This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3043742, IEEE Transactions on Computers

# Scheduling of Real-Time Tasks with Multiple Critical Sections in Multiprocessor Systems

Jian-Jia Chen, Junjie Shi, Georg von der Brüggen, and Niklas Ueter
Department of Informatics, TU Dortmund University, Germany
{jian-jia.chen, junjie.shi, georg.von-der-brueggen, niklas.ueter}@tu-dortmund.de

**Abstract**—The performance of multiprocessor synchronization and locking protocols is a key factor to utilize the computation power of multiprocessor systems under real-time constraints. While multiple protocols have been developed in the past decades, their performance highly depends on the task partition and prioritization. The recently proposed Dependency Graph Approach showed its advantages and attracted a lot of interest. It is, however, restricted to task sets where each task has at most one critical section. In this paper, we remove this restriction and demonstrate how to utilize algorithms for the classical job shop scheduling problem to construct a dependency graph for tasks with multiple critical sections. To show the applicability, we discuss the implementation in LITMUS$^{RT}$ and report the overheads. Moreover, we provide extensive numerical evaluations under different configurations, which in many situations show significant improvement compared to the state-of-the-art.

**Index Terms**—Real-Time Systems, Multiprocessor Resource Synchronization, Job Shop, and Dependency Graph Approaches

## 1 Introduction

Under the von-Neumann programming model, shared resources that require mutual exclusive accesses, such as shared files, data structures, etc., have to be protected by applying synchronization (*binary semaphores*) or locking (*mutex locks*) mechanisms. A protected code segment that has to access a shared resource mutually exclusively is called a *critical section*. For uniprocessor real-time systems, the state-of-the-art are longstanding protocols that have been developed in the 90s, namely the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) by Sha et al. [34], as well as the Stack Resource Policy (SRP) by Baker [3]. Specifically, a variant of PCP has been implemented in Ada (called Ceiling locking) and in POSIX (called Priority Protect Protocol).

Due to the development of multiprocessor platforms, multiprocessor resource synchronization and locking protocols have been proposed and extensively studied, such as the Distributed PCP (DPCP) [33], the Multiprocessor PCP (MPCP) [32], the Multiprocessor SRP (MSRP) [16], the Flexible Multiprocessor Locking Protocol (FMLP) [4], the Multiprocessor PIP [13], the $O(m)$ Locking Protocol (OMLP) [7], the Multiprocessor Bandwidth Inheritance (M-BWI) [15], and the Multiprocessor resource sharing Protocol (MrsP) [8]. Since the performance of these protocols highly depends on task partitioning, several partitioning algorithms were developed in the literature, e.g., for MPCP by Lakshmanan et al. [26] and Nemati et al. [30], for MSRP by Wieder and Brandenburg [42], and for DPCP by Hsiu et al. [21], Huang et. al [22], and von der Brüggen et al. [40]. In addition to the theoretical soundness of these protocols, some of them have been implemented in the real-time operating systems LITMUS$^{RT}$ [5], [9] and RTEMS [1].

For several decades, the primary focus when considering multiprocessor synchronization and locking in real-time systems has been the design and analysis of resource sharing protocols, where the protocols decide the order in which the new incoming requests access the shared resources dynamically. Contrarily, the Dependency Graph Approaches (DGA), that was proposed by Chen et al. [11] in 2018, pre-computes the order in which tasks are allowed to access resources, and consists of two individual steps:

1) A dependency graph is constructed to determine the execution order of the critical sections guarded by *one binary semaphore or mutex lock*.
2) Multiprocessor scheduling algorithms are applied to schedule the tasks by respecting the constraints given by the constructed dependency graph(s).

Chen et al. [11] showed significant improvement against existing protocol-based approaches from the empirical as well as from the theoretical perspective, and demonstrated the practical applicability of the DGA by implementing it in LITMUS$^{RT}$ [5], [9]. However, the original dependency graph approaches presented in [11] has two strong limitations: 1) the construction in the first step allows only one critical section per task, and 2) the presented algorithms 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. The latter has been recently removed by Shi et al. [36], who applied the DGA after unrolling the jobs in the hyper-period. However, the former remains open and is a fundamental obstacle which limits the generality of the dependency graph approaches.

In the original DGA, the assumption that each task has only one non-nested critical section allows the algorithm to partition the tasks according to their shared resources in the first step. However, when a task accesses multiple shared resources, such a partitioning is no longer possible. Therefore, to enable the DGA for tasks with multiple critical sections, an exploration of effective construction mechanisms for a dependency graph that considers the interactions of the

[1] http://www.rtems.org/

shared resources is needed.

**Contribution:** In this paper, we focus on allowing multiple critical sections per task in the dependency graph approaches for both frame-based and periodic real-time task systems with synchronous releases. Our contributions are:

- Our key observation is the correlation between the dependency graph in DGA and the classical *job shop scheduling problem*. With respect to the computational complexity, we present a polynomial-time reduction from the classical *job shop scheduling problem*, which is $\mathcal{NP}$-hard in the strong sense [28]. Intractability results are established even for severely restricted instances of the studied multiprocessor synchronization problem, as detailed in Sec. 3.
- For frame-based task sets, we reduce the problem of constructing the dependency graph in the DGA to the classical *job shop scheduling problem* in Sec. 4, and establish approximation bounds for minimizing the makespan based on the approximation bounds of job-shop algorithms. Sec. 4.3 details how these results can be extended to periodic real-time task systems.
- We explain how we implemented the dependency graph approach with multiple critical sections in LITMUS$^{\text{RT}}$ and report the overheads in Sec. 5, showing that our new implemented approach is comparable to the existing methods with respect to the overheads.
- We provide extensive numerical evaluations in Sec. 6, which demonstrate the performance of the proposed approach under different system configurations. Compared to the state-of-the-art, our approach shows significant improvement for all the evaluated frame-based real-time task systems and for most of the evaluated periodic task systems.

# 2 System Model

## 2.1 Task Model

We consider a set $\mathbf{T}$ of $n$ recurrent tasks to be scheduled on $M$ identical (homogeneous) processors. All tasks can have multiple (non-nested) critical sections and may access several of the $Z$ shared resources. Each task $\tau_i$ is described by $\tau_i = ((\eta_i, C_i),\ T_i,\ D_i)$, where:

- $\eta_i$ is the number of computation segments in task $\tau_i$.
- $C_i$ is the total worst-case execution time (WCET) of the computation segments in task $\tau_i$.
- $T_i$ is the period of $\tau_i$.
- $D_i$ is the relative deadline of $\tau_i$.

We consider constrained deadlines, i.e., $\forall \tau_i \in \mathbf{T},\ D_i \leq T_i$. For the $j$-th segment of task $\tau_i$, denoted as $\theta_{i,j} = (C_{i,j}, \lambda_{i,j})$:

- $C_{i,j} \geq 0$ is the WCET of computation segment $\theta_{i,j}$ with $C_i = \sum_{j=1}^{\eta_i} C_{i,j}$.
- $\lambda_{i,j}$ indicates whether the corresponding segment is a non-critical section or a critical section. If $\theta_{i,j}$ is a critical section, $\lambda_{i,j}$ is 1; otherwise, $\lambda_{i,j}$ is 0.
- If $\theta_{i,j}$ is a non-critical section, then $\theta_{i,j-1}$ and $\theta_{i,j+1}$ must be critical sections (if they exist). That is, $\theta_{i,j}$ and $\theta_{i,j+1}$ cannot be both non-critical sections.
- If $\theta_{i,j}$ is a critical section, it starts from the lock of a mutex lock (or *wait* for a binary semaphore), denoted by $\sigma_{i,j}$, and ends at the unlock of the same mutex lock (or *signal* to the same binary semaphore).

Furthermore, we make following assumptions:

- Each task releases one job in the beginning of each period. Therefore, each computation segment within one task releases one instance accordingly, which is treated as a sub-job of the corresponding job.
- A job cannot be executed in parallel, i.e., the sub-jobs in a job must be sequentially executed.
- The execution of the critical sections guarded by a mutex lock (or one binary semaphore) must be sequentially executed. Hence, if two computation segments share the same lock, they must be executed one after another.
- There are in total $Z$ mutex locks (or binary semaphores).

We consider two kinds of task systems, namely:

- **Frame-based** task systems: all tasks release their jobs at the same time and have the same period and relative deadline, i.e., $\forall i, j,\ T_i = T_j \wedge D_i = D_j$. Hence, the analysis can be restricted to one job of each task.
- **Periodic** task systems (with synchronous release): all tasks release their first job at time $0$ and subsequent jobs are released periodically, but different tasks may have different periods and relative deadlines. The hyper-period of the task set $\mathbf{T}$ is defined as the least common multiple (LCM) of the periods of the tasks in $\mathbf{T}$.

## 2.2 Problem Definition and Approximation

In this subsection, we define the problem of scheduling frame-based real-time tasks with multiple critical sections in homogeneous multiprocessor systems.

We define a schedule from the sub-job's perspective. Suppose that $\mathbf{\Theta}$ is the set of the computation segments, i.e., $\mathbf{\Theta} = \{\theta_{i,j} \mid \tau_i \in \mathbf{T}, j = 1, 2, \ldots, \eta_i\}$. A schedule for $\mathbf{T}$ is a function $\rho : \mathbb{R} \times M \rightarrow \mathbf{\Theta} \cup \{\perp\}$, where $\rho(t, m) = \theta_{i,j}$ denotes that the sub-job $\theta_{i,j}$ is executed at time $t$ on processor $m$, and $\rho(t, m) = \perp$ denotes that processor $m$ is idle at time $t$. Since a job has to be sequentially executed, at any time point $t \geq 0$, only a sub-job of $\tau_i$ can be executed on one of the $M$ processors, i.e., if $\rho(t, m)$ is $\theta_{i,j}$, then $\rho(t, m') \neq \theta_{i,k}$ for any $k \leq \eta_i$ and $m' \neq m$. Moreover, since the sub-jobs of a job must be executed sequentially, $\theta_{i,k}$ cannot be executed before $\theta_{i,j}$ finishes for any $j < k \leq \eta_i$, i.e., if $\rho(t, m)$ is $\theta_{i,j}$ for some $t, m, i, j$, then $\rho(t', m) \neq \theta_{i,k}$ for any $t' \leq t$ and any $k > j$. The critical sections guarded by one mutex lock must be sequentially executed. That is, if $\lambda_{i,j}$ is 1, $\lambda_{k,\ell}$ is 1, and $\sigma_{i,j} = \sigma_{k,\ell}$ then a schedule must guarantee $\rho(t, m') \neq \theta_{k,\ell}$ for any $t \geq 0$ and $m \neq m'$ when $\rho(t, m)$ is $\theta_{i,j}$.

We only consider schedules that can finish the execution demand of the computation segments. Let $R$ be the finishing time of the schedule. In this case, $\sum_{m=1}^{M} \int_0^R [\rho(t, m) = \theta_{i,j}] dt$ must be equal to $C_{i,j}$, where $[P]$ is the Iverson bracket, i.e., $[P]$ is 1 when the condition $P$ holds, otherwise $[P]$ is 0. Note that the integration is used in this paper only as a symbolic notation to represent the summation over time. The earliest moment when all sub-jobs finish their computation segments in the schedule (under all the constraints defined above) is called the *makespan* of the schedule, commonly denoted as $C_{\max}$ in scheduling theory, i.e., $C_{\max}$ of schedule $\rho$ is:

$$\text{min. } R \quad \text{s. t. } \sum_{m=1}^{M} \int_0^R [\rho(t, m) = \theta_{i,j}] dt = C_{i,j}, \forall \theta_{i,j} \in \mathbf{\Theta}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3043742, IEEE Transactions on Computers

A schedule is *non-preemptive* from the sub-job's perspective if a sub-job cannot be preempted, i.e., there is only one interval with $\rho(t,m) = \theta_{i,j}$ on processor $m$ for any sub-job $\theta_{i,j}$in $\mathbf{\Theta}$. A schedule is *preemptive* from the sub-job's perspective if a sub-job can be preempted, i.e., more than one interval with $\rho(t,m) = \theta_{i,j}$ for any task $\theta_{i,j}$ in $\mathbf{\Theta}$ on processor $m$ is allowed. A critical section $\theta_{i,j}$ in a preemptive schedule can be preempted by non-critical sections or other critical sections that are unrelated to $\sigma_{i,j}$.

For a *partitioned* schedule, all sub-jobs of a job have to be executed on one processor, i.e., there is one processor $m$ with $\rho(t,m) = \theta_{i,j}$ for $t \geq 0$ and $j = 1, 2, \ldots, \eta_i$ for every task $\tau_i$ in $\mathbf{T}$. For a *global schedule*, a sub-job can be arbitrarily executed on any of the $M$ processors at any time point. That is, it is possible that $\rho(t,m) = \theta_{i,j}$ and $\rho(t',m') = \theta_{i,j}$ for $m \neq m'$ and $t \neq t'$. For a *semi-partitioned* schedule, a sub-job has to be executed only on one processor.

A partitioned or a semi-partitioned schedule can be preemptive or non-preemptive from the sub-job's perspective. A global schedule in the above definition is always a preemptive schedule from the sub-job's perspective.

The problem of multiprocessor synchronization with multiple critical sections per task can be transferred to the following two general problems:

**Definition 1.** ***Multiprocessor Multiple critical-Sections task Synchronization (*****MMSS*****) makespan problem:*** *Assume $M$ identical (homogeneous) processors and that $n$ tasks are arriving at time $0$. Each task $\tau_i$ is composed of $\eta_i$ computation segments, each of which is either a non-nested critical section or a non-critical section. The objective is to find a schedule that minimizes the makespan.*

A feasible schedule of the *MMSS* makespan problem is a schedule that satisfies all aforementioned non-overlapping constraints. An optimal solution of an input instance of the *MMSS* makespan problem is the makespan of a schedule that has the minimum makespan among the feasible schedules of the input instance. An algorithm $\mathcal{A}$ for the *MMSS* makespan problem has an *approximation ratio* $a \geq 1$, if given any task set $\mathbf{T}$ and $M$ processors, the resulting makespan is at most $a \cdot C^*_{\max}$, where $C^*_{\max}$ is the optimal makespan.

**Definition 2.** ***The*** **MMSS** ***schedulability problem:*** *Assume there are $M$ identical (homogeneous) processors and that $n$ tasks are arriving at time $0$. All tasks $\tau_i$ have the same deadline $D$. Each task is composed of $\eta_i$ computation segments, each of which is either a non-nested critical section or a non-critical section. The objective is to find a feasible schedule that meets the deadline $D$ on the given $M$ processors.*

A feasible schedule of the *MMSS* schedulability problem is a schedule that has a makespan no more than $D$ and satisfies all the non-overlapping constraints. The *MMSS* schedulability problem is a decision problem, in which for a given $D$ and a given algorithm either a feasible schedule is derived that meets the deadlines or no feasible schedule can be derived from the algorithm. For such a decision setting, the *speedup factor* [23], [31] can be used to examine the performance. *Provided that there exists one feasible schedule at the original speed*, the speedup factor $a \geq 1$ of a scheduling algorithm $\mathcal{A}$ for the *MMSS* schedulability problem is the factor $a \geq 1$ by which the overall speed of a system would need to be increased so that the algorithm $\mathcal{A}$ always derives a feasible schedule.

## 2.3 Notation from Scheduling Theory

In this subsection, for completeness, we summarize the classical flow shop and job shop scheduling problems in operations research (OR). In scheduling theory, a scheduling problem is described by a triplet $\alpha|\beta|\gamma$.

- $\alpha$ describes the machine (i.e., processing) environment.
- $\beta$ specifies the characteristics and constraints.
- $\gamma$ is the objective to be optimized.

The widely used machine environment in $\alpha$ are:

- 1: single machine (or uniprocessor).
- $P$: independent machines (or homogeneous multiprocessor systems).
- $F_M$: **flow shop.** The environment $F_M$ consists of $M$ machines and each job $i$ has a chain of $M$ sub-jobs, denoted as $O_{i,1}, O_{i,2}, \ldots, O_{i,M}$, where the $M$ operations are executed in the specified order and $O_{i,m}$ is executed on the $m$-th machine. A job has to finish its operation on the $m$-th machine before it can start any operation on the $(m+1)$-th machine, for any $m = 1, 2, \ldots, M-1$.
- $J_M$: **job shop**, i.e., a job $i$ has a chain of $\eta_i$ sub-jobs, denoted as $O_{i,1}, O_{i,2}, \ldots, O_{i,\eta_i}$, where the $\eta_i$ operations should be executed in the specified order and $O_{i,m}$ is executed on a specified machine. Note that a flow shop is a special case of a job shop environment.

In this paper, we are specifically interested in three constraints specified in $\beta$:

- $prmp$: preemptive scheduling. In classical scheduling theory, preemption in parallel machines implies the possibility of job migration from one machine to another machine.
- $r_j$: with specified arrival time of the job (and deadline).
- $l_{i,j}$: preparation time between dependent job pair, i.e., job $i$ and job $j$.
- $prec$: the jobs have precedence constraints.

Note that the scheduler is implicitly assumed to be non-preemptive if $prmp$ is not specified. Furthermore, the job set is assumed to arrive at time $0$ if $r_j$ is not specified.

In addition, we are specifically interested in two objectives specified in $\gamma$:

- $C_{\max}$: to minimize the makespan, as defined in Sec. 2.2.
- $L_{\max}$: to minimize the maximum lateness over all jobs, in which the lateness of a job is defined as its finishing time minus its absolute deadline.

## 2.4 Critical Sections Access Patterns

Two types of access patterns of the critical sections are considered, which we name according to the applicable algorithms for convenience:

- **Flow-Shop Compatible Access Patterns**: A task set has a pattern where *flow-shop* approaches can be applied, if all tasks access each resource (in a non-nested manner) at most once and a total order $\prec$ in which tasks access the resources can be constructed over all tasks in the set. Hence, a flow-shop pattern means that $\sigma_{i,j'} \prec \sigma_{i,j}$ when $j' < j$ and $\theta_{i,j'}$ and $\theta_{i,j}$ are both critical sections. In such a case, we can assume that the mutex locks are indexed according to the specified total order set.

Although the order must be always respected, a task does not need to access all the mutex locks. That is, the access pattern of the mutex locks of a task is a subset of the specified total order set.

- **Job-Shop Compatible Access Patterns** allow tasks to accesses shared resources multiple times and without any restriction on the order, in which resources are accessed.

*Flow-shop compatible access patterns* are a very restrictive special case of the much more general *job-shop compatible access patterns*. We implicitly assume job-shop compatible access patterns if not specified differently, but examine flow-shop compatible access patterns when showing certain complexity results.

## 3 Computational Complexity Analysis

In this section, we provide a short overview of results regarding job shop and flow shop problems in the literature at first. Afterwards, we explain the connection of the *MMSS* schedulability problem to the job and flow shop problem by showing different reductions that can be later applied for demonstrating different scenarios with respect to their computational complexity.

### 3.1 Literature Review of Shop Scheduling

Since the late 1950s, many computational complexity results, approximation algorithms, heuristic algorithms, and tools for job and flow shop scheduling problems have been established. Intractability results have been well-established even for severely restricted instances of job shop or flow shop problems. The reader is referred to the surveys by Lawler et al. [27] and Chen et al. [10] for details.

Specifically, the following restricted scenarios are $\mathcal{NP}$-complete in the strong sense:

- $J_2||C_{\max}$, see [28].
- $J_3|p_{i,j} = 1|C_{\max}$, i.e., unit execution time, see [28].
- $J_3|n = 3|C_{\max}$, i.e., 3 jobs with multiple operations on 3 shops, see [39].
- $F_3||C_{\max}$, i.e., three-stage flow shop [17].
- $F_2|r_j|C_{\max}$, i.e., two-stage flow shop with arrival times, as shown in [28].
- $F_2|p_{i,j} = 1, t_j|C_{\max}$, i.e., two-stage flow shop with unit processing time and transportation time between the finishing time of the first and the starting time of the second stage [44].

The best polynomial-time approximation algorithm for the general $J_M||C_{\max}$ problem was provided by Shmoys et al. [38], showing an approximation ratio of $O\left(\frac{\log^2(M\mu)}{\log\log(M\mu)}\right)$, where $M$ is the number of shops and $\mu$ is the maximum number of operations per job. The approximation ratio of this algorithm was later improved by Goldberg et al. [18], showing a ratio of $O\left(\frac{\log^2(M\mu)}{(\log\log(M\mu))^2}\right)$.

Whether there exists a polynomial-time algorithm with a constant approximation ratio for the general $F_M||C_{\max}$ or $J_M||C_{\max}$ problem remained open until 2011, when Mastrolilli and Svensson [29] showed that $F_M||C_{\max}$ (and hence $J_M||C_{\max}$) does not admit any polynomial-time approximation algorithm with a constant approximation ratio. Moreover, they also showed that the lower bound on the approximation ratio is very close to the existing upper bound provided by Goldberg et al. [18].

| | |
|---|---|
| Chen et al. in [11] | $M \geq n+1$, any scheduling paradigm |
| Theorem 1 | $M \geq Z$, semi-partitioned scheduling paradigm |
| Theorem 2 and [28] | $M \geq n > Z = 2$, partitioned scheduling paradigm |
| Theorem 2 and [28] | $M \geq n > Z = 3$, unit execution time,<br>partitioned scheduling paradigm |
| Theorem 2 and [39] | $M \geq n = 3, Z = 3$, partitioned scheduling paradigm<br>(with multiple visits to a mutex lock per job) |
| Theorem 3 and [17] | $M \geq n > Z = 3$, partitioned scheduling paradigm<br>(with flow-shop compatible access patterns) |
| Theorem 4 and [28] | $M = Z = 2$, semi-partitioned scheduling paradigm |
| Theorem 5 and [28] | $Z = M = 3$, unit execution time,<br>semi-partitioned scheduling paradigm |
| Theorem 6 and [39] | $n = Z = M = 3$,<br>(semi-)partitioned scheduling paradigm |
| Theorem 7 and [17] | $Z = M = 3$, semi-partitioned scheduling paradigm<br>(with flow shop access patterns) |
| Theorem 8 and [44] | $Z = 1, \eta_i \geq 3, M \geq n$, unit execution time,<br>any scheduling paradigm |

TABLE 1
The complexity results that are known and discussed in this work.

In Sec. 3.3, we demonstrate that the *MMSS* schedulability problem is already $\mathcal{NP}$-complete in the strong sense for very restrictive scenarios, even when $M$ and $Z$ are both extremely small. In Sec. 3.4, we further reduce from the master-slave problem [44] to show that the *MMSS* schedulability problem is $\mathcal{NP}$-complete in the strong sense even when there are two critical sections that access the unique shared resource with unit execution time per task.

### 3.2 Reductions from the Job/Flow Shop Problem

Chen et al. [11] showed that a special case of the *MMSS* makespan problem is $\mathcal{NP}$-hard in the strong sense when a task has only one critical section and $M$ is sufficiently large. The *MMSS* schedulability problem is the decision version of the *MMSS* makespan problem. We therefore focus on the hardness of the decision version in Definition 2. Here, we provide reductions from the job/flow shop scheduling problems to different restricted scenarios of the *MMSS* schedulability problem. Such reductions are used in Sec. 3.3 for demonstrating the $\mathcal{NP}$-completeness for different scenarios. The complexity results are shown in Table 1.

We start from the more general scenario under the semi-partitioned scheduling paradigm.

**Theorem 1.** *Under the semi-partitioned scheduling paradigm, there is a polynomial-time reduction from an input instance of the decision version of the job shop scheduling problem $J_Z||C_{\max}$ with $Z$ shops to an input instance of the* MMSS *schedulability problem that has $Z$ mutex locks on $M$ processors with $M \geq Z$.*

*Proof.* The proof is based on a polynomial-time reduction from an instance of the job shop scheduling problem $J_Z||C_{\max}$ to the *MMSS* schedulability problem. We present a polynomial-time reduction from the job shop scheduling problem $J_Z||C_{\max}$ to the *MMSS* schedulability problem.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3043742, IEEE Transactions on Computers

Suppose a given input instance with $n$ jobs of the job shop scheduling problem $J_Z||C_{\max}$.

- We have $Z$ shops with non-preemptive execution.
- A job $i$ is defined by a chain of $\eta_i$ sub-jobs, denoted as $O_{i,1}, O_{i,2}, \ldots, O_{i,\eta_i}$. The processing time of $O_{i,j}$ is $C_{i,j}$.
- These $\eta_i$ operations should be executed in the specified order and $O_{i,m}$ is executed on one of the given $Z$ shops, i.e., on shop $s(O_{i,m})$, where $s(O_{i,m}) \in \{1, 2, \ldots, Z\}$.

The decision version of the job shop scheduling problem is to decide whether there is a non-preemptive schedule whose makespan is no more than a given $D$. The polynomial-time reduction to the *MMSS* schedulability problem is as follows:

- There are $M \geq Z$ processors.
- There are $Z$ mutex locks, indexed as $1, 2, \ldots, Z$.
- For a job $i$ of the input instance of the job shop scheduling problem, we create a task $\tau_i$, which is composed of $\eta_i$ computation segments. The execution time of $\theta_{i,j}$ is the same as the processing time of the operation $O_{i,j}$. The mutex lock $\sigma_{i,j}$ used by $\theta_{i,j}$ is $s(O_{i,m})$.
- The deadline of the tasks is $D$ and the period is $T = D$.

We denote the above input instance of the job shop scheduling problem as $I$ (the *MMSS* schedulability problem as $I'$, respectively). We show that there exists a feasible schedule $\rho$ for $I$ (in the job shop scheduling problem) if and only if there exists a feasible schedule $\rho'$ for $I'$ (in the *MMSS* schedulability problem).[2]

**Only-if part**: Suppose $\rho$ is a feasible schedule for $I$, i.e.,

$$\left(\sum_{m=1}^{Z} \int_0^D [\rho(t,m) = O_{i,j}] dt\right) = C_{i,j}, \forall O_{i,j} \qquad (1)$$

and $\rho(t,m) \neq O_{i,j}$ for any $t$ and $m$ if $s(O_{i,j}) \neq m$. Since the execution on shops ins non-preemptive, if two operations $O_{i,j}$ and $O_{k,\ell}$ are supposed to be executed on a shop $z$, they are executed sequentially in $\rho$. As a result, without any conflict, for $0 \leq t \leq D$, we can set

$$\rho'(t,m) = \begin{cases} \perp & \text{if } \rho(t,m) = \perp \\ \theta_{i,j} & \text{if } \rho(t,m) = O_{i,j} \end{cases} \qquad (2)$$

In the schedule $\rho'$, critical sections guarded by the mutex lock $z$ are executed sequentially on the $z$-th processor. Therefore,

$$\left(\sum_{m=1}^{Z} \int_0^D [\rho'(t,m) = \theta_{i,j}] dt\right) = C_{i,j}, \forall \theta_{i,j} \in \mathbf{\Theta} \qquad (3)$$

and all the constraints for a feasible schedule for $I'$ are met. Such a schedule is a semi-partitioned and non-preemptive schedule (from the sub-job's perspective), which is also a global preemptive schedule (from the job's perspective).

**If part**: Suppose that $\rho'$ is a feasible schedule for $I'$, i.e.,

$$\sum_{m=1}^{M} \int_0^D [\rho'(t,m) = \theta_{i,j}] dt = C_{i,j}, \forall \theta_{i,j} \in \mathbf{\Theta} \qquad (4)$$

and the schedule $\rho'$ executes any two critical sections $\theta_{i,j}$ and $\theta_{k,\ell}$ with $\sigma_{i,j} = \sigma_{k,\ell} = z$ sequentially. Therefore, for a mutex lock $z \in \{1, 2, \ldots, Z\}$, the critical sections guarded by $z$ must be sequentially executed. As a result, without any conflict, for $0 \leq t \leq D$, we can set

$$\rho(t,z) = \begin{cases} O_{i,j} & \text{if } \exists m \text{ with } \rho'(t,m) = \theta_{i,j} \text{ and } \sigma_{i,j} = z \\ \perp & \text{otherwise} \end{cases} \qquad (5)$$

However, since we do not put any constraint on the feasible schedule $\rho'$, it is possible that the execution of $O_{i,j}$ on shop $z$ is not continuous. Suppose that $a_{i,j}$ ($f_{i,j}$, respectively) is the first (last, respectively) time instant when $O_{i,j}$ is executed on shop $z$ in $\rho$. Since the schedule $\rho'$ executes any two critical sections $\theta_{i,j}$ and $\theta_{k,\ell}$ sequentially when $\sigma_{i,j} = \sigma_{k,\ell} = z$, we know that for any $t$ between $a_{i,j}$ and $f_{i,j}$ either $\rho(t,z) = O_{i,j}$ or $\rho(t,z) = \perp$. Therefore, we can simply set $\rho(t,z)$ to $O_{i,j}$ for any $t$ in the time interval $[a_{i,j}, a_{i,j} + C_{i,j})$ and set $\rho(t,z)$ to $\perp$ for any $t$ in $[a_{i,j} + C_{i,j}, f_{i,j})$. The resulting schedule $\rho$ executes all the operations non-preemptively on the corresponding shops. Therefore, all the scheduling constraints of the job shop scheduling problem are met and

$$\left(\sum_{m=1}^{Z} \int_0^D [\rho(t,m) = O_{i,j}] dt\right) = C_{i,j}, \forall O_{i,j} \qquad (6)$$

We note that there is no specific constraint of scheduling imposed by the schedule $\rho'$. □

The proof of Theorem 1 is not valid for the more restrictive partitioned scheduling paradigm, i.e., all the computation segments of a task must be executed on the same processor, since the constructed schedule $\rho'$ in the proof of the only-if part is not a partitioned schedule. Interestingly, if we use an abundant number of processors, i.e., $M \geq n$, then the reduction in Theorem 1 holds for the partitioned scheduling paradigm as well.

**Theorem 2.** *Under the partitioned scheduling paradigm, there is a polynomial-time reduction which reduces from an input instance of the decision version of the job shop scheduling problem $J_Z||C_{\max}$ with $Z$ shops to an input instance of the* MMSS *schedulability problem that has $n$ tasks and $Z$ mutex locks on $M$ processors with $M \geq n \geq Z$.*

*Proof.* The proof is identical to the proof of Theorem 1 by ensuring that $\rho'$ constructed in the only-if part in the proof of Theorem 1 can be converted to a partitioned schedule. Instead of applying Eq. (2), since $M \geq n$, without any conflict, for $0 \leq t \leq D$ and $i = 1, 2, \ldots, n$, we can set

$$\rho'(t,i) = \begin{cases} \perp & \text{if } \nexists m \text{ with } \rho(t,m) = O_{i,j} \\ \theta_{i,j} & \text{if } \exists m \text{ with } \rho(t,m) = O_{i,j} \end{cases} \qquad (7)$$

Since all computation segments of $\tau_i$ are executed on processor $i$, the schedule $\rho'$ is a partitioned schedule. All the remaining analysis follows the proof of Theorem 1. □

**Theorem 3.** *There is a polynomial-time reduction which reduces from an input instance of the decision version of the flow shop scheduling problem $F_Z||C_{\max}$ with $Z$ flow shops to an input instance of the* MMSS *schedulability problem that has $Z$ mutex locks with a flow-shop compatible access pattern. The conditions in Theorems 1 and 2 for different scheduling paradigms with respect to constraint of $M$ remain the same.*

[2] Although we do not formally define the schedule function of the job shop scheduling problem, we believe that the context is clear enough by replacing the use of the computation segments with the operations.

*Proof.* The proof is identical to the proofs of Theorems 1 and 2. The additional condition is to access to the $Z$ mutex locks by following the index, starting from 1. □

The above theorems show that the computational complexity of the *MMSS* schedulability problem is almost independent from the number of processors (i.e., adding processors may not be helpful) and the underlying scheduling paradigm. The fundamental problem is the sequencing of the critical sections.

### 3.3 Computational Complexity for Small $M$

We can now reach the computational complexity of the *MMSS* schedulability problem when $Z \geq 2$ for small $M$. For completeness, we state the following lemma.

**Lemma 1.** *The* MMSS *schedulability problem is in $\mathcal{NP}$.*

*Proof.* Since the feasibility of a given schedule for the *MMSS* schedulability problem can be verified in polynomial-time, it is in $\mathcal{NP}$. □

The following four theorems are based on the reductions in Theorem 1 and Theorem 3. In general, even very special cases are $\mathcal{NP}$-complete in the strong sense.

**Theorem 4.** *Under the semi-partitioned scheduling paradigm, the* MMSS *schedulability problem is $\mathcal{NP}$-complete in the strong sense when $Z = M = 2$.*

*Proof.* The job shop scheduling problem $J_2||C_{\max}$ with 2 shops is $\mathcal{NP}$-complete in the strong sense [28]. Together with Theorem 1, we conclude the theorem. □

The *MMSS* schedulability problem is also difficult when all computation segments have the same execution time.

**Theorem 5.** *Under the semi-partitioned scheduling paradigm, the* MMSS *schedulability problem is $\mathcal{NP}$-complete in the strong sense when $Z = M = 3$ and $C_{i,j} = 1$ for any computation segment $\theta_{i,j}$.*

*Proof.* The job shop scheduling problem $J_3|p_{i,j} = 1|C_{\max}$ with unit execution time on 3 shops is $\mathcal{NP}$-complete in the strong sense [28]. Together with Theorem 1, we conclude the theorem. □

The following theorem shows that the *MMSS* schedulability problem is also difficult when there are just three tasks, three mutex locks, and three processors.

**Theorem 6.** *The* MMSS *schedulability problem is $\mathcal{NP}$-complete in the strong sense when $n = Z = M = 3$.*

*Proof.* The job shop scheduling problem $J_3|n = 3|C_{\max}$ with 3 jobs (with multiple operations) on 3 shops is $\mathcal{NP}$-complete in the strong sense [39]. Together with Theorem 1, we conclude the theorem for semi-partitioned scheduling paradigm.

For the partitioned scheduling paradigm, since there are exactly 3 tasks, 3 processors, and 3 mutex locks, the computational complexity remains the same, as a semi-partitioned schedule can be mapped to a partitioned schedule. □

**Theorem 7.** *Under the semi-partitioned scheduling paradigm, the* MMSS *schedulability problem* for flow-shop compatible access patterns *is $\mathcal{NP}$-complete in the strong sense when $Z = M = 3$.*

*Proof.* The flow shop scheduling problem $F_3||C_{\max}$ with 3 shops is $\mathcal{NP}$-complete in the strong sense [17]. Together with Theorem 3, we conclude the theorem. □

### 3.4 Computational Complexity When $M \geq n$

Chen et al. [11] showed that a special case of the *MMSS* makespan problem is $\mathcal{NP}$-hard in the strong sense when a task has only one critical section and $M$ is sufficiently large. The following theorem shows that the *MMSS* schedulability problem is $\mathcal{NP}$-complete when there are only two critical sections per task and the critical sections are with unit execution time.

**Theorem 8.** *The* MMSS *schedulability problem is $\mathcal{NP}$-complete in the strong sense when $Z = 1$, $\eta_i \geq 3$ for every $\tau_i \in \boldsymbol{T}$, $C_{i,j} = 1$ for every computation segment $\theta_{i,j}$ with $\lambda_{i,j} = 1$, and $M \geq n$.*

*Proof.* The problem is in $\mathcal{NP}$, since the feasibility of a given schedule can be verified in polynomial-time. Similar to the proof of Theorem 1, we show a polynomial-time reduction from the master-slave scheduling problem with unit execution time on the master [44]. Assume a given input instance with $n$ jobs of the master-slave scheduling problem:

- We assume a sufficient number of slaves, but only one master that can be modeled as a uniprocessor.
- A job $i$ has a chain of three sub-jobs, in which the first and third sub-jobs have to be executed on the master and the second sub-job has to be executed on a slave.
- The processing time of the first and third sub-jobs of a job $i$ is 1. The processing time of the second sub-job of a job $i$ is $O_i > 0$.

The decision version of the master-slave scheduling problem is to decide whether there is a schedule whose makespan is no more than a given target $D$, which is $\mathcal{NP}$-complete in the strong sense [44]. The master-slave scheduling problem is equivalent to the uniprocessor self-suspension problem with two computation segments and one suspension interval.

The polynomial-time reduction to the *MMSS* schedulability problem is as follows:

- There are $M \geq n$ processors.
- There is one mutex lock.
- For a job $i$ of the input instance of the master-slave scheduling problem, we create a task $\tau_i$, which is composed of three computation segments. The execution time $C_{i,1} = C_{i,3}$ and $C_{i,2} = O_i$. Computation segments $\theta_{i,1}$ and $\theta_{i,3}$ are critical sections guarded by the only mutex lock. Computation segment $\theta_{i,2}$ is a non-critical section.
- The deadline of the tasks is $D$ and the period is $T = D$.

It is not difficult to prove that a feasible schedule $\rho$ for the original input of the master-slave scheduling problem exists if and only if there exists a feasible schedule $\rho'$ for the reduced input of the *MMSS* schedulability problem. Details are omitted due to space limitation. □

# 4 The DGA Based on Job/Flow Shop

In this section, we detail the DGA for tasks with multiple critical sections, based on job shop scheduling to construct a dependency graph.

- In the first step, we construct a directed *acyclic* graph $G = (V, E)$. For each sub-job $\theta_{i,j}$ of task $\tau_i$ in $\mathbf{T}$, we create a vertex in $V$. The sub-job $\theta_{i,j}$ is a predecessor of $\theta_{i,j+1}$ for $j = 1, 2, \ldots, \eta_i - 1$. Suppose that $\mathbf{\Theta}^z$ is the set of the computation segments that are critical sections guarded by mutex lock $z$, i.e., $\mathbf{\Theta}^z \leftarrow \{\theta_{i,j} \mid \lambda_{i,j} = 1 \text{ and } \sigma_{i,j} = z\}$. For each $z = 1, 2, \ldots, Z$, the subgraph of the computation segments in $\mathbf{\Theta}^z$ is a directed chain, which represents the total execution order of these computation segments.
- In the second step, we construct a schedule of $G$ on $M$ processors either globally or partitioned, either preemptive or non-preemptive.

For a directed acyclic graph $G$, a **critical path** of $G$ is a longest path of $G$, and its length is denoted by $len(G)$. We now explain how to reduce from an input instance $I^{MS}$ of the *MMSS* makespan problem to an input instance $I^{JS}$ of the job shop scheduling problem $J_{Z+n}||C_{\max}$.

- We create $Z + n$ shops:
  - Shop $z \in \{1, 2, \ldots, Z\}$ is exclusively used to execute critical sections guarded by mutex lock $z$. That is, only critical sections $\theta_{i,j}$ with $\lambda_{i,j} = 1$ and $\sigma_{i,j} = z$ (i.e., $\theta_{i,j} \in \mathbf{\Theta}^z$) can be executed on shop $z$.
  - Shop $Z + i$ is exclusively used to execute non-critical sections of task $\tau_i$. That is, only non-critical sections $\theta_{i,j}$ with $\lambda_{i,j} = 0$ can be executed on shop $Z + i$.
- The operation of each computation segment $\theta_{i,j}$ is transformed to the corresponding shop, and the processing time is the same as the segment's execution time, i.e., $C_{i,j}$.

Suppose that $\rho^{JS}$ is a feasible job shop schedule for $I^{JS}$. Since $\rho^{JS}$ is non-preemptive, the operations on a shop are executed sequentially in $\rho^{JS}$. The construction of the dependency graph $G$ sets the precedence constraints of $\mathbf{\Theta}^z$ by following the total order of the execution of the operations on shop $z$, i.e., the shop dedicated for $\mathbf{\Theta}^z$ in $\rho^{JS}$.

Once the dependency graph $G$ is constructed, a schedule $\rho^{MS}$ of the original input instance $I^{MS}$ can be generated by applying any scheduling algorithms to schedule $G$, as already detailed in [11], [36]. Specifically, for semi-partitioned scheduling, the LIST-EDF in [36] based on classical list scheduling by Graham [19] can be applied, i.e., whenever a processor idles and at least one sub-job is eligible, the sub-job with the earliest deadline starts its execution on the processor. Additionally, its partitioned extension in [37] (P-EDF) can be applied to generate the partitioned schedule.

We assume each computation segment/sub-task executes exactly its WCET for all the releases, i.e., early completion is forbidden, thus the schedule generated for one hyper-period is static and repeated periodically. Accordingly, an exact schedulability test is performed by simply evaluating the LIST-EDF or P-EDF schedule over one hyper-period to check whether there is any deadline miss. Since the schedule is static and repeated periodically, there is no dynamics that can lead to the multiprocessor anomalies pointed out by Graham [19]. To demonstrate the work flow of our approach, we provide an illustrative example in the supplemental material.

## 4.1 Properties of Our Approach

We now prove the equivalence of a schedule of $I^{JS}$ and a directed acyclic graph $G$ for $I^{MS}$.

**Lemma 2.** *Suppose that there is a directed acyclic graph $G$ for $I^{MS}$ whose critical path length is $len(G)$. There is a job shop schedule for $I^{JS}$ whose makespan is $len(G)$.*

*Proof.* This lemma is proved by constructing a job shop schedule $\rho^{JS}$ for $I^{JS}$, in which the makespan of $\rho^{JS}$ is $len(G)$. Suppose that the longest path ended at a vertex $\theta_{i,j}$ in $V$ in the directed acyclic graph $G$ is $L_{i,j}$. There are two cases to schedule $\theta_{i,j}$ in $\rho^{JS}$:

- If $\theta_{i,j}$ is a non-critical section, the schedule $\rho^{JS}$ schedules the operation on shop $i + Z$ from time $L_{i,j} - C_{i,j}$ to $L_{i,j}$.
- If $\theta_{i,j}$ is a critical section guarded by mutex lock $z$, the schedule $\rho^{JS}$ schedules the operation on shop $z$ from time $L_{i,j} - C_{i,j}$ to $L_{i,j}$.

The above schedule has a makespan of $len(G)$ by construction. The only thing that has to be proved is that the schedule is a feasible job shop schedule for $I^{JS}$.

Suppose for contradiction that the schedule $\rho^{JS}$ is not a feasible job shop schedule for $I^{JS}$. This is only possible if the schedule $\rho^{JS}$ has a conflicting decision to schedule two operations at the same time $t$ on a shop $z$. There are two cases:

1) $z$ is an exclusively reserved shop for the non-critical sections of a task. This contradicts to the definition of $G$ since the non-critical sections of task $\tau_i$ form a total order in graph $G$.
2) $z$ is a shop for the critical sections guarded by the mutex lock $z$. This contradicts to the definition of $G$ since the critical sections in $\mathbf{\Theta}^z$ form a total order in graph $G$.

In both cases, we reach the contradiction. Therefore, $I^{JS}$ is a feasible job shop schedule with a makespan of $len(G)$. □

**Lemma 3.** *Suppose that there is a job shop schedule for $I^{JS}$ whose makespan is $\Delta$. Then, there is a directed acyclic graph $G$ for $I^{MS}$ whose critical path length is at most $\Delta$.*

*Proof.* This lemma is proved by constructing a graph $G$ for $I$, in which the critical path length of $G$ is at most $\Delta$. By the definition of $G$, the sub-job $\theta_{i,j}$ is a predecessor of $\theta_{i,j+1}$ for $j = 1, 2, \ldots, \eta_i - 1$ for every task $\tau_i$. For the sub-jobs in $\mathbf{\Theta}^z$, we define their total order and form a chain in $G$ by following the execution order on shop $z$ in the given schedule $\rho^{JS}$ for $I^{JS}$. Such a graph $G$ must be acyclic; otherwise, the schedule $\rho^{JS}$ is not a valid job shop schedule for $I^{JS}$.

We now prove that the critical path length $len(G)$ of $G$ is no more than $\Delta$. Suppose for contradiction that $len(G) > \Delta$. This critical path of $G$ defines a total order of the execution of the computation segments in the critical path, which follows *exactly* the total order of the operations of a job and a shop in $\rho^{JS}$. Therefore, this contradicts to the fact that the makespan of schedule $\rho^{JS}$ for $I^{JS}$ is $\Delta$. □

Based on Lemmas 2 and 3, we get the following theorem:

**Theorem 9.** *An $a$-approximation algorithm for the job shop scheduling problem $J_{Z+n}||C_{\max}$ can be used to construct a dependency graph $G$ with $len(G) \leq a \times len(G^*)$, where $G^*$ is a dependency graph that has the shortest critical path length for the input instance $I^{MS}$ of the* MMSS *makespan problem.*

*Proof.* Suppose that $\Delta^*$ is the optimal makespan for $I^{JS}$. By Lemma 2, we know that $\Delta^* \leq len(G^*)$. By Lemma 3, we know that $\Delta^* \geq len(G^*)$. Therefore, $\Delta^* = len(G^*)$. Suppose that the algorithm derives a solution for $I^{JS}$ with a makespan $\Delta$. By the $a$-approximation for $I^{JS}$ and Lemma 3, we know $\Delta \leq a \times \Delta^*$. Therefore, by Lemma 3 and above discussions, $len(G) \leq \Delta \leq a\Delta^* = a \times len(G^*)$. □

**Lemma 4.** *Let $G^*$ be defined as in Theorem 9. The optimal makespan for the input instance $I^{MS}$ of the* MMSS *makespan problem is at least*

$$\max\left\{\sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M}, len(G^*)\right\} \tag{8}$$

*Proof.* The lower bound $\sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M}$ is due to the pigeon hole principle. The lower bound $len(G^*)$ is due to the definition with an infinite number of processors. □

**Theorem 10.** *Applying list scheduling for the dependency graph $G$ with $len(G) \leq a \times len(G^*)$ results in a schedule with an approximation ratio of $a+1$ for the* MMSS *makespan problem under semi-partitioned scheduling, where $G^*$ is defined in Theorem 9.*

*Proof.* According to Theorem 1 and Section 4 in [19], by applying list scheduling, the makespan of $I^{MS}$ for the *MMSS* makespan problem is at most

$$len(G) + \sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M} \leq a \times len(G^*) + \sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M}$$
$$\leq (a+1) \times \max\left\{\sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M}, len(G^*)\right\}$$

The resulting schedule is a semi-partitioned schedule since two computation segments of a task can be executed on different processors. By Lemma 4, we conclude the theorem. □

Since the 1950s [10], [27], job/flow shop scheduling problems have been extensively studied. Although the problems are $\mathcal{NP}$-complete in the strong sense (even for very restrictive cases), algorithms with different properties have been reported in the literature. If time complexity is not a major concern, applying constraint programming as well as mixed integer linear programming (MILP) or branch-and-bound heuristics can derive optimal solutions for the job shop scheduling problem. In such a case, based on Theorem 10, our DGA has an approximation ratio of 2 for the *MMSS* makespan problem.

## 4.2 Remarks

At first glance, it may seem impractical to reduce the *MMSS* makespan problem to another very challenging problem, i.e., job shop scheduling, in the first step of our DGA algorithms. However, an advantage of considering the job shop scheduling problem is that it has been extensively studied in the literature, related results can directly be applied, and commercial tools, like the Google OR-Tools [3], can be utilized, as we did in our evaluation. In addition, due to Lemma 2, constructing a good dependency graph implies a good schedule for $I^{JS}$.

The last $n$ job shops, i.e., shops $Z+1, Z+2, \ldots, Z+n$, in $I^{JS}$, are in fact created just to match the original job shop scheduling problem. From the literature of flow and job shop scheduling, we know that these additional $n$ job shops can be removed by introducing *delay* ($l_{i,j}$ in Sec. 2.3). If the first computation segment $\theta_{i,1}$ of task $\tau_i$ is a non-critical section, this implies a non-zero release time $r_i$ of task $\tau_i$ in $I^{JS}$.

In our Google OR-Tools implementation for solving $I^{JS}$, the no overlap constraint has to be taken into consideration for both machine and job perspectives. For each machine, it prevents jobs assigned on the same machine from overlapping in time. For each job, it prevents sub-jobs for the same job from overlapping in time. The first constraint can be achieved by applying the `AddNoOverlap` method, by default supported in Google OR-Tools, for each machine. For the second constraint, instead of creating $n+Z$ job shops, we utilize the above concept by creating only $Z$ job shops and adding proper delays between the operations. We configure the start time (denoted as $\theta_{i,j}.start$) of a computation segment based on the end time (denoted as $\theta_{i,j}.end$) of an earlier computation segment. For notational brevity, we assign $\theta_{i,1}.start \geq 0$ and $\theta_{i,0}.end = 0$. For any $j \geq 2$ with $\lambda_{i,j} = 1$:

$$\begin{cases} \theta_{i,j}.start \geq \theta_{i,j-1}.end & \text{if } \lambda_{i,j-1} \text{ is } 1 \\ \theta_{i,j}.start \geq \theta_{i,j-2}.end + C_{i,j-1} & \text{if } \lambda_{i,j-1} \text{ is } 0 \end{cases} \tag{9}$$

In other words, if $\theta_{i,j-1}$ is a non-critical section, the execution time $C_{i,j-1}$ is added directly to the end (finishing) time of $\theta_{i,j-2}$; otherwise $\theta_{i,j}$ is started after the end time of $\theta_{i,j-1}$.

Hence, a proper job shop scheduling problem for $I^{JS}$ is $J_Z|r_j, l_j|C_{\max}$, i.e., scheduling of jobs with release time and delays between operations on $Z$ shops. An $a$-approximation algorithm for the problem $J_Z|r_j, l_j|C_{\max}$ can be used to construct a dependency graph. This problem is not widely studied and only few results can be found in the literature.

For a task system with a *flow-shop compatible access pattern*, i.e., the $Z$ mutex locks have a pre-defined total order, the instance $I^{JS}$ is in fact a flow shop problem. For a special case with three computation segments per task in which the second segment is a non-critical section, and the first and the third segments are critical sections of mutex locks 1 and 2, respectively, the constructed input $I^{JS}$ is a two-stage flow shop problem with delays, i.e., $F_2|l_j|C_{\max}$. For the problem $F_2|l_j|C_{\max}$, several polynomial-time approximation algorithms are known: Karuno and Nagamochi [24] developed a $\frac{11}{6}$-approximation, Ageev [1] developed a 1.5 approximation for a special case when $C_{i,1} = C_{i,3}$ for every task $\tau_i$, and Zhang and van de Velde [45] proposed polynomial-time approximation schemes (PTASes), i.e., $(1+\epsilon)$-approximation for any $\epsilon > 0$.

Specifically, Zhang and van de Velde [45] presented PTASes for different settings of the job/flow shop scheduling problems in [45]. For any of such scenarios, the approxi-

[3] https://developers.google.com/optimization/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3043742, IEEE Transactions on Computers

mation ratio of DGA is at most $2+\epsilon$ for any $\epsilon > 0$, according to Theorem 10.

### 4.3 Extension to Periodic Tasks

The treatment used in [36] to construct dependency graphs can also be applied here. That is, unroll the jobs of all the tasks in one hyper-period and then construct a dependency graph of these jobs. Suppose that the hyper-period $H$ of a task set is the least common multiple (LCM) of the periods of the all the tasks in this set. For each task $\tau_i$ that requests (at least) one resource, we create $H/T_i$ jobs of task $\tau_i$. For the $\ell$-th job of task $\tau_i$, we set its release time to $(\ell-1)T_i$ and its absolute deadline must be no later than $(\ell-1)T_i + D_i$. Since the jobs for one task should not have any execution overlap with each other, we only need one dedicated shop for them. Therefore, there are two modifications of the job shop problem scheduling considered in Sec. 4:

- The release time constraint is added for each job.
- Instead of optimizing the makespan, the objective is to minimize the maximum lateness.

And now the studied problem becomes $J_{Z+n}|r_j, l_{i,j}|L_{\max}$.

Afterwards, a dependency graph for all the jobs in one hyper-period is generated by solving the aforementioned flow/job shop scheduling problem. In the end, the schedules are generated offline by applying LIST-EDF or P-EDF, similar to fame-based task systems. And the generated schedules will be repeated in the upcoming hyper-periods.

Please note that such an extension can be applied to any periodic real-time task system, with the space cost of unrolling all the jobs, and the computation cost of increasing number of considered jobs to the number of jobs in one hyper-period.

## 5 Implementation and Overheads

In this section, we present details on how we implemented the dependency graph approach in LITMUS$^{\text{RT}}$ to support multiple critical sections per task. Afterwards, the implementation overheads are compared with the Flexible Multiprocessor Locking Protocol (FMLP) [4] provided by LITMUS$^{\text{RT}}$ for both partitioned and global scheduling.

### 5.1 Implementation Details

When implementing our approach in LITMUS$^{\text{RT}}$, we can either apply the table-driven scheduling that LITMUS$^{\text{RT}}$ provides, or implement a new binary semaphore which enforces the execution order of critical sections that access the same resource, since this order is defined in advance by the dependency graph. A static scheduling table can be generated over one hyper-period and be repeated periodically in a table-driven schedule. This table determines which sub-job is executed on which processor for each time point in the hyper-period. However, due to the large number of sub-jobs in one hyper-period and possible migrations among processors, the resulting table can be very large. To avoid this problem, we decided to implement a new binary semaphore that supports all the properties of our new approach instead.

Since our approach is an extension of the DGA by Chen et al. [11], and Shi et al. [36], our implementation is based on the source code the authors provided online [35], i.e., it is implemented under the plug-in Partitioned EDF with synchronization support (PSN-EDF), called P-DGA-JS, and the plug-in Global EDF with synchronization support (GSN-EDF), denoted G-DGA-JS.

The EDF feature is guaranteed by the original design of these two plug-ins. Therefore, we only need to provide the relative deadlines for all the sub-jobs of each task, and LITMUS$^{\text{RT}}$ will automatically update the absolute deadlines accordingly during runtime.

In order to enforce the sub-jobs to follow the execution order determined by the dependency graph, our implementation has to: 1) let the all the sub-jobs inside one job follow the predefined order; 2) force all the sub-jobs that access the same resource to follow the order determined by the graph.

The first order is ensured in LITMUS$^{\text{RT}}$ by default. The task deploy tool `rtspin` provided by the user-space library *liblitmus* defines the task structure, e.g., the execution order of non-critical sections and critical sections within one task, the related execution times, and the resource ID that each critical section accesses. Moreover, the resource ID for each critical section is parsed by `rtspin`, so the critical section can find the correct semaphore to lock, and in our implementation we do not have to further consider addressing the corresponding resources. Afterwards, `rtspin` emulates the work load in a CPU according to the task set. A sub-job can be released only when its predecessor (if any) has finished its execution. Please note that for sub-jobs related to critical sections the release time is not only defined by its predecessor's finish time inside the same job, but also related to another predecessor that accesses the same resource (if one exists).

A ticket system with a similar general concept to [35] is applied to enforce the execution order. However, due to different task structure which allows to support multiple critical sections, compared to [35], additional parameters had to be introduced and the structure of existed parameters had to be revised. To be precise, we extended LITMUS$^{\text{RT}}$ data structure `rt_params` that describes tasks, e.g., priority, period, and execution time, by adding:

- `total_jobs`: an integer which defines the number of jobs of the related task in one hyper-period.
- `total_cs`: an integer that defines the number of critical sections in this task.
- `job_order`: an array which defines the total order of the sub-jobs related to critical sections that access the same resource over one hyper-period. In addition, the last $Z$ elements record the total number of critical sections of the task set for each shared resource. Thus, the length of the array is the number of critical sections in one hyper-period plus the number of total shared resources, i.e., `len(job_order)` = `total_jobs` × `total_cs` + $Z$.
- `current_cs`: an integer that defines the index of the current critical section of the task that is being executed.
- `relative_ddls`: an array which records the relative deadlines for all sub-jobs of one task.

Furthermore, we implemented a new binary semaphore, named as `mdga_semaphore`, to make sure the execution order of all the sub-jobs that access the same resource follows the order specified by the dependency graph.

**Algorithm 1** DGA with multi-critical sections implementation

```
Input: New coming task $\tau_i$ {job_no, total_jobs, total_cs, current_cs, relative_ddls}, and
    Requested semaphore $s_z$ {semaphore_owner, serving_ticket, wait_queue};

    Function get_cs_order():
current_jobno ← $\tau_i$.job_no mod $\tau_i$.total_jobs;
index ← current_jobno × $\tau_i$.total_cs + current_cs;
cs_order ← $\tau_i$.job_order[index];

    Function mdga_lock():
if $s_z$.semaphore_owner is NULL and
    $s_z$.serving_ticket equals to $\tau_i$.cs_order then
    $s_z$.semaphore_owner ← $\tau_i$;
    Update the deadline for $\tau_i$;
    $\tau_i$ starts the execution of its critical section;
else
    Add $\tau_i$ to $s_z$.wait_queue;

    Function mdga_unlock():
$\tau_i$ releases the semaphore lock;
Update the deadline for $\tau_i$;
$\tau_i$.current_cs++;
if $\tau_i$.current_cs = total_cs then
    Set $\tau_i$.current_cs ← 0;
$s_z$.serving_ticket++;
if $s_z$.serving_ticket = num_cs then
    Set $s_z$.serving_ticket ← 0;
Next task $\tau_{next}$ ← the head of the wait_queue (if exists);
if serving_ticket equals to $\tau_{next}$.cs_order then
    $s_z$.semaphore_owner ← $\tau_{next}$;
    $\tau_{next}$ starts the execution of its critical section;
else
    $s_z$.semaphore_owner ← NULL;
    Add $\tau_{next}$ to $s_z$.wait_queue;
```

A semaphore has the following common components:

- litmus_lock protects the semaphore structure,
- semaphore_owner defines the current holder of the semaphore, and
- wait_queue stores all jobs waiting for this semaphore.

A new parameter named serving_ticket is added to control the non-work conserving access pattern of the critical sections, i.e., a job can only lock the semaphore and start its critical section if it holds the ticket equals to the corresponding serving_ticket.

The pseudo code in Algo. 1 shows three main functions in our implementation: The function **get_cs_order** returns the position of the sub-job in the execution order for all the sub-jobs that access the same shared resource during the run-time. In LITMUS$^{RT}$, job_no counts the number of jobs that one task releases. In order to find out the exact position of this job in one hyper-period, we apply a modulo operation on job_no and total_jobs. Since a job has multiple critical section and the current_cs represents the position of the critical section in a job, the index is calculated by counting the number of previous jobs' critical sections and the current_cs in this job. After that, the value of cs_order is searched from job_order based on the obtained index. We provide an example with 5 tasks which share two resources in the supplemental material.

The function **mdga_lock** is called in order to lock the semaphore and get access to the corresponding resource. After getting the correct position in the execution order in one hyper-period by applying function get_cs_order(), the semaphore's ownership will be checked. If the semaphore is occupied by another job at that moment, the new arriving job will be added to the wait_queue directly; otherwise, the semaphore's current_serving_ticket and the job's cs_order are compared. If they are equal, the semaphore's owner will be set to that job, and the job will start its critical section; otherwise, the job will be added to the wait_queue as well. In our setting the wait_queue is sorted by the jobs' cs_order, i.e., the job with the smallest cs_order is the head of the waiting queue. Hence, only the head of the wait_queue has to be checked when the current semaphore owner finishes its execution, rather than checking the whole unsorted wait_queue.

The function **mdga_unlock** is called once a job has finished its critical section and tries to unlock the semaphore. The task's current_cs is added by one to point to the next possible critical section in this job. If current_cs reaches to the total_cs, which means all the critical sections in this job have finished their execution, then the current_cs will be reset to zero. Next, the semaphore's serving_ticket is increased by 1, i.e., it is ready to be obtained by the successor in the dependency graph. If serving_ticket reaches the total number of critical sections related to this resource in one hyper-period, i.e., num_cs, the dependency graph is traversed completely, i.e., all sub-jobs that access the related resource finished their executions of the critical sections in the current hyper-period, the parameter serving_ticket is reset to 0 to start the next iteration. Please note, the num_cs can be found in the last $Z$ elements of job_order according to the related resource id. After that, the first job (if any) in the wait_queue, named as $\tau_{next}$ is checked. If $\tau_{next}$ has the cs_order which equals to the semaphore's serving_ticket, the the semaphore's owner is set as $\tau_{next}$, and $\tau_{next}$ can start the execution of its critical section. Otherwise, the semaphore owner is set as NULL, and the task $\tau_{next}$ is put back to the corresponding wait_queue.

Additionally, each sub-job has its own modified deadline accordingly, which means each job can have different deadlines when it is executing different segments. Therefore, we have to take care of the deadline update during the implementation. When we deploy a task using rtspin to the system, we deliver the relative deadline of its first sub-task as the relative deadline of the whole task. Since no two continuous non-critical sections are allowed in the task model, once a sub-job finishes its execution, either mdga_lock or mdga_unlock is called. If mdga_lock is called, the new critical section's deadline is updated by searching the relative_deadline; if mdga_lock is called, only the finished critical section can update related job's deadline for its successor (if any), since $\tau_{next}$'s deadline has been updated when it tries to lock the semaphore already.

The implementations for the global and partitioned plug-ins are similar. However, due to the frequent preemption and/or interrupts in global scheduling, the preemption has to be disabled during the executions of semaphore related functions in order to protect the functionalities of aforementioned functions.

| Max. (Avg.) in $\mu s$ | CXS | RELEASE | SCHED | SCHED2 | SEND-RESCHED |
|---|---|---|---|---|---|
| P-FMLP | 29.51 (0.98) | 17.68 (0.96) | 31.85 (1.31) | 28.77 (0.18) | 66.33 (2.86) |
| P-DGA-JS | 30.65 (1.25) | 18.63 (1.02) | 31.09 (1.64) | 29.43 (0.19) | 59.09 (21.06) |
| G-FMLP | 30.51 (1.05) | 48.53 (3.75) | 45.99 (1.51) | 29.62 (0.16) | 72.26 (2.50) |
| G-DGA-JS | 26.87 (0.94) | 30.01 (2.19) | 30.25 (1.02) | 19.26 (0.14) | 72.53 (21.50) |
| P-LIST-EDF | 18.76 (0.90) | 18.98 (1.06) | 48.50 (1.33) | 29.25 (0.16) | 38.3 (1.61) |
| G-LIST-EDF | 30.87 (1.79) | 61.63 (12.06) | 59.05 (4.46) | 27.17 (0.25) | 72.09 (20.77) |

TABLE 2
Overheads of protocols in LITMUS$^{\text{RT}}$.

### 5.2 Overheads Evaluations

We evaluated the overheads of our implementation in the following platform: a cache-coherent SMP, consisting of two 64-bit Intel Xeon Processor E5-2650Lv4, with 35 MB cache and 64 GB main memory. The FMLP supported in LITMUS$^{\text{RT}}$ was also evaluated for comparisons, including P-FMLP for partitioned scheduling and G-FMLP for global scheduling. These four protocols are evaluated using same task sets where each task has multiple critical sections.

The overheads that we tracked are:

- **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., find the next job to be executed.
- **SCHED2**: time spent to perform post context switch and management activities.
- **SEND-RESCHED**: inter-processor interrupt latency, including migrations.

The overheads are reported in Table 2, which shows that the overheads of our approach and those of P-FMLP, G-FMLP are comparable. Furthermore, the implementations provided in [36], called P-LIST-EDF and G-LIST-EDF, were evaluated to examine the overhead and reported in Table 2. The direct comparison between P-LIST-EDF and P-DGA-JS (G-LIST-EDF and G-DGA-JS, respectively) is not possible because they are designed for different scenarios, depending on the number of critical sections per task. The reported overheads in Table 2 for our approach are for task sets with multiple critical sections per task, whilst the overheads for P-LIST-EDF and G-LIST-EDF were for task sets with one critical section per task. Regardless, they are in the same order of magnitude.

## 6 Evaluations

We evaluated the performance of the proposed approach by applying numerical evaluations for both frame-based task sets and periodic task sets, and measuring its overheads.

### 6.1 Evaluations Setup

We conducted evaluations on $M$ = 4, 8, and 16 processors. Based on $M$, we generated $100$ synthetic task sets with $10M$ tasks each, using the RandomFixedSum method [14]. We set $\sum_{\tau_i \in \mathbf{T}} U_i = M$ and enforced $U_i \leq 0.5$ for each task $\tau_i$, where $U_i = \frac{C_i}{T_i}$ is the utilization of a task. The number of shared resources (binary semaphores) $Z$ was either $4$, $8$, or $16$. Each task $\tau_i$ accesses the available shared resource randomly between 2 and 5 times, i.e., $\sum \lambda_{i,j} \in [2,5]$. The total length of the critical sections $\sum_{\lambda_{i,j}=1} C_{i,j}$ is a fraction of the total execution time $C_i$ of task $\tau_i$, depended on $H \in \{[5\% - 10\%], [10\% - 40\%], [40\% - 50\%]\}$. When considering shared resources in real-time systems, the utilization of critical sections for each task in classical settings is relatively low. However, with the increasing computation demand in real-time systems (e.g., for machine learning algorithms), adopted accelerators, like GPUs, behave like classical shared resources (i.e., they are non-preemptive and mutually exclusive), but have a relatively high utilization. Hence, we chose possible settings of $H$ that cover the complete spectrum. The total length of critical sections and non-critical sections are split into dedicated segments by applying UUniFast [14] separately. For task $\tau_i$, the number of critical sections $Num_{cs}$ equals to $\sum \lambda_{i,j}$, and the number of non-critical sections $Num_{ncs} = Num_{cs} + 1$. In the end, the generated non-critical sections and critical sections are combined in pairs, and the last segment is the last non-critical section. We evaluated all resulting 27 combinations of $M$, $Z$, and $H$.

The dependency graph is generated by applying:

1) The method in Sec. 4 with the objective to minimize the makespan, denoted as **JS**. We utilized the constraint programming approach provided in the Google OR-Tools to solve the job shop scheduling problem,
2) The extension to multiple critical sections sketched in [36], denoted as **PRP**. To check the feasibility of the generated dependency graph, one simulated schedule with respect to the dependency graph is generated.

We name these algorithms by combining:

1) *JS/PRP*: the two different dependency graph generation methods.
2) *LEDF/PEDF*: to schedule the generated graph, we used the LIST-EDF in [36] (LEDF) or partitioned EDF (PEDF) in [37], and a *worst-fit* partitioning algorithm.
3) *P/NP*: preemptive or non-preemptive schedule for critical sections.

We also compare our approach with the following protocols regarding their schedulability by applying the publicly available tool SET-MRTS in [12] with the same naming:

- Resource Oriented Partitioned PCP (ROP-PCP) [22]: Binds the resources on dedicated processors and schedules tasks using semi-partitioned PCP.
- GS-MSRP [41]: THe Greedy Slacker (GS) partitioning heuristic for spin-based locking protocol MSRP [16], using Audsley's Optimal Priority Assignment [2] for priority assignment. (LP) analysis for global FP scheduling using the FMLP [4].
- LP-GFP-PIP: LP-based global FP scheduling using the Priority Inheritance Protocol (PIP) [13].
- LP-PFP-DPCP [6]: DPCP [33] with a Worst-Fit-Decreasing (WFD) task assignment strategy [6]. The analysis is based on a linear-programming (LP).
- LP-PFP-MPCP [6]: MPCP [32] with a Worst-Fit-Decreasing (WFD) task assignment strategy as proposed in [6]. The analysis is based on a LP.
- LP-GFP-FMLP [4]: FMLP [4] for global FP scheduling with a LP analysis.

*Note that a comparison to the original DGA in [11] is not possible, since the approach in [11] is only applicable when there is one*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3043742, IEEE Transactions on Computers

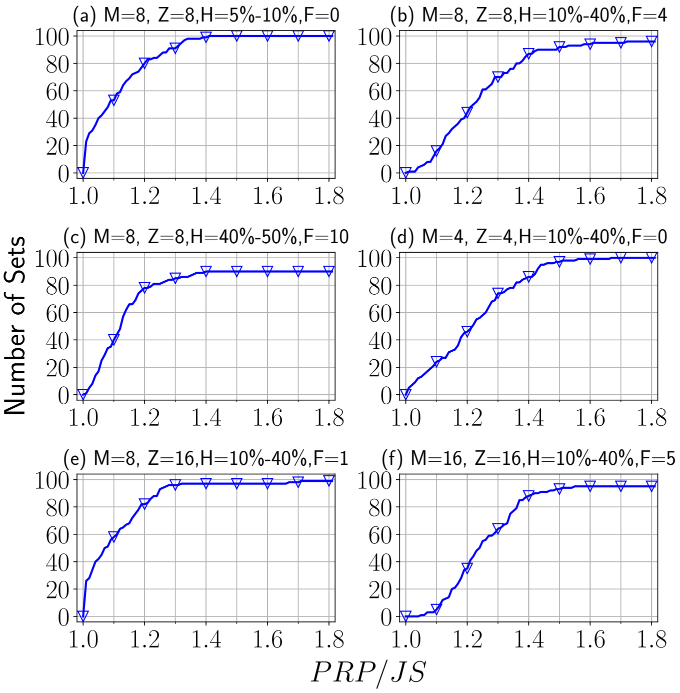


Fig. 1. Comparison of critical paths from the two graph generation methods.

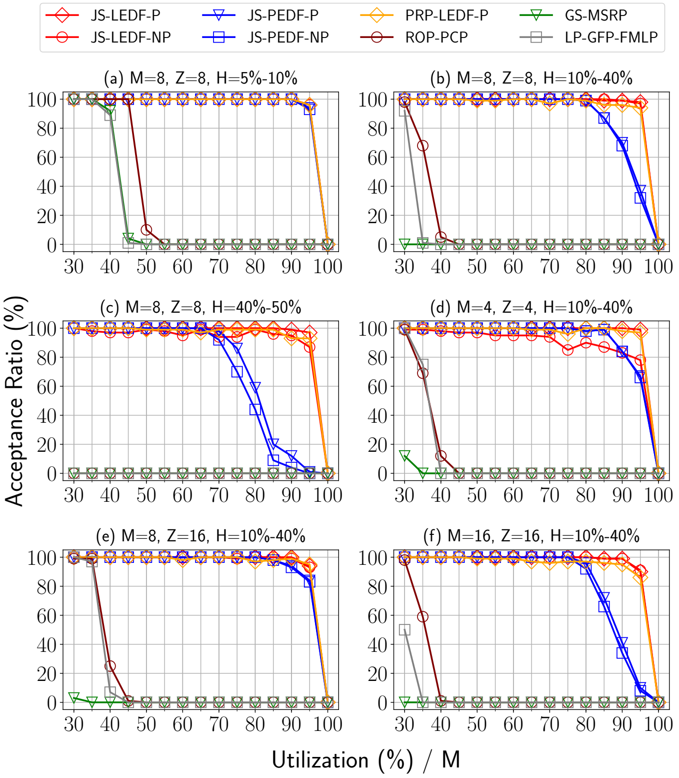


Fig. 2. Schedulability of different approaches for frame-based task sets.

*critical section per task.* We also launched the evaluation of the Priority Inheritance Protocol (PIP) [13] based on LP, but we were not able to collect the complete results because validating a task set took multiple hours. However, according to [11], [36], [43], the PIP based on LP performs similar to LP-GFP-FMLP.

### 6.2 Evaluation Results for Frame-Based Tasks

For frame-based task systems, we set $T = D = 1$ for all the tasks, i.e., the execution time of each task is the same as its utilization. We tracked the number of dependency graphs calculated with PRP where the ratio of $PRP/JS$ is less than a certain factor. The results are shown in Fig. 1, where $F$ represents the number of infeasible dependency graph for the *PRP* method due to cycle detection. The job-shop based dependency graph generation method clearly outperform the method extended from the original DGA. In addition, the failure rate of the *PRP* is increasing when the length of critical sections is increased, i.e., Fig. 1 (a), (b), and (c). The other results show similar trends and are thus omitted due to space limitation.

In our schedulability evaluation, we considered synthetic task sets under the aforementioned settings, testing the utilization level from 0 to $100\% \times M$ in steps of $5\%$. The acceptance ratios of LP-PFP-DPCP and LP-PFP-MPCP are zero for all configurations, even for utilization levels $\leq 20\% \times M$. Hence, we omitted them in Fig. 2. Additionally, considering the readability of the figure, we only show *PRP-LEDF-P*, which has the best performance for the approaches where dependency graphs are generated by *PRP*.

Fig. 2 shows that our approach outperforms the other non-DGA based methods significantly for all evaluated settings, and performs slightly better than the methods using *PRP*. Fig. 1 and Fig. 2 also show that a better dependency graph, i.e, a shorter critical path, not always results in better schedulability in the second step of the DGA.

### 6.3 Evaluation Results for Periodic Tasks

We applied constraint programming to solve the job shop problem $J_Z|r_j, l_j|L_{\max}$ and construct the dependency graph. We extended the settings for frame-based task sets in Sec. 6.2 to periodic task systems by choosing the period $T_i$ randomly from a set of semi-harmonic periods, i.e., $T_i \in \{1, 2, 5, 10\}$, which is a subset of the periods used in automotive systems [20], [25]. We used a small range of periods to generate reasonable task sets with high utilization of the critical sections, which are otherwise by default not schedulable.

Due to space limitation, only a subset of the results is presented in Fig. 3. When the utilization of critical sections is high, i.e., $H = [40\% - 50\%]$ in Fig. 3 (c), or under medium utilization when the number of processor and shared resources are relative high, i.e., $M = H = 16$ in Fig. 3 (f), our approaches outperforms the other methods significantly. However, when the utilization of critical sections is low, i.e., $H = [5\% - 10\%]$ in Fig. 3 (a) and (b), ROP-PCP outperformed the proposed approaches. The reason is that the constraint programming of the problem $J_Z|r_j, l_j|L_{\max}$ has the objective to minimize the maximum lateness, but ignores the execution order of the sub-jobs that do not have any influence on the optimal lateness, which may lead to lower performance when the utilization of the non-critical sections is high. When the utilization of critical section is medium, i.e., $H = [10\% - 40\%]$, and the number of processor is relative small i.e., $M = \{4, 8\}$, the newly proposed

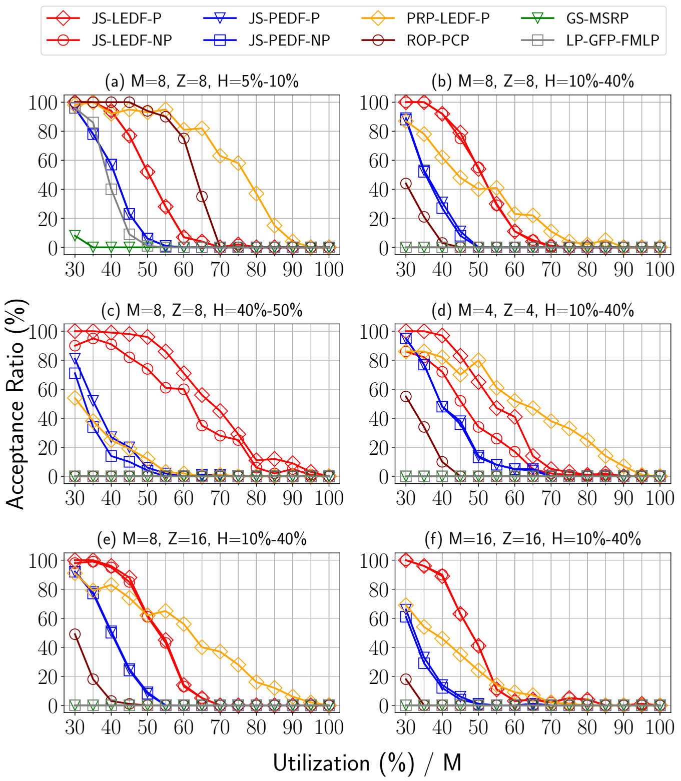


Fig. 3. Schedulability of different approaches for periodic task sets.

DGA-based methods and the extension PRP-LEDF-P both outperform all the other methods significantly, but their relation differs depending on the utilization value.

## 7 Conclusion and Future Work

We have removed an important restriction, i.e., only one critical section per task, of the recently developed dependency graph approaches (DGA). Regarding the computational complexity, we show that the multiprocessor synchronization problem is $\mathcal{NP}$-complete even in very restrictive scenarios, as detailed in Sec. 3. We propose a systematic design flow based on the DGA by using existing algorithms developed for job/flow shop scheduling and provide the approximation ratio(s) for the derived makespan.

The evaluation results in Sec. 6.2 show that our approach is very effective for frame-based real-time task systems. Extensions to periodic task systems are presented in Sec. 4.3, and the evaluation results show that our approach has significant improvements, compared to existing protocols, in most evaluated cases except light shared resource utilization. This paper significantly improves the applicability of the DGA by allowing arbitrary configurations of the number of non-nested critical sections per task.

In this paper, we focus on the long-standing problem of resource sharing of periodic tasks and on providing a good solution for this most adopted real-time task model. As a result, we achieved a solution that outperforms the methods in the literature which can be applicable to this task model. In the future, we plan to explore the possibility to apply the dependency graph approach on sporadic task systems, which do not have predefined arrival times of jobs.

## Acknowledgments

This paper is supported by DFG, as part of the Collaborative Research Center SFB876, project A1 and A3 (http://sfb876.tu-dortmund.de/). The authors thank Zewei Chen and Maolin Yang for their tool SET-MRTS [12] (Schedulability Experimental Tools for Multiprocessors Real Time Systems) to evaluate the GS-MSRP, LP-GFP-FMLP, and ROP-PCP in Fig. 2 and Fig. 3.

## References

[1] A. A. Ageev. A 3/2-approximation for the proportionate two-machine flow shop scheduling with minimum delays. In *Approximation and Online Algorithms, 5th International Workshop, WAOA*, 2007.

[2] 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.

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

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

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

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

[7] B. B. Brandenburg and J. H. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS*, 2010.

[8] 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.

[9] 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 *RTSS*, 2006.

[10] B. Chen, C. N. Potts, and G. J. Woeginger. *A Review of Machine Scheduling: Complexity, Algorithms and Approximability*, pages 1493–1641. Springer US, Boston, MA, 1998.

[11] J.-J. Chen, G. von der Brüggen, J. Shi, and N. Ueter. Dependency graph approach for multiprocessor real-time synchronization. In *IEEE Real-Time Systems Symposium, RTSS*, pages 434–446, 2018.

[12] Z. Chen. SET-MRTS: Schedulability Experimental Tools for Multiprocessors Real Time Systems. https://github.com/RTLAB-UESTC/SET-MRTS-public, 2018.

[13] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *RTSS*, 2009.

[14] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS*, pages 6–11, 2010.

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

[16] 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.

[17] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Co., 1979.

[18] L. A. Goldberg, M. Paterson, A. Srinivasan, and E. Sweedyk. Better approximation guarantees for job-shop scheduling. *SIAM J. Discrete Math.*, 14(1):67–92, 2001.

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

[20] 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*, 2017.

[21] 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.

[22] 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*, 2016.

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

[24] Y. Karuno and H. Nagamochi. A better approximation for the two-machine flowshop scheduling problem with time lags. In *Algorithms and Computation, 14th International Symposium*, 2003.

[25] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmark for free. In *WATERS*, 2015.

[26] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Real-Time Systems Symposium*, pages 469–478, 2009.

[27] E. L. Lawler, J. K. Lenstra, A. H. R. Kan, and D. B.Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in Operations Research and Management Science*, 4:445–522, 1993.

[28] J. Lenstra and A. Rinnooy Kan. Computational complexity of discrete optimization problems. *Ann. Discrete Math.*, 4, 1979.

[29] M. Mastrolilli and O. Svensson. Hardness of approximating flow and job shop scheduling problems. *Journal of the ACM*, 58(5):20:1–20:32, Oct. 2011.

[30] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *Principles of Distributed Systems - International Conference*, pages 253–269, 2010.

[31] 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.

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

[33] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, 1988.

[34] 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.

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

[36] J. Shi, N. Ueter, G. von der Brüggen, and J.-j. Chen. Multiprocessor synchronization of periodic real-time tasks using dependency graphs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 279–292, 2019.

[37] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen. Partitioned scheduling for dependency graphs in multiprocessor real-time systems. In *Proceedings of the 25th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, 2019.

[38] D. B. Shmoys, C. Stein, and J. Wein. Improved approximation algorithms for shop scheduling problems. *SIAM J. Comput.*, 23(3):617–632, 1994.

[39] Y. Sotskov and N. Shakhlevich. NP-hardness of shop-scheduling problems with three jobs. *Discrete Appl. Math.*, 59(3):237–266, 1995.

[40] G. von der Brüggen, J.-J. Chen, W.-H. Huang, and M. Yang. Release enforcement in resource-oriented partitioned scheduling for multiprocessor systems. In *RTNS*, 2017.

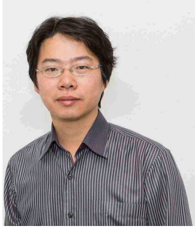
[41] A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *RTSS*, 2013.

[42] 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.

[43] M. Yang, A. Wieder, and B. B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *Real-Time Systems Symposium (RTSS)*, pages 1–12, 2015.

[44] W. Yu, H. Hoogeveen, and J. K. Lenstra. Minimizing makespan in a two-machine flow shop with delays and unit-time operations is np-hard. *J. Scheduling*, 7(5):333–348, 2004.

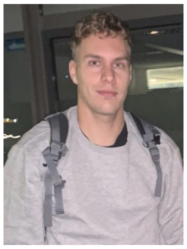
[45] X. Zhang and S. L. van de Velde. Polynomial-time approximation schemes for scheduling problems with time lags. *J. Scheduling*, 13(5):553–559, 2010.



**Jian-Jia Chen** is Professor at Department of Informatics in TU Dortmund University in Germany. He was Juniorprofessor at Department of Informatics in Karlsruhe Institute of Technology (KIT) in Germany from May 2010 to March 2014. He received his Ph.D. degree from Department of Computer Science and Information Engineering, National Taiwan University, Taiwan in 2006. He received his B.S. degree from the Department of Chemistry at National Taiwan University 2001. Between Jan. 2008 and April 2010, he was a postdoc researcher at ETH Zurich, Switzerland. His research interests include real-time systems, embedded systems, energy-efficient scheduling, power-aware designs, temperature-aware scheduling, and distributed computing. He received the European Research Council (ERC) Consolidator Award in 2019. He has received more than 10 Best Paper Awards and Outstanding Paper Awards and has been part of Technical Committees in many international conferences.

**Junjie Shi** received his master degree in electronic technology and information technology from TU Dortmund University, Germany, in 2017 and now is a PhD student at TU Dortmund University, supervised by Prof. Dr. Jian-Jia Chen. His research interests are resource-sharing protocols for real-time systems, resource aware scheduling for machine learning algorithms, and computation offloading for real-time systems.

**Georg von der Brüggen** is a Postdoctoral Researcher at the Max Planck Institute for Software Systems in Kaiserslautern, Germany. He received his PhD from TU Dortmund University, Germany, in 2019 and his Diploma degree in computer science from TU Dortmund University, Germany, in 2013. His research interests are in the area of embedded and real-time systems with a focus on real-time scheduling. He participated in the program committee of multiple international conferences and workshops in the area of real-time systems, like RTSS, RTCSA, and RTNS, and was the program chair of the RTNS junior workshop JRWRTC in 2018.



**Niklas Ueter** received his master degree in computer science from TU Dortmund University, Germany, in 2018 and now is a PhD student at TU Dortmund University, supervised by Prof. Dr. Jian-Jia Chen. His research interests are in the area of embedded and real-time systems with a focus on real-time scheduling.