Therefore, the scheduling problem $P|prec, prmp|C_{\max}$ allows job preemption and migration, i.e., preemptive global scheduling.

2.4 Approximation Metrics

Since many scheduling problems are $\mathcal{NP}$-hard in the strong sense, polynomial-time approximation algorithms are often used. In the realm of real-time systems, there are two widely adopted metrics:

The *Approximation Ratio* compares the resulting objectives of (i) scheduling algorithm $\mathcal{A}$ and (ii) an optimal algorithm when scheduling any given task set. Formally, an algorithm $\mathcal{A}$ for the makespan problem (i.e., Definition 1) has an approximation ratio $\alpha \geq 1$, if given any task set **T**, the resulting makespan is at most $\alpha C^*_{\max}$ on M processors, where $C^*_{\max}$ is the minimum (optimal) makespan to schedule **T** on M processors. An algorithm $\mathcal{A}$ for the bin packing problem (i.e., Definition 2) has an approximation ratio $\alpha \geq 1$, if given any task set **T**, it finds a schedule of **T** on αM^* processors to meet

[3]In real-time systems, this is not necessarily the case. For instance, under preemptive partitioned scheduling a job can be preempted and resumed later on the same processor without migration.

the common deadline, where M^* is the minimum (optimal) number of processors required to feasibly schedule $\mathbf{T}$.

The *Speedup Factor* [27], [37] of a scheduling algorithm $\mathcal{A}$ indicates the factor $\alpha \geq 1$ by which the overall speed of a system would need to be increased so that the scheduling algorithm $\mathcal{A}$ always derives a feasible schedule to meet the deadline, provided that there exists one at the original speed. This is used for the problem in Definition 3.

We note that an algorithm that has an approximation ratio α for the makespan problem in Definition 1 also has a speedup factor α for the schedulability problem in Definition 3.

Please note that the speedup factor is only a means to analyze the worst-case behavior of an algorithm. Algorithms with similar speedup factors may differ largely regarding their performance. This fact and how considering speedup factors during the algorithm design can lead to reduced performance has been recently discussed by Chen et al. [16]. To the best of our knowledge, the presented algorithms in this paper do not suffer from any of the potential pitfalls pointed out in [16].

3 Dependency Graph Approach for Multiprocessor Synchronization

To handle the studied makespan problem in Definition 1, we propose a **Dependency Graph Approach** with two steps:

- In the first step, a directed graph $G = (V, E)$ is constructed. A subjob (i.e., a critical or a non-critical section) is a vertex in V and the edges in E describe the precedents constraints of these jobs. The subjob $C_{i,1}$ is a predecessor of the subjob $A_{i,1}$, and $A_{i,1}$ is a predecessor of the subjob $C_{i,2}$. If two jobs of τ_i and τ_j share the same binary semaphore, i.e., $\sigma(\tau_{i,1}) = \sigma(\tau_{j,1})$, then either the subjob $A_{i,1}$ is the predecessor of $A_{j,1}$ or the subjob $A_{j,1}$ is the predecessor of $A_{i,1}$. All the critical sections guarded by a binary semaphore form a chain in G, i.e., the critical sections of the binary semaphore follow a total order. Therefore, we have the following properties in set E:
  - The two directed edges $(C_{i,1}, A_{i,1})$ and $(A_{i,1}, C_{i,2})$ are in E.
  - Suppose that $\mathbf{T}_k$ is the set of tasks which require the same binary semaphore s_k. Then, the $|\mathbf{T}_k|$ tasks in $\mathbf{T}_k$ follow a certain total order π such that $(A_{i,1}, A_{j,1})$ is a directed edge in E when $\pi(\tau_i) = \pi(\tau_j) - 1$.

  Fig. 1 provides an example for a task dependency graph with one binary semaphore. Since there are z binary semaphores in the task set, the task dependency graph G has in total z connected subgraphs, denoted as $G_1, G_2, \ldots, G_z$. In each connected subgraph G_ℓ, the corresponding critical sections of the tasks that request critical sections guarded by the same semaphore form a chain and have to be executed sequentially. For example, in Fig. 1, the dependency graph forces the scheduler to execute the critical section $A_{1,1}$ prior to any of the other three critical sections.
- In the second step, a corresponding schedule of G on M processors is generated. The schedule can be based on system's restrictions or user's preferences, i.e., either preemptive or non-preemptive schedules, either global, semi-partitioned, or partitioned schedules.

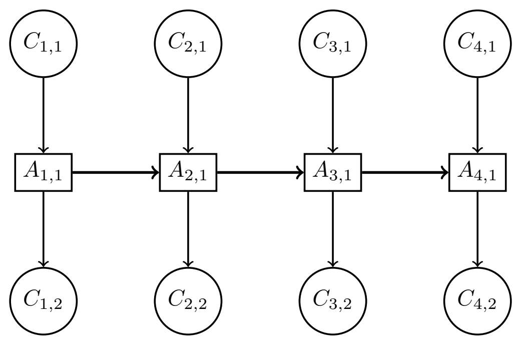
$$\begin{tikzcd}
C_{1,1} \arrow[d] & C_{2,1} \arrow[d] & C_{3,1} \arrow[d] & C_{4,1} \arrow[d] \\
A_{1,1} \arrow[r] \arrow[d] & A_{2,1} \arrow[r] \arrow[d] & A_{3,1} \arrow[r] \arrow[d] & A_{4,1} \arrow[d] \\
C_{1,2} & C_{2,2} & C_{3,2} & C_{4,2}
\end{tikzcd}$$

Fig. 1. A task dependency graph for a task set with one binary semaphore.

The second step of the dependency graph has been wildly studied in scheduling theory. A solution of the problem $P|prec|C_{\max}$ results in a semi-partitioned schedule, since the dependency graph is constructed by considering a critical section or a non-critical section as a subjob. Moreover, a solution of the problem $P|prec, prmp|C_{\max}$ results in a global schedule. For deriving a partitioned schedule, we can force the subjobs generated by a job to be *tied* to one processor. That is, $P|prec, tied|C_{\max}$ targets a partitioned non-preemptive schedule and $P|prec, prmp, tied|C_{\max}$ targets a partitioned preemptive schedule.

Therefore, the key issue is the construction of the dependency graph, i.e., the first step. An alternative view of the dependency graph approach is to build the dependency graph assuming an always sufficient number of processors, i.e., assuming an infinite number of processors, and then the second step considers the constraint of the number of processors. Towards the first step, we need the following definition:

Definition 4: A **critical path** of a task dependency graph G is one of the longest paths of G. The critical path length of G is denoted by $len(G)$.

For the rest of this paper, we denote a dependency task graph of the input task set $\mathbf{T}$ that has the minimum critical path length as G^*. Note that G^* is independent of M.

Lemma 1: *$len(G^*)$ is the lower bound of the TS-OCS makespan problem for task set* $\mathbf{T}$ *on* M *processors.*

Proof: This comes from the setting of the problem, i.e., each task τ_i has only one critical section guarded by one binary semaphore, and the definition of the graph G^*, i.e., using as many processors as possible. ■

Definition 5: A **feasible schedule** $S(G)$ of a task dependency graph G respects the precedence constraints defined in G and the specified scheduling requirement, e.g., being global/semi-partitioned/partitioned and preemptive/non-preemptive. $L(S(G))$ is the makespan of $S(G)$.

With the above definitions, we can recap the objectives of the two steps in the dependency graph approach. In the first step, we would like to construct a dependency graph G to minimize $len(G)$, and in the second step, we would like to construct a schedule $S(G)$ to minimize $L(S(G))$.

We conclude this section by stating the following theorem:

Theorem 1: *The optimal makespan of the TS-OCS* makespan problem for $\mathbf{T}$ on M processors is at least

$$\max\left\{\sum_{\tau_i \in \mathbf{T}} \frac{C_{i,1}+A_{i,1}+C_{i,2}}{M}, len(G^*)\right\} \quad (1)$$

where G^* is a dependency task graph of $\mathbf{T}$ that has the minimum critical path length.

Proof: The lower bound $len(G^*)$ comes from Lemma 1 and the lower bound $\sum_{\tau_i \in \mathbf{T}} \frac{C_{i,1}+A_{i,1}+C_{i,2}}{M}$ is due to the pigeon hole principle. ■

4 Computational Complexity and Lower Bounds

This section presents the computational complexity and lower bounds for the approximation ratios of the dependency graph approach.

4.1 Computational Complexity

The following theorem shows that constructing G^* is unfortunately $\mathcal{NP}$-hard in the strong sense.

Theorem 2: Constructing a dependency task graph G^* that has the minimum critical path length is $\mathcal{NP}$-hard in the strong sense.

Proof: This theorem is proved by a reduction from the decision version of the scheduling problem $1|r_j|L_{\max}$, i.e., uniprocessor non-preemptive scheduling, in which the objective is to minimize the maximum lateness assuming that each job J_j in the given job set $\mathbf{J}$ has its known processing time $p_j \geq 0$, arrival time $r_j \geq 0$, and absolute deadline d_j. This problem is $\mathcal{NP}$-hard in the strong sense by a reduction from the 3-Partition problem [31]. Suppose that the decision version of the scheduling problem $1|r_j|L_{\max}$ is to validate whether there exists a schedule in which the finishing time of each job J_j is no less than d_j.

Let H be any positive integer greater than $\max_{j\in\mathbf{J}} d_j$. For each job J_j in $\mathbf{J}$, we construct a task τ_j with one critical section, where $C_{j,1}$ is set to r_j, $C_{j,2}$ is set to $H-d_j$, and $A_{j,1}$ is set to p_j. By the setting, $C_{j,1} \geq 0, C_{j,2} \geq 0$, and $A_{j,1} \geq 0$ for every constructed task τ_j. The critical sections of all the constructed tasks are guarded by *only one* binary semaphore. Let the task set constructed above be $\mathbf{T}$. The above input task set $\mathbf{T}$ by definition is a feasible input task set for the one-critical-section task synchronization problem (*TS-OCS*).

We now prove that there is a non-preemptive uniprocessor schedule for $\mathbf{J}$ in which all the jobs can meet their deadlines if and only if there is a dependency task graph G^* with a critical path length less than or equal to H for the constructed task set $\mathbf{T}$.

If part, i.e., $len(G^*) \leq H$ holds: Without loss of generality, we index the tasks in $\mathbf{T}$ so that the critical section of $A_{i,1}$ is the immediate predecessor of the critical section $A_{i+1,1}$ in G^*, e.g., as in Fig. 1. Suppose that $G^*(\tau_i)$ is the subgraph of G^* that consists of only the vertices representing $\{C_{k,1}, A_{k,1}, C_{k,2} \mid k = 1, 2, \ldots, i-1\} \cup \{C_{i,1}, A_{i,1}\}$ and the corresponding edges. Let f_i be the longest path in $G^*(\tau_i)$ that *ends at the vertex representing* $A_{i,1}$.

By definition, f_1 is $C_{1,1} + A_{1,1}$. Moreover, f_i is $\max\{f_{i-1}, C_{i,1}\}+A_{i,1}$ for every task τ_i in $\mathbf{T}$. Since $len(G^*) \leq H$ and $C_{i,2} = H-d_i$, we know that $f_i+C_{i,2} \leq H \Rightarrow f_i \leq d_i$ for every task τ_i in $\mathbf{T}$.

We can now construct the uniprocessor non-preemptive schedule for $\mathbf{J}$ by following the same execution order. Here, we index the jobs in $\mathbf{J}$ corresponding to $\mathbf{T}$. The finishing time of job J_1 is $r_1+p_1 = C_{1,1}+A_{1,1} = f_1$. The finishing time of job J_i is $\max\{f_{i-1}, r_i\}+p_i = \max\{f_{i-1}, C_{i,1}\}+A_{i,1} = f_i$.

This proves the if part.

Only-If part, i.e., there is a uniprocessor non-preemptive schedule in which all the deadlines of the jobs in $\mathbf{J}$ are met: The proof for the **if part** can be reverted and the same arguments can be applied. Due to space limitation, details are omitted. ■

Theorem 3: The makespan problem with task synchronization for $\mathbf{T}$ on M processors is $\mathcal{NP}$-hard in the strong sense even if M is sufficiently large under any scheduling paradigm.

Proof: This follows directly from Theorem 2. Consider $M \geq |\mathbf{T}|+1$ processors. The if-and-only-if proof in Theorem 2 can be extended by introducing a concrete schedule that executes the two non-critical sections of task τ_i on processor i and the critical section of task τ_i on processor $|\mathbf{T}|+1$.[4] ■

Theorem 3 expresses the fundamental difficulty of the multiprocessor synchronization problem and shows that a very simplified version of this problem is $\mathcal{N}P$-hard in the strong sense regardless of the number of processors and the underlying scheduling paradigm. Therefore, the allowance of preemption or migration does not reduce the computational complexity. The fundamental problem is the sequencing of the critical sections, which is independent from the underlying scheduling paradigm. Therefore, no matter what flexibility the scheduling algorithm has (unless aborting and restarting a critical section is allowed), the computational complexity remains $\mathcal{N}P$-hard in the strong sense.

4.2 Remarks: Bin Packing

Although the focus of this paper is the makespan problem in Definition 1 and the schedulability problem in Definition 3, we also state the following theorems to explain the difficulty of the bin packing problem in Definition 2.

Theorem 4: Minimizing the number of processors for a given common deadline of $\mathbf{T}$ with task synchronization for $\mathbf{T}$ (i.e., Definition 2) is $\mathcal{NP}$-hard in the strong sense under any scheduling paradigm.

Proof: As the decision problem is Definition 3, we reach the conclusion based on Theorem 3. ■

Theorem 5: There is no polynomial-time (approximation) algorithm to minimize the number of processors for a given common deadline of $\mathbf{T}$ with task synchronization for $\mathbf{T}$ under any scheduling paradigm unless $\mathcal{P} = \mathcal{NP}$.

Proof: This is based on Theorems 2 and 3. If such a polynomial-time algorithm exists, then the problem $1|r_j|L_{\max}$ is solvable in polynomial time, which implies $\mathcal{P} = \mathcal{NP}$. ■

[4]The same statement also holds for using $M = |\mathbf{T}|$ processors, but the proof is more complicated.

4.3 Lower Bounds

The dependency graph approach requires two steps. The following theorem shows that even if both steps are optimized, the resulting schedule for the makespan problem with task synchronization is not optimal and has an asymptotic lower bound 2 of the approximation ratio.

Theorem 6: The optimal schedule on M identical processors for the dependency graph G^* that has the minimum critical path length is not optimal for the *TS-OCS* makespan problem and can have an approximation bound of at least

- $2 - \frac{2}{M} + \frac{1}{M^2}$ under any scheduling paradigm, and
- $2 - \frac{1}{M}$ under partitioned or semi-partitioned scheduling.

Proof: We prove this theorem by providing a concrete input instance as follows:

- Suppose that M is a given integer with $M \geq 2$ and we have $N = M^2 - M + 1$ tasks.
- We assume a very small positive number δ and a number Q which is much greater than δ, i.e., $\frac{Q}{MN} \gg \delta > 0$.
- All N tasks have a critical section guarded by the same binary semaphore.
- Task τ_1 has $C_{1,1} = \delta, A_{1,1} = Q - \frac{Q}{M}$, and $C_{1,2} = \frac{Q}{M} + N\delta$
- Task τ_i has $C_{i,1} = \delta, A_{i,1} = \delta$, and $C_{i,2} = \frac{Q}{M}$ for $i = 2, 3, \ldots, N$.

We need to show that the optimal dependency graph of this input instance in fact leads to the specified bound. The proof is in Appendix. ■

5 Algorithms to Construct G

The key to success is to find G^*. Unfortunately, as shown in Theorem 2, finding G^* is $\mathcal{NP}$-hard in the strong sense. However, finding good approximations is possible. The problem to construct G is called the *dependency-graph construction problem*. Here, instead of presenting new algorithms to find good approximations of G^*, we explain how to use the existing algorithms of the scheduling problem $1|r_j|L_{\max}$ to derive good approximations of G^*.

It should be first noted that the problem $1|r_j|L_{\max}$ cannot be approximated with a bounded approximation ratio because the optimal schedule may have no lateness at all and any approximation leads to an unbounded approximation ratio. However, a variant of this problem can be easily approximated. This is known as the *delivery-time* model of the problem $1|r_j|L_{\max}$. In this model, each job J_j has its release time r_j, processing time p_j, and delivery time $q_j \geq 0$. After a job finishes its execution on a machine, its result (final product) needs q_j amount of time to be delivered to the customer. The objective is to minimize the makespan K. Therefore, the *effective* deadline d_j of job J_j on the given single machine is $d_j = K - q_j$. Since K is a constant, this is effectively equivalent to the case when d_j is set to $-q_j$.

The delivery-time model of the problem $1|r_j|L_{\max}$ can then be effectively approximated. Moreover, our problem to construct a good dependency graph for $\mathbf{T}$ is indeed equivalent to the delivery-time model of the problem $1|r_j|L_{\max}$. To show such equivalence, Algorithm 1 presents the detailed transformation. For each semaphore s_k, suppose that $\mathbf{T}_k$ is the set of tasks that use s_k (Line 1 in Algorithm 1). For each task set $\mathbf{T}_k$, we transform the problem to construct G_k to an equivalent delivery-time model of the problem $1|r_j|L_{\max}$ (Line 3 to Line 8). Then, we construct the graph G_k based on the derived schedule of an approximation algorithm for the delivery-time model of the problem $1|r_j|L_{\max}$.

Theorem 7: An α-approximation algorithm for the delivery-time model of the problem $1|r_j|L_{\max}$ applied in Algorithm 1 guarantees to derive a dependency graph G with $len(G) \leq \alpha \times len(G^*)$.

Proof: This theorem can be proved by a counterpart of the proof of Theorem 2. We will show that Algorithm 1 is in fact an L-reduction (i.e., a reduction that preserves the approximation ratio) from the input task set to the delivery-time model of the problem $1|r_j|L_{\max}$. In this L-reduction, there is no loss of the approximation ratio.

First, by definition, two tasks are independent if they do not share any semaphore. Moreover, since the *TS-OCS* problem assumes that a task accesses at most one binary semaphore, a task τ_i can only appear at most in one $\mathbf{T}_k$ for a certain k. Therefore, $len(G^*) = \max_{k=1,2,\ldots,z} len(G^*_k)$.

To show that the reduction preserves the approximation ratio, we only need to prove the one-to-one mapping. One possibility is to prove that a schedule for the input instance of the problem $1|r_j|L_{\max}$ delivers the last result at time X if and only if the corresponding graph G_k constructed by using Lines 9 and 10 in Algorithm 1 has a critical path length X. This is unfortunately not possible because a (*technically bad but possible*) schedule for the input instance of the problem $1|r_j|L_{\max}$ can be arbitrarily alerted by inserting useless delays.

Fortunately, for a given permutation to order the $|\mathbf{T}_k|$ tasks in $\mathbf{T}_k$, we can always construct a schedule for the input instance of the problem $1|r_j|L_{\max}$ by respecting the given order and their release times. Such a schedule for the input instance of the problem $1|r_j|L_{\max}$ delivers the last result at time X if and only if the corresponding graph G_k constructed by using Lines 9 and 10 in Algorithm 1 has a critical path length X. Moreover, the schedule for one such permutation is optimal for the input instance of the problem $1|r_j|L_{\max}$.

Therefore, the approximation ratio is perserved while constructing G_k. According to the above discussions, $len(G_k) \leq \alpha \times len(G^*_k)$. Moreover,

$$\begin{aligned} len(G) \leq & \max_{k=1,2,\ldots,z} len(G_k) \\ & \leq \alpha \times \max_{k=1,2,\ldots,z} len(G^*_k) = \alpha \times len(G^*) \end{aligned}$$

■

According to Theorem 7 and Algorithm 1, we can simply apply the existing algorithms of the scheduling problem $1|r_j|L_{\max}$ in the delivery-time model to derive G^* by using well-studied branch-and-bound methods, see for example [14], [33], [36], or good approximations of G^*, see for example [23], [38]. Here, we will summarize several polynomial-time approximation algorithms. The details can be found in [23].

For the delivery-time model of the scheduling problem $1|r_j|L_{\max}$, the **extended Jackson's rule** (**JKS**) is as follows: "Whenever the machine is free and one or more jobs is

Algorithm 1 Graph Construction Algorithm

```
Input: set $\mathbf{T}$ of $N$ tasks with $z$ shared binary semaphores;
 1: $\mathbf{T}_k \leftarrow \{\tau_i \mid \sigma(\tau_{i,1}) = s_k\}$ for $k = 1, 2, \ldots, z$;
 2: for $k \leftarrow 1$ to $z$ do
 3:     $\mathbf{J} \leftarrow \emptyset$;
 4:     for each $\tau_i \in \mathbf{T}_k$ do
 5:         create a job $J_i$ with $r_i \leftarrow C_{i,1}$, $p_i \leftarrow A_{i,1}$, and $q_i \leftarrow C_{i,2}$,
            where $q_i$ is the delivery time;
 6:         $\mathbf{J} \leftarrow \mathbf{J} \cup \{J_i\}$;
 7:     end for
 8:     apply an approximation algorithm to derive a non-preemptive schedule
        $\rho_k$ for the delivery-time model of the problem $1|r_j|L_{\max}$ on one
        machine;
 9:     construct the initial dependency graph $G_k$ for $\mathbf{T}_k$, with the directed
        edges $(C_{i,1}, A_{i,1})$ and $(A_{i,1}, C_{i,2})$ for every task $\tau_i \in \mathbf{T}_k$;
10:     create a directed edge from $A_{i,1}$ to $A_{j,1}$ in $G_k$ if job $J_j$ is executed
        right after (but not necessarily consecutively to) job $J_i$ in $\rho_k$;
11: end for
12: return $G = G_1 \cup G_2 \cup \ldots \cup G_z$;
```

available for processing, schedule an available job with largest delivery time," as explained in [23].

Lemma 2: The extended Jackson's rule (**JKS**) is a polynomial-time 2-approximation algorithm for the dependency-graph construction problem.

Proof: This is based on Theorem 7 and the approximation ratio of **JKS** for the problem $1|r_j|L_{\max}$, where the proof can be found in [28]. ■

Potts [38] observed some nice properties when the extended Jackson's rule is applied. Suppose that the last delivery is due to a job J_c. Let J_a be the earliest scheduled job so that the machine in the problem $1|r_j|L_{\max}$ is not idle between the processing of J_a and J_c. The sequence of the jobs that are executed sequentially from J_a to J_c is called a *critical sequence*. By the definition of J_a, all jobs in the critical sequence must be released no earlier than the release time r_a of job J_a. If the delivery time of any job in the critical sequence is not shorter than the delivery time q_c of J_c, then it can be proved that the extended Jackson's rule is optimal for the problem $1|r_j|L_{\max}$. However, if the delivery time q_b of a job J_b in the critical sequence is shorter than the delivery time q_c of J_c, the extended Jackson's rule may start a non-preemptive job J_b too early. Such a job J_b that appears last in the critical sequence is called the *interference job* of the critical sequence.

Potts [38] suggested to *attempt at improving the schedule by forcing some interference job to be executed after the critical job* J_c, i.e., by delaying the release time of J_b from r_b to $r'_b = r_c$. This procedure is repeated for at most n iterations and the best schedule among the iterations is returned as the solution.

Lemma 3: Potts' iterative process (**Potts**) is a polynomial-time 1.5-approximation algorithm for the dependency-graph construction problem.

Proof: This is based on Theorem 7 and the approximation ratio of **Potts** for the problem $1|r_j|L_{\max}$, where the proof can be found in [23]. ■

Hall and Shmoys [23] further improved the approximation ratio to $4/3$ by handling a special case when there are two jobs J_i and J_h with $p_i > P/3$ and $p_h > P/3$ where P is $\sum_{J_j} p_j$ and running Potts' algorithm for $2n$ iterations.[5]

Lemma 4: Algorithm **HS** is a polynomial-time $4/3$-approximation algorithm for the dependency-graph construction problem.

Proof: This is based on Theorem 7 and the approximation ratio of **HS** for the problem $1|r_j|L_{\max}$, where the proof can be found in [23]. ■

The algorithm that has the best approximation ratio for the delivery-time model of the problem $1|r_j|L_{\max}$ is a polynomial-time approximation scheme (PTAS) developed by Hall and Shmoys [23].

Lemma 5: The dependency-graph construction problem admits a polynomial-time approximation scheme (PTAS), i.e., the approximation bound is $1 + \epsilon$ under the assumption that $\frac{1}{\epsilon}$ is a constant for any $\epsilon > 0$.

6 Algorithms to Schedule Dependency Graphs

This section presents our heuristic algorithms to schedule the dependency graph G derived from Algorithm 1. We first consider the special case when there is a sufficient number of processors, i.e., $M \geq N$.

Lemma 6: Assume a given task set $\mathbf{T}$, M identical processors, and a given dependency graph G. The makespan of the schedule which executes task τ_i on exactly one processor i as early as possible by respecting to the precedence constraints defined in G is $len(G)$ if $M \geq N$. By definition, this is a partitioned schedule for the given jobs which is non-preemptive with respect to the subjobs.

Proof: Since $M \geq N$, all the tasks can start their first non-critical sections at time 0. Therefore, the critical section of task τ_i arrives exactly at time $C_{i,1}$. Then, the finishing time of the critical section of task τ_i is exactly the longest path in G that finishes at the vertex representing $A_{i,1}$. Therefore, the makespan of such a schedule is exactly $len(G)$. ■

For the remaining part of this section, we will focus on the other case, i.e., when $M < N$. We will heavily utilize the concept of list schedules developed by Graham [22] and extensions of list scheduling to schedule the dependency graph G derived from Section 5. 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 G have finished), one of the eligible subjobs is executed on the processor. When the number of eligible subjobs is larger than the number of idle processors, many heuristic strategies exist to decide which subjobs should be executed with higher priorities. Graham [22] showed that list schedules can be generated in polynomial time and have a $2 - \frac{1}{M}$ approximation ratio for the scheduling problem $P|prec|C_{\max}$.

We will now explain how to use or extend list schedules to generate partitioned or semi-partitioned as well as preemptive or non-preemptive schedules based on G.

[5]Hall and Shmoys [23] further use the concept of forward and inverse problems of the input instance of $1|r_j|L_{\max}$. As they are not highly related, we omit those details.

6.1 Semi-Partitioned Scheduling

Since the subjobs of a task are scheduled individually in list scheduling, a task may migrate among different processors in the generated list schedule, i.e., resulting in a semi-partitioned schedule. However, a subjob by default is non-preemptive in list schedules.

The following lemma is widely used in the literature for the list schedules developed by Graham [22]. All the existing results of federated scheduling, e.g., [6], [15], [32], for scheduling sporadic dependent tasks (that are not due to synchronizations) all implicitly or explicitly use this property.

Lemma 7: The makespan of a list schedule of a given task dependency graph G for task set $\mathbf{T}$ on M processors is at most $\frac{\sum_{\tau_i \in \mathbf{T}}(C_{i,1}+A_{i,1}+C_{i,2})-len(G)}{M} + len(G)$.

Proof: The original proof can be traced back to Theorem 1 by Graham [22] in 1969. We omit the proof here as this is a standard procedure in the proof of list schedules for the scheduling problem $P|prec|C_{\max}$. ■

Lemma 8: If $len(G) \leq \alpha \times len(G^*)$ for a certain $\alpha \geq 1$, the makespan of a list schedule of the task dependency graph G for task set $\mathbf{T}$ on M processors has an approximation bound of $1 + \alpha - \frac{\alpha}{M}$ if $M < N$.

Proof: Since $M < N$, the makespan of a list schedule of G, denoted as $L(List(G))$, is

$$\begin{aligned}
& L(List(G)) \\
\overset{\text{Lemma 7}}{\leq} \;& \frac{(\sum_{\tau_i \in \mathbf{T}} C_{i,1} + C_{i,2} + A_{i,1}) - len(G)}{M} + len(G) \\
= \;& \frac{\sum_{\tau_i \in \mathbf{T}} C_{i,1} + C_{i,2} + A_{i,1}}{M} + len(G)(1 - \frac{1}{M}) \\
\overset{\text{assumption}}{\leq} \;& \frac{\sum_{\tau_i \in \mathbf{T}} C_{i,1} + C_{i,2} + A_{i,1}}{M} + \alpha \times len(G^*)(1 - \frac{1}{M}) \\
\overset{\text{Theorem 1}}{\leq} \;& (1 + \alpha - \frac{\alpha}{M})OPT \qquad (2)
\end{aligned}$$

■

We now conclude the approximation ratio.

Theorem 8: When applying **JKS** ($\alpha = 2$, from Lemma 2), **Potts** ($\alpha = 1.5$, from Lemma 3), **HS** ($\alpha = 4/3$, from Lemma 4), and PTAS ($\alpha = \epsilon$ for any $\epsilon > 0$, from Lemma 5) to generate the task dependency graph G, the *TS-OCS* Makespan problem admits polynomial-time algorithms to generate a semi-partitioned schedule that has an approximation ratio of

$$\begin{cases} \alpha & \text{if } M \geq N \\ 1 + \alpha - \frac{\alpha}{M} & \text{if } M < N \end{cases} \qquad (3)$$

Proof: The case when $M < N$ comes from Lemma 8. The case when $M \geq N$ comes from Lemma 6 and the fact that a partitioned schedule is also a semi-partitioned schedule by definition. ■

The default list schedulers are non-preemptive in the subjob level. However, it may be more efficient if the second non-critical section of a task can be preempted by a critical section. Otherwise, some processors may be busy executing second non-critical sections and a critical section has to wait. As a result, not only this critical section itself but also its successors in G may be unnecessary postponed and therefore increase the makespan. Allowing such preemption in the scheduler design can be achieved easily as follows:

- In the algorithm, the scheduling decision is made at a time t when there is a subjob eligible or finished.
- Whenever a subjob representing a critical section is eligible, it can be assigned to a processor that executes a second non-critical section of a job by preempting that subjob.

The makespan of the resulting schedule remains at most $\frac{\sum_{\tau_i \in \mathbf{T}}(C_{i,1}+A_{i,1}+C_{i,2})-len(G)}{M} + len(G)$ as in Lemma 7. Therefore, the approximation ratios in Theorem 8 still hold even if preemption of the second non-critical sections is possible.

6.2 Partitioned Scheduling

In a partitioned schedule of the frame-based task set $\mathbf{T}$, all subjobs of a task must be executed on the same processor. Therefore, the list scheduling algorithm variant must ensure that once the first subjob $C_{i,1}$ of task τ_i is executed on a processor, all subsequent subjobs of task τ_i are tied to the same processor in any generated list schedule. Specifically, the problem is termed as $P|prec, tied|C_{\max}$ in Section 2.3.

A special case of $P|prec, tied|C_{\max}$ has been recently studied to analyze OpenMP systems by Sun et al. [43] in 2017. They assumed that the synchronization subjob of a task always takes place *at the end of* the task. Our dependency graph G unfortunately does not satisfy the assumption because the synchronization subjob is in fact in the middle of a task. Nevertheless, the algorithm in [43] can still easily be applied. We illustrate the key strategy by using Fig. 2. The subgraph $\bar{G}$ of G that consists of only the vertices of the first non-critical sections and the critical sections in fact satisfies the assumption made by Sun et al. [43]. Therefore, we can generate a multiprocessor schedule for the dependency graph $\bar{G}$ on M processors by using the BFS* algorithm (an extension of the breadth-first-scheduling algorithm) by Sun et al. [43]. It can be imagined that the subjobs that represent the second non-critical sections $C_{i,2}$ are *background* workload and can be executed only at the end of the schedule or when the available idle time is sufficient to complete $C_{i,2}$.

Alternatively, in order to improve the parallelism, another heuristic algorithm can be applied where all the first non-critical sections are scheduled before any of the critical sections, using list scheduling. Once the first non-critical section $C_{i,1}$ of task τ_i is assigned on a processor, the remaining execution of task τ_i is forced to be executed on that processor. For completeness, we illustrate this in Algorithm 2 in the Appendix, assuming that second non-critical sections can be preempted by critical sections and are handled as background workload.

7 Timing Anomaly

So far, we assume that $C_{i,1}$, $A_{i,1}$, and $C_{i,2}$ are exact for a task τ_i. However, the execution of a subjob of task τ_i can be finished earlier than in the worst case. It should be noted that list schedules are in this case not sustainable, i.e., the reduction of the execution time of a subjob can lead to a worse makespan due to the well-known multiprocessor timing

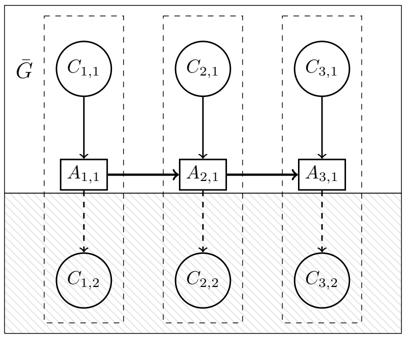

Fig. 2. A schematic of a dependency graph with tied tasks.

anomaly observed by Graham [22]. There are three ways to handle such timing anomaly: 1) ignore the early completion and stick to the offline schedule, 2) reclaim the unused time (slack) carefully without creating timing anomaly, e.g., [51], or 3) use a safe upper bound, e.g., Lemma 7 to account for all possible list schedules. Each of them has advantages and disadvantages. It is up to the designers to choose whether they want to be less effective (Option 1), pay more runtime overhead (Option 2), or be more pessimistic by taking always a safe upper bound (Option 3).

Due to multiprocessor timing anomaly, a dependency graph with a longer critical path may have a better makespan in the resulting list schedule. Our approach can be easily improved by returning and scheduling the intermediate dependency graphs in Algorithms Potts and HS.

8 Periodic Tasks with Different Periods

Our approach can be extended to periodic tasks with different periods under an assumption that a binary semaphore is only shared among the tasks that have the same period. For each of the z semaphores, a DAG is constructed using Algorithm 1. Afterwards, the z resulting DAGs can be scheduled using any approach for multiprocessor DAG scheduling, e.g., global scheduling [30], Federated Scheduling [32] as well as enhanced versions like Semi-Federated Scheduling [26] and Reservation-Based Federated Scheduling [44].

9 Evaluations

This section presents the evaluations of the proposed approach. We first provide the measured overhead of our approach in LITMUS$^{\text{RT}}$. Details regarding the LITMUS$^{\text{RT}}$ implementation can be found in the Appendix. Afterwards, we evaluate the performance by applying numerical evaluations under different configurations.

9.1 Implementations and Overheads

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. Both the partitioned and the semi-partitioned scheduling algorithms for the dependency graph approach, presented in Section 6, have been implemented in LITMUS$^{\text{RT}}$ in order to investigate the overheads, under the plug-in Partitioned Fixed Priority (P-FP). Details can be found in the Appendix. Out implementation has been released in [42].

Max.(Avg.) in μs	DPCP	MPCP	PDGA	SDGA
CXS	30.93 (1.51)	31.1 (0.67)	31.21 (0.71)	30.95 (1.54)
RELEASE	32.63 (3.96)	19.48 (3.91)	19.77 (4.03)	21.64 (4.3)
SCHED2	28.7 (0.18)	29.78 (0.15)	29.91 (0.16)	29.74 (0.2)
SCHED	31.43 (1.2)	31.38 (0.78)	31.4 (0.83)	31.26 (1.11)
SEND-RESCHED	47.01 (14.42)	31.83 (3.45)	45.23 (4.33)	41.53 (7.24)

TABLE I. OVERHEADS OF DIFFERENT PROTOCOLS IN LITMUS$^{\text{RT}}$.

To analyze the applicability of our approach, we tracked different overheads under LITMUS$^{\text{RT}}$:

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

Table I reports the overheads of different protocols in LITMUS$^{\text{RT}}$, namely of the existing implementations of DPCP) [40] and (MPCP) [39] in LITMUS$^{\text{RT}}$ and our implementation of the partitioned dependency graph approach (PDGA) and the semi-partitioned dependency graph approach (SDGA). Table I shows that the overheads of our approach and of other protocols are comparable in LITMUS$^{\text{RT}}$.

9.2 Numerical Performance Evaluations

We conducted evaluations with M = 4, 8, and 16 processors. Depending on M, we generate 1000 task sets, each with $10M$ tasks. For each task set **T**, we generated synthetic tasks with $\sum_{\tau_i \in \mathbf{T}} C_{i,1} + C_{i,2} + A_{i,1} = M$ by applying the RandomFixedSum method [19] and enforced that $C_{i,1}+C_{i,2}+A_{i,1} \leq 0.5$ for each task τ_i. The number of shared resources (binary semaphores) was set to $z \in \{4, 8, 16\}$. The length of the critical section $A_{i,1}$ is a fraction of the total execution time $C_{i,1}+C_{i,2}+A_{i,1}$ of task τ_i, depended on $\beta \in \{5\%-50\%\}$. The remaining part C_i was split into $C_{i,1}$ and $C_{i,2}$ by drawing $C_{i,1}$ randomly uniform from $[0, C_i]$ and setting $C_{i,2}$ to $C_i - C_{i,1}$.

For a generated task set **T**, we calculated a lower bound LB on the optimal makespan based on Eq. (1). Since deriving $len(G^*)$ is computationally expensive, we used $\min_{\tau_i \in \mathbf{T}} C_{i,1} + \min_{\tau_i \in \mathbf{T}} C_{i,2} + \max_{k=1,\ldots,z} CriticalSum_k$ as a safe approximation for $len(G^*)$, where $CriticalSum_k$ is the summation of the lengths of the critical sections that share semaphore s_k. If the relative deadline of the task set is less than LB, the task set is not schedulable by any algorithm. We compare the performance of different algorithms according to the *acceptance ratio* by setting the relative deadline $D = T$ in the range of $[LB, 1.8LB]$. We name the developed algorithms using the following rules: 1) *JKS/POTTS* in the first part: using the extended Jackson's rule or Potts to construct the dependency graph;[6] 2) *SP/P* in the second part: semi-partitioned or partitioned scheduling algorithm is applied[7]; 3) *P/NP* in the

[6]We did not implement Lemma 5 due to the complexity issue. Algorithm HS in general has similar performance to POTTS.

[7]In Section 6.2, we presented two strategies for task partitioning: one is based on [43] (detailed in Appendix) and another is a simple heuristic by performing the list scheduling algorithm based on the first non-critical sections. In all the experiments regarding partitioned scheduling, we observed that the latter (i.e., the simple heuristic) performed better. All the presented results for partitioned scheduling are therefore based on the simple heuristic.

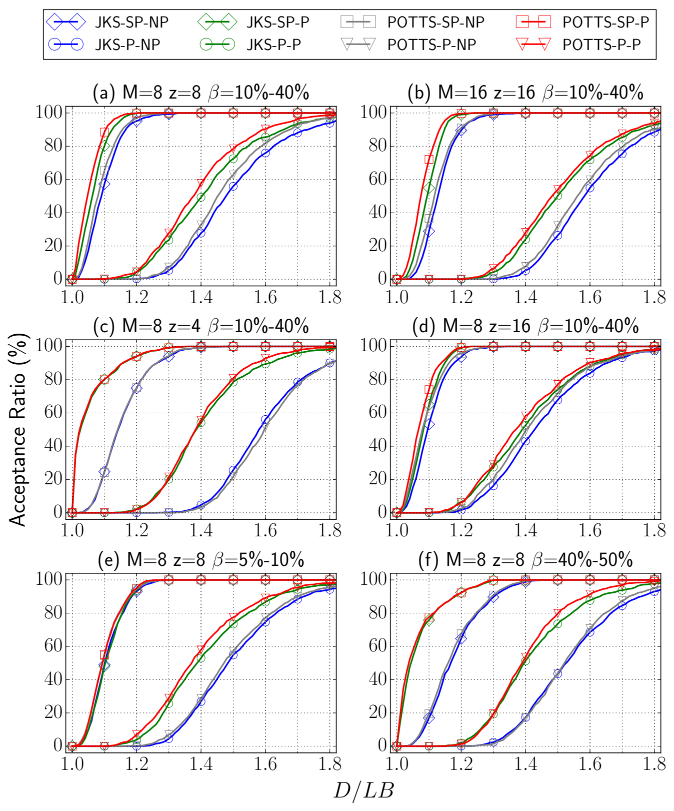

Fig. 3. Comparison of different approaches assuming different deadlines.

third part: preemptive or non-preemptive for the second non-critical sections.

We evaluated all 8 combinations under different settings as shown in Fig. 3. Due to space limitation, only a subset of the results is presented. In general, the semi-partitioned scheduling algorithms clearly outperform the partitioned strategies, independently from the algorithm used to construct the dependency graph. In addition, the preemptive scheduling policy with respect to the second computation segment is superior to the non-preemptive strategy and POTTS (usually) performs slightly better than JKS. We analyze the effect of the three parameters individually by changing:

1) $M = z \in \{8, 16\}$ (Fig. 3(a) and Fig. 3(b)): increasing z and M also slightly increases the difference between the semi-partitioned and the partitioned approaches.
2) **z for a fixed M**, i.e., $z \in \{4, 8, 16\}$ and $M = 8$ (Fig. 3(c), Fig. 3(a), and Fig. 3 (d)): when the number of resources is decreased compared to the number of processors, the performance gap between preemptive and non-preemptive scheduling increases.
3) **Workload of Shared Resources, i.e., $\beta \in \{[5\% - 10\%], [10\% - 40\%], [40\% - 50\%]\}$** (Fig. 3(e), Fig. 3 (a), and Fig. 3 (f)): if the workload of the critical sections is increased, the difference between preemptive and non-preemptive scheduling approaches is more significant.

We also compare our approach with the Resource Oriented Partitioned (ROP) scheduling with release enforcement by von der Brüggen et al. [45] which is designed to schedule periodic tasks with one critical section on a multiprocessor platform. The concept of the ROP is to have a resource centric view instead of a processor centric view. The algorithm 1) binds the critical sections of the same resource to the same processor, thus enabling well known uniprocessor protocols like PCP to handle the synchronization, and 2) schedules the non-critical sections on the remaining processors using a state-of-the-art scheduler for segmented self-suspension tasks, namely SEIFDA [46]. Among the methods in [45], we evaluated ROP-FP (under fixed-priority scheduling) and ROP-EDF (under dynamic priority scheduling, namely earliest deadline first). It has been shown in [45] that ROP-EDF dominates all existing methods. We performed another set of evaluations by adopting the aforementioned settings and testing the utilization level in a step of 5%, where the utilization of a task set $\mathbf{T}$ is $\sum_{\tau_i \in \mathbf{T}} \frac{C_{i,1}+C_{i,2}+A_{i,1}}{T_i}$. Fig. 4 presents the evaluation results. Due to space limitation, only a subset of the results is presented, but the others have very similar tendencies. For readability, we only select two combinations in our proposed approach that outperform the others. The results in Fig. 4 show that for frame-based tasks, our approach outperforms ROP significantly. We note that Fig. 4 is only for frame-based tasks, and the results for periodic task systems discussed in Section 8 are presented in Appendix.

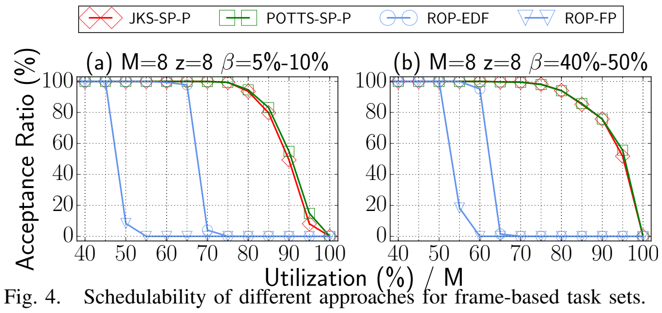

Fig. 4. Schedulability of different approaches for frame-based task sets.

10 Conclusion

This paper tries to answer a few fundamental questions when real-time tasks share resources in multiprocessor systems. Here is a short summary of our findings:

- The fundamental difficulty is mainly due to the sequencing of the mutual exclusive accesses to the share resources (binary semaphores). Adding more processors, removing periodicity and job recurrence, introducing task migration, or allowing preemption does not make the problem easier from the computational complexity perspective.
- The performance gap of partitioned and semi-partitioned scheduling in our study is mainly due to the capability to schedule the subjobs constrained by the dependency graph. Although partitioned scheduling may seem much worse than semi-partitioned scheduling in our evaluations, this is mainly due to the lack of understanding of the problem $P|prec, tied|C_{\max}$ in the literature. Further explorations are needed to understand these scheduling paradigms for a given dependency graph.
- The dependency graph approach is not work-conserving for the critical sections, since a critical section may be ready but not executed due to the artificially introduced precedence constraints. Existing multiprocessor synchronization protocols mainly assume work-conserving for granting the accesses of the critical sections via priority boosting. Our study reveals a potential to consider cautious and non-work-conserving synchronization protocols in the future.

Acknowledgement: This paper is supported by DFG, as part of the Collaborative Research Center SFB876, project A3 and B2 (http://sfb876.tu-dortmund.de/). The authors thank Zewei Chen and Maolin Yang for their tool SET-MRTS (Schedulability Experimental Tools for Multiprocessors Real Time Systems, https://github.com/RTLAB-UESTC/SET-MRTS-public) to evaluate the LP-GFP-FMLP, LP-PFP-DPCP, LP-PFP-MPCP, GS-MSRP, and LP-GFP-PIP in Fig. 5.

Appendix

Proof of Theorem 6. Due to the design of the task set, there are only N different dependency graphs, depending on the position of τ_1 in the execution order. Suppose that the critical section of task τ_1 is the j-th critical section in the dependency graph. It can be proved that the critical path of this dependency graph is $j\delta + Q + N\delta$. We *sketch* the proof:

- The non-critical section $C_{1,2}$ must be part of the critical path since $C_{1,2} = \frac{Q}{M} + N\delta$, which is greater than any $(N-1)A_{i,1} + C_{i,2}$ for any $i = 2, 3, \ldots, N-1$.
- The longest path that ends at the vertex representing $A_{1,1}$ has 1) one non-critical section, 2) $j-1$ critical sections from τ_i for $i = 2, 3, \ldots, N$, and 3) 1 critical section from task τ_1. Therefore, this length is $\delta + (j-1)\delta + Q - \frac{Q}{M} = j\delta + Q - \frac{Q}{M}$.
- Combining the two scenarios, we reach the conclusion.

Therefore, the dependency graph G^* that has the minimum critical path length is the one where τ_1's critical section is the first one among the N critical sections. The optimal schedule of the dependency graph G^* on M processors has the following properties:

- Task τ_1 finishes its critical section at time $\delta + Q - \frac{Q}{M}$.
- Before time $\delta + Q - \frac{Q}{M}$, none of the second non-critical sections is executed. Therefore, the makespan of any feasible schedule $S(G^*)$ of G^* on M processors is

$$\begin{aligned} L(S(G^*)) &\geq \delta + Q - \frac{Q}{M} + \sum_{i=1}^{N} \frac{C_{i,2}}{M} \\ &= \delta + Q - \frac{Q}{M} + \frac{(M^2 - M + 1)\frac{Q}{M} + N\delta}{M} \\ &= \left(1 + \frac{N}{M}\right)\delta + \left(2 - \frac{2}{M} + \frac{1}{M^2}\right)Q \end{aligned}$$

- Moreover, when the scheduling policy is either semi-partitioned or partitioned scheduling, by the pigeon hole principle, at least one processor must execute $\left\lceil \frac{N}{M} \right\rceil$ of the N second non-critical sections no earlier than $\delta + Q - \frac{Q}{M}$. Therefore, the makespan of a feasible semi-partitioned or partitioned schedule S_p of G^* on M processors is

$$\begin{aligned} L(S_p(G^*)) &\geq \delta + Q - \frac{Q}{M} + \left\lceil \frac{N}{M} \right\rceil \frac{Q}{M} \\ &= \delta + Q - \frac{Q}{M} + \left\lceil M - 1 + \frac{1}{M} \right\rceil \frac{Q}{M} \\ &= \delta + Q - \frac{Q}{M} + M\frac{Q}{M} \\ &= \delta + \left(2 - \frac{1}{M}\right)Q \end{aligned}$$

We can have another feasible partitioned schedule S^*:

- The first non-critical section τ_1 is executed on processor M, and the first non-critical sections of the other $N-1$ tasks are executed on the first $M-1$ processors based on list scheduling. All the first non-critical sections finish no later than $M\delta$. Each of the first $M-1$ processors executes *exactly* M tasks since there are $N-1 = M(M-1)$ tasks with identical properties on these $M-1$ processors.
- The critical sections of tasks $\tau_N, \tau_{N-1}, \ldots, \tau_1$ are executed sequentially by following the above reversed-index order on the same processor of the corresponding first non-critical sections, starting from time $M\delta$.
- At time $M\delta + N\delta$, all the second non-critical sections of $\tau_2, \ldots, \tau_N$ are eligible to be executed. We execute them in parallel on the first $M-1$ processors by respecting the partitioned scheduling strategy. That is, each of the first $M-1$ processors executes *exactly* M tasks with $C_{i,2} = Q/M$. The makespan of these $N-1$ tasks is $(N+M)\delta + \frac{(N-1)\frac{Q}{M}}{M-1} = (N+M)\delta + Q$.
- At time $M\delta + N\delta$, the critical section of τ_1 starts its execution on processor M. Furthermore, at time $(N+M)\delta + Q - \frac{Q}{M}$, the second non-critical section of τ_1 is executed on processor M and it is finished at time $(N+M)\delta + Q + N\delta = (2N+M)\delta + Q$.
- As a result, the makespan of the above partitioned schedule S^* is *exactly* $(2N+M)\delta + Q$.

Therefore, the approximation bound of the optimal task dependency graph approach is at least $\frac{L(S(G^*))}{L(S^*)}$ under any scheduling paradigm and is at least $\frac{L(S_p(G^*))}{L(S^*)}$ under partitioned or semi-partitioned scheduling paradigm. We reach the conclusion by taking $\delta \to 0$. □

Pseudo-code of the Partitioned Preemptive Scheduling in Section 6.2 For notational brevity, we define two vertices $v_{i,1}$ and $v_{i,3}$ to represent the first and second non-critical sections of task τ_i and $v_{i,2}$ to represent the critical section of task τ_i. Let $\mathbf{T}_m$ be the set of tasks in $\mathbf{T}$ assigned to processor m for $m = 1, 2, \ldots, M$. The pseudo-code is listed in Algorithm 2. It consists of three blocks: initialization from Line 1 to Line 4, scheduling of the first non-critical sections and the critical sections of the tasks according to $\bar{G}$ from Line 5 to Line 23, and scheduling of the second non-critical sections of the tasks from Line 24 to Line 28.

The first block is self-explained in Algorithm 2. We will focus on the second and third blocks of Algorithm 2. Our scheduling algorithm executes the first non-critical sections and the critical sections non-preemptively. Whenever a subjob finishes at time t, we examine the following scenarios on each processor m for $m = 1, 2, \ldots, M$:

- If there is a pending critical section on processor m that is eligible at time t according to the dependency graph G, this critical section is executed as soon as it is eligible and the processor idles (i.e., Lines 12-13).
- Else if there is a task in $\mathbf{T}_m$ in which its first non-critical section has not finished yet at time t, it is executed (Line 14-15)
- Otherwise, there is no eligible subjob to be executed at time t. If there is still an unassigned task, we select one and assign it to processor m by starting its first non-critical section at time t (Lines 16-19).

Algorithm 2 Tied List-Scheduling (Partitioned Preemptive)

```
Input: $G, \mathbf{T}, M$ with $|\mathbf{T}| > M$;
1: $current \leftarrow 0$;
2: assign one task $\tau_i$ in $\mathbf{T}$ to task set $\mathbf{T}_m$ to be executed on processor $m$;
3: $\mathbf{T} \leftarrow \mathbf{T} \setminus \cup_{m=1}^{M} \mathbf{T}_m$;
4: execute $v_{i,1}$ of the unique task $\tau_i$ in $\mathbf{T}_m$ on processor $m$ from time 0,
   i.e., $\rho(t, m) \leftarrow \tau_i$ for $t \in [0, C_{i,1})$, for each $m = 1, 2, \ldots, M$;
5: while $\exists \tau_i$ such that $v_{i,2}$ has not finished yet at time $current$ do
6:   let $t$ be the minimum time instant greater than $current$ such that the
     schedule finishes a subjob at time $t$;
7:   $current \leftarrow t$;
8:   for $m = 1, 2, \ldots, M$ do
9:     if processor $m$ is busy executing a subjob at time $t$ then
10:      continue;
11:    else if processor $m$ idles (or just finishes a subjob) at time $t$ then
12:      if $\exists \tau_i \in \mathbf{T}_m$, in which $v_{i,2}$ has not finished yet and $v_{i,2}$ is
         eligible according to $G$ at time $t$ then
13:        execute $\tau_i$'s critical section from time $t$ to $t + A_{i,1}$ non-
           preemptively on processor $m$, i.e., $\rho(\theta, m) \leftarrow \tau_i$ for $\theta \in$
           $[t, t + A_{i,1})$;
14:      else if $\exists \tau_i \in \mathbf{T}_m$, in which $v_{i,1}$ has not finished yet at $t$ then
15:        execute $v_{i,1}$ from time $t$ on $t + C_{i,1}$ processor $m$, i.e.,
           $\rho(\theta, m) \leftarrow \tau_i$ for $\theta \in [t, t + C_{i,1})$;
16:      else if $\mathbf{T}$ is not empty then
17:        select a task $\tau_i$ and remove $\tau_i$ from $\mathbf{T}$, i.e., $\mathbf{T} \leftarrow \mathbf{T} \setminus \{\tau_i\}$;
18:        assign task $\tau_i$ to processor $m$, i.e., $\mathbf{T}_m \leftarrow \mathbf{T}_m \cup \{\tau_i\}$;
19:        execute $v_{i,1}$ from time $t$ to $t + C_{i,1}$ on processor $m$, i.e.,
           $\rho(\theta, m) \leftarrow \tau_i$ for $\theta \in [t, t + C_{i,1})$;
20:      end if
21:    end if
22:  end for
23: end while
24: for $m = 1, 2, \ldots, M$ do
25:   for each task $\tau_i$ in $\mathbf{T}_m$ do
26:     schedule the second non-critical section $v_{i,3}$ of task $\tau_i$ as back-
        ground workload with the lowest priority preemptively as early as
        possible but no earlier than the finishing time of its critical section;
27:   end for
28: end for
```

In all the above steps, task τ_i can be arbitrarily selected if there are multiple tasks satisfying the specified conditions. We note that the schedule is in fact *offline*. Therefore, after we finish the schedule of the first non-critical sections and the critical sections, in the third block in Algorithm 2, we can pad the idle time of the schedule on a processor m with the second non-critical sections assigned on processor m, starting from time 0, as soon as they critical section of the task is finished.

Implementation in LITMUS$^{\text{RT}}$ To force the tasks to follow the pre-defined order to execute the critical sections, we added several elements into the rt_params structure which is used to define the property for each task, i.e., priority, period, execution time, etc. Two parameters are added: 1) rt_order to define the order of the task to execute the critical section, and 2) rt_total to define the number of the tasks that shared the same resource. To implement the binary semaphores under the dependency graph approach, we created two new structures, $pdga\_semaphore$ for the partitioned dependency graph approach (PDGA), and $sdga\_semaphore$ for the semi-partitioned dependency graph approach (SDGA). In these structures, one parameter is defined to control the order of the execution named $current\_serving\_ticket$. When a task requests the resource, it will compare its rt_order with the semaphore's $current\_serving\_ticket$, if they are equal, the task will be granted to access the resource and start its critical section; if not, the task will be added to the wait-queue, which is sorted by the tasks' parameter rt_order. Once a task has finished its critical section, it will increase the semaphore's current serving ticket by 1, and check the head of the wait-queue do the comparison again. Once the $current\_serving\_ticket$ reaches to the rt_total, which means one dependency graph has finished its execution of the critical sections, then the parameter $current\_serving\_ticket$ will be reset to 0 to start the next iteration. The only difference between PDGA and SDGA is that we added the migration function for SDGA to support the semi-partitioned algorithm.

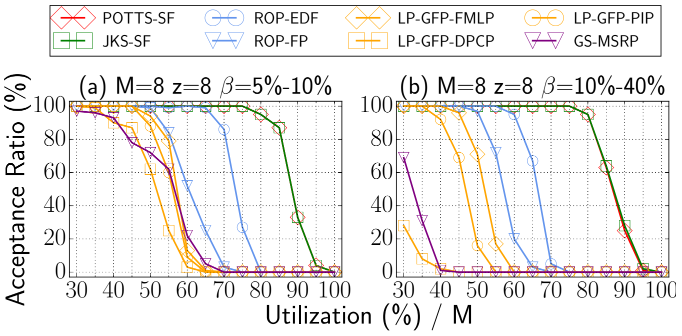

Fig. 5. Comparison of different approaches for periodic task sets.

Evaluations for Periodic Task Sets We also performed evaluations for periodic task systems, when a binary semaphore is only shared by the tasks with the same period described in Section 8. We used similar configurations as in Section 9.2 to generate the task sets. For the tasks that share the same semaphore, they have the same period in the range of $[1, 10]$. The following algorithms were evaluated:

- LP-GFP-FMLP [7]: a linear-programming-based (LP) analysis for global FP scheduling using the FMLP [7].
- LP-PFP-DPCP [9]: LP-based analysis for partitioned FP and DPCP [40]. Tasks are assigned using Worst-Fit-Decreasing (WFD) as proposed in [9].
- LP-PFP-MPCP [9]: LP-based analysis for partitioned FP using MPCP [39]. Tasks are partitioned according to WFD as proposed in [9].
- GS-MSRP [49]: the Greedy Slacker (GS) partitioning heuristic with the spin-based locking protocol MSRP [21] under Audsley's Optimal Priority Assignment [4].
- LP-GFP-PIP: LP-based global FP scheduling using the Priority Inheritance Protocol (PIP) [18].
- ROP-FP [45]: The ROP under fixed-priority scheduling and release enforcement.
- ROP-EDF [45]: The ROP under dynamic-priority scheduling and release enforcement.
- POTTS-SF: Our approach by applying algorithm Potts for generating G and semi-federated scheduling in [26].
- JKS-SF: Our approach by applying algorithm JKS for generating G and semi-federated scheduling in [26].

For one evaluation point, 100 synthetic task sets were generated and tested. Only a subset of the results is presented in Fig. 5, and LP-PFP-MPCP is not presented for better readability since it performs the worst for the evaluations in Fig. 5. The figure clearly shows that POTTS-SF and JKS-SF significantly outperform the other approaches.

References

[1] S. Afshar, M. Behnam, R. J. Bril, and T. Nolte. An optimal spin-lock priority assignment algorithm for real-time multi-core systems. In *RTCSA*, pages 1–11, 2017.

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

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

[4] 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.

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

[6] S. Baruah. The federated scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 15th International Conference on Embedded Software (EMSOFT)*, 2015.

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

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

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

[10] 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.

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

[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. 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.

[14] J. Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11(1):42 – 47, 1982.

[15] J.-J. Chen. Federated scheduling admits no constant speedup factors for constrained-deadline dag task systems. *Real-Time Systems*, 52(6):833–838, November 2016.

[16] 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.

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

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

[19] 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.

[20] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 90–99, 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] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[23] 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.

[24] 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.

[25] 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.

[26] X. Jiang, N. Guan, X. Long, and W. Yi. Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Proceedings of the 38nd IEEE Real-Time Systems Symposium, RTSS*, 2017.

[27] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of ACM*, 47(4):617–643, July 2000.

[28] H. Kise, T. Ibaraki, and H. Mine. Performance analysis of six approximation algorithms for the one-machine maximum lateness scheduling problem with ready times. *Journal of the Operations Research Society of Japan*, 22(3):205–224, 1979.

[29] 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.

[30] K. Lakshmanan, S. Kato, and R. R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 259–268, 2010.

[31] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[32] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. D. Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *26th Euromicro Conference on Real-Time Systems, ECRTS*, pages 85–96, 2014.

[33] G. McMahon and M. Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3):475–482, 1975.

[34] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 251–261, 2011.

[35] 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.

[36] E. Nowicki and S. Zdrzałka. A note on minimizing maximum lateness in a one-machine sequencing problem with release dates. *European Journal of Operational Research*, 23(2):266 – 267, 1986.

[37] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *ACM Symposium on Theory of Computing*, pages 140–149, 1997.

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

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

[40] 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.

[41] 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.

[42] J. Shi. DGA-LITMUS-RT. https://github.com/Strange369/Dependency-Graph-Approaches-for-LITMUS-RT, 2018.

[43] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi. Real-time scheduling and analysis of OpenMP task systems with tied tasks. In *IEEE Real-Time Systems Symposium, RTSS*, pages 92–103, 2017.

[44] N. Ueter, G. von der Brüggen, J.-J. Chen, J. Li, and K. Agrawal. Reservation-based federated scheduling for parallel real-time tasks. *CoRR*, abs/1712.05040, 2017.

[45] 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.

[46] 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.

[47] B. C. Ward and J. H. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Euromicro Conference on Real-Time Systems ECRTS*, pages 223–232, 2012.

[48] B. C. Ward and J. H. Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In *International Conference on Real-Time Networks and Systems, RTNS*, pages 67–76, 2013.

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

[50] 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.

[51] D. Zhu, R. G. Melhem, and B. R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium RTSS*, pages 84–94, 2001.