Partitioned Scheduling for Dependency Graphs in Multiprocessor Real-Time Systems

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

***Abstract*—Effectively handling precedence constraints and resource synchronization is a challenging problem in the era of multiprocessor systems even with massively parallel computation power. One common approach is to apply list scheduling to a given task graph with precedence constraints. However, in some application scenarios, such as the OpenMP task model and multiprocessor partitioned scheduling for resource synchronization using binary semaphores, several operations can be forced to be *tied* to the same processor, which invalidates the list scheduling. This paper studies a special case of this challenging scheduling problem, where a task comprised of (at most) three subtasks is executed sequentially on the same processor and the second subtasks of the tasks may have sequential dependencies, e.g., due to synchronization. We demonstrate the limits of existing algorithms and provide effective heuristics considering preemptive execution. The evaluation results show a significant improvement, compared to the existing multiprocessor partitioned scheduling strategies.**

I. Introduction

In real-time systems, tasks are typically assumed to release an infinite number of task instances, called jobs, under a given inter-arrival constraint. When concurrent real-time tasks have to access shared resources, e.g., external devices, shared memory, or files, ensuring the timeliness is a challenging problem, since the resource access is often the bottleneck. Such a resource access is typically called a *critical section* of a task. To prevent race conditions and data corruption, mutual exclusion is enforced, i.e., once a task has been granted access to a shared resource, no other task is allowed to access the same resource until it is released by the resource holding task. One possible solution to achieve mutual exclusion is to use system programming (e.g., based on the pthread library) and *semaphores* offered by operating systems. However, especially for nested resource accesses, utilizing semaphores introduces additional problems, e.g., deadlocks and unbounded priority inversions, that can render any timing analysis infeasible. To avoid these problems and to allow the verification of hard real-time constraints for multiprocessor systems, multiple resource synchronization protocols have been proposed, analyzed, and evaluated, for instance, the Distributed Priority Ceiling Protocol (DPCP) [28], the Multiprocessor Priority Ceiling Protocol (MPCP) [27], the Multiprocessor Stack Resource Policy (MSRP) [10], and the Flexible Multiprocessor Locking Protocol (FMLP) [5]. These protocols typically focus on managing the resources' access under a given situation and provide real-time guarantees by bounding the resulting worst-case response time of tasks. However, the performance of these protocols is highly dependent on settings such as task partitions, priority assignments, local or global execution of shared resources, and the tasks' waiting semantics (self-suspension or spinning). Regardless, only a small number of studies evaluate how these settings can be optimized for a given protocol, e.g., [1], [8], [36]. Which resource-sharing protocols are applicable depends on the underlying multiprocessor scheduling paradigm:

- A *partitioned* schedule allocates a task to a processor and for each job of the task all subjobs of that job, i.e., critical and non-critical sections, are executed on that processor.
- A *global* schedule maintains a single ready queue and allows tasks to migrate among processors at any time.
- A *semi-partitioned* schedule divides a task into several sub-tasks, and then assigns these subtasks individually, e.g., it may assign the non-critical section(s) on one processor and the critical section(s) on a different processor.

However, since resource access rather than the processor capacity is often the bottleneck for timing guarantees, a focus onto the shared resources seems reasonable [15]. Recently, approaches that construct a dependency graph have been proposed. They calculate precedence constraints for tasks that share the same resource offline and schedule these graphs online, i.e., at run-time. While Chen et al. in [7] consider *frame-based* real-time task sets, Shi et al. [30] extended this method to periodic real-time task sets. Both methods consider various algorithms to calculate the order in which tasks/jobs access a given resource, resulting in a directed-acyclic graph (DAG) representation of the subtasks. Therefore, dependency graph approaches (DGAs) are not work-conserving regarding the resource access, since the order in which a shared resource is accessed is determined by the preconstructed DAG and not by the moment when a critical section becomes eligible for execution. Contrary to other approaches, this allows DGAs to schedule periodic task sets where the longest critical section of a resource is larger than the shortest period of a task that accesses the same resource.

Both Chen et al. [7] and Shi et al. [30] use global scheduling as the underlying scheduling paradigm. One main reason to exploit global scheduling is the assumption that it leads to a higher overall system utilization then the other paradigms which justifies the additional scheduling overhead. However, a comprehensive experimental study by Brandenburg et al. [6] demonstrates that the performance of global scheduling algorithms can be matched by using semi-partitioned scheduling algorithms. Furthermore, Biondi et al. [4] show that the current response-time analysis for global scheduling algorithms is

978-1-7281-3197-9/19/$31.00 ©2019 IEEE

inherently pessimistic due to imprecise interference accounting and thus not able to achieve a higher schedulability compared to partitioned scheduling algorithms given real-time constraints. Therefore, we believe that good partitioned scheduling algorithms for DGAs are beneficial in the design of verifiable and practically good systems, especially in the automotive or aerospace domain.

Therefore, we consider the task partitioning problem for the dependency graph approach with the additional constraint that each task is *tied* to one processor. The models studied in this paper fit the dependency graph approach for binary semaphores [7], [30] and the task synchronization model under OpenMP [25]. Note however that our algorithms are not restricted to only these two cases, but can be extended to any other application that shares similar characteristics.

Contributions: We detail how the directed acyclic graphs resulting from the dependency graph approach for multiprocessor resource sharing of strictly periodic constrained-deadline task sets can be scheduled using partitioned earliest deadline first. Our detailed contributions are as follows:

- We describe how the schedulability of preconstructed dependency graphs for periodic real-time tasks can be determined for a given task partition in Section IV.
- We propose two partitioning algorithms, one based on federated scheduling and one based on a worst-fit heuristic, for dependency graphs of periodic task sets on homogeneous multiprocessor systems in Section V.
- We evaluate the performances of the proposed algorithms based on randomly generated task sets under different configurations in Section VII. The results show that one of the proposed algorithms outperform existing partitioned approaches and perform reasonably compared to global List-EDF for both periodic and frame-based task sets.

II. Related Work

Most task partitioning problems for multiprocessor platforms are NP-hard (in the strong sense), even if the dependency of tasks (due to shared resources) is not considered. Hence, algorithms to construct optimal solutions are computationally infeasible and it is mandatory to design heuristics or approximations to find a good partition efficiently. To that end, several heuristic partitioning algorithms have been proposed. Here, we restrict ourselves on partitioning-heuristics designed for the context of multiprocessor resource synchronization.

Lakshmanan et al. [18] presented a synchronization-aware partitioned heuristic for MPCP, which classifies the tasks that request the same resource into one group and tries to assign each group of tasks to one processor. Following the same principle, Nemati et al. [24] developed a blocking-aware partitioning method that uses an advanced cost heuristic algorithm to split a task group when the entire group is not able to be assigned on one processor. After that, Hsiu et al. [14] proposed a dedicated-core framework to separate the execution of critical sections and non-critical sections, and a priority-based mechanism is applied for resource sharing. In this method, each request for a shared resource can be blocked by at most one lower-priority task's request. Furthermore, Wieder and Brandenburg [36] proposed a greedy slacker partitioning heuristic in the presence of spin-based resource synchronizations. Huang et. al [15] proposed the *resource-oriented partitioned* (ROP) scheduling which was later refined by von der Brüggen et al. [33] with release enforcement for a special case. The ROP-based algorithms rely on task migrations and uniprocessor locking protocols, where the tasks requesting a specific resource must migrate to the related synchronization processor. The evaluations in [15] and [33] showed that ROP-based algorithms are the state of the art for sporadic real-time task systems when each task uses at most one binary semaphore.

Recently, Sun et al. [31] have proposed the heuristic algorithm BFS* as an extension of Breadth First Scheduling (BFS) [20] for task sets with precedence constraints when analyzing OpenMP [25] systems by first executing currently tied tasks. They assume that the critical section of a task always takes place at the end of the task, and that the tasks that request the same resource are chained. The performance of their approach highly depends on the execution order of ready tasks which request different resources.

Moreover, a heuristic that improves the robustness and parallel resource usage is presented in [7], where all first non-critical sections are scheduled before the execution of any critical section by applying list scheduling. To minimize the idle time of each processor, it allows preemption for the non-critical sections but disallows preemption for the critical sections to trivially guarantee sequentialized resource accesses.

III. System Model

The problem studied in this paper is identical to our earlier paper [30]. We consider a set $\mathbf{T}$ of n recurrent tasks to be scheduled on M identical (homogeneous) processors. All tasks have exactly one (non-nested) critical section where they access exactly one of the z shared resources in the system, and are each described by $\tau_i = ((C_{i,1},\ A_{i,1},\ C_{i,2}),\ T_i,\ D_i)$, where:

- $C_{i,1}$ is the worst-case execution time (WCET) of the first non-critical section of the job.
- $A_{i,1}$ is the WCET of the critical section of the job, accessing a dedicated shared resource.
- $C_{i,2}$ is the WCET of the second non-critical section.
- T_i is the period of τ_i.
- D_i is the relative deadline. We consider a constrained deadline task system, i.e., $\forall \tau_i \in \mathbf{T},\ D_i \leq T_i$.

All tasks release an infinite number of task instances, called jobs, strictly periodically, i.e., if a job of τ_i is released at time t the subsequent job is released exactly at time $t + T_i$, and the first instance of all tasks is released at time 0. In order to fulfill the timing requirements, a job of τ_i released at time t must finish its execution before its absolute deadline $t + D_i$. For each shared resource s, one dependency graph is given, denoted as $G_s = (V_s, E_s)$. Each subjob of a task accessing s is a vertex in V_s, the subjob $C_{i,1}^{\ell}$ is a predecessor of the subjob $A_{i,1}^{\ell}$, and $A_{i,1}^{\ell}$ is a predecessor of the subjob $C_{i,2}^{\ell}$, where ℓ

indicates the ℓ-th job of task τ_i. If ℓ_i-th job of τ_i and ℓ_j-th job of τ_j share the same resource, then either the subjob $A_{i,1}^{\ell_i}$ is the predecessor of $A_{j,1}^{\ell_j}$ or the subjob $A_{j,1}^{\ell_j}$ is the predecessor of $A_{i,1}^{\ell_i}$. All critical sections guarded by the same resource form a chain in G_s, i.e., the critical sections of the resource have to be executed sequentially in a predefined order. Figure 1 provides an example for a dependency graph for periodic tasks with one binary semaphore. Since there are z shared resources in the system, the complete the dependency graph G has in total z independent subgraphs, denoted as $G_1, G_2, \ldots, G_z$.

For each task τ_i, the total utilization is defined as $U_{\tau_i} = \frac{C_{i,1}+C_{i,2}+A_{i,1}}{T_i}$. For the individual sub-tasks, the corresponding utilization is defined as $U_{C_{i,1}} = \frac{C_{i,1}}{T_i}$, $U_{A_{i,1}} = \frac{A_{i,1}}{T_i}$, and $U_{C_{i,2}} = \frac{C_{i,2}}{T_i}$. We further assume that:

- For each task τ_i in $\mathbf{T}$, $C_{i,1} \geq 0$, $A_{i,1} \geq 0$, and $C_{i,2} \geq 0$.
- The execution of subjobs within one job must fulfill the task's precedence constraints, i.e., a job must be sequentially executed in the order of $C_{i,1}, A_{i,1}, C_{i,2}$.
- Critical sections of a shared resource must be sequentially executed and mutually exclusive, i.e., if two tasks share the same resource, their critical sections have to be executed without any overlap.

Accordingly, the utilization for each graph U_{G_s} is defined as the summation of the utilizations of all the tasks in this graph, i.e., $U_{G_s} = \sum U_{\tau_i}$ for all $\tau_i \in G_s$.

We consider constrained-deadline task systems, a task is feasibly schedulable if its worst-case response time is no more than its deadline. The hyper-period H of the task set $\mathbf{T}$ is defined as the least common multiple (LCM) of the periods of all tasks in $\mathbf{T}$. Additionally, we also define the hyper-period for each subgraph as $H_s = LCM(T_i, \forall \tau_i \in G_s)$. In our approach, we unroll the jobs in one hyper-period and design a schedule for all of them. To make sure that the time and space complexity is affordable, we assume that the task set has one of the following properties:

- *Harmonic Periods*: T_i is an integer multiple of T_j if $T_i \geq T_j$ for any two tasks τ_i and τ_j in $\mathbf{T}$.
- *Semi-Harmonic Periods*: For all tasks $\tau_i \in \mathbf{T}$ there is a small integer value n_i, such that $T_i \cdot n_i = H$.

One prime example for task sets with semi-harmonic periods are automotive applications where the periods of the tasks are in $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ ms [13], [17], [29], [32], [35]. Note that our methods can still be applied to any periodic real-time task systems at the cost of higher complexity if the hyper-period is large compared to the task periods. Details can be found in Section VI-B.

We will also evaluate a special case of periodic task sets, namely the *frame-based* real-time task system where all tasks share the same period and release all their jobs simultaneously.

Please note that we focus on how to schedule given dependency graphs under partitioned scheduling. How such a graph can be constructed is explained in [30]. Furthermore, strict periodicity is enforced since the approach in [30] only works for periodic tasks and how to handle sporadic task sets with the dependency graph approach considered in [7], [30] remains an open problem.

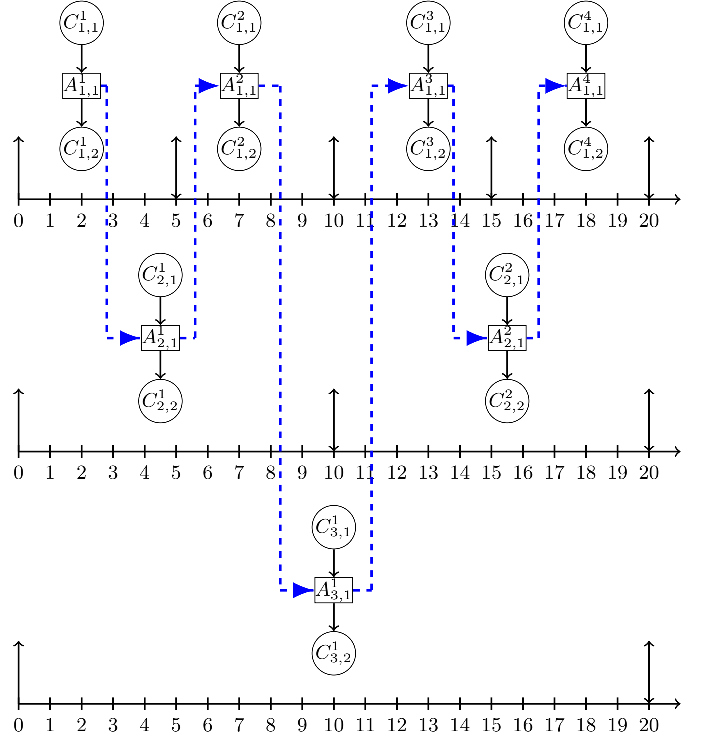

Fig. 1. The dependency graph for the three periodic tasks detailed in Table I that access resource 1 over one hyper-period.

IV. Partitioned Earliest Deadline First

In this section, we describe and exemplify how to schedule the unrolled dependency graphs of the periodic tasks over their hyper-period $H = LCM(H_1, H_2, \ldots, H_z)$ by using Partitioned Earliest Deadline First (P-EDF). Hence, we first shortly summarize the construction of the dependency graph in Shi et al. [30] and explain the deadline setting for P-EDF in Section IV-A. We note that Section IV-A is presented merely to improve the clarify and completeness of the paper. Afterwards, we provide a detailed example in Table I and Figure 2 which is explained in Section IV-B.

A. P-EDF for Dependency Graphs

Each job J_i^ℓ has three subjobs denoted as $J_{i,1}^\ell, J_{i,2}^\ell, J_{i,3}^\ell$ that represent the related subtasks $C_{i,1}, A_{i,1}$, and $C_{i,2}$ respectively. Furthermore, we use $r_{i,j}^\ell$ to denote the release time of the subjob $J_{i,j}^\ell$ and $d_{i,j}^\ell$ to denote this job's absolute deadline. For the l-th job, the release time of the first subjob $r_{i,1}^\ell$ is given by $(\ell-1) \cdot T_i$, and the absolute deadline of the last subjob $d_{i,3}^\ell$ is given by $(\ell-1) \cdot T_i + D_i$, since an constrained-deadline system is considered.

In an initial step, the release times of the second and third subjob as well as absolute deadlines of the first and second subjob are set based on the precedence constraints imposed by the task. To be precise, the release time of the second subjob $r_{i,2}^\ell$ is set to $(\ell-1) \cdot T_i + C_{i,1}$, the release time of the third subjob $r_{i,3}^\ell$ is set to $(\ell-1) \cdot T_i + C_{i,1} + A_{i,1}$, the absolute deadline of the second subjob $d_{i,2}^\ell$ is set to $(\ell-1) \cdot T_i + D_i - C_{i,2}$, and the absolute time of the first subjob $d_{i,1}^\ell$

is set to $(\ell-1)\cdot T_i + D_i - C_{i,2} - A_{i,1}$. This means that during the construction of the dependency graphs, we assume that all non-critical sections can be executed as soon as they are released without considering if sufficient processors are available to schedule all available subjobs.

The earliest possible release time of the second subjob (which accesses the critical section), i.e., $r_{i,2}^{\ell}$, is bounded by the earliest possible finishing time of its predecessor, i.e., the release time of the first subjob plus its WCET $C_{i,1}$. The earliest possible release time of the third subjob i.e., the second non-critical section, equals to the earliest possible release time of the second subjob plus its WCET $A_{i,1}$. These earliest possible release times are used in the dependency graph calculation, hence assuming that there are always sufficient processors to execute all non-critical sections in the system. The reason is that not considering the combined workload of the non-critical sections when constructing the dependency graph allows us to construct the graphs for different resources individually and to use well-known algorithms for uniprocessor non-preemptive scheduling. The actual release times of the second and third subjobs depend on the schedule at runtime, i.e., on the finishing times of the predecessor(s) on the corresponding processors. However, as long as the precedence constraints imposed by the constructed DAGs are respected, the actual release times of the subjobs do not have to be known beforehand by the scheduler. Hence, it is not necessary to precalculate the exact release times.

Contrarily, the deadlines for the first and second subjobs, i.e., the first non-critical section and the critical section, are necessary for both the dependency graph construction and the scheduling decision of the partitioned EDF scheduler. Again, the deadlines used in the dependency graph construction only depend of the deadline of the job and the WCET of the second non-critical section. Afterwards, the deadlines used by the scheduler are determined according to the corresponding dependency graph. To be precise, if the absolute deadline of an immediate predecessor of $J_{i,j}^{\ell}$, denoted as $IPre(J_{i,j}^{\ell})$ with WCET C_{pre}, is larger than $d_{i,j}^{\ell} - C_{pre}$, the absolute deadline of the immediate predecessor is reassigned to $d_{i,j}^{\ell} - C_{pre}$. This is a standard procedure for scheduling jobs subject to release dates and precedence constraints. Details can be found in [3]. For the rest of this paper, we assume that the absolute deadline assignment is adjusted accordingly.

We assume that each dependency graph $\mathbf{G}_s$ for a binary semaphore s is constructed for the corresponding jobs released (strictly) within one hyper-period H. If $H_s < H$, then $\frac{H}{H_s}$ copies of $\mathbf{G}_s$ are applied in a consecutive order to represent the precedence constraints of the critical sections, where H_s is the hyper-period of the tasks in the corresponding dependency graph $\mathbf{G}_s$.

After the construction of $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_z$, the scheduling problem becomes a classical multiprocessor scheduling problem. Once the tasks partition is given, tasks are executed on the assigned processor using EDF with the modified deadlines, i.e., whenever the processor is idle and there are subjobs eligible to be executed, the one with the earliest deadline is executed on that processor. The P-EDF in our setting is a preemptive algorithm, i.e., whenever a new (eligible) subjob has an earlier absolute deadline than an executing subjob on the corresponding processor m, the new subjob can preempt the one that is executing on that processor. Such flexibility to allow preemption does not create any problem for the mutual-exclusive constraint of the critical sections guarded by one binary semaphore s, because their execution order has been predefined in the dependency graph $\mathbf{G}_s$, i.e., only when the critical section of the predecessor finishes its execution, the successor can release its critical section if the first non-critical section of the successor also has finished its execution. Therefore, a critical section guarded by a semaphore s can only be preempted by either non-critical sections or by critical sections guarded by other semaphores.

In addition, we define that if two or more subjobs have the same deadline on the assigned processor, the subjob with the larger remaining execution time is scheduled.

B. An Example for P-EDF

To further explain the work flow, we provide an example to demonstrate how our algorithm works. We consider a task set consisting of five tasks with two shared resources as defined in Table I. The tasks τ_1, τ_2, and τ_3 are requesting resource 1, and the dependency graph is constructed based on Figure 1, which results in the following order for the executions of the associated critical sections: $J_{1,2}^1 \rightarrow J_{2,2}^1 \rightarrow J_{1,2}^2 \rightarrow J_{3,2}^1 \rightarrow J_{1,2}^3 \rightarrow J_{2,2}^2 \rightarrow J_{1,2}^4$. Furthermore, τ_4 and τ_5 are requesting resource 2, and the execution order for the critical sections is defined as $J_{4,2}^1 \rightarrow J_{5,2}^1 \rightarrow J_{4,2}^2$. These five tasks are scheduled on $M=2$ processors, the partition is defined by applying the worst-fit based algorithm in Section V-B, i.e., τ_3 and τ_4 are assigned to processor 1, and τ_1, τ_2, and τ_5 are assigned to processor 2. Afterwards, the P-EDF is applied to schedule the tasks accordingly.

Based on the earlier explanation, the earliest possible release times and deadlines for the individual subjobs can be determined for all releases of all subjobs of all tasks as displayed in Table I. Regarding the release times, we only detail the earliest possible release time of the critical sections and of the second non-critical sections resulting from the dependency graph, assuming no early completion of jobs. The release times and deadlines of the critical sections are changed based on the order and the resulting restrictions in the dependency graph (colored red), which propagates to later releases of the second non-critical section or an earlier deadline of the first non-critical section (colored blue) in Table I. Note that the actual release times may be later, depending on the actual schedule.

All first subjobs, i.e., first non-critical sections, are released exactly periodically. For all other subjobs, the release times may be adjusted based on the earliest times their predecessors may finish. This is done in a forward manner, i.e., starting from time 0 up until time H, i.e., 20 in Table I. Hence, $J_{2,2}^1$ is released at time 0.8 (marked red in Table I) due to the earliest possible finishing time of $J_{1,2}^1$ at 0.8. Therefore, the release of $J_{2,3}^1$ is postponed as well (marked blue here and for all other third subjobs that are postponed). The second subjob of

Task	WCETs			Other Parameters				Release Times			Deadlines		
	$C_{i,1}$	$A_{i,1}$	$C_{i,2}$	$T_i = D_i$	ϕ_i	$s(\tau_i)$	ℓ	$J^\ell_{i,1}$	$J^\ell_{i,2}$	$J^\ell_{i,3}$	$J^\ell_{i,1}$	$J^\ell_{i,2}$	$J^\ell_{i,3}$
τ_1	0.2	0.6	0.2	5	0	1	1	0	0.2	0.8	4.2	4.8	5
							2	5	5.2	5.8	5.6	6.2	10
							3	10	13.8	14.4	14.2	14.8	15
							4	15	15.2	15.8	19.2	19.8	20
τ_2	0.2	0.6	3.7	10	0	1	1	0	0.8	1.4	5	5.6	10
							2	10	14.4	15	15.7	16.3	20
τ_3	4	8	5	20	0	1	1	0	5.8	13.8	6.2	14.2	20
τ_4	0.3	0.4	0.3	10	0	2	1	0	0.3	0.7	9.3	9.7	10
							2	10	10.3	10.7	19.3	19.7	20
τ_5	2	2	2	20	0	2	1	0	2	4	16	18	20

TABLE I
THE INFORMATION OF TASK SET FOR REQUESTING 2 RESOURCES ALONG WITH THE (EARLIEST POSSIBLE) RELEASE TIMES AND THE DEADLINES

the second release of τ_1 can finish no earlier than at time 5.8, hence the release of $J^1_{3,2}$ is postponed accordingly. Due to the long critical section of task τ_3, the releases of $J^3_{1,2}$ and $J^2_{2,2}$ are postponed to time 13.8 and 14.4 as well.

The deadlines in Table I are constructed in a backward manner, i.e., from the end of the hyper-period to the beginning. Here, all third subjobs have a deadline identical to the end of the related period. The deadlines of the second subjobs are based on the dependency graph. For $J^4_{1,2}, J^2_{2,2}, J^3_{1,2}$, and $J^1_{1,2}$ the deadlines directly result from that job's third subjob. However, the short deadline of $J^3_{1,2}$, leads to an earlier deadline of $J^1_{3,2}$ (14.2 instead of 15), which also leads to the deadline of $J^2_{1,2}$ (6.2 instead of 9.8), which again leads to the deadline of 5.6 (instead of 6.3) for $J^1_{2,2}$. Additionally, the changed deadlines for the second subjobs are marked red while the resulting adjustment for the first subjobs is marked blue in Table I. Regarding the tasks τ_4 and τ_5 (that access resource 2) shown in Table I as well, since no deadlines or release times are adjusted, details are omitted.

The schedule based on partitioned EDF is displayed in Figure 2 and considers the deadlines provided in Table I. Execution on processor 1 is marked blue while execution on processor 2 is marked red. In addition, the access to the critical sections related to resource 1 and resource 2 are shown with different hatching patterns as detailed in Figure 2. Recall, that we assume that the subjob with the larger remaining workload is preferred in the scheduling decision if two tasks with the same deadline compete for a processor.

Due to the high utilization of τ_3, only two tasks, i.e., τ_3 and τ_4 are assigned on processor 1. And the rest three tasks, i.e., τ_1, τ_2, and τ_5 are assigned on processor 2. There are several properties in our P-EDF schedule, which can be observed in the example:

- Partitioned: All the jobs of one task are assigned to the same processor, i.e., one row only has one color.
- Earliest Deadline First: At time 0 the subjobs $J^1_{3,1}$ and $J^1_{1,1}$ are scheduled due to their deadlines of 6.2 and 4.2, which are earlier than for the other subjobs released on the related processors.
- Preemptive schedule: when $J^2_{1,1}$ is released at time 5, it preempts the execution of $J^1_{2,3}$, due to the earlier deadline ($d^2_{1,1}$ is 5.6 and $d^1_{2,3}$ is 10).
- Critical sections can be preempted by non-critical sections or critical sections of other resources: at time 10 the critical section of resource 2 of task τ_5 is preempted by the non-critical section of the third job of τ_1.
- Larger remaining execution time first: After $J^2_{1,2}$ finished its execution, $J^1_{2,3}$ resumes on processor 1. Although $J^1_{2,3}$ has the same deadline as $J^2_{1,3}$, it has a larger remaining execution time, thus $J^1_{2,3}$ has higher priority to be executed here.
- Precedence constraints: At time 5, processor 1 is idle and $J^1_{3,1}$ has finished its execution, in a work-conserving schedule, $J^1_{3,2}$ would start the execution of its critical section. However, due to the precedence constraints in Fig1, $J^1_{3,2}$ cannot be executed until $J^2_{1,2}$ has finished its critical section for resource 1 at time 5.8.

The provided example shows that our proposed approach is able to schedule a task set with the total utilization of 190% on two processors. This utilization can even be increased slightly while ensuring the schedulability, by increasing the WCET for the second non-critical section of τ_1 and τ_4, i.e., $C_{1,2} = 0.3$ and $C_{4,2} = 0.5$, resulting in a total utilization of 194%.

When comparing Figure 2 and Table I, the difference between the actual released times and the minimum release times considered in the dependency graph construction becomes clear. For instance, in the actual schedule in Figure 2, $J^1_{4,2}$ and $J^1_{5,2}$ the second subjobs of the first job of τ_4 and τ_5 are released much later than at time 0.3 and 2, which are the earliest possible release times in Table I.

V. PARTITIONING ALGORITHMS FOR DEPENDENCY GRAPHS

When considering partitioned scheduling for dependency graphs, all subtasks of τ_i are tied to the same processor. Hence, once the first subtask, i.e., the first non-critical section $C_{i,1}$, has been assigned to one processor, the remaining two subtasks $A_{i,1}$ and $C_{i,2}$ will be executed on that processor as well. In

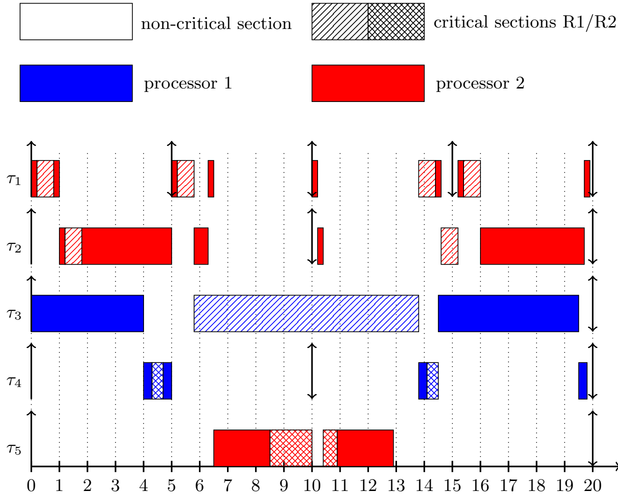

Fig. 2. An example of P-EDF with two shared resources.

addition, once the first job of a task has been assigned on one processor, all the following jobs of this task are also assigned to that processor. Typically, in a partitioned scheduling tasks are partitioned one by one in some predefined order. Since the the dependency graph for the periodic task set is constructed individually for each resource, the jobs of these tasks share some precedence constrains. Therefore, it seems reasonable to consider the possibility to partition all task that access the same resource, and therefore subgraphs, together rather than considering tasks individually. Once the partition of a task set has been defined, tasks on each processor will be scheduled by applying Partitioned Earliest Deadline First (P-EDF), which has been explained in Section IV.

In this section, two partitioning algorithms for dependency graphs resulting from periodic task sets are proposed. One is based on federated scheduling [19], i.e., tasks are partitioned graph by graph, and another is based on global worst-fit heuristic, i.e., all the tasks are partitioned one by one.

A. Federated Based Partitioning Algorithm

Federated scheduling was proposed by Li et al. [19] in order to schedule parallel real-time task systems with internal precedence constraints that can be modeled as a directed-acyclic graph (DAG). The foremost intention of this scheduling algorithm is to provide provably good approximations with respect to an optimal scheduling algorithm whilst considering implementation constraints, e.g., cache hit-rates and memory accesses during runtime. The idea of Federated scheduling is to assign DAGs, in our case the DAGs resulting from the dependency graph construction, that need to utilize more than one processor, so called *heavy* graphs, to those processors exclusively. Analogously, the graphs that can be feasibly scheduled on a single processor are denoted as *light* graphs and are scheduled jointly on the remaining processors, i.e., non-exclusively allocated processors. After this initial partition, the actual scheduling is done by a work-conserving scheduler on the assigned processors. Our Federated scheduling heuristic for DGAs is shown in Alg. 1.

Algorithm 1 Federated Based Partitioning Algorithm

Input: Task set $\mathbf{T}$, dependency graph $G(\mathbf{T})$, number of processors M, number of resources z, the total utilization for each graph U_z;

1: Initialize: Schedule $S_s \leftarrow \emptyset$ for each graph, Heavy graphs $G_H \leftarrow \emptyset$, Light graphs $G_L \leftarrow \emptyset$, Partition P_z for each graph, Available processor $M_a \leftarrow M$;
2: Divide the graphs from $G(\mathbf{T})$ to either G_H or G_L
3: Sort the tasks in each G_s decreasingly *w.r.t* utilizations;
4: **for** all $G_h \in G_H$ **do**
5: $\quad m_h \leftarrow \lceil U_{G_h} \rceil$;
6: $\quad$ Initialize the temporary schedule $S'_h \leftarrow \emptyset$ for G_h;
7: $\quad$ **while** $m_h \leq M_a$ and S'_h is unschedulable; **do**
8: $\quad\quad$ Generate the P_h for G_h on m_h using worst-fit;
9: $\quad\quad$ Create S'_h based on P_h and m_h using P-EDF;
10: $\quad\quad$ **if** S'_h is unschedulable **then**
11: $\quad\quad\quad$ Assign one more processor to G_h: $m_h \leftarrow m_h + 1$;
12: $\quad\quad$ **else**
13: $\quad\quad\quad$ $M_a \leftarrow M_a - m_h$;
14: $\quad\quad$ **if** $m_h > M_a$ **then**
15: $\quad\quad\quad$ Return unschedulable;
16: **for** all $G_l \in G_L$ **do**
17: $\quad$ schedule light graphs using greedy algorithm;
18: Return task partition;

In the first stage, all graphs are categorized into either the set of *heavy* graphs, or the set of *light* graphs. All graphs with utilizations larger than 100% are *heavy* by default. For the remaining graphs, with utilizations less or equal to 100%, an EDF schedule is simulated in order to decide whether the graph is *light* or *heavy*, i.e., whether it can be feasibly scheduled on one processor or not. This test is necessary, since even for implicit-deadline tasks a total utilization of 100% may not be reachable on one processor, even for a very low resource utilization. For example, let two tasks $\tau_a = ((0, \varepsilon, 3), 6, 6)$ and $\tau_b = ((0, 6, 0), T_b, T_b)$ share the same resource, with $\varepsilon > 0$ but very small and T_b arbitrary large. This task set has a total utilization of $(50 + 2 \cdot \varepsilon)\%$ and a resource utilization of $(2 \cdot \varepsilon)\%$. However, due to the fact that a critical section cannot be preempted by a critical section for the same resource, at some point in time the critical section of τ_b must be scheduled between two critical sections of τ_a. This results in a total workload of $2 \cdot (\varepsilon + 3) + 6$, which is larger then 12 and, therefore, the critical section of τ_b is not schedulable in two consecutive periods of τ_a. Nevertheless, the task set can easily be scheduled by assigning both tasks to individual processors.

In each group, i.e., heavy and light, the graphs are sorted in non-increasing order with respect to the graph utilization, i.e., largest graph utilization first. For each *heavy* graph G_h, the minimal number of required processors, in order to feasibly schedule the graph, has to be determined (line 4-14 in Alg. 1). The initial number of processors m_h is given by the ceiling of the utilization of G_h. The tasks in G_h are partitioned on m_h processors based on the individual task utilization, using the worst-fit strategy. Once the partition is generated, P-EDF

is simulated to verify whether the m_h processors are sufficient to feasibly schedule G_h. In case of an infeasible schedule, the number of processors is incremented and the above procedure is repeated until either the generated schedule is feasible or the number of allocated processors exceeds the number of available processors. After feasibly assigning a G_h, the number of available processors is updated, i.e., the available processors for the following graphs is set to the number of currently available processors minus the number of processor necessary to schedule G_h.

Algorithm 2 Greedy Algorithm to Partition Light Graphs

Input: Set of light graphs $\mathbf{G_L}$ and number of remaining processors M_a;
1: Sort light graphs $\mathbf{G_L}$ in non-increasing order with respect to the graph's utilization;
2: Initialize: Partitions $P_1 \leftarrow \emptyset, P_2 \leftarrow \emptyset, \ldots, P_{M_a} \leftarrow \emptyset$;
3: **for** $i \leftarrow 1$ to M_a **do**
4:   **for each** graph G_ℓ in G_L **do**
5:     **if** $P_i \leftarrow P_i \cup G_\ell$ is not *EDF* schedulable **then**
6:       continue;
7:     **else**
8:       $P_i \leftarrow P_i \cup G_\ell$;
9:       $\mathbf{G_L} \leftarrow \mathbf{G_L} \setminus G_\ell$;
10: **if** All graphs are partitioned, i.e., $\mathbf{G_L}$ is empty; **then**
11:   **return** Task partition;
12: **else**
13:   **return** Infeasible

On the remaining processors, the greedy algorithm in Algo. 2 is applied to assign the light tasks in decreasingly order with respect to the graph utilization. Aiming to waste as little processor capacity as possible, we apply a best-fit strategy, where we first assign the graph in G_L with the largest utilization to a new processor. Afterwards, we traverse the remaining graphs in G_l and assign them to the same processor if possible (Line 4 - 9 in Algo. 2). Whether a graph can be assigned is determined by running EDF on the related processor. Thereafter, we remove all graphs that are assigned to the processor from G_L and continue with the next processor. This processes is repeated either until all light graphs are assigned to a processor or until no processor remains where the remaining tasks in G_L could be assigned.

If the graphs in both the *heavy* group and the *light* group can be scheduled feasibly, the corresponding partition is returned.

Note that the schedule repeats in each hyper-period if subtasks always run for their WCET. Therefore, in addition to running P-EDF online based on the partition, it is also possible to apply a time driven scheduler. In this case, the partitioning algorithm can also return the related schedule since we test the schedulability by running P-EDF.

B. Worst-Fit Heuristic

In addition, a worst-fit heuristic is proposed in Alg. 3, where the tasks are partitioned one by one. The tasks are first sorted according to a sorting strategy. After that, they are partitioned to the available processors using a worst-fit strategy, i.e., each task is assigned to the processor with the currently lowest utilization. Again, P-EDF is applied to verify whether the resulting partition on M processors is feasible.

We proposed two sorting strategies: 1) sort all the tasks decreasingly *w.r.t* the tasks' utilizations, no matter which resources they request. 2) sort the graphs decreasingly *w.r.t* the graph utilizations at first, then the tasks inside each graph are sorted decreasingly *w.r.t* the tasks' utilizations. In our proposed heuristic, both sorting strategies are applied. If the partition P generated by the first sorting strategy is not applicable, i.e., the task set is not schedulable on M processors based on current partition P using P-EDF, the second sorting strategy and the resulting partition P' are considered, and P-EDF is applied to verify the new partition P' once again. The algorithm only returns infeasible both aforementioned sorting strategies cannot generate a schedulable partition. Otherwise, the task set is schedulable and the partition is returned. Again, if a time driven schedule should be created the schedule can be returned as well.

Different to the federated scheduling, in this heuristic, tasks share the same resource may be partitioned on all the available processors, i.e., all the M processors may have some tasks share the same resource.

Algorithm 3 Worst-Fit Based Heuristic

Input: Task set $\mathbf{T}$, dependency graph $G(\mathbf{T})$, number of processors M;
1: Initialize: Partition P
2: Sort all the tasks in $\mathbf{T}$ decreasingly *w.r.t* the utilizations;
3: **for** all $\tau_i \in \mathbf{T}$ **do**
4:   Generate the P on M using worst-fit;
5: Create schedule S based on P and M using P-EDF;
6: **if** S is unschedulable **then**
7:   Sort the graphs in $G(\mathbf{T})$ decreasingly *w.r.t* to the U_z;
8:   Sort the tasks in G_z decreasingly *w.r.t* the utilizations;
9:   **for** all $\tau_i \in \mathbf{T}$ **do**
10:     Generate the P' on M using worst-fit;
11:   Create new S' based on P' and M using P-EDF;
12:   **if** S' is schedulable **then**
13:     Return task partition
14:   **else**
15:     Return infeasible;
16: **else**
17:   Return task partition

VI. Schedulability and Complexity

We discuss the exact simulation based schedulability test of our new proposed algorithms, especially with respect to multiprocessor timing anomalies. Afterwards, both the time and space complexity for these methods are discussed and the scalability to periodic task sets with arbitrary periods is discussed.

A. Schedulability Test and Multiprocessor Timing Anomalies

We perform an exact schedulability test by simulating the P-EDF schedule over one hyper-period, assuming the WCET

for each computation segment in this schedule. While the schedulability for this schedule can be easily verified, we have to ensure that analyzing this specific schedule is sufficient to guarantee schedulability in the general case.

The main concern here are the multiprocessor timing anomalies known in the real-time systems community, which were discovered by Graham [11] in 1969. To be precise, he showed that the response time of a task can be increased if either 1) precedence constraints are removed, 2) the number of processors is increased, or 3) the execution time of a job is reduced. As a result, a previously scheduled task set may become unschedulable under each of these three conditions. While we assume the number of processors to be given and the precedence constraints to be given (and fixed) before the schedule is determined, we do not prove that P-EDF does not have multiprocessor timing anomalies when a subjob finishes earlier than its worst case. Hence, if the execution time of the three subjobs of a certain task can change for different releases, the schedule determined under the assumption that all subjobs execute according to their WCET may not be the actual worst case under an online EDF scheduling algorithm, i.e., due to subjobs that are released earlier as a result of early completion another job may finish later.

However, the schedulability can be ensured if the same schedule is applied every hyper-period, i.e., the schedule is static. One option is to apply table-driving scheduling to ensure a repetitive schedule. Such a table can be determined by running P-EDF. Nonetheless, such a scheduling table may be large, especially when preemptive scheduling is considered. Alternatively, a static schedule can be achieved by enforcing the actual execution time of each subjob to be the same as its worst-case execution time, i.e., when no early completion is allowed. Since the EDF scheduling algorithm is deterministic, the scheduler will always take the same scheduling decisions every hyper-period and the schedule is always repeated if early completion is forbidden. We applied this solution in Section VII to evaluate the performances of the proposed algorithms. As a result, although we applied an online EDF version of our new method, the schedule is repeated for each hyper-period.

B. Complexity and Scalability

When applying P-EDF to the dependency graph approach for periodic tasks, we perform the following four steps:

1) Unroll the jobs J_i^ℓ for all the tasks τ_i in one complete hyper-period H.
2) Apply one of the non-preemptive uniprocessor scheduling techniques to generate the dependency graphs.
3) Select one of the proposed partitioning algorithms to assign the tasks on M processors.
4) Perform P-EDF to schedule these dependency graphs according the partition to determine the schedulability.

The space complexity of our method is $O(n_j)$, where n_j is the number of jobs in one hyper-period, since we have to unroll all jobs. For each resource s, let n_s be the number of jobs related to s in H_s, and let n_{max} be $max_{s\in\{1,\dots,z\}}(n_s)$.

We consider the time complexity step by step. The time complexity for unrolling the jobs to the hyper-period is $O(n_j)$ as well. In the second step, we applied a construction algorithm for each resource s based on non-preemptive uniprocessor scheduling techniques that were utilized in [7] and [30]. We detail the Extended Jacksons Rule as well as Potts algorithm here and consider the related time complexity. However, other approaches for non-preemptive uniprocessor scheduling may be applied as well. In this case, the time and space complexity depends on the time and space complexity of the applied algorithm. For each shared resource s, there are n_s jobs that are considered for one sub-graph by applying:

1) Extended Jacksons Rule (**JKS**) [16]: whenever the machine is free and one or more jobs are available for processing, schedule an available job with largest delivery time, which is $C_{i,2}$ for frame-based task sets and $H-(\ell-1)T_i-D_i+C_{i,2}$ for periodic task sets. JKS has a runtime $O(n_s^2)$ [16] for each resource, hence the total runtime is in $O(z\cdot n_{max}^2)$.
2) **POTTS** [26]: it further modifies the results from JKS by delaying the release of the interference jobs. A job J_a is called a interference job, if J_a is released earlier than J_b but has shorter delivery time. Then the release of job J_a will be delayed at the same time as J_b, and JKS is applied again, until there is no interference job or n_s iterations have been performed. The algorithm has a time complexity of $O(n_s^2 log\ n_s)$ for an individual resource according to [26], resulting in $O(z\cdot(n_{max} log\ n_{max}))$ for all resources.

Both provided partitioning algorithms cost $O(n\log n)$ time to sort the tasks' according to the utilization, where n is the number of tasks, and $O(nM)$ for the partition. Running partitioned EDF takes $O(n_j\log n_j)$ time.

Thus, our proposed methods have polynomial time complexity w.r.t. the number of jobs in one hyper-period, where the runtime is dominated by the amount of time needed to perform the construction of the dependency graph.

We note that, independent from the considered algorithm to construct the dependency graph, the time and space complexity of the approach is polynomial in the number of jobs in the hyper-period, which is exponential with respect to the number of tasks if periods are arbitrary. While the schedule itself can be created online, the dependency graph over one hyperperiod must be predefined in our approach. Hence, the graph must not only be calculated offline but must also be saved in a suitable data structure, e.g., as a list. The size of this data structure can be reduced since the schedule for a specific resource s repeats after H_s. It is also possible to use more complex data structures with compressing, e.g., by saving repeated subgraphs only ones. Regardless, periods in commercial embedded real-time systems are usually not arbitrary, but jobs have a relatively small sets of possible periods, e.g., the periods used in automotive systems are usually $\{1,2,5,10,20,50,100,200,1000\}$ [13], [17], [29], [32], [35]. If the hyper-period is small compared to the largest period in the system, as in the automotive case, a time and

space complexity that is polynomial in the number of jobs in the hyper-period is affordable.

VII. EVALUATIONS

In this section, we examine the performance of the proposed approaches by evaluations based on synthesized task sets for different configurations. Due to space limitations, we only present a subset of the conducted evaluations.

A. Evaluation Setup

We randomly generated task sets based on the number of processors M, shared resources z, and relative utilization of the critical sections β as parameters. In course of our evaluation, we considered $M \in \{4, 8, 16\}$, $z \in \{4, 8, 16\}$, and $\beta \in \{[5\% - 10\%], [10\% - 40\%], [40\% - 50\%]\}$.

For a given configuration of M, z, and β, we generated task sets with $10 \cdot M$ tasks for each total utilization value $\sum_{\tau_i \in \mathbf{T}} U_{\tau_i} \in [0, M]$ with a step 2%, applying the RandFixedSum method [9]. We enforced that $U_{\tau_i} \leq 0.5$ for each task τ_i. To determine the subtask utilization for $U_{C_{i,1}}$, $U_{C_{i,2}}$, and $U_{A_{i,1}}$, first the utilization of the critical section $U_{A_{i,1}}$ was decided by randomly drawing a percentage of the task's total utilization U_{τ_i} based on the parameter β. Afterwards, the remaining utilization U_{C_i} was split by drawing $U_{C_{i,1}}$ randomly uniform from $[0, U_{C_i}]$ and setting $U_{C_{i,2}}$ to $U_{C_i} - U_{C_{i,1}}$. The resource the critical section of a task access was drawn randomly uniform based on the number of resources. In addition, we generated two two kinds of task sets:

(1) Periodic task sets with semi-harmonic periods: The task periods T_i are selected randomly from a set of semi-harmonic periods, i.e., $T_i \in \{1, 2, 5, 10\}$, that is a subset of the periods used in automotive systems [13], [17], [29], [32], [35]. We used a small range of periods to generate reasonable task sets with high utilization of the critical sections. Otherwise, these task sets are by default not schedulable, i.e., at least one task has a critical section with a WCET ≥ 2. Hence, for the WCET values of the subtasks we get $C_{i,1} = U_{C_{i,1}} \times T_i$, $A_{i,1} = U_{A_{i,1}} \times T_i$, and $C_{i,2} = U_{C_{i,2}} \times T_i$.

(2) Frame-based task sets: A special case of periodic task sets, where all the tasks have the common period as 1. Hence, i.e., $C_{i,1} = U_{C_{i,1}}$, $A_{i,1} = U_{A_{i,1}}$, and $C_{i,2} = U_{C_{i,2}}$.

For each of these in total 54 configurations and each of the utilization step values, 1000 task sets were randomized according to the workflow detailed above.

B. Evaluated Approaches

When evaluation the heuristics presented in Section V, we applied the same approaches to construct the dependency graphs as used in [7], [30], since the algorithms to construct the dependency graphs are not the focus of this paper. For frame-based tasks, we applied the construction based on non-preemptive uniprocessor scheduling techniques that were utilized in [7], i.e., Extended Jacksons Rule **JKS** [16], **POTTS** [26], and **HS** [12].

For periodic task sets we applied **POTTS** [26] and Extended Jacksons Rule **JKS** [16] as proposed in [30]. Please note, that other techniques for non-preemptive uniprocessor scheduling can be applied as well, e.g., an iterative improvement algorithm by Hall and Shmoys [12], the Precautious-RM by Nasri et al. [21], [23], and the critical time window-based EDF scheduling policy (CW-EDF) by Nasri and Fohler [22].

The methods evaluated to schedule the tasks sets were:

- Two of our proposed methods: the algorithm based on federated scheduling in Alg. 1, denoted **FED-P-EDF**, and the algorithm based on global worst-fit partitioning in Alg. 3, denoted **WF-P-EDF**. We only display the results for dependency graphs construction using **POTTS** [26], since these approches outperformed the others.
- **LIST-EDF**: the List schedule based approach to schedule the dependency graphs [30]. Here, the deadlines for subjobs are redefined according to the precedence constraints at first, *List-EDF* is applied to schedule the subjobs accordingly. Furthermore, we modified the method when subjobs have the same deadline in order to improve the schedulability. Previously, the subjob with the longer rest execution time will be scheduled. However, in our new modification, once there is an interrupt, i.e., preemption due to the subjob with earlier deadline, the scheduled subjobs may be preempted by two cases: 1) new coming subjob has earlier deadline, or 2) another subjob in the ready queue has the same deadline but longer rest execution time. Such modification can highly improve the performance when this method is applied to *frame-based* task sets, since it balance the workload among processors in a finer grained manner.
- Resource Oriented Partitioned (ROP) scheduling with release enforcement by von der Brüggen et al. [33] which is designed to schedule periodic tasks with one critical section on a multiprocessor platform. The concept of ROP is to have a resource centric view instead of a processor centric view. The algorithm 1) binds the critical sections of the same resource to the same processor, thus enabling well known uniprocessor protocols like PCP to handle the synchronization, and 2) schedules the non-critical sections on the remaining processors using a state-of-the-art scheduler for segmented self-suspension tasks, namely SEIFDA [34]. Among the methods in [33], we evaluated **ROP-FP** (under fixed-priority) and **ROP-EDF** (under dynamic-priority), i.e., the best performing fixed-priority and the best performing dynamic priority method according to the evaluation in [33]. It has been shown in [33] that **ROP-EDF** dominates all existing methods suitable for sporadic real-time task systems.
- **LP-GFP-FMLP** [5]: a linear-programming-based (LP) analysis for global FP scheduling using the FMLP [5].
- **LP-GFP-PIP**: LP-based global FP scheduling using the Priority Inheritance Protocol (PIP) [8].
- **GS-MSRP** [36]: the Greedy Slacker (GS) partitioning heuristic with the spin-based locking protocol MSRP [10] under Audsley's Optimal Priority Assignment [2].

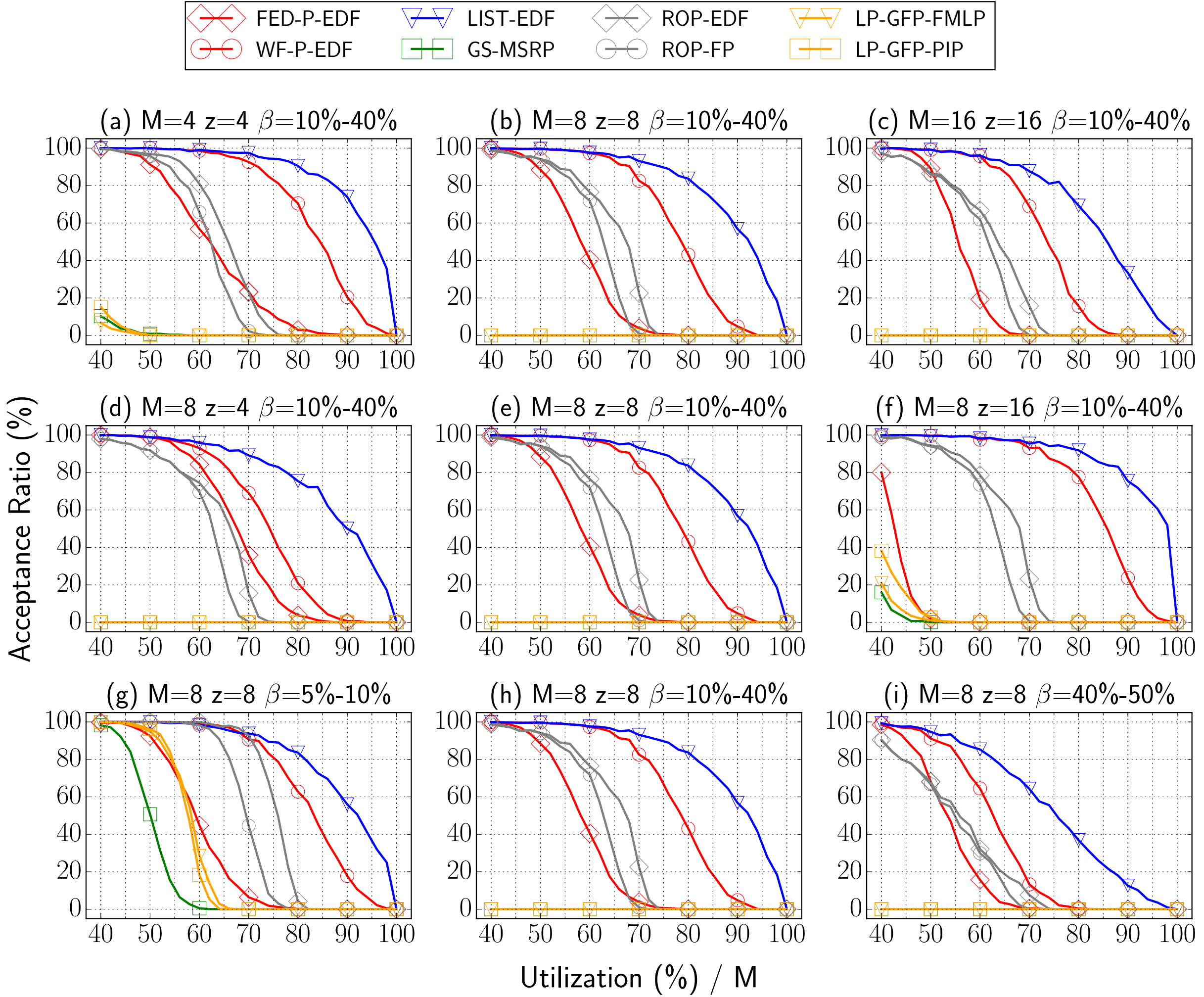

Fig. 3. Schedulability of different approaches for periodic task sets.

C. Schedulability for Periodic Task Sets

Evaluating the periodic task sets, due to the space limitation and similar performance, only the subsets of the evaluation results are presented in Figure 3. In general, one of our proposed methods clearly outperforms other (semi-)partitioned approaches.[1]

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

1) $\boldsymbol{M = z \in \{4, 8, 16\}}$ (Figure 3 (a) to Figure 3 (c)): Increasing z and M at the same time does not have significant impact on the WF-P-EDF but the other approaches perform worse.
2) $\boldsymbol{z}$ **for a fixed** $\boldsymbol{M}$, i.e., $z \in \{4, 8, 16\}$ and $M = 8$ (Figure 3 (d) to Figure 3 (f)): When the number of resources is increased, compared to the number of processors, the performance gap between the WF-P-EDF and the FED-P-EDF approaches increases. This indicates that when the number of resources becomes large, the advantage of balancing workloads among processors of the WF-P-EDF, and the disadvantage of isolate the tasks according the requested resources of the FED-P-EDF is significant.
3) **Workload of Shared Resources, i.e.,** $\boldsymbol{\beta \in \{[5\% - 10\%], [10\% - 40\%], [40\% - 50\%]\}}$ (Figure 3(g) to Figure 3 (i)): If the workload of the critical sections is increased, the performance of all methods is reduced, and the difference of different methods is decreased as well. The reason is that, when $\beta = [40\% - 50\%]$, the execution time of the critical section for tasks with period 10 can be large, i.e., longer than 2. Therefore, tasks with period 1 directly miss the deadline by default for all other approaches, no matter what kind of the partitioning algorithm is applied, the performance drops down quickly when the utilization is increased and the critical section workload is large as shown in Figure 3 (i).

[1]LIST-EDF is not a partitioned scheduling, used here as the baseline for state-of-the-art.

D. Schedulability for Frame-Based Task Sets

Regarding schedulability for frame-based task sets, we compare our approaches with all the aforementioned methods. Due to the space limitation and similar performances (w.r.t. acceptance ratios), only a subset of the evaluation results is presented in Figure 4. The proposed worst-fit heuristic **WF-P-**

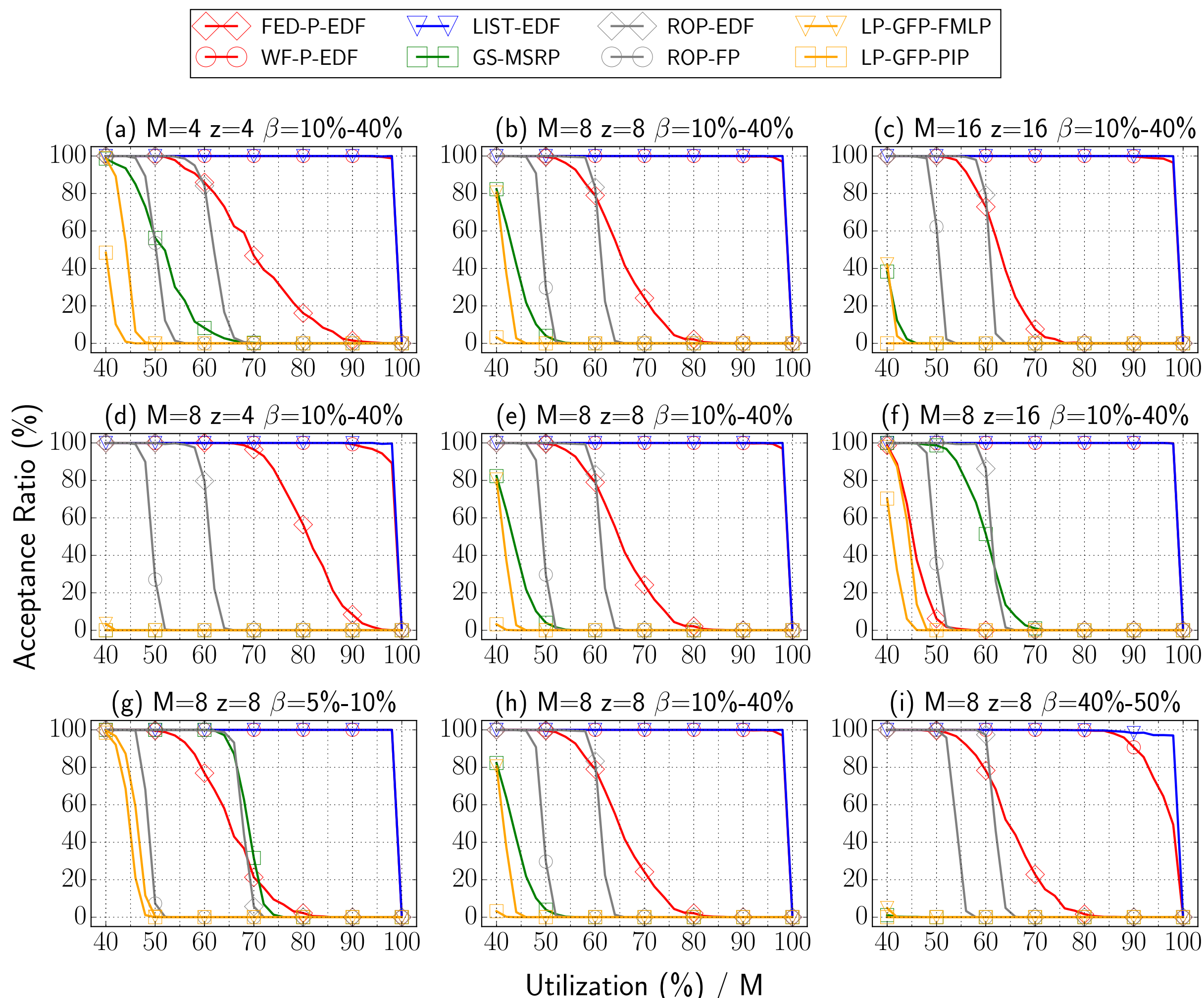

Fig. 4. Schedulability of different approaches for frame-based task sets.

EDF outperforms **ROP-EDF** and other partitioned scheduling methods significantly. Furthermore, Figure 4 shows that **WF-P-EDF** has a good performance compared to **LIST-EDF**. In most cases, both **LIST-EDF** and **WF-P-EDF** can reach 100 % acceptance ratio even for 98 % utilization per processor. The reasons are as follows:

- Since P-EDF is a preemptive schedule, the (potentially long) idle slots caused by the precedence constraints for the critical sections can be filled by other task's normal executions without influencing the executions of critical sections, even if the critical sections are relatively long.
- The modified deadlines for each sub task help to avoid deadline misses.
- The worst-fit decreasing strategy allows to balance the workload on the processors nearly optimally.

However, when the utilization for critical section become extremely large in Figure 4(i), our new proposed **WF-P-EDF** performs worse than **LIST-EDF**, since some of the precedence constraints may push some of the critical sections back so far, that a the processor actually runs idle due to the partitioned property. The probability for such a situation is increased when the utilizations of critical sections become large, and therefore the resulting dependency graphs related to the individual resources are long. In such a situation, global scheduling is able to fill these idle slots by migrating tasks.

VIII. Conclusion

This paper considers partitioned scheduling for dependency graphs in multiprocessor real-time resource synchronization i.e., tasks are *tied* to a processor. We detail how the schedulability can be determined for a given partition and proposes two partitioning algorithms, which are based on federated scheduling and a worst-fit heuristic. For the latter, different pre-sorting strategies for these tasks are considered. The evaluations based on *periodic* and *frame-based* task sets under different configurations show that our proposed methods can outperform existing partitioned scheduling algorithms and perform reasonably compared to a global scheduling approach. Our proposed methods can also be applied for given dependency graphs under a less restrictive task model, e.g., multiple critical sections within one task. However, how to construct the dependency graph in such a situation remains an open problem.

Acknowledgement: This paper is supported by DFG, as part of the Collaborative Research Center SFB876, project A3 and B2 (http://sfb876.tu-dortmund.de/). The authors thank Zewei Chen and Maolin Yang for their tool SET-MRTS (Schedulability Experimental Tools for Multiprocessors Real Time Systems, https://github.com/RTLAB-UESTC/SET-MRTS-public) to evaluate the GS-MSRP, LP-GFP-FMLP, and LP-GFP-PIP in Figure 3 and Figure 4.

REFERENCES

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

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

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

[6] B. B. Brandenburg and M. Gül. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 99–110. IEEE, 2016.

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

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

[9] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[10] 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.

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

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

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

[14] 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.

[15] 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.

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

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

[18] 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.

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

[20] G. J. Narlikar. Scheduling threads for low space requirement and good locality. *Theory of Computing Systems*, 35(2):151–187, 2002.

[21] M. Nasri, S. K. Baruah, G. Fohler, and M. Kargahi. On the optimality of RM and EDF for non-preemptive real-time harmonic tasks. In *RTNS*, page 331, 2014.

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

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

[24] 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.

[25] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, Nov. 2015.

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

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

[28] 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.

[29] 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.

[30] 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 25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 279–292. IEEE, 2019. Proceedings not yet availible, download link: https://ls12-www.cs.tu-dortmund.de/daes/media/multi-sync-periodic-dga.pdf.

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

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

[33] 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.

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

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

[36] 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.