Replication-Based Scheduling of Parallel Real-Time Tasks

Federico Aromolo
Scuola Superiore Sant'Anna, Pisa, Italy

Geoffrey Nelissen
Eindhoven University of Technology, Eindhoven, The Netherlands

Alessandro Biondi
Scuola Superiore Sant'Anna, Pisa, Italy

Abstract

Multiprocessors have become the standard computing platform for real-time embedded systems. To efficiently leverage the computational power of such platforms, software tasks are often characterized by an internal structure where concurrent subtasks can execute in parallel on different processors. Existing strategies for the scheduling of parallel real-time tasks on multiprocessor platforms, such as partitioned, global, and federated scheduling, were inspired by earlier techniques that were not conceived to explicitly support parallel tasks, thus carrying advantages but also well-known limitations. This paper introduces replication-based scheduling, a specialized scheduling paradigm for parallel real-time DAG tasks. Replication-based scheduling leverages the internal structure of the parallel tasks to assign replicas of the subtasks to different processors, while ensuring that exactly one replica of each subtask will be executed at runtime for every task instance. This approach aims at preserving the advantages of partitioned scheduling while simplifying the timing analysis. The replication-based scheduling framework is first defined, together with a strategy for implementing replication-based scheduling in real-time operating systems. Then, offline allocation strategies for subtask replicas and a response-time analysis are presented. In the provided experiments, the schedulability achieved with replication-based scheduling is compared with that of existing techniques for the scheduling of parallel real-time tasks on multiprocessors.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

Keywords and phrases Real-Time Systems, Scheduling Algorithms, Schedulability Analysis, Parallel Tasks

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.18

Acknowledgements The authors would like to thank José Fonseca for the fruitful discussions that triggered the idea for the approach presented in the paper.

1 Introduction

With the emergence of multiprocessor systems as the standard enabling platform for high-performance real-time embedded computing systems, computational workloads have evolved towards highly parallel structures to match the enhanced processing capabilities offered by the underlying hardware. Numerous models exist to capture and analyze the timing behavior of the scheduling system and guide the allocation of the computational activities to the available processing elements, both at design time and at runtime, in order to maximize resource usage while ensuring timely execution of all software activities in the system. However, existing scheduling solutions for parallel tasks are characterized by either achieving low resource utilization levels, or by excessive complexity in their runtime behavior and implementation, leading to conservative analyses and significant runtime overheads [9, 8, 11, 23, 19].

Contributions. This paper presents the replication-based scheduling paradigm (RBS) for managing the execution of sporadic parallel real-time tasks on multiprocessor computing platforms, which aims at improving the achieved system utilization and schedulability performance by employing a flexible allocation and execution scheme based on subtask replication. The main contributions are as follows:

1. Definition of the replication-based scheduling approach and discussion on its distinguishing features in comparison with existing approaches.
2. Description of a pattern of implementation of replication-based scheduling in real-time operating systems.
3. Real-time analysis for parallel tasks executing on a multiprocessor platform under preemptive fixed-priority replication-based scheduling.
4. Experimental evaluation of the schedulability performance of replication-based scheduling, in comparison with existing techniques for the scheduling of real-time parallel tasks.

2 System model

We consider a set $\tau = \{\tau_1, \dots, \tau_n\}$ of n sporadic parallel real-time tasks, to be scheduled on a multiprocessor platform consisting of m identical processors $P_1, \dots, P_m$ under preemptive fixed-priority scheduling. Each task τ_i releases a potentially infinite sequence of jobs, each separated from the next by at least a minimum inter-arrival time T_i, and subject to a constrained relative deadline D_i, such that $D_i \leq T_i$. The parallel computational structure of each task τ_i is modeled as a directed acyclic graph (DAG) $G_i = (V_i, E_i)$, where $V_i = \{v_{i,1}, v_{i,2}, \dots, v_{i,n_i}\}$ is a set of n_i nodes (or vertices), and $E_i \subseteq V_i \times V_i$ is a set of directed edges between nodes in V_i. Each node $v_{i,a} \in V_i$ represents a sequential computational unit, or subtask, of the task τ_i, and is characterized by a worst-case execution time (WCET) $C_{i,a}$. Each edge in E_i represents a precedence constraint between two nodes of the DAG G_i. If $e_i^{a,b} = (v_{i,a}, v_{i,b})$ is an edge connecting the vertices $v_{i,a}$ and $v_{i,b}$, then, for every job of τ_i, subtask $v_{i,b}$ cannot execute before $v_{i,a}$ is completed. Each task τ_i is assigned an unique fixed scheduling priority π_i. Subtasks inherit the priority of the corresponding task. The set of tasks with priorities higher than or equal to that of a task τ_i, excluding τ_i itself, is denoted by $\mathrm{hep}(\tau_i)$. Overall, a task τ_i is characterized by the tuple (G_i, T_i, D_i, π_i).

The cumulative worst-case execution time (WCET) C_i of a task τ_i is defined as $C_i = \sum_{v_{i,a} \in V_i} C_{i,a}$. The utilization factor U_i of τ_i is defined as $U_i = C_i / T_i$. The response time of a job of a task τ_i is defined as the difference between its finishing time, that is, the time at which the job completes its execution, and its arrival time. The worst-case response time (WCRT) R_i of a task τ_i is defined as the maximum response time across all possible jobs of τ_i in all possible schedules of task set τ, with respect to the adopted scheduling algorithm. Analogously, the response time of an instance of a subtask $v_{i,a}$ within a job of τ_i is defined as the difference between the finishing time of that instance and the arrival time of the corresponding job, while the WCRT $R_{i,a}$ of a subtask $v_{i,a}$ is defined as the maximum possible response time of $v_{i,a}$ across all possible jobs of τ_i in all possible schedules of task set τ.

Whenever an edge from $v_{i,a}$ to $v_{i,b}$ exists, $v_{i,a}$ is said to be an immediate predecessor of $v_{i,b}$, whereas $v_{i,b}$ is said to be an immediate successor of $v_{i,a}$. The set of immediate predecessors of $v_{i,a}$ is denoted by $\mathrm{ipred}(v_{i,a})$, while the set of immediate successors of $v_{i,a}$ is denoted by $\mathrm{isucc}(v_{i,a})$. When the immediate predecessor and the immediate successor definitions are applied transitively starting from a node $v_{i,a}$ over the topology of the DAG G_i, the set of predecessors and the set of successors of $v_{i,a}$ are obtained, respectively. Two different nodes are said to be independent from each other if neither is a predecessor or a

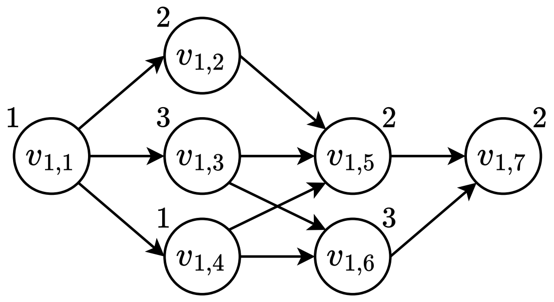

Figure 1 Example of computational structure of a parallel task τ_1, modeled as a DAG G_1.

successor of the other. A node with no incoming edge is referred to as a *source* node, while a node with no outgoing edge is referred to as a *sink* node. The set of sink nodes in a DAG G_i is denoted by $\text{sink}(G_i)$. In the following we assume, without loss of generality, that a single source node, denoted by $v_{i,S}$, is present in each DAG G_i. A *path* in a DAG G_i is defined as an ordered sequence of nodes where a directed edge exists between any two adjacent nodes in the sequence, each node in the sequence is an immediate predecessor of the following node, and the sequence starts from a source node and ends on a sink node. Given a path λ, $V(\lambda)$ represents the set of nodes belonging to the path. The set of all paths in a DAG G_i is denoted by $\text{path}(G_i)$.

Running example. Figure 1 depicts the DAG topology G_1 of an example parallel task τ_1 composed of $n_1 = 7$ nodes. In the figure, the WCET $C_{1,a}$ of each subtask $v_{1,a}$ is reported next to the corresponding node.

2.1 Scheduling requirements

Designing a suitable scheduling paradigm for parallel tasks requires satisfying the following requirements for each task set τ scheduled under that paradigm.

- **Requirement 1.** For each task $\tau_i \in \tau$, in all jobs of τ_i, the precedence constraints in G_i must be properly enforced, meaning that each node in G_i cannot start executing before all of its predecessors have completed.
- **Requirement 2.** For each task $\tau_i \in \tau$, in all jobs of τ_i, each node in G_i must execute exactly once.

3 Background and motivations

Several parallel task models have been proposed in the literature to represent the different forms of workload generated by well-known parallel programming models. In the fork-join model [22, 27, 4], tasks are represented as an interleaved sequence of sequential and parallel segments, where synchronization is assumed at the boundary of every segment. The sporadic DAG model was introduced by Saifullah et al. [28] to support less restrictive parallel structure structures. A number of works demonstrated that the DAG task model resembles commonly used parallel programming models such as OpenMP [30, 25].

The following techniques are considered the primary options when dealing with the scheduling of sporadic DAG tasks on multiprocessor platforms.

Global scheduling. Under fixed-priority global scheduling, the m highest-priority pending subtasks are scheduled at any given time. If tasks are preemptive, the execution of a low-priority subtask may be preempted by a higher-priority one. When resuming its execution, the preempted subtask may migrate to a different processor than that on which it was preempted. Therefore, global scheduling requires the underlying system to be capable of moving the execution context of a job from one processor to another. Migrations are typically costly and increase the execution time of jobs. An advantage of global scheduling is that all scheduling decisions are taken at runtime, and no design time resource assignment is required. Global scheduling for parallel tasks was investigated in numerous works, such as those by Bonifaci et al. [12], Baruah [5], and Fonseca et al. [17, 18].

Partitioned scheduling. Under partitioned scheduling, each subtask is statically assigned to a specific processor at design time, and can only execute on that processor at runtime. Execution of the subtasks allocated to each processor is then managed by a dedicated uniprocessor scheduler. As a result, the partitioned scheduling approach entails solving a complex allocation problem to map subtasks on processors. On the other hand, partitioned scheduling can be easily implemented in a real-time operating system by reusing techniques from uniprocessor scheduling. Parallel tasks under partitioned scheduling were analyzed by Fonseca et al. [19], Casini et al. [13], and Aromolo et al. [2] by means of model transformation techniques to self-suspending task models [14].

Federated scheduling. Federated scheduling, originally proposed by Li et al. [23], splits the task set into two disjoint sets: the set of *high-utilization tasks*, which contains all tasks τ_i such that $U_i \geq 1$, and the set of *low-utilization tasks*, which contains all tasks τ_i such that $U_i < 1$. The two sets of tasks are then treated separately. First, each high-utilization task τ_i is assigned a set of $m_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ *dedicated* processors, with $L_i = \max_{\lambda \in \text{path}(G_i)} \left\{ \sum_{v_{i,a} \in V(\lambda)} C_{i,a} \right\}$. Each high-utilization task is scheduled on its dedicated m_i processors by any work-conserving scheduler. Low-utilization tasks are treated as sequential tasks and executed with any multiprocessor scheduling algorithm for sequential tasks on the processors that were not assigned to high-utilization tasks. Subsequent works, such as those by Baruah [6, 7], Jiang et al. [21], Ueter et al. [29], Dinh et al. [15], and Jiang et al. [20], explored the application of federated scheduling under different assumptions.

3.1 Motivations

The scheduling approaches mentioned above come with both advantages (e.g., simplicity or load balancing) and disadvantages (e.g., resource over-provisioning or limited analyzability), which were properly documented in previous work [9, 8, 11, 2]. It is worth highlighting the benefits of partitioned scheduling, which include the possibility of accurately controlling the contention for memory resources and the typically lower overheads implied by its implementations compared to schedulers that support job migrations. Nevertheless, partitioned scheduling of parallel tasks proved to introduce significant complexity in the response-time analysis, which inevitably also affects the performance of partitioning algorithms [19].

Motivated by these observations, this work investigates a specialized scheduling approach for parallel tasks that aims at preserving the overall philosophy of partitioned scheduling on a per-job basis, while at the same time drastically simplifying the timing analysis.

The proposed replication-based scheduling algorithm leverages the internal structure of each parallel task to assign replicas of its nodes to different processors, while ensuring that exactly one replica of each node will be executed at runtime for every job.

Other scheduling approaches leveraging replication have been investigated in the context of high-performance computing, with the aim of reducing communication costs and improving the expected response times, but are characterized by different scheduling behaviors. For instance, duplication-based scheduling [1] statically assigns nodes of parallel tasks redundantly to different processors in order to minimize the overheads incurred due to inter-processor communication. Duplication-based scheduling was also adopted to devise a specialized partitioning strategy for real-time parallel DAG tasks which aims at eliminating inter-processor dependencies between subtasks, thus simplifying the resulting schedulability analysis [16]. However, in duplication-based approaches, all copies of each node are executed for each job, whereas the replication-based scheduling approach ensures that exactly one node replica executes for each job, meaning that the overall computational workload of the task is not increased. Concerning distributed server systems, replication-based load balancing techniques were proposed to minimize the expected response time for the incoming job requests by creating multiple replicas of the job on different servers [26]. Unlike replication-based load balancing, which aims at minimizing the expected latency for job requests in distributed systems, our solution focuses on ensuring that precedence constraints and real-time properties are satisfied in the scheduling of parallel real-time tasks.

4 Replication-based scheduling

This section presents the replication-based scheduling strategy for parallel tasks, for the specific case of preemptive fixed-priority systems. The aim of the proposed replication-based scheduling paradigm is to mitigate the limitations and performance loss suffered by existing techniques by leveraging the knowledge of the internal computational structure of the parallel tasks, in terms of their DAG topology.

4.1 Overview

As with partitioned and federated scheduling, replication-based scheduling consists of two phases; namely, an allocation phase at design time, and a dynamic scheduling phase at runtime. The core feature of replication-based scheduling is that, during the allocation phase, each node of the DAG of a given parallel task can be replicated and made available for execution on a subset of processors. For each job, a single replica of each node is then selected for execution at runtime, depending on the dynamic scheduling situation.

4.1.1 Allocation phase

In the system design phase, the computational parallel structure of each task is first decomposed into a set of linear node sequences by means of a specialized *sequence expansion* algorithm, specified later in Section 4.2. Each node sequence generated by this algorithm represents a *subpath* in the DAG which should be executed sequentially and *without suspension* on a single processor. Then, in a *sequence allocation* step, each node sequence is allocated to a specific processor, meaning that it can only execute on that specific processor at runtime. In the following, let $S_{i,q} = \langle v_{i,a}, v_{i,b}, \dots \rangle$ represent an ordered sequence of nodes of task τ_i, and let $\mathcal{S}_i = \{S_{i,1}, S_{i,2}, \dots\}$ represent the set of node sequences generated by the sequence expansion algorithm for a task τ_i. Then, $P\left(S_{i,q}\right)$ represents the processor to which a sequence $S_{i,q}$ of a task τ_i is assigned.

Running example. Figure 2 illustrates the four linear node sequences $\{S_{1,1}, S_{1,2}, S_{1,3}, S_{1,4}\}$ obtained for the example task τ_1 of Figure 1 with the *sequence expansion* algorithm specified later in Section 4.2. Sequence $S_{1,1} = \langle v_{1,1}, v_{1,2}, v_{1,5}, v_{1,7}\rangle$ contains all nodes in the upper path of τ_1. Sequence $S_{1,2} = \langle v_{1,3}, v_{1,5}, v_{1,7}\rangle$ contains the subpath starting at node $v_{1,3}$. $S_{1,3}$ starts at node $v_{1,4}$ and $S_{1,4}$ at node $v_{1,6}$. The arrows in Figure 2 represent the precedence constraints between nodes in the sequences, inherited from the DAG G_1 of τ_1.

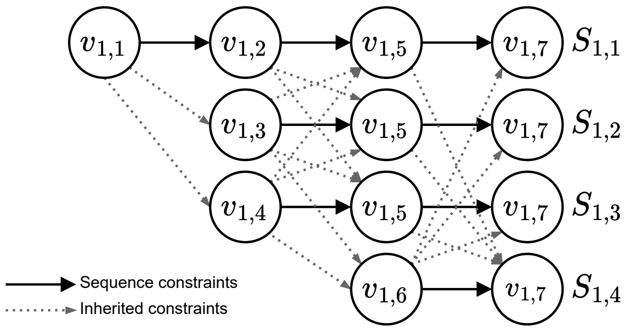

Figure 2 Example set of node sequences $\mathcal{S}_1$ obtained from the decomposition of parallel task τ_1.

As can be seen from the above example, each node of the original DAG of a given parallel task can be present in multiple sequences (e.g., $v_{1,5}$ and $v_{1,7}$ appear 3 and 4 times, respectively), which are then potentially allocated to different processors. Therefore, the nodes that belong to multiple sequences allocated to different processors are said to be replicated.

4.1.2 Runtime phase

As in the partitioned approach, the scheduling of the node sequences on each processor is managed by a dedicated uniprocessor scheduler. The runtime scheduler is designed in a way that ensures that exactly one replica of each node is executed for each job of a parallel task while enforcing the precedence constraints of the original DAG.

To do so, whenever a sequence completes the execution of a node, it checks whether all the precedence constraints of the next node in the sequence are satisfied. If they are, then the next node in the sequence is executed. If they are not, the execution of the sequence is terminated. This provides two properties. First, during system execution, a node sequence can be modeled as a sequential task without suspensions but with varying execution time executed on a single processor. Second, the combination of structural properties observed on the sequences obtained in the decomposition algorithm and of the early termination mechanism guarantees that nodes do not execute before their corresponding precedence constraints are satisfied (i.e., a sequence is ended if precedence constraints of the next node are not respected), and that exactly one replica of each node will execute for each job of a task (i.e., the replica in the last sequence reaching that node).

This means that resources are initially set aside for the execution of a specific node on multiple processors, but those resources will only be utilized in one processor for each job of the task, depending on the ongoing dynamic scheduling situation.

Running example. Assume that the sequences of Figure 2 are each assigned to a different processor of a multicore platform, so that $P(S_{1,1}) = P_1$, $P(S_{1,2}) = P_2$, $P(S_{1,3}) = P_3$, and $P(S_{1,4}) = P_4$. Note that there are four replicas of $v_{1,7}$ and three replicas of $v_{1,5}$ in this case.

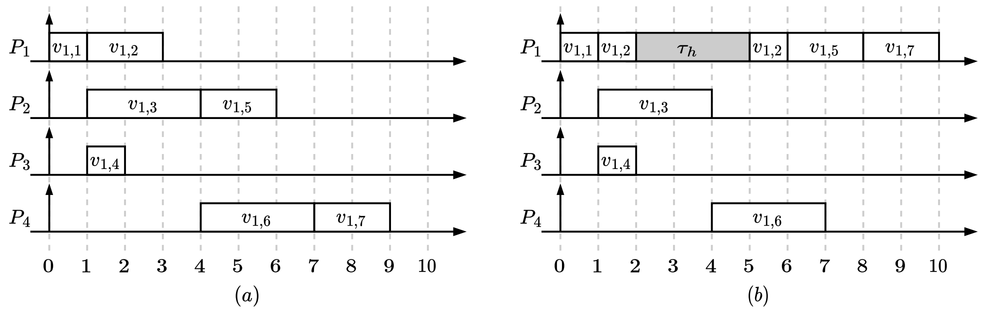

Figure 3 Example schedules of parallel task τ_1 under replication-based scheduling, in isolation (a) and with preemption by a higher-priority task τ_h (b).

Figure 3(a) provides an example schedule of a job of task τ_1 when all its subtasks execute for their WCET in isolation according to the algorithm explained above. The figure depicts the schedule of each sequence in $\mathcal{S}_1$ on the corresponding processor. In the example, the job of τ_1 is released at time 0. Therefore, the corresponding sequences arrive on the associated processor at time 0, as represented by the upward arrows. Since it corresponds to the only source node in the DAG G_1, subtask $v_{1,1}$ starts executing on processor P_1 as part of the sequence $S_{1,1}$. Once $v_{1,1}$ terminates at time 1, subtasks $v_{1,2}$, $v_{1,3}$, and $v_{1,4}$ can start their execution and sequences $S_{1,2}$ and $S_{1,3}$ are released on processors P_2 and P_3. At time 2, subtask $v_{1,4}$ terminates. At this point, the incoming precedence constraints towards the next subtask in the sequence $S_{1,3}$, i.e., $v_{1,5}$, are not yet satisfied. For this reason, the execution of the sequence $S_{1,3}$ is forcibly terminated for this job, and it will not execute its replicas of the nodes $v_{1,5}$ and $v_{1,7}$. Similarly, at time 3, the execution of subtask $v_{1,2}$ terminates, but the next subtask in the sequence $S_{1,1}$, i.e., $v_{1,5}$, cannot start executing at this time because the precedence constraint incoming from node $v_{1,3}$ is not yet satisfied. Therefore, the execution of sequence $S_{1,1}$ is also terminated early for this job. $S_{1,1}$ does not execute its replicas of the nodes $v_{1,5}$ and $v_{1,7}$. When, at time 4, subtask $v_{1,3}$ terminates its execution as part of the sequence $S_{1,2}$, all precedence constraints towards node $v_{1,5}$ are satisfied. This means that the replica of node $v_{1,5}$ belonging to the sequence $S_{1,2}$ can start its execution at time 4. At the same time, the precedence constraints towards node $v_{1,6}$ are satisfied, so that sequence $S_{1,4}$ can start its execution on processor P_4. When subtask $v_{1,5}$ terminates at time 6, node $v_{1,7}$ cannot start executing because the precedence constraint incoming from $v_{1,6}$ is not yet satisfied, therefore the corresponding sequence $S_{1,2}$ is terminated early. Finally, subtask $v_{1,7}$ starts executing on processor P_4 once subtask $v_{1,6}$ terminates at time 7, since all of its precedence constraints are satisfied at that time. The job of τ_1 finishes at time 9, when $S_{1,4}$ completes the execution of $v_{1,7}$.

Figure 3(b) provides an example schedule of a job of task τ_1 and another higher-priority task τ_h executing some workload on processor P_1 between time instants 2 and 5. This example highlights how the overall scheduling scenario on the multiprocessor system dynamically affects the selection of the replica to be executed for a job of any parallel task in the system.

In this scenario, node $v_{1,2}$ is preempted at time 2 on processor P_1, and cannot execute until time 5, when the processor becomes again available for execution of τ_1. Since node $v_{1,2}$ is the last of the predecessors of $v_{1,5}$ to terminate in this schedule, the replica of $v_{1,5}$ to be executed in this case is the one in $S_{1,1}$, instead of $S_{1,2}$ as in the previous schedule (Figure 3(a)). Similarly, the replica which is executed for the sink node $v_{1,7}$ is the one in $S_{1,1}$, again differently from the previous case.

Note that, within the schedules in Figure 3(a) and Figure 3(b), (1) each subtask of τ_1 is executed exactly once, (2) replicated nodes ($v_{1,5}$ and $v_{1,7}$) are always executed by the last sequence that reaches that node, (3) all precedence constraints between nodes of the DAG are respected, and (4) sequences may suffer release jitter and early-termination but never suffer self-suspension. These observations will be leveraged in Section 5 in order to derive a real-time analysis for replication-based scheduling.

4.2 Specification of the allocation algorithms

In the following, we specify the two algorithms that are part of the design phase of replication-based scheduling; namely, sequence expansion and sequence allocation.

4.2.1 Sequence expansion

The sequence expansion algorithm performs a decomposition of the DAG G_i of each task τ_i into sequences corresponding to subpaths in the DAG topology. The purpose of this algorithm is to generate a set of node sequences that can be executed as sequential sporadic tasks with release jitter and execution time variation.

A possible approach to perform the sequence expansion step for a given task $\tau_i \in \tau$ is described in Algorithm 1, where head$(S_{i,q})$ and tail$(S_{i,q})$ represent the first and the last node in a sequence $S_{i,q}$, respectively. First, the set $\mathcal{S}_i$ is initialized with a single sequence $S_{i,1}$, initially only including the source node $v_{i,S}$ of G_i (Lines 2-3). Then, the set $\mathcal{S}_i$ is extended in an incremental procedure (Lines 6-20). In this procedure, each sequence in $\mathcal{S}_i$, starting with $S_{i,1}$, is expanded by appending nodes to the sequence, following one path of the DAG G_i until a sink node of G_i is reached (Lines 7-10). When expanding a sequence $S_{i,q}$, all immediate successors of the last node in $S_{i,q}$ that are not added to $S_{i,q}$ initiate new sequences in $\mathcal{S}_i$ that are added to $\mathcal{S}_i$ (Lines 11-17). The sequences in $\mathcal{S}_i$ are expanded in the order in which they were created. A pair of indices, q and c, is used to keep track of the sequence that is currently being explored and of the last sequence added to $\mathcal{S}_i$ (Lines 4-5), and the procedure continues until all sequences in $\mathcal{S}_i$ have been expanded. Whenever a node is added to sequence, one of the successors isucc$(v_{i,L})$ of the last node $v_{i,L}$ of $S_{i,q}$ is selected according to the policy implemented in the SelectSuccessor procedure and is appended to the sequence $S_{i,q}$ (Lines 8-10); then, for all the other nodes $v_{i,K}$ in isucc$(v_{i,L})$, an additional sequence, initially containing $v_{i,K}$ only, is added to $\mathcal{S}_i$ if no other sequence starting with node $v_{i,K}$ exists in $\mathcal{S}_i$ (Lines 11-17).

The selection of the successor to be appended to the sequence $S_{i,q}$ that is being explored (SelectSuccessor at Line 8) can be performed according to different policies. In the following, we assume that a *static* successor selection policy is adopted, meaning that, for each node $v_{i,a} \in V_i$, the node selected to follow a replica of $v_{i,a}$ must be the same for all sequences in $\mathcal{S}_i$ in which $v_{i,a}$ appears. For instance, the immediate successor node $v_{i,r}$ with smallest index r might be selected to be added as the next element in $S_{i,q}$. Another option could be that the next selected node is the node among the candidate successors which comes first in a fixed topological ordering of the nodes of the DAG G_i. Different policies may bring to different outcomes of the sequence expansion algorithm in terms of number and structure of the resulting node sequences. In future work, one may propose a non-static successor selection policy which, for example, may consist in keeping track of the number of times each node is visited as a successor of other nodes within the sequence expansion algorithm, and then selecting the node which was visited the least number of times. Figure 2 shows the sequences resulting from applying Algorithm 1 to the DAG of Figure 1, when SelectSuccessor selects the immediate successor with smallest index.

Algorithm 1 Sequence expansion algorithm for a task τ_i.

```
procedure SequenceExpansion($\tau_i$)
    $S_{i,1} \leftarrow \langle v_{i,S} \rangle$
    $\mathcal{S}_i \leftarrow \{S_{i,1}\}$
    $q \leftarrow 1$                                    ▷ Index of the next sequence to be expanded
    $c \leftarrow 1$                                    ▷ Index of the last sequence added to $\mathcal{S}_i$
    while $q \leq c$ do
        while tail($S_{i,q}$) $\notin$ sink($G_i$) do
            $v_{i,L} \leftarrow$ tail($S_{i,q}$)
            $v_{i,A} \leftarrow$ SelectSuccessor($G_i, v_{i,L}$)
            $S_{i,q} \leftarrow S_{i,q} \parallel v_{i,A}$              ▷ Append $v_{i,A}$ to $S_{i,q}$
            for all $v_{i,K} \in$ isucc($v_{i,L}$) $\setminus v_{i,A}$ do
                if $\forall S_{i,p} \in \mathcal{S}_i, v_{i,K} \neq$ head($S_{i,p}$) then
                    $c \leftarrow c + 1$
                    $S_{i,c} \leftarrow \langle v_{i,K} \rangle$
                    $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup S_{i,c}$
                end if
            end for
        end while
        $q \leftarrow q + 1$
    end while
    return $\mathcal{S}_i$
end procedure
```

4.2.2 Sequence allocation

Following the sequence expansion, each sequence in the set $\mathcal{S}_i$ for each task τ_i in τ must be allocated to a specific processor, on which it is bound to execute at runtime. This procedure is akin to the partitioning problem for partitioned scheduling of sequential and parallel tasks, and can be approached with different techniques.

A possible sequence allocation scheme is described in Algorithm 2. Under this approach, tasks are allocated in order of decreasing utilization (Line 2), and each sequence is allocated in topological order of the first node of the sequence (Line 3). The choice for the allocation of each sequence is determined based on the impact on the schedulability of a partial version of the task set following a tentative allocation of the sequence on each available processor, thus determining the allocation of each task incrementally. The schedulability can be evaluated with the response-time analysis that will be presented in Section 5. In Algorithm 2, the choice for the allocation of each sequence $S_{i,q}$ of a task τ_i is determined by tentatively allocating $S_{i,q}$ to each of the processors $P_1, \ldots, P_m$, one after the other, followed, for every such allocation, by applying the schedulability analysis to all the sequences in the partial task set including the tentatively allocated sequence $S_{i,q}$ and all the other sequences that were already allocated to a processor (Lines 4-12). Note that, since the allocation of tasks does not follow a priority order, the schedulability results obtained for tasks that were already allocated cannot be reused; therefore, the schedulability of the partial task set composed of all the sequences that were already allocated must be reevaluated for each allocation attempt. In case none of the allocations produces a schedulable task set, the task set is deemed not schedulable, and the allocation returns a failure (Lines 13-15). Otherwise, the preferred allocation is selected according to a specific policy among those that result in a schedulable partial task

Algorithm 2 Sequence allocation algorithm for a task set τ.

```
1: procedure SequenceAllocation($\tau$)
2:     for all $\tau_i \in \tau$ in decreasing utilization order do
3:         for all $S_{i,q} \in \mathcal{S}_i$ in topological order of the first node head($S_{i,q}$) do
4:             $\mathcal{P}_{i,q} \leftarrow \emptyset$                    ▷ Set of schedulable allocations of $S_{i,q}$
5:             for all $P_k \in P$ do
6:                 $P(S_{i,q}) \leftarrow P_k$                    ▷ Tentatively allocate $S_{i,q}$ to $P_k$
7:                 Test the schedulability of $S_{i,q}$ assuming it is allocated to $P_k$
8:                 For all $S_{j,p}$ that have already been allocated, test the schedulability of $S_{j,p}$
                       assuming $S_{i,q}$ is allocated to $P_k$
9:                 if $P(S_{i,q}) = P_k$ yields a schedulable task set then
10:                    $\mathcal{P}_{i,q} \leftarrow P_k$
11:                end if
12:            end for
13:            if $\mathcal{P}_{i,q} = \emptyset$ then
14:                return Failure (no valid allocation was found)
15:            end if
16:            $P(S_{i,q}) \leftarrow$ SelectProcessor($\mathcal{P}_{i,q}$)          ▷ Select a schedulable allocation
17:        end for
18:    end for
19:    return Success (all sequences were allocated)
20: end procedure
```

set (SELECTPROCESSOR at Line 16). One possible selection strategy is to first determine the slack $\overline{S}_i$ of the partial version of task τ_i within the partially allocated task set, computed as $\overline{S}_i = D_i - \overline{R}_i$, where $\overline{R}_i$ is an upper bound on the WCRT of τ_i (computed with respect to the sequences that were already allocated), and then apply a Worst Fit, Best Fit, or First Fit heuristic (or a combination of them) with respect to the available slack to select the allocation. Specifically, Worst Fit and Best Fit select the allocation producing, respectively, the largest and the smallest slack $\overline{S}_i$, while First Fit simply selects the first processor that fits the sequence.

A variant of this approach is inspired by the dual allocation scheme proposed in federated scheduling. In this case, tasks are allocated in order of decreasing utilization, where the sequences of the high-utilization tasks, i.e., those tasks τ_i with utilization factor $U_i \geq 1$, are allocated as in the above approach. Instead, for sequences of low-utilization tasks, i.e., those tasks τ_i with utilization factor $U_i < 1$, an attempt is first made to allocate the full task to a processor as a single linearized sequence of τ_i, corresponding to a topological sorting of the nodes in G_i, similar to how low-utilization tasks are treated in federated scheduling. If the attempt fails, the task is allocated as in the above approach leveraging the slack.

4.3 Runtime phase and implementation pattern

The runtime phase of replication-based scheduling decides which replica of each subtask should be executed at runtime. As discussed in Section 4.1, the executed replica varies for each job of a task and depends on runtime properties like the actual execution time of predecessors or the interference suffered by subtasks. In order to realize a runtime mechanism for replication-based scheduling that is consistent with the requirements for the scheduling of parallel tasks, the following rules govern the execution of the task sequences.

- **Rule 1.** Each sequence $S_{i,q} \in \mathcal{S}_i$ of every task $\tau_i \in \tau$ is scheduled on the assigned processor $P(S_{i,q})$ according to a preemptive fixed-priority policy using the priority π_i of the corresponding task τ_i. If two or more sequences have equal priority, then the one that was released the earliest is considered as having higher priority.
- **Rule 2.** For all tasks $\tau_i \in \tau$ and each sequence $S_{i,q} \in \mathcal{S}_i$, when a job of τ_i is released, the sequence $S_{i,q}$ arrives on the corresponding processor, but is released and becomes eligible for execution only once all the precedence constraints incoming into the first node of the sequence, head$(S_{i,q})$, are satisfied.
- **Rule 3.** For all tasks $\tau_i \in \tau$ and each sequence $S_{i,q} \in \mathcal{S}_i$, nodes in the sequence $S_{i,q}$ are executed in the order in which they appear in the sequence.
- **Rule 4.** For all tasks $\tau_i \in \tau$ and each sequence $S_{i,q} \in \mathcal{S}_i$, for each pair $(v_{i,a}, v_{i,b})$ of consecutive nodes in the sequence $S_{i,q}$, whenever node $v_{i,a}$ terminates its execution as part of $S_{i,q}$ and at least one of the precedence constraints incoming into $v_{i,b}$ is not satisfied, then the execution of the sequence $S_{i,q}$ is terminated (i.e., $v_{i,b}$ and the following nodes are not executed).
- **Rule 5.** For all tasks $\tau_i \in \tau$, for each job of τ_i, no more than one replica of each node $v_{i,b} \in V_i$ can start its execution across the sequences in $\mathcal{S}_i$. In case multiple replicas of a node $v_{i,b}$ could start executing in different sequences at the same time, one of those sequences executes $v_{i,b}$ and the other sequences are terminated early, according to an arbitrary tie-breaking rule.

In the following, we describe a possible implementation pattern for the runtime rules of replication-based scheduling in a real-time operating system.

Rules 1-3 can be obtained by extending the runtime support for a preemptive fixed-priority scheduler for uniprocessors to support delaying the release of a sequence with respect to its arrival and signaling the corresponding events. In particular, Rule 1 is implemented by executing sequences according to a uniprocessor preemptive fixed-priority scheduling policy on the processor on which they are assigned. Rule 2 is implemented by delaying the release of the sequence until all the precedence constraints incoming into the first node in the sequence are satisfied. Then, Rule 3 is obtained by executing nodes in a sequence in order, one after the other.

Rules 4 and 5 require implementing an efficient inter-processor synchronization mechanism. A simple and efficient way to implement this kind of synchronization is to leverage the availability of atomic instructions (e.g., store-exclusive instructions in Arm architectures) or higher-level operating system constructs emulating their behavior. These instructions can be used by the replication-based scheduler to control the contents of a small memory area dedicated to the scheduling of a specific task, which contains the completion state of each subtask for the current job of the task, assuming that each task releases at most one job at a time. In particular, consider a task $\tau_i \in \tau$. The scheduler reserves a memory area $\mathcal{B}_i$ for τ_i that is shared among all processors to which at least one sequence of τ_i is allocated. The bits in this shared memory area can be accessed and manipulated as a bitmap using atomic instructions. The bits in $\mathcal{B}_i$ are interpreted as a vector $[B_{i,1}, B_{i,2}, \ldots, B_{i,n_i}]$ of n_i consecutive data elements, where each element $B_{i,a}$ distinguishes the completion status of the corresponding subtask $v_{i,a}$ in τ_i for the job which is currently pending among three possible states, i.e., pending but not started ($B_{i,a} = S^P$), started but not completed ($B_{i,a} = S^S$), and completed ($B_{i,a} = S^C$). In particular, when a job of the task is released, the scheduler sets the state of all nodes in $\mathcal{B}_i$ to the state S^P. Then, the code that controls the execution of each sequence $S_{i,q}$ is instrumented such that the following rules are applied.

- Whenever a node $v_{i,a}$ starts executing as part of a sequence of τ_i, the corresponding state $B_{i,a} \in \mathcal{B}_i$ is set to S^S, meaning that the node has started its execution for the current job but did not complete yet.
- Whenever a node $v_{i,a}$ completes its execution as part of a sequence of τ_i, the corresponding state $B_{i,a} \in \mathcal{B}_i$ is set to S^C, meaning that the node has completed its execution for the current job.
- Whenever a node $v_{i,a}$ with precedence constrains incoming from other sequences is the next subtask to execute in a sequence $S_{i,q}$, the shared memory area is accessed as a bitmap by the sequence $S_{i,q}$ to simultaneously check the value of all bits corresponding to the completion state of the nodes from which the precedence constraints incoming into $v_{i,a}$ originate. If at least one of those states is not S^C, meaning that the corresponding node has not yet completed in the current job of τ_i, the sequence $S_{i,q}$ is terminated early (enforcing Rule 4).
- Whenever a node $v_{i,a}$ could start executing in a sequence $S_{i,q}$, the completion status of $B_{i,a}$ is accessed and, if $B_{i,a} \neq S^P$, meaning that the corresponding node has already started executing in the current job of τ_i, the sequence $S_{i,q}$ is terminated early (enforcing Rule 5).

Running example. In the schedule in Figure 3(a), node $v_{1,5}$ cannot execute as part of sequence $S_{1,3}$ at time 2 since the completion states of nodes $v_{1,2}$ and $v_{1,3}$ are $B_{1,2} = S^S$ and $B_{1,3} = S^S$ at this time. Therefore, the corresponding sequence $S_{1,3}$ is terminated at time 2. Instead, node $v_{1,5}$ is executed as part of the sequence $S_{1,2}$ since, at time 4, all of the elements $B_{1,2}$, $B_{1,3}$, and $B_{1,4}$ corresponding to the set of nodes with precedence constraints towards $v_{1,5}$ (i.e., nodes $v_{1,2}$, $v_{1,3}$ and $v_{1,4}$) signal a completion state S^C.

The most important advantage with respect to global scheduling is that replication-based scheduling does not require support for the migration of jobs and subtasks among processors. Instead, whenever data needs to be transferred from one DAG subtask to one of its successors, it only requires that such data can be accessed by the replicas of the successor node, which can be deployed in different sequences assigned to different processors. This can be achieved by using message passing or shared memory, like in any other partitioned or global scheduler. All the node replicas must have access to the shared memory or message queue, but only one will write into it and read from it at each job execution.

4.4 Properties

In the following, we derive a set of properties of replication-based scheduling, also proving that the requirements presented in Section 2.1 for a correct execution of parallel tasks are respected with replication-based scheduling.

▶ **Lemma 1.** *For each task $\tau_i \in \tau$ and each node $v_{i,a} \in V_i$, $v_{i,a}$ is present in at least one sequence in $\mathcal{S}_i$.*

Proof. By Algorithm 1, the source node of the DAG of τ_i is the first node of sequence $S_{i,1}$ (Line 3). Now, for any node that is in a sequence $S_{i,q} \in S_i$, all its immediate successors are either in the sequence $S_{i,q}$ (Lines 8-10 of Algorithm 1) or are the first node of another sequence in S_i (Lines 11-17 of Algorithm 1). Therefore, by induction starting from the source node of τ_i, all nodes of τ_i are at least in one sequence in S_i. ◀

▶ **Lemma 2.** *Replication-based scheduling satisfies Requirement 1; i.e., for each task $\tau_i \in \tau$, each node in G_i does not start executing before all of its predecessors have completed.*

Proof. Consider a task $\tau_i \in \tau$. Consider a replica of a node $v_{i,a} \in V_i$ in a sequence $S_{i,q}$. If that replica is located at the start of the sequence $S_{i,q}$, then, by Rule 2, the release of the sequence, and thus the start of the replica, is delayed until all the precedence constraints incoming into node $v_{i,a}$ are satisfied. Instead, if that replica is not the first node of the sequence $S_{i,q}$, then, by Rule 3, nodes in $S_{i,q}$ are executed in the order in which they appear in the sequence, and, by Rule 4, the sequence is terminated if at least one of the predecessors of the replica of $v_{i,a}$ did not yet execute when the replica of $v_{i,a}$ is reached in $S_{i,q}$. Therefore, a replica of $v_{i,a}$ may only execute if all precedence constraints are satisfied. ◀

▶ **Lemma 3.** *Replication-based scheduling satisfies Requirement 2; i.e., for each task $\tau_i \in \tau$, in all jobs of τ_i, each node in G_i executes exactly once.*

Proof. By Rule 5, each node in G_i does not execute more than once in all jobs of τ_i. It then remains to prove that each node in G_i executes at least once in all jobs of τ_i. We prove it by structural induction on the topology of the DAG G_i. The base case of the structural induction corresponds to proving that the source node $v_{i,S}$ executes at least once in each job of τ_i. This holds by considering that the source node $v_{i,S}$ has no incoming precedence constraints and a single replica of $v_{i,S}$ is present as the starting node of $S_{i,1}$ in $\mathcal{S}_i$; therefore, that replica can immediately start executing once the job of the task is released (Rule 2) and can never be prevented from executing as a result of Rules 4 or 5. For the inductive step, we prove that, in a generic job of τ_i, if all the predecessors of any node $v_{i,a} \in V_i$ execute at least once, then $v_{i,a}$ executes at least once. First, by Lemma 1, for each task τ_i, each node $v_{i,a} \in V_i$ is present in at least one sequence in $\mathcal{S}_i$. If at least one replica of $v_{i,a}$ appears as the first node of a sequence $S_{i,q} \in \mathcal{S}_i$, then that replica can never be terminated as a result of Rule 4, and can in fact only be terminated if another replica of $v_{i,a}$ is selected for execution according to the tie-breaking in Rule 5, therefore $v_{i,a}$ will be executed in $\mathcal{S}_i$ once all the precedence constraints incoming into $v_{i,a}$ are satisfied (Rule 2). In the following, consider the case where $v_{i,a}$ never appears as the first node of a sequence in $\mathcal{S}_i$. Consider the last sequence that executes an immediate predecessor of $v_{i,a}$. Call that sequence $S_{i,L}$ and the executed predecessor $v_{i,L}$. $S_{i,L}$ must exist since, by the induction assumption, all immediate predecessors of $v_{i,a}$ must execute at least once for each job of τ_i. By Algorithm 1 (Lines 7-18), all sequences that include an immediate predecessor $v_{i,L}$ of $v_{i,a}$ must have a replica of $v_{i,a}$ right after $v_{i,L}$. Thus, $S_{i,L}$ either executes $v_{i,a}$ after $v_{i,L}$, which would complete the proof, or $S_{i,L}$ is terminated early after executing $v_{i,L}$. In the latter case, it means that, by Rule 4, at least one of the predecessors of $v_{i,a}$ must not have completed its execution when $S_{i,L}$ completes the execution of $v_{i,L}$. However, this contradicts the assumptions that all predecessors of $v_{i,a}$ execute, and that $S_{i,L}$ is the last sequence executing a predecessor of $v_{i,a}$. Thus, $v_{i,a}$ certainly executes as part of $S_{i,L}$. ◀

5 Schedulability analysis

Unlike other scheduling algorithms (e.g., partitioned or global fixed-priority or Earliest Deadline First scheduling), replication-based scheduling was designed from the ground up so as to simplify its schedulability analysis and avoid analytical pessimism introduced by a limited understanding of which schedule may lead to the worst-case response time of each DAG task. In fact, a task set scheduled with replication-based scheduling may simply be analyzed as a set of sequential sporadic tasks with release jitter scheduled on single core platforms.

The proposed response-time analysis for replication-based scheduling derives a WCRT upper bound $\overline{R}_{i,a}$ for each node $v_{i,a}$ in V_i. Then, the WCRT upper bound $\overline{R}_i$ of a task τ_i is given by the maximum value of $\overline{R}_{i,a}$ for any $v_{i,a}$ in V_i, or, equivalently, for any $v_{i,a} \in \text{sink}(G_i)$. The task set τ is then deemed schedulable if $\overline{R}_i \leq D_i$ holds for each task τ_i. The main observation behind the analysis is that each sequence of a parallel task τ_i in τ behaves as a sporadic task with release jitter and execution time variation.

5.1 Response-time analysis with model transformation

Consider the following properties of replication-based scheduling to support our claim that each sequence can be modeled as an independent sequential sporadic task with release jitter and execution time variation executing on a single core platform.

▶ **Property 1** (From Rule 3)**.** *A sequence $S_{i,q}$ starts by executing its first node* $\text{head}(S_{i,q})$*, and all the following nodes will execute sequentially.*

▶ **Property 2** (From Rule 2)**.** *The first node of a sequence $S_{i,q}$, $v_{i,s} = \text{head}(S_{i,q})$, arrives at the same time as the job of the task τ_i, but is released and becomes eligible for execution only once all the precedence constraints incoming into $v_{i,s}$ have been fulfilled.*

▶ **Property 3.** *A sequence $S_{i,q}$ never self-suspends as part of its execution.*

Proof. None of the Rules 1-5 allows a sequence $S_{i,q}$ to perform a self-suspension. ◀

▶ **Property 4.** *A sequence $S_{i,q}$ never migrates.*

Proof. By Rule 1, $S_{i,q}$ can only execute on the processor $P(S_{i,q})$ on which it is assigned. ◀

From the above properties, it is evident that the behavior of a sequence $S_{i,q}$ is equivalent to executing a sequential sporadic task τ'_i on a single-core platform, subject to a release jitter J'_i, where the jitter is given by the largest amount of time by which the precedence constraints of the first nodes of the sequence are fulfilled, i.e., by the maximum response time among the immediate predecessors of the first node of $S_{i,q}$, while the WCET C'_i is simply given by the sum of the WCETs of the nodes in $S_{i,q}$.

Since the above observation holds for every sequence of every task in τ, the WCRT of a sequence $S_{i,q}$ can be obtained by means of a model transformation of the set of sequences allocated to the same processor $P(S_{i,q})$ as $S_{i,q}$ into a set of sporadic tasks with release jitter.

The WCRT R'_i of a task τ'_i, and thus of a sequence $S_{i,q}$, in a set τ' of sporadic tasks with release jitter can then be computed with the response-time analysis by Audsley et al. [3]. That is, $R'_i = r'_i + J'_i$, where r'_i is the smallest positive solution of the recurrent equation[1]

$$r'_i = C'_i + \sum_{\tau'_k \in \{\tau' \setminus \tau'_i\} | \pi_k = \pi_i} C'_k + \sum_{\tau'_j \in \text{hp}(\tau'_i)} \left\lceil \frac{r'_i + J'_j}{T'_j} \right\rceil C'_j, \tag{1}$$

where $\text{hp}(\tau'_i)$ denotes the set of tasks with priority higher than that of τ'_i.

Applying the above transformation requires deriving an upper bound on the release jitter J'_i of a sequence $S_{i,q}$, which is a function of the WCRT of the predecessors of the first node of $S_{i,q}$. In fact, a sequence $S_{i,q}$ is released only when all nodes with a precedence constraint towards $S_{i,q}$ have completed their execution. Therefore, for every DAG task τ_i, the proposed analysis computes WCRT upper bounds for each node of τ_i in their topological order in G_i.

[1] Note that, since jobs with identical priorities are executed in FIFO order, at most one job of a task with identical priority to τ'_i can interfere with a job of τ'_i, hence the second term of Equation (1).

Algorithm 3 Derivation of WCRT upper bounds $\overline{R}_i$ for each task τ_i in τ.

```
1: procedure ComputeWCRTUpperBounds($\tau$)
2:     for all $\tau_i \in \tau$ in decreasing priority order do
3:         for all $v_{i,a} \in V_i$ in topological order do
4:             for all $S_{i,q} \in \mathcal{S}_i \mid v_{i,a} \in S_{i,q}$ do
5:                 $\tau' \leftarrow$ Transformation($\tau, (i,a,q)$)
6:                 $\overline{R}_{i,a,q} \leftarrow$ RTA($\tau', (i,a)$)
7:             end for
8:             $\overline{R}_{i,a} \leftarrow \max\{\overline{R}_{i,a,q} \mid v_{i,a} \in S_{i,q} \land S_{i,q} \in \mathcal{S}_i\}$
9:         end for
10:        $\overline{R}_i \leftarrow \max\{\overline{R}_{i,a} \mid v_{i,a} \in V_i\}$
11:    end for
12: end procedure
```

More specifically, as detailed in Algorithm 3, tasks in τ are analyzed in decreasing priority order, and the subtasks of each task τ_i are analyzed in topological order. A WCRT upper bound $\overline{R}_{i,a}$ is derived for each subtask $v_{i,a}$, by taking the maximum value of the WCRT bounds of all replicas of that node across all sequences of τ_i (Lines 3-10). The WCRT upper bound $\overline{R}_i$ of τ_i is then given by the response time of the node of τ_i with the largest response time (Line 10). At Line 5, the WCRT bound $\overline{R}_{i,a,q}$ for the replica of a node $v_{i,a}$ in a sequence $S_{i,q}$ is calculated by transforming τ_i and all higher-priority and equal-priority tasks into a set of equivalent sporadic tasks τ' using Algorithm 4, detailed later in this section. The WCRT upper bound of $v_{i,a}$ in a sequence $S_{i,q}$ is then obtained by applying the response-time analysis by Audsley et al. [3] presented above to the equivalent sporadic task $\tau'_{i,a} \in \tau'$ (RTA at Line 6).

The model transformation procedure (Transformation at Line 5) is detailed in Algorithm 4. The procedure constructs a set τ' of sporadic tasks with release jitter. For the analysis of a replica of node $v_{i,a}$ of τ_i in sequence $S_{i,q}$, Algorithm 4 creates one sporadic task for each node of every task with priority higher than or equal to that of τ_i that has a replica assigned to the same processor as $S_{i,q}$. The procedure is based on the following three lemmas.

▶ **Lemma 4.** *The interference generated by a sequence $S_{h,p}$ with release jitter $J_{h,p}$ and WCET $\sum_{v_{h,k} \in S_{h,p}} C_{h,k}$ in an interval of length Δ is upper bounded by the sum of the interference generated by each of its nodes $v_{h,k}$ modeled as sporadic tasks with release jitter $J_{h,p}$ and WCET $C_{h,k}$.*

Proof. Since $S_{h,p}$ behaves as a sporadic sequential task, the interference generated by $S_{h,p}$ during an interval Δ is upper bounded by $\left\lceil \frac{\Delta + J_{h,p}}{T_h} \right\rceil \times \sum_{v_{h,k} \in S_{h,p}} C_{h,k} = \sum_{v_{h,k} \in S_{h,p}} \left\lceil \frac{\Delta + J_{h,p}}{T_h} \right\rceil \times C_{h,k}$, which is equivalent to the interference generated by a set of sporadic tasks made of one task per node $v_{h,k} \in S_{h,p}$ with release jitter $J_{h,p}$ and execution time $C_{h,k}$. ◀

▶ **Lemma 5.** *Maximizing the release jitter of a node $v_{h,k}$ maximizes the interference it generates.*

Proof. Equation (1) is monotonically non-decreasing with respect to the release jitter of each task. ◀

▶ **Lemma 6.** *Let $v_{i,b}$ be a node of τ_i that is not in sequence $S_{i,q}$ and has a precedence constraint towards the first node of $S_{i,q}$. The node $v_{i,b}$ cannot interfere with $S_{i,q}$.*

Algorithm 4 Model transformation algorithm.

```
1: procedure TRANSFORMATION($\tau, (i, a, q)$)
2:     $\tau' \leftarrow \emptyset$                                   ▷ Transformed task set
       ▷ For all sequences of higher or equal priority assigned to the same processor as $S_{i,q}$
3:     for all $\tau_h \in \text{hep}(\tau_i)$ do
4:         for all $S_{h,p} \in \mathcal{S}_h \mid P(S_{h,p}) = P(S_{i,q})$ do
5:             $J_{h,p} \leftarrow \max\left\{0, \max_{v_{h,b} \in \text{ipred}(\text{head}(S_{h,p}))} \left\{\overline{R}_{h,b}\right\}\right\}$      ▷ Release jitter of $S_{h,p}$
6:         end for
               ▷ For all nodes of $\tau_h$ with a replica assigned to the same processor as $S_{i,q}$
7:         for all $v_{h,k} \in V_h \mid \exists S_{h,p} \in \mathcal{S}_h,\ v_{h,k} \in S_{h,p} \wedge P(S_{h,p}) = P(S_{i,q})$ do
8:             $J'_{h,k} \leftarrow \max_{S_{h,p} \mid v_{h,k} \in S_{h,p} \wedge P(S_{h,p}) = P(S_{i,q})} \{J_{h,p}\}$    ▷ Max. release jitter of $v_{h,k}$
9:             $\tau'_{h,k} \leftarrow$ Create a sporadic task with release jitter $J'_{h,k}$ and WCET $C_{h,k}$
10:            $\tau' \leftarrow \tau' \cup \tau'_{h,k}$
11:        end for
12:    end for
                        ▷ For all nodes of $\tau_i$ assigned to the same processor as $S_{i,q}$
13:    for all $v_{i,b} \in \{V_i \setminus v_{i,a}\} \mid \exists S_{i,p} \in \mathcal{S}_i,\ v_{i,b} \in S_{i,p} \wedge P(S_{i,p}) = P(S_{i,q})$ do
14:        if $v_{i,b}$ is independent from head($S_{i,q}$) in $G_i$ then
15:            $\tau'_{i,b} \leftarrow$ Create a sporadic task with WCET $C_{i,b}$ and release jitter $J'_{i,b} = 0$
16:            $\tau' \leftarrow \tau' \cup \tau'_{i,b}$
17:        end if
18:    end for
19:    $S^\star_{i,q} \leftarrow$ The sequence obtained by removing all nodes after $v_{i,a}$ in $S_{i,q}$
20:    $J'_{i,q} \leftarrow \max\left\{0, \max_{v_{i,b} \in \text{ipred}(\text{head}(S_{i,q}))} \left\{\overline{R}_{i,b}\right\}\right\}$        ▷ Max. release jitter of $S_{i,q}$
21:    $\tau'_{i,a} \leftarrow$ Create a sporadic task with WCET $\sum_{v_{i,j} \in S^\star_{i,q}} C_{i,j}$ and release jitter $J'_{i,q}$
22:    $\tau' \leftarrow \tau' \cup \tau'_{i,a}$
23:    return $\tau'$
24: end procedure
```

Proof. Let head($S_{i,q}$) be the first node of $S_{i,q}$. The node $v_{i,b}$ is either a predecessor or a successor of head($S_{i,q}$). In case $v_{i,b}$ is a predecessor of head($S_{i,q}$), then $v_{i,b}$ must be completed when $S_{i,q}$ is released. Therefore, $v_{i,b}$ does not interfere with $S_{i,q}$. In case $v_{i,b}$ is a successor of head($S_{i,q}$), then a job of $v_{i,b}$ can only be released after $S_{i,q}$. Since jobs with equal priority execute in FIFO order, $v_{i,b}$ executes after $S_{i,q}$, and thus does not interfere with $S_{i,q}$. ◀

Following the result in Lemma 4, Algorithm 4 creates one sporadic task $\tau'_{h,k}$ per node $v_{h,k}$ of each sequence of every higher-priority or equal-priority task different from τ_i (i.e., of every task in the set hep(τ_i)) assigned to the same core as the sequence $S_{i,q}$ under analysis (Lines 3-12), in order to upper bound the interference generated by those sequences. According to Lemma 3, for each job released by a task, each of its nodes executes at most once, irrespective of its number of replicas. Therefore, Algorithm 4 only generates one sporadic task per node instead of one sporadic task per replica. The WCET of the generated task $\tau'_{h,k}$ is then equal to the WCET of the node $v_{h,k}$, and its release jitter is the maximum release jitter of all the sequences in which $v_{h,k}$ appears (Line 8), so as to maximize the interference it generates (see Lemma 5). The minimum inter-arrival time and the priority of the generated task $\tau'_{h,k}$ are inherited from the corresponding task τ_h.

After generating equivalent sporadic tasks for all the tasks in hep(τ_i), Algorithm 4 generates sporadic tasks to model the self-interference of nodes of τ_i on the sequence $S_{i,q}$ under analysis (Lines 13-18). One such task is generated for each node of τ_i, except $v_{i,a}$

itself, that is independent from the first node in $S_{i,q}$ and has a replica assigned to $P(S_{i,q})$ (in accordance with Lemma 6). Note that, according to Equation (1), since the sporadic tasks modeling the self-interference of nodes of τ_i have the same priority as $S_{i,q}$, their release jitter does not influence the WCRT of the sequence under analysis. Therefore, Algorithm 4 arbitrarily sets their release jitter to 0.

Finally, since we aim at computing the WCRT of node $v_{i,a}$ in sequence $S_{i,q}$, Algorithm 4 models the partial sequence $S^{\star}_{i,q} \subseteq S_{i,q}$ ending at $v_{i,a}$ as a sporadic task $\tau'_{i,a}$, with WCET equal to the sum of the execution time of its nodes and release jitter equal to the maximum WCRT upper bound of the predecessors of the first node of $S_{i,q}$ (Lines 19-22).

5.2 Analysis improvements

Although the analysis presented in Section 5.1 is an efficient approach to test the schedulability of a set of parallel tasks executing under replication-based scheduling, the analysis might yield pessimistic WCRT upper bounds in some cases.

In order to identify a potential source of such analytical pessimism, consider a replica of the node under analysis $v_{i,a}$ in sequence $S_{i,q}$ and a replica of another node $v_{i,b}$ in $S_{i,p}$ that triggers the release of $S_{i,q}$ (i.e., it is an immediate predecessor of head$(S_{i,q})$ that causes the release jitter on $S_{i,q}$). According to the analysis in Section 5.1, the WCRT of $v_{i,a}$ in $S_{i,q}$ is upper bounded by the WCRT upper bound of $v_{i,b}$ added to the solution to Equation (1) for $v_{i,a}$. Assume that there is a node v_h of a higher-priority task with replicas assigned to the processors where $S_{i,q}$ and $S_{i,p}$ execute. Then, v_h interferes with both $v_{i,a}$ and $v_{i,b}$. Since the analysis in Section 5.1 analyses the WCRT of $v_{i,a}$ and $v_{i,b}$ independently from each other, it may account for the same jobs of v_h as interfering with both $v_{i,a}$ and $v_{i,b}$, thus overestimating the overall interference those jobs may generate.

The following lemma provides a lower bound on the redundant interference caused by the higher-priority node v_h in the computation of $\overline{R}_{i,a,q}$, with reference to the replicas of $v_{i,a}$ in $S_{i,q}$ and of $v_{i,b}$ in $S_{i,p}$.

▶ **Lemma 7.** *Let $r_{i,a,q}$ and $r_{i,b,p}$ represent the solutions to Equation* (1) *for, respectively, $v_{i,a}$ in $S_{i,q}$ and $v_{i,b}$ in $S_{i,p}$. Assume that the release jitter of $S_{i,q}$ is equal to the WCRT upper bound $\overline{R}_{i,b,p}$ of the replica of $v_{i,b}$ in $S_{i,p}$. The redundant interference caused by v_h on both the replica of v_a in $S_{i,q}$ and the replica of v_b in $S_{i,p}$, i.e., the amount of interference caused by v_h included in the computation of both $r_{i,a,q}$ and $r_{i,b,p}$, is lower bounded by*

$$\left(\left\lceil \frac{r_{i,b,p} + J'_h}{T_h} \right\rceil + \left\lceil \frac{r_{i,a,q} + J'_h}{T_h} \right\rceil - \left\lceil \frac{r_{i,b,p} + r_{i,a,q} + J'_h}{T_h} \right\rceil \right) \cdot C'_h. \tag{2}$$

Proof. Let J'_h and T_h be the release jitter and minimum inter-arrival time of node v_h. The number of jobs of v_h considered as causing direct interference on the replica of $v_{i,a}$ in $S_{i,q}$ as part of $r_{i,a,q}$ is given by $\left\lceil \frac{r_{i,a,q}+J'_h}{T_h} \right\rceil$ (from Equation (1)). Similarly, the number of jobs of v_h considered as causing interference on the replica of $v_{i,b}$ in $S_{i,p}$ is given by $\left\lceil \frac{r_{i,b,p}+J'_h}{T_h} \right\rceil$. Since the analysis in Section 5.1 adds the WCRT upper bound of $v_{i,b}$ (as part of the release jitter of $v_{i,a}$) to $r_{i,a,q}$ to calculate the WCRT upper bound of $v_{i,a}$, it considers that, in total, $\left\lceil \frac{r_{i,a,q}+J'_h}{T_h} \right\rceil + \left\lceil \frac{r_{i,b,p}+J'_h}{T_h} \right\rceil$ jobs of v_h participate to the WCRT upper bound of $v_{i,a}$. However, since $v_{i,b}$ in $S_{i,p}$ triggers the release of $S_{i,q}$, the time between the release of $v_{i,b}$ in $S_{i,p}$ and the completion of $v_{i,a}$ in $S_{i,q}$ is upper bounded by $r_{i,b,p} + r_{i,a,q}$. Therefore, the number of jobs of v_h released between the release time of $v_{i,b}$ and the completion of $v_{i,a}$ cannot be larger than $\left\lceil \frac{r_{i,b,p}+r_{i,a,q}+J'_h}{T_h} \right\rceil$. Thus, the analysis in Section 5.1 considers at least

Algorithm 5 Analysis improvements for the derivation of the WCRT of a node $v_{i,a}$ in sequence $S_{i,q}$ of a task $\tau_i \in \tau$.

```
 1: procedure ReduceJitter($\tau, (i, a, q)$)
 2:     $r_{i,a,q} \leftarrow$ The solution to Equation (1) for $v_{i,a}$ in $S_{i,q}$
 3:     $J_{i,a,q} \leftarrow 0$                                ▷ Jitter of $v_{i,a}$ in $S_{i,q}$
 4:     $v_{i,s} \leftarrow \text{head}(S_{i,q})$
 5:     for all $v_{i,b} \in \text{ipred}(v_{i,s})$ do
 6:         for all $S_{i,p} \in \mathcal{S}_i \mid v_{i,b} \in S_{i,p}$ do
 7:             $r_{i,b,p} \leftarrow$ The solution to Equation (1) for $v_{i,b}$ in $S_{i,p}$
 8:             $\mathcal{V}_h \leftarrow$ The set of all nodes with higher priority than $\tau_i$ with replicas assigned
                    to both $P(S_{i,q})$ and $P(S_{i,p})$
 9:             for all $v_h \in \mathcal{V}_h$ do
10:                 $J'_h \leftarrow$ The maximum release jitter of $v_h$ as an interfering sequential task
11:                 $C'_h \leftarrow$ The WCET of node $v_h$
12:                 $I_h^{\text{redundant}} \leftarrow \left(\left\lceil \frac{r_{i,b,p}+J'_h}{T_h} \right\rceil + \left\lceil \frac{r_{i,a,q}+J'_h}{T_h} \right\rceil - \left\lceil \frac{r_{i,b,p}+r_{i,a,q}+J'_h}{T_h} \right\rceil\right) \cdot C'_h$
13:             end for
14:             $J^{\star}_{i,a,q} \leftarrow \overline{R}_{i,b,p} - \sum_{v_h \in \mathcal{V}_h} I_h^{\text{redundant}}$
15:             $J_{i,a,q} \leftarrow \max\{J_{i,a,q}, J^{\star}_{i,a,q}\}$
16:         end for
17:     end for
18:     $\overline{R}_{i,a,q} \leftarrow J_{i,a,q} + r_{i,a,q}$
19:     return $\overline{R}_{i,a,q}$
20: end procedure
```

$\left\lceil \frac{r_{i,b,p}+J'_h}{T_h} \right\rceil + \left\lceil \frac{r_{i,a,q}+J'_h}{T_h} \right\rceil - \left\lceil \frac{r_{i,b,p}+r_{i,a,q}+J'_h}{T_h} \right\rceil$ too many jobs of v_h as contributing to the WCRT of $v_{i,a}$ in $S_{i,q}$. Every such job of v_h has a WCET of C'_h. Therefore, Equation (2) is a lower bound on the redundant interference caused by v_h on both the replica of $v_{i,a}$ in $S_{i,q}$ and the replica of $v_{i,b}$ in $S_{i,p}$. ◀

We use Lemma 7 to improve the analysis in Section 5.1. We introduce an additional step in Algorithm 3 right after the WCRT upper bound $\overline{R}_{i,a,q}$ for a node $v_{i,a}$ within a sequence $S_{i,q}$ is obtained (i.e., right after Line 6). The additional analysis step computes a reduced value for the release jitter of the sporadic task modeling the sequence $S_{i,q}$ in the analysis of $v_{i,a}$ by discounting redundant interference caused by higher-priority nodes that interfere both with immediate predecessors of head$(S_{i,q})$, whose WCRT upper bounds determine the release jitter of $S_{i,q}$, and with $S_{i,q}$ itself.

Algorithm 5 details how the WCRT upper bound $\overline{R}_{i,a,q}$ is updated for the node $v_{i,a}$ within the sequence $S_{i,q}$, by computing a reduced release jitter $J_{i,a,q}$ for $S_{i,q}$. In the algorithm, the jitter $J_{i,a,q}$ is initially set to 0 (Line 3). Then, at Lines 4-17, the procedure examines each immediate predecessor of head$(S_{i,q})$ to determine which predecessors may generate the largest release jitter $J_{i,a,q}$ for $S_{i,q}$. For every predecessor $v_{i,b}$ of head$(S_{i,q})$, and for every sequence $S_{i,p}$ in $\mathcal{S}_i$ containing a replica of $v_{i,b}$, a candidate value $J^{\star}_{i,a,q}$ for the release jitter is obtained by subtracting redundant interference from the WCRT upper bound $\overline{R}_{i,b,p}$ (Lines 5-16). Specifically, the total redundant interference on $v_{i,a}$ in $S_{i,q}$ and $v_{i,b}$ in $S_{i,p}$ to be subtracted from $\overline{R}_{i,b,p}$ is derived by first identifying all nodes with higher priority than τ_i that have at least one replica assigned to processor $P(S_{i,q})$ and one replica assigned to processor $P(S_{i,p})$, meaning that they contribute interference in the computation of both $r_{i,a,q}$ and $r_{i,b,p}$. For every such higher-priority node v_h, we use Equation (2) to compute the redundant interference and discount it from $\overline{R}_{i,b,p}$ to obtain $J^{\star}_{i,a,q}$. The value of $J_{i,a,q}$ is then set to the maximum between $J^{\star}_{i,a,q}$ and the current value of $J_{i,a,q}$. Thus, the final value of $J_{i,a,q}$ is given by the maximum release jitter candidate among those computed for all replicas of every immediate predecessor of head$(S_{i,q})$. As a result, the replicas of the immediate predecessor of head$(S_{i,q})$ that produced a candidate release jitter value equal to the final value of $J_{i,a,q}$ satisfy the assumption in Lemma 7. Finally, given the resulting value of $J_{i,a,q}$, the WCRT upper bound of $v_{i,a}$ within the sequence $S_{i,q}$ is recomputed as $\overline{R}_{i,a,q} = J_{i,a,q} + r_{i,a,q}$ (Line 18).

6 Experimental results

This section presents the results of an experimental evaluation of the proposed replication-based scheduling approach, including a comparison with state-of-the-art variants of federated scheduling [23] and partitioned scheduling [2].

6.1 Experimental setup

The experimental campaign is based on the analysis of randomly generated task sets. The task set generation procedure works as follows. The number of tasks n composing each task set τ is a generation parameter which is fixed for each experiment. For each parallel task τ_i, the topology of the DAG G_i is generated according to the technique by Melani et al. [24]. This approach generates a series-parallel graph with multiple levels of nested parallel branches in a recursive approach which starts from an initial graph composed of two nodes and then recursively expands non-terminal nodes to either terminal nodes or additional parallel subgraphs, until a maximum recursion depth is reached. The maximum recursion depth is modeled as a generation parameter n_{rec}, and another generation parameter p_{par} is used to represent the probability with which a non-terminal node is expanded to a parallel subgraph within the recursion. The level of parallelism of the parallel subgraph is controlled with an additional parameter, n_{par}. In particular, the number of branches to which a node is expanded is selected from the discrete uniform distribution $[2, n_{par}]$.

Given the value of the system utilization U, the UUniFast algorithm by Bini and Buttazzo [10] was used to generate the utilization U_i for each task $\tau_i \in \tau$. In particular, UUniFast is used to uniformly select n real values $\hat{U}_i \in [0, 1]$ such that $\sum_{i=1}^{n} \hat{U}_i = 1$; then, the utilization U_i of each task τ_i is set to $U_i = U \cdot \hat{U}_i$. Once the DAG topology G_i of a task τ_i is generated, the minimum inter-arrival time T_i of τ_i is selected from a discrete uniform distribution with range $[T_{min}, T_{max}]$, where T_{min} and T_{max} are generation parameters. The deadline of each task τ_i is set to $D_i = T_i$ (implicit deadlines). The cumulative WCET C_i of τ_i is set to $C_i = U_i \cdot T_i$; then, the WCET $C_{i,a}$ of each node $v_{i,a} \in V_i$ is generated using the UUniFast algorithm by distributing the WCET C_i among the nodes of G_i in such a way that $\sum_{v_{i,a} \in V_i} C_{i,a} = C_i$. In particular, UUniFast is used to uniformly select n_i real values $\hat{C}_{i,a} \in [0, 1]$ such that $\sum_{v_{i,a} \in V_i} \hat{C}_{i,a} = 1$; then, the WCET $C_{i,a}$ of each node $v_{i,a} \in V_i$ is set to $C_{i,a} = C_i \cdot \hat{C}_{i,a}$. Finally, the priority level π_i of each task τ_i is assigned according to the Rate Monotonic algorithm, which assigns higher priority levels to tasks with smaller minimum inter-arrival time T_i.

In order to limit the amount of non-feasible task sets generated for the experiments, the generation procedure for each task τ_i is repeated (up to 5000 times) in case either **(i)** $C_{i,a} > D_i$ holds for some node $v_{i,a} \in V_i$; or **(ii)** $\sum_{v_{i,a} \in V(\lambda)} C_{i,a} > D_i$ holds for some path $\lambda \in$ path(G_i).

In each experiment, the number of processors m in the platform was fixed to a specific value, and the system utilization U was varied between 0 and m in increments of 0.5. For each value of U, 100 task sets were generated and analyzed using the replication-based, federated, and partitioned scheduling approaches. The performance metric considered in the experiments is the schedulability ratio with respect to the system utilization U, computed as the ratio between the number of task sets deemed schedulable by a given analysis approach and the total number of task sets evaluated for the utilization point U.

The following scheduling approaches and respective analyses were tested: **(RBS-WBF)** replication-based scheduling, testing all the available heuristics (Worst Fit, Best Fit, First Fit) and applying the analysis in Section 5.1; **(RBS-WBF-I)** like RBS-WBF, but leveraging the improved analysis in Section 5.2; **(RBS-DUAL)** replication-based scheduling using the variant allocation approach which treats high-utilization and low-utilization tasks differently, testing the Worst Fit heuristic and applying the analysis in Section 5.1; **(RBS-DUAL-I)** like RBS-DUAL, but leveraging the improved analysis in Section 5.2; **(RBS-OR)** logic OR combination of RBS-WBF and RBS-DUAL, which deems a task set schedulable if it is deemed schedulable by at least one of RBS-WBF and RBS-DUAL; **(RBS-OR-I)** logic OR combination of RBS-WBF-I and RBS-DUAL-I; **(FED-WBF)** federated scheduling [23], allocating low-utilization tasks by decreasing utilization order and testing all the standard Worst Fit, Best Fit, and First Fit heuristics; and **(PART-EDD)** partitioned scheduling, analyzed using an approach leveraging mixed-integer linear programming following a transformation to the event-driven delay-induced task model, and with allocation determined according to the best performing variant of the pseudo-federated approach, which treats high-utilization and low-utilization tasks similarly to federated scheduling and distributes nodes of low-utilization tasks on underutilized dedicated processors of high-utilization tasks [2].

6.2 Experimental results

Figure 4 reports the results of the experiments. For all system configurations, the values of n_{rec}, p_{par}, T_{min}, and T_{max} were set to $n_{rec} = 2$, $p_{par} = 0.8$, $T_{min} = 100$, and $T_{max} = 1000$, while the other parameters (m, n, n_{par}) were varied among the experiments, and their value for each experiment is reported above the corresponding graph. The PAR-FEAS curve represents the ratio of task sets which satisfy both feasibility conditions in the generation, i.e., $C_{i,a} \leq D_i$ for all nodes $v_{i,a} \in V_i$, and $\sum_{v_{i,a} \in V(\lambda)} C_{i,a} \leq D_i$ for all paths $\lambda \in \text{path}(G_i)$. This curve represents an upper bound on the attainable performance of the evaluated scheduling and analysis approaches.

The results for $n_{par} = 3$ (Figures 4(a-c)) share a common overall trend, with replication-based scheduling outperforming both federated and partitioned scheduling by a significant margin. For what concerns replication-based scheduling, the most significant performance loss occurs at utilization values U that are above 50% of the available system utilization m across all processors, with the overall performance decline starting at around 37.5% of m. Partitioned scheduling follows with an intermediate level of performance, while federated scheduling exhibits the worst performance among the evaluated approaches. The same general pattern is observed in the experiments in which a larger number of nodes is generated for each task, i.e., when $n_{par} = 5$ (Figures 4(d-f)). In this case, the drop-off for replication-based scheduling with respect to U occurs again at around 37.5% of m, but with a sharper performance loss after that point. Partitioned scheduling suffers a similar loss in performance, whereas federated scheduling exhibits robust performance with respect to the previous case. Across all experiments, the two tested allocation approaches for replication-based scheduling, RBS-WBF and RBS-DUAL, show comparable performance, with the combined approach

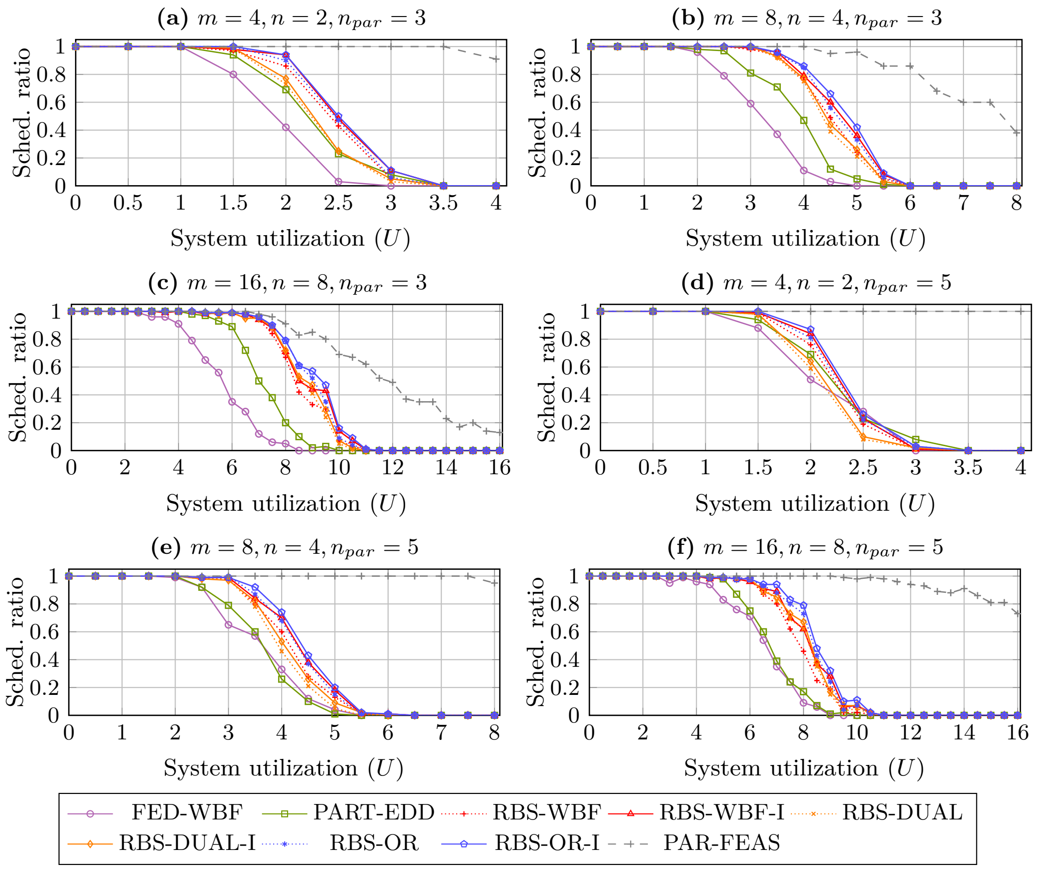

Figure 4 Schedulability ratio with respect to the system utilization U obtained for different system configurations.

RBS-OR granting an additional edge in performance, meaning that neither of the methods dominates the other. Finally, in all the tested scenarios, the RBS-WBF-I, RBS-DUAL-I, and RBS-OR-I approaches utilizing the improved analysis in Section 5.2 provided slightly improved performance with respect to the corresponding RBS-WBF, RBS-DUAL, and RBS-OR approaches adopting the analysis in Section 5.1.

Overall, the experiments show that replication-based scheduling can outperform both partitioned and federated scheduling by a large margin across several system configurations.

7 Conclusions and future work

This paper presented replication-based scheduling, a specialized scheduling approach for parallel real-time tasks executing on a multiprocessor platform which leverages the internal topology of the DAG of each task to provide enhanced schedulability performance with limited expected runtime overhead and analysis complexity. In addition to the overall scheduling paradigm, design-time allocation strategies were discussed, and a response-time analysis for the case of fixed-priority preemptive scheduling was provided. Experimental results showed that replication-based scheduling significantly outperforms state-of-the-art variations of both federated and partitioned scheduling. Future work includes implementing replication-based scheduling in a real-time operating system, investigating further improvements to the provided analysis, and exploring variants of replication-based scheduling supporting Earliest Deadline First scheduling and non-preemptive execution of nodes. Finally, given the flexibility of the proposed scheduling framework, future work should also evaluate further variations to the design-time allocation algorithms.

References

1 Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, 1998.

2 Federico Aromolo, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Event-driven delay-induced tasks: Model, analysis, and applications. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021)*, pages 53–65. IEEE, 2021.

3 Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering*, 8(5):284–292, 1993.

4 Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, pages 215–224. IEEE, 2013.

5 Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, pages 97–105. IEEE, 2014.

6 Sanjoy Baruah. The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the 18th IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE 2015)*, pages 1323–1328. IEEE, 2015.

7 Sanjoy Baruah. Federated scheduling of sporadic DAG task systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2015)*, pages 179–186. IEEE, 2015.

8 Andrea Bastoni, Björn Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)*, pages 33–44, 2010.

9 Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 14–24. IEEE, 2010.

10 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

11 Alessandro Biondi and Youcheng Sun. On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling. *Real-Time Systems*, 54:515–536, 2018.

12 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic DAG task model. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, pages 225–233. IEEE, 2013.

13 Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018)*, pages 421–433. IEEE, 2018.

14 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, and Georg von der Brüggen. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207, 2019.

15 Son Dinh, Christopher Gill, and Kunal Agrawal. Efficient deterministic federated scheduling for parallel real-time tasks. In *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2020)*, pages 1–10. IEEE, 2020.

16 José Fonseca. *Multiprocessor Scheduling and Mapping Techniques for Real-Time Parallel Applications*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2019.

17 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Improved response time analysis of sporadic DAG tasks for global FP scheduling. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS 2017)*, pages 28–37. ACM, 2017.

18 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Schedulability analysis of DAG tasks with arbitrary deadlines under global fixed-priority scheduling. *Real-Time Systems*, 55(2):387–432, 2019.

19 José Fonseca, Geoffrey Nelissen, Vincent Nélis, and Luís Miguel Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES 2016)*, pages 1–10. IEEE, 2016.

20 Xu Jiang, Nan Guan, Haochun Liang, Yue Tang, Lei Qiao, and Wang Yi. Virtually-federated scheduling of parallel real-time tasks. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium (RTSS 2021)*, pages 482–494. IEEE, 2021.

21 Xu Jiang, Nan Guan, Xiang Long, and Wang Yi. Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS 2017)*, pages 80–91. IEEE, 2017.

22 Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 259–268. IEEE, 2010.

23 Jing Li, Jian-Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, pages 85–96. IEEE, 2014.

24 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, pages 211–221. IEEE, 2015.

25 Alessandra Melani, Maria A Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quinones, and Giorgio Buttazzo. A static scheduling approach to enable safety-critical OpenMP applications. In *Procedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC 2017)*, pages 659–665. IEEE, 2017.

26 Amir Nahir, Ariel Orda, and Danny Raz. Replication-based load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):494–507, 2015.

27 Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 321–330. IEEE, 2012.

28 Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

29 Niklas Ueter, Georg Von Der Brüggen, Jian-Jia Chen, Jing Li, and Kunal Agrawal. Reservation-based federated scheduling for parallel real-time tasks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018)*, pages 482–494. IEEE, 2018.

30 Roberto Vargas, Eduardo Quinones, and Andrea Marongiu. OpenMP and timing predictability: A possible union? In *Proceedings of the 18th IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE 2015)*, pages 617–620. IEEE, 2015.