technische universität
dortmund

Graph-Based Optimizations for Multiprocessor Nested Resource Sharing

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

Citation: 10.xxx.xxx.xxx

BibTeX:

```
@inproceedings{DBLP:conf/rtcsa/ShiUBC21,
  author    = {Junjie Shi and
               Niklas Ueter and
               Georg von der Br{\"{u}}ggen and
               Jian{-}Jia Chen},
  title     = {Graph-Based Optimizations for Multiprocessor Nested Resource Sharing},
  booktitle = {27th {IEEE} International Conference on Embedded and Real-Time Computing
               Systems and Applications, {RTCSA} 2021, Houston, TX, USA, August 18-20,
               2021},
  pages     = {129--138},
  publisher = {{IEEE}},
  year      = {2021},
  url       = {https://doi.org/10.1109/RTCSA52859.2021.00023},
  doi       = {10.1109/RTCSA52859.2021.00023},
  timestamp = {Tue, 05 Oct 2021 08:59:33 +0200},
  biburl    = {https://dblp.org/rec/conf/rtcsa/ShiUBC21.bib},
  bibsource = {dblp computer science bibliography, https://dblp.org}
}
```


CS 12 computer science 12

Graph-Based Optimizations for Multiprocessor Nested Resource Sharing

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

***Abstract*—Multiprocessor resource synchronization and locking protocols are of great importance to utilize the computation power of multiprocessor real-time systems. Hence, in the past decades a large number of protocols have been developed and analyzed. The recently proposed dependency graph approach has significantly improved the schedulability for frame-based and periodic real-time task systems. However, the dependency graph approach only supports non-nested resource access, i.e., each critical section can only access one shared resource.**

In this paper, we develop a dependency graph based protocol that allows nested resource access, where a critical section can access multiple shared resources at the same time. First, constraint programming is applied to construct a dependency graph that determines the execution order of critical sections. Afterwards, a schedule is generated based on this order. To show the feasibility of our proposed protocol, we provide extensive numerical evaluations under different configurations. The evaluation results show that our approach has very good performance with respect to schedulability for frame-based and periodic real-time task systems, whereas the existing results applicable for sporadic task systems have worse performance under such a limited setting.

I. Introduction

In concurrent real-time systems, mutually exclusive (mutex) accesses to shared resources, e.g., shared memory, external devices, and shared data, prevent data corruptions and race conditions. This means that when a task is accessing a mutex shared resource, no other tasks can access the shared resource at the same time. Code segments that access shared resource(s) are called *critical sections*. Semaphores and mutex locks are widely used to ensure mutual exclusion when accessing critical sections. Such a mutually exclusive resource access can result in priority inversion where a higher-priority task has to wait for a resource that is held by a lower-priority task.

To avoid priority inversion as much as possible and hence improve the schedulability of the system, resource synchronization and locking protocols have been developed and analyzed since the 1990s. In uni-processor systems, the Priority Inheritance Protocol (PIP) [36], the Priority Ceiling Protocol (PCP) [36], and the Stack Resource Policy (SRP) [4] have been widely accepted and used. For multiprocessor systems, multiple protocols like the Multiprocessor PCP (MPCP) [33], the Distributed PCP (DPCP) [34], the Multiprocessor SRP (MSRP) [18], the Multiprocessor Resource Sharing Protocol (MrsP) [10], and the Flexible Multiprocessor Locking Protocol (FMLP) [6] have been proposed.

The Dependency Graph Approach (DGA) that was recently proposed by Chen et al. [13] provides a promising new direction, since the DGA has shown significant schedulability improvement compared to existing protocols. The DGA mechanism has two steps:

1) A dependency graph (DG) is constructed to determine the execution order of critical sections for a shared resource.
2) A schedule is generated by applying multiprocessor scheduling algorithms, respecting the execution order given by the DG(s).

The original approach [13] can only be applied to a restrictive setting, namely frame-based task systems with only one critical section per task, but has been extend to periodic task systems [37] and to multiple critical sections per task when each critical section accesses only one shared resource [12].

However, the analysis and discussion focused on non-nested resource sharing, i.e., each critical section can only access one shared resource. When nested resource sharing is allowed, two fundamental problems are considered.

First, *deadlocks* must be prevented. A deadlock is a situation where (in the simplest case) the execution of two tasks τ_i and τ_j is postponed indefinitely, because τ_i holds a mutually exclusive resource 1 and waits a mutually exclusive resource 2 held by τ_j, while τ_j in turn is waiting to get access to resource 1. These two conditions, called *hold-and-wait* and *circular waiting*, are essential for a deadlock to happen. One trivial approach to break the *hold-and-wait* condition is to use a coarse-grained group lock, i.e., each critical section has to lock all the requested resources before its execution starts. However, this approach introduces significant unnecessary blocking time which results in reduced system performance.

Second, *transitive blocking chains* should be broken. When nested resource sharing is allowed, these transitive blocking chains can result in unnecessary blocking time for tasks even though no conflict resources are requested among them. For example, consider that task τ_1 requests resource 1 and 2, task τ_2 requests resource 2 and 3, and task τ_3 requests resource 3 and 4. Although τ_1 and τ_3 can be executed in parallel (by only considering the resource access condition), τ_3 can be potentially blocked by τ_1 if it is blocked by τ_2 (due to resource 3), and τ_2 is blocked by τ_1 (due to resource 1).

Contribution: Taking the aforementioned two problems into consideration, in this paper, we study nested resource accesses for both frame-based and periodic real-time task systems with synchronous releases under the DGA to overcome the limitation of existing DGA mechanisms. Our contributions are:

- We propose a graph based approach to synchronize (multiple) nested resource accesses per task for frame-based real-time task systems in Section III by reducing the dependency graph construction to constraint programming. An illustrative example is provided to demonstrate the detailed work flow of our approach.
- We detail properties of our approach in Section IV to explain how the two fundamental problems, namely *deadlocks* and *transitive blocking chains*, are solved.
- Two extensions of our method are explained and discussed, 1) from a special nested locking pattern (i.e., lock all at once) to normal locking pattern (i.e., lock only when needed) in Section V; 2) from frame-based task systems to periodic task systems in Section VI and discuss the incurred complexity of our approach.
- We provide exhaustive numerical evaluations in Section VII, demonstrating the performance of the proposed approach. We evaluate a range of system configurations that are still solvable by our approach, i.e., we only consider relatively small periods. For the class of frame-based and periodic task systems, our approach outperforms the state-of-the-art significantly in most evaluated settings. In addition, the advantages and limitations of our approach comparing with the state-of-the-art are discussed.

We note that the DGA is limited to periodic task systems with sequential and segmented accesses to critical and non-critical sections, while the methods in [31] and [24] are applicable for sporadic task systems and not limited to sequential access patterns to be defined in Section II.

II. System Model

We consider scheduling of a set of n periodic real-time tasks $\mathbb{T} = \{\tau_1, \ \ldots, \ \tau_n\}$ on M identical (homogeneous) processors $\mathbb{P} = \{P_1, P_2, \ldots, P_M\}$. The system has Z shared resources $\mathbb{Z} = \{z_1, z_2, \ldots, z_Z\}$. All tasks release an infinite number of task instances, called jobs, strictly periodically and the first job of all tasks is released at time 0. Each task is described by the tuple $\tau_i = (\Theta_i, C_i, D_i, T_i)$, where:

- $\Theta_i = \{\theta_{i,1}, \theta_{i,2}, \ldots, \theta_{i,\eta_i}\}$ denotes the totally-ordered set of computation segments with the constraint that if $\theta_{i,j}$ is a non-critical section then $\theta_{i,j-1}$ and $\theta_{i,j+1}$ (if they exist) must be critical sections. The total-order implies precedence constraints between these computation segments, enforcing sequential execution.
- Each computation segment $\theta_{i,j}$ is defined by its worst-case execution time (WCET) $c_{i,j}$ and its resource access sequence $\sigma_{i,j}$.
- C_i denotes the total WCET of τ_i, i.e., $C_i = \sum_{j=1}^{\eta_i} c_{i,j}$.
- T_i is the inter-arrival time (period) of task τ_i, i.e., if a τ_i releases a job at time t, then the next job of τ_i is released exactly at $t + T_i$.
- D_i is the relative deadline of τ_i, i.e., a job of τ_i released at time t must finish its execution no later than its absolute deadline $t + D_i$. We consider constrained-deadline task systems, i.e., $D_i \leq T_i$ for every task $\tau_i \in \mathbb{T}$.

A *frame-based* task system is a special case of periodic task systems, where all tasks are released simultaneously and have the same period. That the original DGA approach [13] was extended to periodic task systems [37] suggest that a good solution of nested resource synchronization for frame-based real-time task systems can be a cornerstone for real-time tasks in a periodic setting. For the rest of this paper, we discuss the methodology for frame-based task systems and the extension to periodic tasks is presented in Section VI.

Definition 1. ***Resource Access Sequence:*** *The j-th resource access sequence of task τ_i (related to computation segment $\theta_{i,j}$) is a finite sequence of tuples $\sigma_{i,j} = a_1, a_2, \ldots, a_\ell$, where each $a_i \in (\mathbb{R}_+, \mathcal{P}(\mathbb{Z}))$ and $\mathcal{P}(\mathbb{Z})$ denotes the power set of $\mathbb{Z}$. The first element in each access a_i, i.e., a_i^0, denotes the execution duration while holding these resources and the second element, i.e., a_i^1, denotes the set of locked resources.*

If resource z_j is used in a_k but not in a_{k+1} then z_j is unlocked after a_k finished. If z_j is used in a_k and in a_{k+1} then the lock for z_j remains. If resource z_j is not used in a_k but in a_{k+1} then z_j is locked after a_k finished.

Definition 2. ***Computation Segment:*** *The j-th computation segment of a task τ_i is defined by $\theta_{i,j} = (c_{i,j}, \sigma_{i,j})$, where $c_{i,j} = \sum_{a_k \in \sigma_{i,j}} a_k^0$ and denotes the overall execution time of the resource access sequence $\sigma_{i,j}$. If $\cup_{a_k \in \sigma_{i,j}} a_k^1 = \emptyset$ then the computation segment is a non-critical section and a critical section otherwise.*

For example, the computation segment $(4, (4, \{\emptyset\}))$ denotes a non-critical section with 4 units of execution time.

We consider two distinct locking strategies for nested resource requests, namely, *nested locking* and *all-at-once locking*. In *nested locking*, resources may be requested when a critical section already holds resource(s).

Example 1. *If the j-th resource access sequence of τ_i is described by $\sigma_{i,j} = ((1, \{z_1\}), (2, \{z_1, z_2\}), (1, \{z_1\}))$, it first locks z_1 for the entire duration (1+2+1), and z_2 is locked after 1 time unit and released 2 time units later.*

Definition 3. ***All-at-once Locking:*** *A critical section is said to be locked* all-at-once *if that critical section can only be entered after all the requested mutex locks are granted simultaneously (all-at-once) successfully; otherwise, a job is blocked until the resource access is granted. At the end of a critical section, all of its locked mutex locks are unlocked. Formally any resource access sequence $\sigma_{i,j}$ is said to be locked* all-at-once *if $\sigma_{i,j} = (\sum_{a_k \in \sigma_{i,j}} a_k^0, \cup_{a_k \in \sigma_{i,j}} a_k^1)$.*

Please note, that nested locking can be transferred to a *all-at-once* locking. For instance, when applying the *all-at-once locking* strategy to Example 1, the notation becomes $\sigma_{i,j} = (4, \{z_1, z_2\})$, which denotes that the j-th resource access sequence of τ_i locks z_1 and z_2 all-at-once and holds them for the entire duration of 4 time units.

III. Graph Based Approach

This section presents the proposed graph-based approach for frame-based real-time task systems, where each task can have (multiple) critical sections that request nested resources. Firstly, the two consecutive steps of our approach are explained. Afterwards, an illustrative example is given to demonstrate the workflow in detail.

A. Step I: Dependency Graph Construction

We construct a directed acyclic graph (DAG) $G = (V, E)$. For each computation segment $\theta_{i,j}$ of all tasks τ_i in $\mathbb{T}$ a vertex is created and, to ensure the sequential execution of tasks, $\theta_{i,j}$ is a predecessor of $\theta_{i,j+1}$ in G for $j = 1, 2, \ldots, \eta_i - 1$, i.e., $\forall \tau_i, j \in \{1, 2, \ldots, \eta_i - 1\} : (\theta_{i,j}, \theta_{i,j+1}) \in E$.

Let the set of all computation segments that are critical sections guarded by a mutex for resource z_k be denoted by $\Theta(k)$. For each $z_k \in \mathbb{Z}$, the sub-graph of the computation segments in $\Theta(k)$ can be constructed as a directed chain, representing the execution order of these computation segments with respect to the corresponding shared resource. Since we consider nested resources, each critical section may request multiple resources. Hence, each critical section may have multiple predecessors, depending on the number of resources it requests.

The construction of a dependency graph for nested resource sharing can be formulated as a constraint programming. We represent each shared resource in this system by a particular machine, i.e., m_k, and each task τ_i is assigned to a dedicated machine, i.e., m_{Z+i}. Thus, in total $Z+n$ machines are created:

- Machine m_k, where $k \in \{1, 2, \ldots, Z\}$ exclusively executes critical sections guarded by mutex lock k, i.e., only if the critical section $\theta_{i,j}$ requests resource z_k it is executed on machine m_k.
- Machine m_{Z+i} is only used to execute non-critical sections $\theta_{i,j}$ of τ_i.

We note that the machines are purely conceptual with the intention to generate an execution order irrespective of the actual number of processors in our studied problem.

The operation of each computation segment $\theta_{i,j}$ is expressed as a processing on the corresponding machine for the duration of the segment's execution time. To be precise, each task τ_i is assigned to the machine m_{Z+i} for the execution of its non-critical section. Once a task τ_i has to access a shared resource, the execution of its critical section will be migrated to the assigned machine for the shared resource. Moreover, the following four constraints have to be satisfied by any feasible solution:

Constraint 1 (No-overlap Constraint)**.** *Any two tasks (or segments) cannot be executed on a machine at the same time. That is, for any machine at any time point, there is at most one task (or segment) executed on that machine, i.e.,*

$$\forall m_{k \in \mathbb{Z}}, i \neq g : \theta_{i,j}.start \geq \theta_{g,\ell}.finish \text{ or } \theta_{g,\ell}.start \geq \theta_{i,j}.finish \quad (1)$$

Constraint 2 (Precedence Constraint)**.** *For any two computation segments with precedence constraints, e.g., $\theta_{i,j} \prec \theta_{g,\ell}$, the starting time of $\theta_{g,\ell}$ is no earlier than the finishing time of $\theta_{i,j}$ if $\theta_{i,j} \prec \theta_{g,\ell}$, i.e.,*

$$\forall m_{k \in \mathbb{Z}'} : \ \theta_{g,\ell}.start \geq \theta_{i,j}.finish \ \ if \ \ \theta_{i,j} \prec \theta_{g,\ell} \quad (2)$$

and

$$\forall m_{k \in \mathbb{Z}'} : \ \theta_{g,\ell}.start \geq 0 \ \ if \ \ \theta_{k,\ell} \text{ has no predecessor} \quad (3)$$

where $\mathbb{Z}'$ is the conflict nested resource set.

Constraint 3 (Non-preemption Constraint)**.** *The execution of computation segments on the machines is non-preemptive in order to enforce exclusive execution of the critical sections and to respect the sequential execution order of the non-critical sections of task τ_i on machine m_{Z+i}. That is, if a computation segment $\theta_{i,j}$ with WCET $C_{i,j}$ is scheduled at time t_0, the finishing time of $\theta_{i,j}$ has to be $t_0 + C_{i,j}$. Then the time interval $[t_0, t_0 + C_{i,j}]$ is appended to the corresponding machine, i.e.,*

$$\forall m_{k \in \mathbb{Z}} : \ \theta_{i,j}.finish = \theta_{i,j}.start + C_{i,j}^{m_k} \quad (4)$$

where $C_{i,j}^{m_k}$ is the execution time of $\theta_{i,j}$ on machine m_k (which is not defined directly in $\theta_{i,j}$, but can be calculated by using the information in $\sigma_{i,j}$).

Constraint 4 (All-at-once Constraint)**.** *All resources that are requested within a critical section must be assigned to their corresponding machines at the same time, execute for the same amount of time, and finish at the same time. This means that for the construction of the dependency graph a nested locking scheme is transferred into all-at-once locking. That is, the time interval $[t_0, t_0 + C_{i,j}]$ of $\theta_{i,j}$ is appended to all the machines representing for these resources that the segment requests, i.e.,*

$$\forall m_{k \in \mathbb{Z}'} : \ if \ \theta_{i,j}.start = t_0, \ then \ \theta_{i,j}.finish = t_0 + C_{i,j} \quad (5)$$

where the $\mathbb{Z}'$ is the set of resources that $\theta_{i,j}$ requests. This constraint is similar to gang scheduling [32], i.e., multiple processors are requested simultaneously for a single task.

Thereby, the dependency graph construction problem is formulated as a constraint programming problem by generating a feasible schedule for a frame-based task set on $Z+n$ machines with respect to Constraint 1 to Constraints 4. The optimization objective is to minimize the makespan for the generated schedule, i.e., the latest finishing time of any job on any machine. This is formulated to minimize $\max_{i,j} \theta_{i,j}.finish$. Since the generated schedule is non-preemptive, the computation segments on each machine are executed sequentially. The initial dependency graph G, which is given by the tasks internal precedence constraints, is refined by the execution order of the critical sections given by the order of computation segments on the respective machines.

B. Step II: Dependency Graph Scheduling

In the second step, the dependency graph G is scheduled on M processors using either global or partitioned and either preemptive or non-preemptive scheduling. Namely, *LIST-EDF* [37] or its partitioned extension (*P-EDF*) [38] can be applied. We assume that all tasks release task instances, called

jobs, strictly periodically, i.e., if a job of τ_i is released at time t_0 the subsequent job is released exactly at time $t_0 + T_i$. Note that the description in this subsection allows general periodic tasks which all release a job at time 0 and that frame-based tasks are a special case where all tasks have the same period.

Each job consists of a set of sub-jobs, which are instances of corresponding computation segments. For both *LIST-EDF* and *P-EDF*, the deadlines for all sub-jobs are modified according to the generated dependency graph, i.e., the deadline of one sub-job is the minimum of all the successors' latest release times where the latest release time of a sub-job is defined by its deadline minus its WCET.

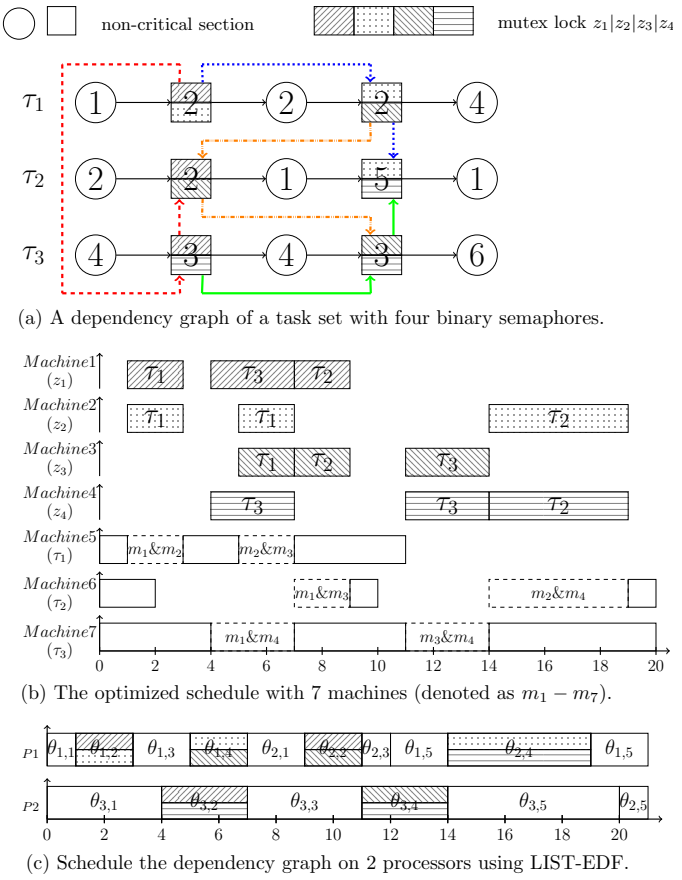
For all-at-once locking, the executions of the critical sections that access at least one identical resource are mutually exclusive. This means that if two computation segments have a (partial) overlap for the resources they request, i.e., at least one shared resource is requested by both of them, their critical sections on the same shared resource(s) must be executed one after another. Two critical sections from two tasks can be (partially) executed at the same time on different processors, if they do not request the same resource(s) during the overlapped execution. Meanwhile, for a *nested locking* strategy, a critical section can start its execution once it has locked all the currently requested resource(s), e.g., for a critical section with the resource access sequence $\sigma_{i,j} = ((1, \{z_1\}), (2, \{z_1, z_2\}), (1, \{z_1\}))$, the execution of the first part that only requests resource z_1 can be started once resource z_1 is locked, rather than waiting for both resource z_1 and z_2 to be locked. Please note, if resource z_2 is not available when the first time units of execution for resource z_1 has finished, the critical section will suspend itself and wait for the release of resource z_2. However during this waiting time, no other critical section is able to access resource z_1, since it is still locked by τ_i.

In addition, we enforce that for each job all computation segments execute for the duration of their WCET. That is, although a segment can have a shorter real execution time than its WCET, it has to spin at the corresponding processor until the time reaches to WCET. Therefore, the generated schedule for one frame/hyper-period is static and repeated periodically, which avoids the multiprocessor timing anomalies described by Graham [20], and simulating a *LIST-EDF* or *P-EDF* schedule over one frame/hyper-period, checking whether any deadline is missed, is an exact schedulability test.

C. An Illustrative Examples

Fig. 1 exemplifies our approach for a frame-based task set with three tasks and four shared resources, where all tasks have the same period of 30 time units. Each task consists of five segments, three non-critical sections (circles in Fig. 1 (a)) and two critical sections (rectangles). Each critical section requests two shared resources at the same time, i.e., with the *all-at-once* locking strategy. The execution of each of the critical sections is protected by the corresponding mutex locks.

The computation segments within one task have to be executed sequentially according to the predefined order (black



(a) A dependency graph of a task set with four binary semaphores.

(b) The optimized schedule with 7 machines (denoted as $m_1 - m_7$).

(c) Schedule the dependency graph on 2 processors using LIST-EDF.

Fig. 1. An example of the dependency graph based nested resource synchronization protocol.

solid arrows in Fig. 1 (a)). The numbers in the circles and rectangles denote the execution times of the corresponding computation segments.

To construct a dependency graph for the task set, we apply constraint programming as described in III-A, with $7 = 4 + 3$ exclusively assigned machines: machine 1 to machine 4 are executing the critical sections of the four shared resources, and machine 5 to machine 7 are for the non-critical sections of tasks τ_1 to τ_3. Hence, once a task needs to access the shared resource, the execution is migrated to the corresponding machines, e.g., to machine 1 for resource z_1. Since *all-at-once* locking is applied, each critical section requests two shared resources at the same time. That is, a critical section is executed on multiple machines (representing the shared resources) at the same time and without any overlap with the execution of other critical sections using the same machine.

Fig.1 (b) shows an optimized schedule for the given task set. We note that the schedule in Fig. 1 (b) is only used to generate the dependency graph. The execution order for critical sections of tasks τ_1, τ_2, and τ_3 on machine 1 to 4 determines the precedence constraints in the generated dependency graph with respect to all the critical sections in Fig. 1 (a), where the precedence constraints are represented by dashed red arrows for mutex lock z_1, dotted blue arrows for mutex lock z_2,

densely dashed and dotted orange arrows for mutex lock z_3, and the solid green arrows for mutex lock z_4.

The concrete schedule on two processors for LIST-EDF [37] is shown in Fig. 1 (c).

IV. Properties of Our Approach

In this section, we prove properties of our graph based approach to schedule task sets with nested resource-sharing. Specifically, we prove that our approach is deadlock-free, free of transitive blocking, and has bounded approximation factors.

A. Deadlock-Free

Whenever nested resource-requests are considered, the possibility of deadlocks is a major concern. A simple way to avoid deadlocks is to specify an order for all available resources, and to require that nested locks are acquired according to this determined order and therefore avoid *circular waiting*, but it is not obvious how to determine this access order. Alternatively, dynamic group locks (DGLs) [44], where a super set of the actual requested resources by a critical section are requested simultaneously, break the *hold-and-wait* condition. However, the proposed DGA is by design deadlock free and avoids transitive blocking as shown in this subsection.

We first introduce the required notation. We use

$$pre(v_i) : \{v_j \in V | (v_j, v_i) \in E\} \text{ and } v_j \prec v_i \text{ if } v_j \in pre(v_i)$$
$$suc(v_i) : \{v_j \in V | (v_i, v_j) \in E\} \text{ and } v_j \succ v_i \text{ if } v_j \in suc(v_i)$$

to denote precedence constraints and paths in a given DAG.

Definition 4. ***Path:*** *A path Δ in a directed-acyclic graph G is any sequence of sub-jobs $v_{i_1} \prec v_{i_2} \prec \ldots \prec v_{i_k}$ for $v_{i_j} \in V$ such that each sub-job in the sequence is an immediate successor of the previous sub-job in terms of precedence constraints and $pre(v_{i_1}) = \emptyset$ and $suc(v_{i_k}) = \emptyset$.*

Based on the definition of a path we can describe the longest path in a DAG called critical path more formally stated in the following definition.

Definition 5. ***Length:*** *Then the length of a path is given by $\ell en(\Delta) := \sum_{v_i \in \Delta} \ell en(v_i)$ where the length of a sub-job denotes its execution time. Subsequently, the length of DAG G is given by $\ell en(G) := \max\{\ell en(\Delta) \mid \Delta \text{ is a path in } G\}$.*

Definition 6. ***Volume:*** *The volume of the DAG G is given by the graph's cumulative execution time, that is, $vol(G) := \sum_{v_i \in V} \ell en(v_i)$.*

First, we show that the dependency graph G constructed in Section III-A has no cycles.

Theorem 1. *The generated dependency graph G that respects the all-at-once locking constraint for nested resources in critical sections from Section III-A is a directed acyclic graph.*

Proof. Before the optimization by the constraint programming, each task τ_i is given by a chain that is composed of the computation segments $\theta_{i,1} \prec \theta_{i,2} \prec \ldots \prec \theta_{i,\eta_i}$ and hence does not contain any cycles. Let any two critical sections of different tasks, e.g., $\theta_{i,j}$ and $\theta_{g,\ell}$, have conflicting nested resource requests, i.e., a subset of resources $\mathbb{Z}' \in \mathcal{P}(\mathbb{Z})$ is requested in both critical sections. By assuming any generated feasible graph G must respect the constraints $1-4$, if any resource of $\theta_{i,j}$ is granted, then all resources in $\mathbb{Z}'$ are granted to that critical section as well. By the non-preemption constraint (Constraint 3) the resources are held until the completion of that critical section. In consequence, for any two critical sections that have conflicting nested resource requests, either $\theta_{i,j} \prec \theta_{g,\ell}$ or $\theta_{g,\ell} \prec \theta_{i,j}$ must hold. Since the internal order inside a task is respected by the generation as well, the generated dependency graph does not contain any cycles. □

Theorem 2. *Any schedule on M processors that respects the precedence constraints of the dependency graph G as described in Section III-A is deadlock-free even if resources in critical sections are locked as soon as they are required, i.e., when not enforcing all-at-once locking at run-time.*

Proof. We disprove the possibility of *circular waiting* in any schedule that respects the precedence constraints in G by contradiction. Let S be a schedule that respects the precedence constraints in G and at let τ_i and τ_j be in the state of *circular waiting* at some point in time. This implies that task τ_i holds at least one resource z and waits for at least one other resource z' which is held by task τ_j, which in return waits for resource z (held by task τ_i). However, this means that there exist critical sections, e.g., $\theta_{j,k}$ and $\theta_{i,l}$, that are in conflict. By Theorem 1, we know that the set of conflicting critical sections is ordered, i.e., $\theta_{i,l} \prec \theta_{j,k}$ or $\theta_{j,k} \prec \theta_{i,l}$. Therefore if S respects the precedence constraints then any resource from the critical section $\theta_{j,k}$ could not have been scheduled before all resources used in $\theta_{i,j}$ have finished execution. That is, the circular waiting implies a violation of constraints given by G, which contradicts the assumption. □

B. No Transitive-Blocking

In addition to being deadlock-free, the proposed approach also avoids transitive blocking.

Theorem 3. *Any schedule on M processors that respects the precedence constraints of the dependency graph G with nested resources sharing as described in Section III-A breaks the transitive blocking chain.*

Proof. Since the makespan minimization of the constraint program only generates precedence constraints for conflicting critical sections, computation segments that do not conflict can be executed in-parallel by a schedule that respects the precedence constraints. □

C. Approximation Factor

In this section we prove that our algorithm has a bounded approximation factor for any variant of list-scheduling when a dependency graph with a bounded approximation factor α compared to an optimal dependency graph is given. We formally define a dependency graph with approximation factor α as follows.

Definition 7. *A dependency graph G is an α-approximation of a dependency graph G' if for some $\alpha \geq 1$ the following constraints are satisfied:*

- $vol(G) = vol(G')$
- $\ell en(G) = \alpha \cdot \ell en(G')$

The optimization quality is determined related to an optimal dependency graph, which is a dependency graph with minimum length.

Theorem 4. *LIST-EDF of an α-approximated optimal dependency graph G^* is an $(1+(1-\frac{1}{M})\cdot\alpha)$ approximation algorithm for frame-based task sets that use all-at-once locking to access critical sections with nested resources.*

Proof. Let G be the dependency graph that α-approximates an optimal dependency graph G^*. By the property of list scheduling, the makespan $L(G)$ on M processors is at most

$$L(G) \leq \frac{vol(G)}{M} + (1 - \frac{1}{M}) \cdot \alpha \cdot \ell en(G^*) \tag{6}$$

By the fact that an optimal makespan can be no shorter than the length of the longest path and the perfectly distributed workload, we know that $L(G^*) \geq \max\{vol(G^*)/M, \ell en(G^*)\}$. In conclusion,

$$L(G) \leq L(G^*) \cdot (1 + (1 - \frac{1}{M}) \cdot \alpha)$$

□

Corollary 1. *The Dependency Graph Approach provides an $(1+\alpha)$-approximation for frame-based task sets that use all-at-once locking to access critical sections with nested resources.*

V. Extension to Normal Locking Patterns

In this section, we extend the dependency graph approach to nested resource sharing with normal locking patterns, where each critical section can request each resource at most once.

Definition 8. ***Normal Locking Pattern:*** *A critical section is said to lock nested resources with a* normal locking pattern*, if the critical section requests a resource only when it is needed. Hence, not all resources are necessarily locked when the execution of the critical section starts.*

It is well known that the normal locking pattern can potentially lead to a deadlock, once two tasks request two shared resources in reversed order, since then both the *hold-and-wait* as well as the *circular waiting* condition are fulfilled. To prevent this, a new constraint is designed to replace the Constraint 4 in Section III-A:

Constraint 5 (Pattern-respect Constraint)**.** *For any computation segment (representing for critical section) with normal locking pattern, the locking pattern has to be respected. For a computation segment $\theta_{i,j} = (c_{i,j}, \sigma_{i,j})$, where $\sigma_{i,j} = a_1, a_2, \ldots a_\ell$, we have to determine stating times and finishing times for all tuples. We explain this explicitly for the first and second tuple. The starting time on machines in the first tuple is $\theta_{i,j}.start = t_0\ \forall m_{k\in a_1^1}$; the starting time of $\theta_{i,j}$ on machines that are in the second tuple but not in the first tuple is $\theta_{i,j}.start = t_0 + a_1^0\ \forall m_{k\in(a_2^1-(a_1^1\cap a_2^1))}$; and the finishing time of $\theta_{i,j}$ on machines that are occupied in the first tuple but not occupied any more in second tuple is $\theta_{i,j}.finish = t_0 + a_1^0\ \forall m_{k\in(a_1^1-(a_1^1\cap a_2^1))}$. Such a calculation is applied for all the tuples in $\sigma_{i,j}$.*

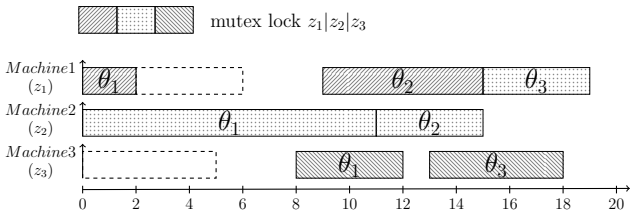


Fig. 2. An example of the dependency graph that with normal locking pattern.

Example 2. *A computation segment given by $\theta_{i,j} = (4, ((1, \{1\}), (2, \{1, 2\}), (1, \{2\})))$, the pattern is respected if $\theta_{i,j}$ is scheduled on machine m_1 at time t_0 (by locking of resource 1) and scheduled on machine m_2 (by locking resource 2) at time $t_0 + 1$. Combined with the Non-preemption constraint, $\theta_{i,j}$ has to finish its execution at time $t_0 + 3$ on machine m_1 (by releasing resource 1), at time $t_0 + 4$ on machine m_2 (by releasing resource 2).*

Theorem 5. *The generated dependency graph G for nested resources with the normal locking pattern in critical sections that respects Constraints 1, 2, 3, and 5 is a DAG.*

Proof. For each task, the computation segments are still chained, i.e, $\theta_{i,1} \prec \theta_{i,2} \prec \ldots \prec \theta_{i,\eta_i}$, and hence the graph does not contain any cycles. Let any two critical sections of different tasks, e.g., $\theta_{i,j}$ and $\theta_{g,\ell}$, have conflicting nested resource requests. That is, a subset of resources $\mathbb{Z}' \in \mathcal{P}(\mathbb{Z})$ is requested in both critical sections. On any two machines, e.g., m_a and m_b, that are requested by $\theta_{i,j}$ or $\theta_{g,\ell}$ simultaneously, the execution time (occupation time) of $\theta_{i,j}$ on two machines has an overlap, i.e., $(\theta_{i,j}^{m_a}.finish - \theta_{i,j}^{m_a}.start) \cap (\theta_{i,j}^{m_b}.finish - \theta_{i,j}^{m_b}.start) \neq \emptyset$ (the same for $\theta_{g,\ell}$), since otherwise it is not a nested resource access. The overlap on both machines for each computation segment can be treated as an *all-at-once* lock. Therefore the execution order of these two segments on both machines are unified. Combined with the non-overlap constraint and non-preemption constraint, the extra execution of each computation segment on both machines follows the same order as the overlapped parts. Therefore, no cycle is included during the generation of graph(s). □

Example 3. *Consider the following task set with 3 shared resources and 3 computation segments:*
$\theta_1 = (12, ((2, \{1, 2\}), (6, \{2\}), (3, \{2, 3\}), (1, \{3\})))$,
$\theta_2 = (6, ((2, \{1\}), (4, \{1, 2\})))$, *and*
$\theta_3 = (6, ((2, \{3\}), (3, \{1, 3\}), (1, \{1\})))$.
A feasible schedule with respect to the constraints 1, 2, 3, and

5 *is shown in Fig. 2. Due to Constraint* 5*, although machine* m_1 *is already free at time* 2*,* θ_2 *starts its execution at time* 9*, in order to start its execution on machine* m_2 *at time* 11 *(since* θ_2 *on machine* m_2 *has to start its execution* 2 *time units after it starts the execution on machine* m_1*).*

Please note, the optimal schedule in Fig. 2 is that θ_2 is schedule on the dashed slots on machine m_1 and m_3. Although θ_1 requested 3 resources, it does not request resource 1 and 3 at the same time. Therefore, θ_1 is not considered as accessing nested resource 1 and 3. Hence θ_3 has no nested conflict with θ_1, and its executions on machine m_1 and m_3 do not necessarily follow the same precedence constraints with regard to the whole computation segment.

VI. Extension to Periodic Task Systems

A similar treatment as presented in [37] can be applied to extend our approach from frame-based task systems to periodic task systems. First, a job-level dependency graph is constructed after unrolling all jobs of all tasks that are released in one hyper-period (where the hyper-period is the lowest common multiple (LCM) of all the periods in a task set). The precedence constraints are generated for each job by applying the method described in Section III-A. Afterwards, a feasible schedule for the generated job-level dependency graph needs to satisfy all precedence constraints. Since all jobs of each task must execute in sequence to fulfill their constrained deadline, only one machine for each task is needed. In comparison to the approach for frame-based task systems, two additional modifications are needed for periodic task systems:

1) For the ℓ-th job fo τ_i we set its release time to $(\ell-1)\cdot T_i$ and its absolute deadline to $(\ell-1)\cdot T_i + D_i$.
2) In periodic task systems, minimizing the makespan is not a useful objective since there is no connection between a job's deadline and the makespan. Instead, the objective of the constraint programming is to minimize the maximum lateness over all jobs, where the lateness of a job is its finishing time minus its deadline.

The above extension can be applied to any periodic real-time task system, however the space cost for storing all unrolled jobs and the computational costs for examining all unrolled jobs in a hyper-period limit the practical applicability of this approach.

VII. Evaluation

We numerically evaluated the performance of the proposed approach for a wide range of different configurations. The hardware platform used in our experiments is a cache- coherent SMP, consisting of two 64-bit AMD EPYC 7742 64-Core Processors running at 1.5 GHz, with 256 GB of main memory.

A. Evaluation Setup

We conducted evaluations for M = 4, 8, and 16 processors. Based on the value of M, we generate randomized task sets with $10 \times M$ tasks each. The utilization of each task τ_i is denoted as $U_i = \frac{C_i}{T_i}$, hence the execution time for each task is $C_i = U_i \times T_i$. We generated synthetic task sets with total utilization level, i.e., $\sum_{\tau_i \in \mathbf{T}} U_i$, from 0 to $100\% \times M$ in steps of $5\% \times M$ by applying the RandomFixedSum method [15], enforcing that $U_i \leq 0.5$ for each task τ_i. The number of shared resources (binary semaphores) Z was either 4, 8, or 16.

The task sets that we generated are either frame-based or periodic. For the frame-based task sets, all the tasks share the same period and relative deadline, i.e., $\forall \tau_i : T_i = D_i = 1$. For the periodic task sets, the task periods T_i are selected 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 [21], [25], [35], [39], [41]. We used a small range of periods to generate reasonable task sets with high utilization of the critical sections. These are otherwise by default not schedulable, since critical sections would be longer than the smallest period. Other configurations that related to nested resource accesses are as follows:

- The nested depth d is in $[2, 4]$, i.e., each critical section can access at most 2 or 4 resources at the same time. Each task τ_i contains at most 5 critical sections.
- The nested probability q, i.e., the percentage of nested resource accesses over all critical sections in a task set is chosen from $\{10\%, 20\%, 50\%, 80\%\}$.
- The total length of the critical sections is a fraction of the total execution time C_i of task τ_i, depended on $H \in \{[0.5\% - 1\%], [1\% - 5\%], [5\% - 10\%], [10\% - 40\%], [40\% - 50\%]\}$. The total length of the critical sections and non-critical sections are split into dedicated segments by applying UUniFast [15] separately.

In many traditional real-time applications, the utilization of nested resource accesses are relative low, i.e., less than 10%. However, when GPUs are included as accelerators (for example in machine learning applications with real-time constraints like autonomous driving systems), the utilization of nested resource accesses become relative high. Therefore, a range of $[40\% - 50\%]$ is added in our settings, and the gap, i.e., $[10\% - 40\%]$, is filled as well.

100 randomized tasks sets for each of the 21 utilization steps were generated for each of the 360 configurations (i.e., all possible combinations of the M, Z, d, q, and H values) for frame-based and periodic task systems.

For our graph-based approach, the dependency graph is generated by applying the method presented in Section III. Google OR-Tools [1] is utilized to solve the constraint programming problem. We name the considered algorithms as follows:

1) *LIST-EDF/P-EDF*: To schedule the generated graph, we used the LIST-EDF [37] or partitioned EDF (P-EDF), which is a partitioned extension of LIST-EDF and a *worst-fit* partitioning algorithm *w.r.t.* task's utilization U_i is applied [38]. Both methods modify the deadlines according to the rule introduced by Baker et al. [3].
2) *P/NP*: preemptive or non-preemptive for critical sections.

In addition, the state-of-the-art methods are evaluated, namely the Concurrency Group Locking Protocol (CGLP) [31]

and the Uniform Contention-sensitive Real-time Nested Locking Protocol (C-RNLP) [24].

- CGLP-G: CGLP [31] where concurrency groups are assigned by a greedy algorithm.
- CGLP-N: CGLP [31] where concurrency groups are assigned by minimizing the number of groups.
- UC-RNLP: C-RNLP [24], which grants resources accesses to sets of requests, and the sets are determined dynamically during the run time.
- GC-RNLP: General C-RNLP [24], that grants resources accesses contention-sensitively on a per-request basis.

Please note, although the aforementioned methods support concurrent *read* resource accesses as well, only mutually exclusive *write* resource accesses are evaluated.

B. Evaluation Results for Frame-Based Task Systems

Only a subset of the results is presented here, as the other results show similar trends. The evaluation results in Fig. 3 show that our graph based approach outperforms the state-of-the-art significantly for frame-based task sets. The required time to solve the constraint programming to generate the DAG (step 1 of our approach) highly depends on the configuration. More precisely, it took 0.5, 5.8, 1.3, 2.2, 1.6, and 1420 CPU hours respectively to solve the constraint programming for the configurations shown in sub-figures of Fig. 3.

The existing methods (denoted by brown and orange lines) can only handle the situations where the utilization of critical sections with nested resource requests is extremely low, i.e., $[0.5\% - 1\%]$ in Fig. 3 (c) and (d). When the utilization of critical sections increases (Fig. 3 (b), and (e)), their performance (w.r.t the schedulability) degrades a lot and they do not work at all when the utilization of critical section is $[40\% - 50\%]$ (Fig. 3 (f)). The results also show that when the number of processors and available shared resources increase at the same time (Fig. 3 (a), and (b)), the performance of existing methods degrades while our new methods can still work well. In addition, when increasing only the nested depth (Fig. 3 (c), and (d)) or when increasing both nested depth and possibility at the same time (Fig. 3 (b), and (e)) without changing other configurations does not obviously affect the performance of these methods. Furthermore, when the utilization of nested critical sections is extremely high (Fig. 3 (f)), the performance of both methods scheduled by P-EDF degrades as well. Contrarily, since the execution order for all critical sections has been optimized offline by minimizing the makespan, our approach with LIST-EDF works quiet well even when a high utilization of critical sections is considered.

C. Evaluation Results for Periodic Task Systems

A subset of the evaluation results for periodic task systems is shown in Fig. 4. The constraint programming in step 1 of our approach took 1.3, 7, 2.5, 110, 5.3, and 3.3 CPU hours for the configurations in these sub-figures of Fig. 4 respectively. Our graph based approach still outperforms the state-of-the-art methods in all the evaluated configurations. The results show that when the utilization of critical sections

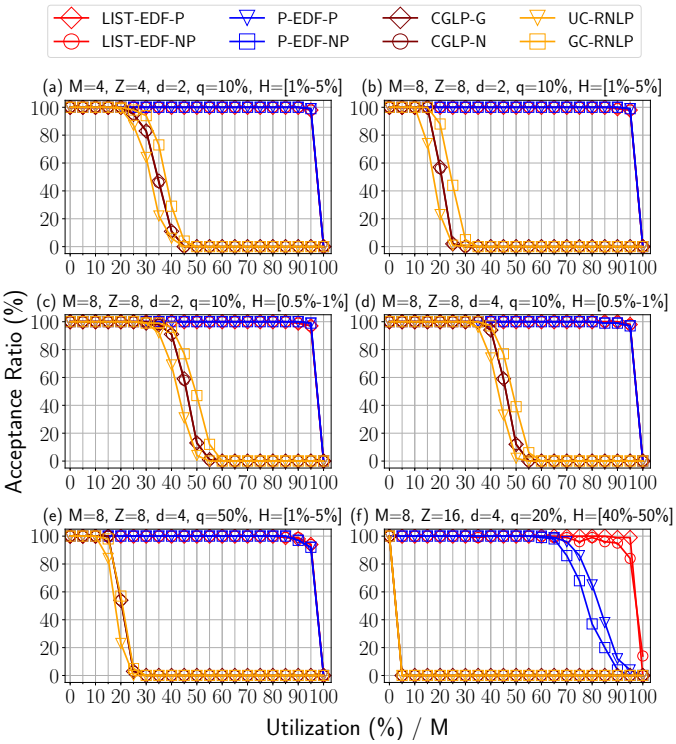


Fig. 3. Schedulability of different approaches for frame-based task sets.

is increased (Fig. 4 (a) to (b)), the performance of existing methods decreases a lot, while the performance of our new proposed approach has not been affected. The increasing of both the number of processor and of available shared resources does not affect the performance of our methods but slightly degrades the performance of existing methods (Fig. 4 (a) to (c)). All existing methods cannot handle the situation when the utilization of critical sections is extremely high, i.e., up to $[40\% - 50\%]$ in Fig. 4 (d), while our newly proposed methods still provides a reasonable acceptance ratio. Fig. 4 (e) and (f) show that increasing the number of shared resources without modifying other configuration does not affect the performance of all the evaluated methods significantly. In addition, the performance of preemptive and non-preemptive scheduling algorithms trends to be similar, since the optimized dependency graph and the pre-calculated deadline for each computational segment reduced the potential preemption due to earlier deadlines.

D. Summary

The evaluation results show that our proposed approach is highly effective for frame-based and periodic real-time tasks. However, our approach is limited to periodic task systems with sequential and segmented accesses to critical and non-critical sections, while the methods in [31] and [24] are applicable for sporadic task systems and not limited to sequential access patterns as defined in Section II. Therefore, the schedulability tests for existing methods are pessimistic for both frame based

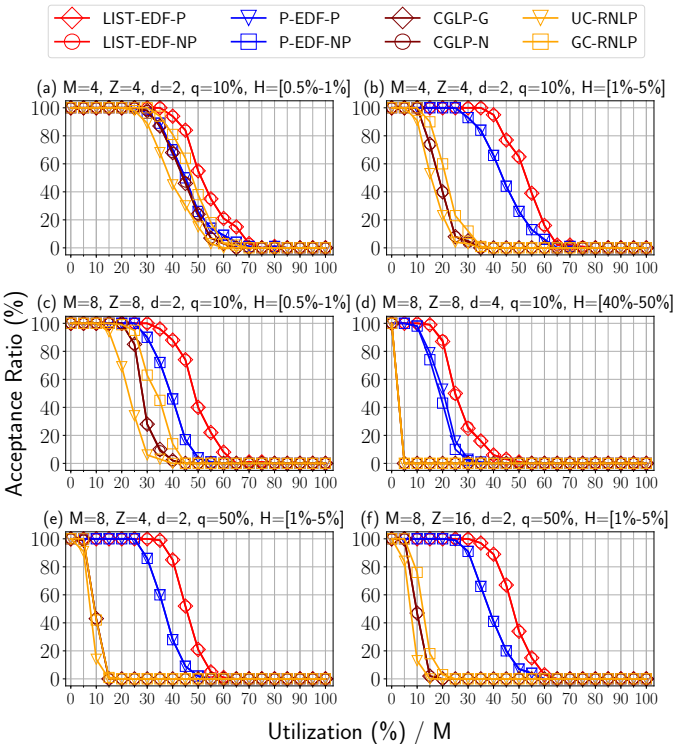


Fig. 4. Schedulability of different approaches for periodic task sets.

and periodic task sets, which explains the performance of the state-of-the-art methods in our evaluation.

Our extension for periodic task systems requires to unroll all jobs that are released within the hyper-period, which results in high space demand. Additionally, the constraint programming is used to construct the dependency graph(s) based on all unrolled jobs, which is time consuming and limits the scalability and flexibility of our proposed approach. The methods in [31] and [24] do not have such limitations, and are more applicable in more general systems. When the task set has sporadic activation patterns, branches or with a more general system model, then our approach cannot be utilized.

However, our approach shows significant performance improvements for periodic task sets with segmented execution and sequential resource access patterns as studied in this paper. Note that the recent empirical study by Akesson et al. [2] shows that

> *While periodic activation is the most common at 82%, over 60% of systems included aperiodic activations [2].*

Therefore, having good solutions for periodic activations is useful for industrial practice. Hence, in the future, finding more efficient DAG generation strategies and adding aperiodic activations together with periodic tasks will further improve the practicability of our approach.

VIII. Related Work

In multiprocessor real-time systems, three scheduling paradigms are widely adopted, i.e., *global*, *partitioned*, and *semi-partitioned* scheduling. Under *global* scheduling, tasks are dynamically dispatched among all the available processors, where tasks are allowed to migrate among processors freely. Under *partitioned* scheduling, each task is assigned on a dedicated processor and migration is not allowed, i.e., all jobs of a task must be executed on the same processor. Under *semi-partitioned* scheduling, each task is split into several computation segments, and each computation segment is partitioned statically on a processor. To prevent race condition and reduce priority inversion, a large number of multiprocessor resource synchronization and locking protocols have been developed and analyzed in the past decades. Many of them are extensions of the well known uni-processor protocols, e.g., the Priority Inheritance Protocol (PIP) [36], the Priority Ceiling Protocol (PCP) [36], and the Stack Resource Policy (SRP) [4]. A comprehensive survey of multiprocessor real-time locking protocols can be found in [7].

Rajkummar et al. [34] proposed Distributed-PCP (DPCP), where each resource is assigned on a processor statically, called the resources synchronization processor. A task has to be migrated to the dedicated synchronization processor for the execution of critical section, and critical sections are executed on the a synchronization processor by following the uni-processor PCP. DPCP applies *semi-partitioned* scheduling. The extension Multiprocessor PCP (MPCP) [33] allows tasks to execute their critical section locally. In order to minimize the usage of stack memory in real-time systems, Gai et al. [18] proposed Multiprocessor SRP. Both MPCP and MSRP apply *partitioned* scheduling. Block et al. [6] introduced Flexible Multiprocessor Locking Protocol (FMLP), where resources are divided into two groups, i.e., long and short. For short resources, critical sections are executed in a non-preemptable manner and tasks are spinning on their processors while waiting for resources. For long resources, tasks suspend themselves into a FIFO queue while waiting. FMLP is also the first protocol that supports both *global* and *partitioned* scheduling. Easwaran and Brandenburg [14] introduced Parallel PCP (P-PCP), considering global fixed priority preemptive multiprocessor systems. Brandenburg and Anderson [8] proposed $O(m)$ Locking Protocol (OMLP), which ensures $O(m)$ maximum pi-blocking for any task set and supports both *global* and *partitioned* scheduling. Burns et al. [10] proposed the Multiprocessor resource sharing Protocol (MrsP), that allows tasks help other tasks during spinning cycles, and *(semi-)partitioned* scheduling is applied.

Since the performance of these protocols highly depends on how the tasks are partitioned, several partitioning algorithms were developed, e.g., by Lakshmanan et al. [26] and Nemati et al. [28] for MPCP, by Wieder and Brandenburg [45] for MSRP, by Hsiu et al. [22], Huang et. al [23], and von der Brüggen et al. [40] for DPCP.

More recently, Chen et al. [13] introduced the Dependency Graph Approach (DGA), where dependency graphs are constructed in the first step and scheduled on multiprocessors either partitioned or globally in the second step. DGA was further extended to periodic task systems by Shi et al. [37],

and partitioning algorithm was developed by Shi et. al [38].

However, only a few of these protocols support nested resource sharing in a fine-grained manner[1]. The first protocol that supports nested resource sharing is DPCP, since uni-processor PCP is applied on synchronization processors. Once nested resources are assigned on the same processor, the nested resource sharing is supported by uni-processor PCP by default. Chen et al. [11] developed MDPCP for periodic task systems by carefully defining the inter-processor ceilings. Besides, the Multiprocessor BandWidth Inheritance protocol (M-BWI) [16], [17] and MrsP [10], [19] allow nested resource accesses without deadlocks if all the resources or mutex locks are accessed according to a specified total order. The family of Real-time Nested Locking Protocols (RNLP) [43], [42], [24], [29], [30] support nested resource sharing with different variants by considering: 1) different waiting mechanisms, i.e., suspension or spinning, 2) different progress mechanisms, i.e., priority boosting [27], [8], priority inheritance [36], and priority donation [9], and 3) how pi-blocking is analyzed. However most of them do not handle the *transitive blocking chain problem*. That is, the traditional first in first out (FIFO) resource accessing order can result in a single request blocking a chain of requests even though some of them have no conflict of requested resources. Only C-RNLP [24] breaks the transitive blocking chains for nested write requests by applying a *cutting ahead* mechanism, where the lengths of critical sections are taken into consideration for lock the and unlock logic. Dynamic group locks (DGLs), where all resources in the corresponding group that the nested request belongs to are requested simultaneously when starting a critical section, also breaks the *hold-and-wait* condition [44]. Moreover, a fine-grained blocking bound for nested non-preemptive FIFO spin locks under P-FP scheduling is presented in [5]. The analysis is based on a graph abstraction that reflects all possible resource conflicts and transitive delays. As the state-of-the-art, the newly proposed Concurrency Group Locking Protocol (CGLP) by Nemitz et al. [31] supports lock nesting using *group* locking. In addition, concurrency groups are utilized to break transitive blocking, where a concurrency group is a group of lock requests that can safely execute together.

IX. Conclusion

We propose a graph based approach to synchronize nested resource requests on multiprocessor real-time systems. We show the feasibility as well as that our approach is deadlock free, transitive blocking free, and has bounded approximation factors. The evaluation results in Sec. VII show that our approach significantly improves the schedulability for both frame-based and periodic task systems with the relatively high space cost and computational cost, compared to the state-of-the art, in all of the evaluated configurations.

[1]Nested resource sharing can be supported by using a coarse-grained group lock for all of these protocols.

Acknowledgment

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

References

[1] Google OR-Tools. https://developers.google.com/optimization/. visit on 15.11.2020.

[2] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (Proceedings)*, 2020.

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

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

[5] A. Biondi, B. B. Brandenburg, and A. Wieder. A blocking bound for nested FIFO spin locks. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*, pages 291–302. IEEE Computer Society, 2016.

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

[7] B. B. Brandenburg. Multiprocessor real-time locking protocols: A systematic review. *CoRR*, abs/1909.09600, 2019.

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

[9] B. B. Brandenburg and J. H. Anderson. Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks. In *EMSOFT*, pages 69–78. ACM, 2011.

[10] A. Burns and A. J. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *ECRTS*, pages 282–291, 2013.

[11] C. Chen, S. K. Tripathi, and A. Blackmore. A resource synchronization protocol for multiprocessor real-time systems. In *ICPP (3)*, pages 159–162. CRC Press, 1994.

[12] J.-J. Chen, J. Shi, G. von der Brüggen, and N. Ueter. Scheduling of real-time tasks with multiple critical sections in multiprocessor systems. *IEEE Transactions on Computers*, (01):1–1, 2020.

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

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

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

[16] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *ECRTS*, pages 90–99, 2010.

[17] D. Faggioli, G. Lipari, and T. Cucinotta. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems*, 48(6):789–825, 2012.

[18] 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.

[19] J. Garrido, S. Zhao, A. Burns, and A. Wellings. Supporting nested resources in mrsp. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 73–86. Springer, 2017.

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

[21] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst. Communication centric design in complex automotive embedded systems. In *29th Euromicro Conference on Real-Time Systems, ECRTS*, 2017.

[22] 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.

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

[24] C. E. Jarrett, B. C. Ward, and J. H. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In J. Forget, editor, *Proceedings of the 23rd RTNS*. ACM, 2015.

[25] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmark for free. In *6th International Workshop WATERS*, 2015.
[26] 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.
[27] K. Lakshmanan, D. De Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, pages 469–478, 2009.
[28] 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.
[29] C. E. Nemitz, T. Amert, and J. H. Anderson. Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts (artifact). *Dagstuhl Artifacts Ser.*, 4(2):02:1–02:3, 2018.
[30] C. E. Nemitz, T. Amert, and J. H. Anderson. Real-time multiprocessor locks with nesting: optimizing the common case. *Real Time Syst.*, 55(2):296–348, 2019.
[31] C. E. Nemitz, T. Amert, M. Goyal, and J. H. Anderson. Concurrency groups: a new way to look at real-time multiprocessor lock nesting. In J. Ermont, Y. Song, and C. Gill, editors, *Proceedings of the 27th RTNS*, pages 187–197. ACM, 2019.
[32] J. K. Ousterhout et al. Scheduling techniques for concurrent systems. In *ICDCS*, volume 82, pages 22–30, 1982.
[33] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings.,10th International Conference on Distributed Computing Systems*, pages 116 – 123, 1990.
[34] 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.
[35] A. Sailer, S. Schmidhuber, M. Deubzer, M. Alfranseder, M. Mucha, and J. Mottok. Optimizing the task allocation step for multi-core processors within AUTOSAR. In *2013 International Conference on Applied Electronics*, pages 1–6, Sept 2013.
[36] 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.
[37] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen. Multiprocessor synchronization of periodic real-time tasks using dependency graphs. In *Proceedings of the RTAS*, pages 279–292, 2019.
[38] 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 RTCSA*, 2019.
[39] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein. System-level timing feasibility test for cyber-physical automotive systems. In *SIES*, pages 1–10, May 2016.
[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 *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, pages 287–296, 2017.
[41] G. von der Brüggen, N. Ueter, J.-J. Chen, and M. Freier. Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems. In *Proceedings of the 25th RTNS*, pages 108–117, 2017.
[42] B. C. Ward and J. H. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*.
[43] B. C. Ward and J. H. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS*, pages 223–232, 2012.
[44] B. C. Ward and J. H. Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In M. Auguin, R. de Simone, R. I. Davis, and E. Grolleau, editors, *21st RTNS*, pages 67–76. ACM, 2013.
[45] 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.