technische universität
dortmund

DAG Scheduling with Execution Groups

Junjie Shi[1] and Mario Günzel[1] and Niklas Ueter[1] and Georg von der Brüggen[1] and Jian-Jia Chen[1,2]

[1]TU Dortmund University, Germany

[2]Lamarr Institute for Machine Learning and Artificial Intelligence, Germany

Citation: RTAS2024.05

BibTeX:

```
@inproceedings{DBLP:conf/rtas/ShiGUBC24,
  author       = {Junjie Shi and
     Mario G{\"u}nzel and
                  Niklas Ueter and
                  Georg von der Br{\"u}ggen and
                  Jian{-}Jia Chen},
  title        = {DAG Scheduling with Execution Groups},
  booktitle    = {30th {IEEE} Real-Time and Embedded Technology and Applications Symposium,
                  {RTAS}},
  publisher    = {{IEEE}},
  year         = {2024},
  url          = {},
  doi          = {}
}
```


CS 12 computer science 12

DAG Scheduling with Execution Groups

Junjie Shi[1], Mario Günzel[1], Niklas Ueter[1], Georg von der Brüggen[1] and Jian-Jia Chen[1,2]
[1] TU Dortmund University, Germany
[2] Lamarr Institute for Machine Learning and Artificial Intelligence, Germany

***Abstract*—In many modern safety-critical cyber-physical systems, such as in the automotive or robotic domain, the application complexity requires the use of multi-core platforms to execute all workloads under strict hard real-time constraints. The sporadic DAG task model is a parallel task model adept at representing tasks comprised of subtasks, which possess internal data flow and precedence constraints induced by synchronization. A significant challenge to the system's performance and its real-time verification stems from the communication-centric nature of applications in these domains. Inter-core communication, required for data sharing among subtasks across different cores, depends on either a shared bus or a network-on-chip, culminating in significant overhead due to latency, congestion, and synchronization. To improve performance and reduce these overheads, it is advantageous to execute subtasks, those that either exchange large volumes of data or access the same data, on a singular physical processor, thereby utilizing more efficient intra-core communication.**

In this paper, we tackle this issue by introducing the DAG task model with execution groups, incorporating a constraint that mandates the execution of grouped subtasks on the same processor. We provide an analysis of worst-case response times and propose optimizations for our DAG task model with execution groups, subsequently evaluating our approach against existing solutions. The evaluation results demonstrate that our approach, even with the imposition of group execution constraints, remains competitive in comparison to existing approaches that do not take group execution constraints into account. Additionally, we explore implementation strategies and potential extensions for multi-task systems.

***Index Terms*—DAG Tasks, Gang Scheduling, Cyber-Physical Systems, Real-Time Systems**

I. Introduction

In many cyber-physical systems such as in the automotive or robotic domain, real-time requirements must be met to ensure system safety and performance. The multitude of generated sensor data in modern systems requires sensor data processing algorithms, sensor fusion algorithms, and optimization-based control algorithms. Consequently, *heavy* and *complex* workloads have to be executed under strict timing constraints to meet the application's quality-of-service (QoS) requirements. Hence, multi-core and multi-processor systems have become the standard computing platforms, due to their parallel and concurrent computing capabilities, facilitating the fulfillment of these requirements.

In the context of hard real-time systems, formal guarantees for the worst-case response time (WCRT) of each task must be provided. The formal guarantee (or analysis) is based on a task model, the execution model, and the task parameters, such as activation model (inter-arrival times) and worst-case execution times (WCET). In regard to parallel task models, the sporadic DAG task model is a widely studied and generic parallel task model, capable of modeling tasks which are composed of subtasks with internal precedence constraints. The precedence constraints can be due to a data producer consumer relation, or due to synchronization barriers, determining the partial execution order of the subtasks.

When scheduling DAG task sets, the objective is to efficiently utilize the parallelism provided by multiprocessors for task sets with inter- and intra-task parallelism, while guaranteeing that each task meets its deadline. Parallelism can be categorized into inter-task parallelism (which refers to the parallel execution of distinct tasks, each of which executes sequentially) and intra-task parallelism (which refers to the parallel execution of a single task). Intra-task parallelism requires task models with subtask level granularity that can be scheduled in parallel, e.g., Fork-join models [30], synchronous parallel task models, or DAG (directed-acyclic graph) based task models. A plethora of real-time scheduling algorithms and response time analyses thereof have been proposed, e.g., for generalized parallel task models [37], and for DAG (directed-acyclic graph) based task models [3], [4], [12], [18], [19], [25], [33], [49]. For DAG-based task models, improvements in the response time analyses can be categorized into analyses that improve inter-task interference, e.g., in [12], [18], or intra-task interference such as in [25], [26], [32], [49]. In general, intra-task interference analyses build upon the interference analysis along the execution of the envelope (also known as critical path or key path).

A common critique in the context of multiprocessor, and parallel task scheduling models in particular, is that the complex interference patterns on the shared resources, such as memory and cache, are neglected despite having been shown to have a significant impact on the efficiency and execution time of parallel tasks [2], [9], [10], [35], [48], [50].

In multi-core systems, there are two typical approaches to implement inter-core communication, namely memory sharing (or pointer passing) and physical copying of data (or data moving). In the case of memory sharing, addressing and accesses to shared memory must be managed. For instance, each core has a local physical memory that can be addressed — on that respective core — by using the core-local memory map. The respective core-local physical memories can be addressed from other cores by using global addresses, which will then be routed through a crossbar to access the physical memory. Data-copying (or data-moving-based) approaches rely on the

sending core to copy the data to the receiving core and to notify the receiving core once the transfer has finished. Data can be moved to fast accessible memory, e.g., L1/L2-cache, however the transfer latency has to be considered. Intra-core communication, in contrast, can be achieved through the use of cache memory — if the communicated data size does not exceed the L1-cache size — where each core has its own fast L1-cache to store frequently accessed data. Notably, many modern processors also support more efficient intra-core sharing of data; for instance, via hardware support for thread-level parallelism, which allows multiple threads to share data more efficiently by providing dedicated caches and other resources.

In multiprocessor multi-threaded systems, cache memory primarily contributes negatively to schedulability by increasing WCET values. This increase is affected by evictions from concurrent execution as well as cache coherency delays across cores via communication. To address this issue, various cache-aware multicore real-time scheduling algorithms have been proposed and studied [6], [7], [11], [46], [47]. Feljan and Carlons [17] demonstrated that inter-core communication could take up to three times longer than intra-core communication — if the shared data fits into L1-cache. In our motivated application domain of robotics and automotive systems, many shared data satisfy this requirement. According to the real world automotive benchmarks by Kramer et al. [29], more than 90% of the shared labels are of *atomic data type*, i.e., 1, 2, and 4 bytes, which fit the L1-cache in most modern multiprocessor architectures.

Most applications in the automotive and robotics domain are communication-centric, i.e., accesses to shared data by subtasks are very frequent and even occur across different tasks. Typically, communication overheads are the dominating factor for communication-centric tasks' WCET estimates [1]. Consequently, communication overheads in parallel applications have a significant impact on the average-case system performance, and are detrimental to resource utilization under hard real-time constraints, if not properly addressed.

Therefore, we propose to consider more restrictive DAG task models, which try to reduce the communication overheads by grouping subtasks — which share a large amount of data — to execute on the same processor. Consequently, we avoid expensive inter-core communication in favor of more efficient intra-core communication and ideally use the private L1-cache of the respective processor for most of the time.

From an analysis perspective however, subtask grouping degrades parallelism, because executing certain subtasks on the same processor requires them to be executed sequentially, which must hence be handled carefully.

To the best of our knowledge, the only other DAG task model addressing subtask group execution constraints is the *tied* DAG task model from OpenMP. However, our proposed model allows for the grouping of arbitrary subtasks, whereas the tied DAG task model restricts groupings to specific subpaths of a DAG, tying them to a particular thread or processor. Moreover, while there are published response time analyses for the tied DAG task model [38], [39] specific to OpenMP scheduling, they predominantly address the response time problem associated with the extended Breadth First Scheduling (BFS*) algorithm [39].

Contributions. To the best of our knowledge, our presented work is the first to focus on complying with design constraints where certain vertices of DAG tasks must be executed on the same core. This decision might be made by system designers to mitigate unacceptable overheads in communication, synchronization, memory access, and other factors inherent in the general sporadic DAG task model on multiprocessor systems. We address this problem by imposing group execution constraints and provide hard real-time response-time analyses thereof.

We summarize our contributions as follows:

- We introduce an extension to the DAG task model in Section II, namely the EG-DAG, which accommodates arbitrary group execution constraints for subtasks. This model ensures that group-constrained subtasks must be executed on the same physical core, effectively reducing communication overhead among different cores. Unlike conventional models, EG-DAG does not enforce a specific group-to-processor binding but dynamically binds a group to a processor when the first subjob of that group executes on it, maintaining this binding for the duration of the respective DAG task instance.
- We propose a scheduling mechanism tailored to the EG-DAG model in Section III. We present a worst-case response time analysis in Section IV along with a group merging heuristic in Section V in the case that the number of groups exceeds the number of available cores.
- We propose an algorithm to improve the WCRT analysis in Section VI, by adding additional edges for subjobs within a group.
- We provide practical insights into the implementation of the EG-DAG model and its associated scheduling mechanisms in Section VII.
- We conduct a comprehensive numerical evaluation in Section VIII, comparing the performance of our proposed approaches against existing algorithms for DAG task scheduling without group execution constraints. Our evaluation results demonstrate that the proposed approaches are highly competitive with existing algorithms, with performance differences typically less than 5%, especially when the probability of edges between subtasks is relatively high.

II. System Model and Specification

In the proposed model, similar to the standard DAG model, each task is composed of several subtasks. Those subtasks need to comply to precedence constraints, that is, for example, if a subtask utilizes data produced by another subtask, then those subtasks should be executed one after the other. Moreover, and distinct from the standard DAG model, we assume that some subtasks are required to be executed on the same processor — we call this restriction a *group execution*

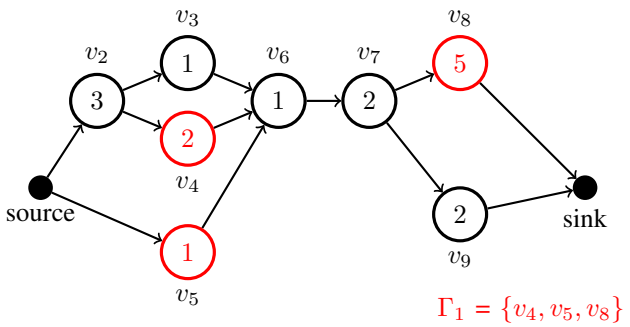


Fig. 1: An exemplary DAG under group execution constraints. Edges denote precedence constraints between subtasks. The execution group Γ_1 contains the 3 red subtasks.

constraint. In Figure 1, a DAG with one group is illustrated. In the following we first define the DAG task model, and afterwards extend it with the group execution constraints.

A. DAG Task Model

A sporadic DAG task τ is specified by the tuple

$$\tau := (G, D, T) \tag{1}$$

where $G = (V, E)$ is a Directed Acyclic Graph (DAG) describing the precedence constraints among subtasks, D is the tasks relative deadline, and T is the tasks minimum inter-arrival time. In the following, we formalize those variables further and make useful definitions for the analysis.

Inter-Arrival Time T. Task τ releases an infinite number of instances, called jobs $(J_\ell^\tau)_{\ell \in \mathbb{N}}$, and any two consecutive job releases are separated by at least T time units.

Relative Deadline D. The relative deadline of task τ is D. Each job J_ℓ^τ must be finished before its absolute deadline $d_\ell^\tau = a_\ell^\tau + D$, where a_ℓ^τ is the arrival time of job J_ℓ^τ. The deadline is assumed to be constrained, i.e., $D \leq T$.

Precedence Constraints (formalized by $G = (V, E)$). The DAG $G = (V, E)$ describes a finite set of subtasks with precedence constraints. Specifically, each DAG is composed of a finite set of subtasks $v \in V$, and a finite set of edges $(u, v) \in E \subseteq V \times V$. We assume that the graph G is acyclic and weakly connected. The number of elements in V is denoted as $|V|$. When job J_ℓ^τ is released, one subjob J_ℓ^v of each subtask v is released which must be executed according to the precedence constraints, i.e., if the edge $(u, v) \in E$ exists then J_ℓ^v can only be executed once J_ℓ^u is finished.

Worst-case Execution Time (formalized by vol). Each subtask $v \in V$ is associated with a worst-case execution time (WCET) $\mathrm{vol} : V \mapsto \mathbb{R}^+$, that specifies the maximum execution time for any instance of subtask v. The volume of any subset of subtasks $U \subseteq V$ is given by $\mathrm{vol}(U) := \sum_{v \in U} \mathrm{vol}(v)$, including the case where $U = V$, representing the total volume of the DAG task. It is assumed that every subtask has a positive execution time, i.e., $\mathrm{vol}(v) > 0$ for all $v \in V$.

Path $\pi \in \Psi(G)$. For each subtask $v \in V$, the set of predecessors of v and the set of successor of v is defined as $pred(v) = \{w \in V | (w, v) \in E\}$ and $succ(v) = \{w \in V | (v, w) \in E\}$, respectively. A path is an ordered set of subtasks $\pi = \{v_1, \ldots, v_n\}$, such that $pred(v_1) = \emptyset$, $succ(v_n) = \emptyset$, and $\forall k \in \{1, \ldots, n-1\}, v_k \in pred(v_{k+1})$. For a given DAG task, the set of all possible paths is denoted as $\Psi(G)$, and the critical path is defined as $\pi^* := \arg\max_{\pi \in \Psi(G)} \mathrm{vol}(\pi)$. The length of the critical path is denoted as $\mathrm{vol}(\pi^*)$.

B. Execution Groups

Inter-core communication can take up to three times longer than intra-core communication [17]. In practical scenarios, this necessitates the strategic placement of interdependent subtasks (i.e., those involved in extensive data sharing) on the same core, thereby mitigating communication overhead. Furthermore, such a placement strategy promotes cache efficiency, as it enables the aggregation of substantial data in the cache, subsequently accessible by multiple subtasks within the same core. To facilitate this optimization, this work introduces the concept of *execution groups*, denoted as $\{\Gamma_1, \ldots, \Gamma_\gamma\}$, to systematically group subtasks that benefit from co-location on the same processing unit.

Definition 1. *An* execution group $\Gamma_i \subseteq V$ *is a set of subtasks that are constrained to be executed on the same processor.*

We assume that each subtask is part of at most one execution group, that is, $\Gamma_i \cap \Gamma_j = \emptyset$ for all $i \neq j$. The set of all subtasks which are not part of any execution group is denoted by

$$\Gamma^\perp := V \setminus \bigcup_{i=1}^{\gamma} \Gamma_i. \tag{2}$$

We assume that the execution groups are known (e.g., given by the application design).

Note that according to our definition a group may be executed on different cores for different jobs. For instance, the subtasks in Γ_1 may be executed on processor 4 for the first and on processor 3 for the second job. However, for each job, the processor assignment of a group is fixed as soon as its first subjob starts executing on any processor.

One exemplary DAG task with execution groups is shown in Figure 1. In this example, subtasks v_4, v_5, and v_8 share a lot of data and should be executed on the same core. As a result, the execution group $\Gamma_1 = \{v_4, v_5, v_8\}$ is specified.

III. Scheduling Mechanism

In this section we discuss the mechanisms to schedule one DAG task on a set of m dedicated processors. We refer to Section IX for the extension to the scheduling mechanism involving several DAG tasks.

The scheduling mechanism is based on the list scheduling algorithm. In the original list scheduling algorithm, all currently ready subjobs of a DAG job are maintained in a list. Whenever a processor runs idle, the first subjob in the list is dispatched and executed on that processor until completion.

To account for the group execution constraints, we consider a list scheduling algorithm with additional lists — one for each execution group. Whenever all predecessors of a subjob finish, the subjob is moved to the corresponding ready queue. At each point in time, a processor executes jobs from the ready queues according to the following description.

Definition 2 (List Scheduling with Execution Groups)**.** *For each task instance (job), each subjob $v \in V$ is scheduled on m dedicated processors according to the following rules:*

- *A subjob is ready if all preceding subjobs have executed until completion, i.e., the subjob arrival time a_v for each subjob v is given by $\max\{f_w \mid (w,v) \in E\}$. A subjob arrives to its respective per-group ready list Queue:Γ_i for $i \in \{1,\dots,\gamma\}$ or to the ungrouped ready list Queue:$\Gamma^{\perp}$. An arrived but not yet finished subjob is called* pending.
- *Subjobs from per-group ready lists are initially admitted executing on any reservation. However, at the first time that any subjob of a per-group ready list Queue:Γ_i is executed on some of the m processors, then all subjobs that arrive to Queue:Γ_i are restricted to execute on that reservation. Once a group is restricted to a processor, it cannot be used for another group, i.e., no two subjobs of different groups can execute on the same processor.*
- *Subjobs from Queue:$\Gamma^{\perp}$ are admitted to execute on any of the m processors.*
- *At any time t, on each of the m processors, a group subjob admitted for this processor is executed (if such a job exists), otherwise a non-group subjob is executed (if such a job exists). When a group subjob arrives that is admitted for the processor, the execution of a non-group subjob on that processor is preempted.*

Figure 2 illustrates the proposed scheduling mechanism. The Mapping between processors and groups is depicted as a table, and the processor follows the rules formulated in the box. Please note that the actual implementation of the scheduling mechanism is specified in Section VII (including a discussion on how the processor behavior can be ensured).

IV. Response-Time Analysis

In this section, we provide a response-time upper bound for an arbitrary job J_ℓ^τ of the DAG task τ under the proposed scheduling mechanism from Section III. In the end we argue that the response-time bound is independent of the job number ℓ and therefore the response-time bound is an upper bound on the worst-case response time of task τ as well.

Let a and f be the arrival time and finishing time of job J_ℓ^τ, respectively, and let a_v and f_v be the arrival time and finishing time of subjob J_ℓ^v, respectively. Our analysis is based on the *envelope* of the interval $[a,f)$. The envelope partitions the whole interval $[a,f)$ into subintervals. During each of those subintervals, one subjob of J_ℓ^τ is pending.

Definition 3 (Envelope)**.** *We define the* envelope *of G as a list intervals*

$$[a_{v_1}, f_{v_1}), [a_{v_2}, f_{v_2}), \dots, [a_{v_p}, f_{v_p}) \tag{3}$$

(with $v_i \in V$ and $p \in \{1,2,\dots,|V|\}$) backwards in an iterative manner as follows:

1) *$J_\ell^{v_p}$ is the subjob of J_ℓ^τ with the maximal finishing time.*
2) *For all $i \in \{p, p-1, \dots, 2\}$, $J_\ell^{v_{i-1}}$ is the subjob with the maximal finishing time such that $v_{i-1} \in pred(v_i)$ is a predecessor of v_i.*
3) *v_1 is a source node, i.e., it has no predecessor.*

We call $\pi_e := \langle J_\ell^{v_1}, J_\ell^{v_2}, \dots, J_\ell^{v_p} \rangle$ the envelope path*.*

We note that the definition of an envelope for a DAG job may not be unique if there are subjobs with the same finishing time. In that case, ties can be broken arbitrarily.

By definition, $f_{v_p} = f$ and all envelope path subjobs are executed one after the other, that is $f_{v_i} \leq a_{v_{i+1}}$ for all $i = 1,\dots,p-1$. Moreover, at each time during the interval $[a,f)$ there is always a subjob of the envelope path being executed, except if (i) all processors are busy executing non-envelope subjobs, or (ii) the envelope path subjob is delayed by the scheduling mechanism due to the execution group constraint. We formalize this observation.

Definition 4 (Envelope, Busy, and Delay Interval)**.** *We partition the interval $[a,f)$ into three disjoint subsets $I_e \sqcup I_b \sqcup I_d = [a,f)$ as follows:*

- *I_e: An envelope path subjob is* executed*.*
- *I_b: All processors are* busy *executing jobs which are not in the envelope path.*
- *I_d: An envelope path subjob is delayed due to group execution constraints, i.e., the subjob is not scheduled although there are empty processors.*

By utilizing the Lebesgue measure (length) λ of the subsets, we can formulate a response time of J_ℓ^τ as:

$$f - a = \lambda(I_e \sqcup I_d \sqcup I_b) \tag{4}$$
$$= \lambda(I_b) + \lambda(I_e) + \lambda(I_d) \tag{5}$$

In the remainder of this section, we construct safe upper bounds for $\lambda(I_b)$, $\lambda(I_e)$ and $\lambda(I_d)$ to derive an upper bound on the worst-case response time. We start with the bound for I_b.

Lemma 1 (Bound for I_b)**.** *The measure of the set I_b (i.e., the time that all processors are executing non-envelope subjobs) is upper bounded by*

$$\lambda(I_b) \leq \frac{vol(V) - \lambda(I_e) - \lambda(I_d)}{m} \tag{6}$$

Proof. To prove this lemma, we examine the amount of executed workload during the sets I_b, I_e and I_d.

- During I_e, envelope jobs are executed. Therefore, the executed workload during I_e is *at least* $\lambda(I_e)$.
- During I_d, the envelope subjob is delayed due to group execution constraints. That means that a subjob of the envelope path is waiting for its dedicated processor to finish the execution of another subjob. In particular, there is workload of at least one subjob being executed during I_d. Therefore, the executed workload during I_d is *at least* $\lambda(I_d)$.

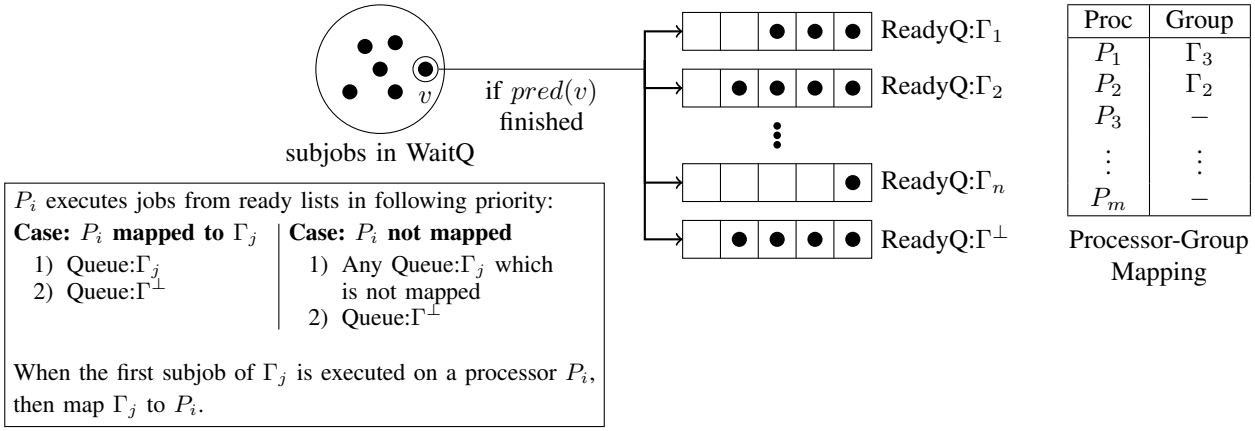


Fig. 2: Explanation of scheduling concept of a DAG job on m processors.

- During I_b, all m processors are busy. Therefore, the executed workload during I_b is *at least* $m \cdot \lambda(I_b)$.

In total, the amount of executed workload during $[a, f) = I_e \sqcup I_d \sqcup I_b$ is *at least* $\lambda(I_e) + \lambda(I_d) + m \cdot \lambda(I_b)$. Moreover, $\mathrm{vol}(V)$ is an *upper bound* on the executed workload during $[a, f)$. We conclude

$$\mathrm{vol}(V) \geq \lambda(I_e) + \lambda(I_d) + m \cdot \lambda(I_b) \tag{7}$$

which is equivalent to Equation (6). □

Second, we derive a bound on the amount of time that the envelope is executed.

Lemma 2 (Bound for I_e)**.** *The measure of the set I_e (i.e., time that the envelope is executed) is upper bounded by the volume of the envelope subjobs $v_1, \ldots, v_p \in \pi_e$. That is,*

$$\lambda(I_e) \leq \mathrm{vol}(\pi_e) = \sum_{i=1}^{p} \mathrm{vol}(v_i). \tag{8}$$

Proof. At each time during I_e an envelope path subjob is executed. Each subjob v_i can be executed for at most its worst-case execution time, that is, for $\mathrm{vol}(v_i)$ time units. In consequence, the measure of I_e is upper bounded by $\sum_{i=1}^{p} \mathrm{vol}(v_i)$. □

Last, we determine the bound for the delay induced by the execution groups.

Lemma 3 (Bound for I_d)**.** *The measure of I_d (i.e., when envelope subjobs are delayed due to group execution constraints) is upper bounded by $\Delta(\pi_e) := \sum_{i=1}^{p} \Delta(v_i)$, where*

$$\Delta(v_i) := \begin{cases} 0 & v_i \in \Gamma^{\perp} \\ \sum\limits_{w \in paral(v_i) \bigwedge w \in \Gamma_j} vol(w) & v_i \in \Gamma_j \end{cases} \tag{9}$$

and where $paral(v_i)$ is the set of potentially parallel subjobs*, that is, $paral(v_i)$ is the set of all $w \in V$ such that there exists no path from v_i to w or from w to v_i.*

Proof. For each envelope subjob v_i for $i \in \{1, \ldots, p\}$, if $v_i \in \Gamma^{\perp}$ then it can execute on any idle processor and thus v_i can not be delayed. Hence, the cumulative amount of time that v_i is delayed is given by $\Delta(v_i) = 0$.

Conversely, if v_i is part of an execution group Γ_j that is mapped to a processor P_k, then it may be delayed while P_k is busy executing other subjobs. We distinguish the delay by different types of "other" subjobs:

- Delay by subjobs in $\Gamma^{\perp}$: There is no delay by other subjobs in $\Gamma^{\perp}$ because they are executed with lower priority and would be delayed when the envelope path subjob v_i arrives.
- Delay by subjobs in $\Gamma_{j'} \neq \Gamma_j$: Since Γ_j is mapped to processor P_k, subjobs of $\Gamma_{j'}$ cannot be executed on P_k. Hence, there is no delay by subjobs of $\Gamma_{j'}$.
- Delay by subjobs in Γ_j: Delay of subjobs of the same group Γ_j is only possible if they are executed (i.e., arrived but not already finished) during a_{v_i}. This is only possible for subjobs of parallel subtasks.

Hence, we can upper bound the amount of delay by

$$\Delta(v_i) = \sum_{w \in paral(v_i) \bigwedge w \in \Gamma_j} \mathrm{vol}(w). \tag{10}$$

We conclude that the delay at each subjob $J_\ell^{v_i}$ of the envelope path is at most $\Delta(v_i)$ (from Equation (9)). Hence, the total delay is at most $\sum_{i=1}^{p} \Delta(v_i)$. □

With the results of Lemmas 1, 2 and 3, we can state the following response time bound.

Theorem 1. *The response time of J_ℓ^τ is upper bounded by*

$$vol(\pi) + \Delta(\pi) + \frac{vol(V \setminus \pi) - \Delta(\pi)}{m}, \tag{11}$$

where π is the path that maximizes $\mathrm{vol}(\pi) + \Delta(\pi)$, and $\Delta(\pi) := \sum_{v \in \pi} \Delta(v)$ is defined as in Lemma 3.

Algorithm 1 Execution Group Merging

Input: DAG task τ, group execution constraints Γ, number of available processors m;

1: **if** $|\Gamma| > m$ **then**
2: Sort the $\Gamma_i \in \Gamma$ decreasingly w.r.t the utilization;
3: Partition Γ on m slots using worst-fit;
(Merge same slot groups:)
4: **for** $i = 1, \ldots, m$ **do**
5: $\Gamma'_i := \{v \in V | v \text{ in group in slot } i\}$;
6: **end for**
7: **end if**

Algorithm 2 Adding Edges

Input: DAG $G = (V, E)$ of task τ, execution groups $\Gamma_1, \ldots, \Gamma_\gamma$;

1: **for** $v \in V$ **do**
2: Calculate the longest path π_v from source to v;
3: **end for**
4: **for** $j = 1, \ldots, \gamma$ **do**
5: Sort Γ_j according to $\text{vol}(\pi_v) - \text{vol}(v)$;
6: For two subsequent vertices v_i, v_{i+1} in Γ_j add one edge (v_i, v_{i+1}) to E;
7: **end for**

Proof. By using Lemmas 1, 2 and 3, we obtain

$$f - a \leq \lambda(I_e) + \lambda(I_d) + \frac{vol(V) - \lambda(I_e) - \lambda(I_d)}{m} \tag{12}$$

$$= \frac{vol(V) + (m-1) \cdot (\lambda(I_e) + \lambda(I_d))}{m} \tag{13}$$

$$\leq \frac{vol(V) + (m-1) \cdot (vol(\pi_e) + \Delta(\pi_e))}{m} \tag{14}$$

Since by definition $vol(\pi_e) + \Delta(\pi_e) \leq vol(\pi) + \Delta(\pi)$, we obtain:

$$f - a \leq \frac{vol(V) + (m-1) \cdot (vol(\pi) + \Delta(\pi))}{m} \tag{15}$$

$$= vol(\pi) + \Delta(\pi) + \frac{vol(V) - vol(\pi) - \Delta(\pi)}{m} \tag{16}$$

Moreover, since $\pi \subseteq V$, we can replace $vol(V) - vol(\pi)$ with $vol(V \setminus \pi)$, which yields the claim. □

The response time bound formulated in Theorem 1 is independent of the job number ℓ. Since ℓ was chosen arbitrarily, the response time bound from Theorem 1 is an upper bound on the worst-case response time of task τ.

V. Execution Group Merging

The scheduling mechanism presented in Section III is only applicable if the number of execution groups does not exceed the number of processors, i.e., $|\Gamma| \leq m$. However, in general the number of execution groups might be significantly higher than the number of processors. To address this problem, we discuss a group merging algorithm using the worst-fit strategy.

First, we sort the execution groups according to their decreasing total utilization. Second, we partition the execution groups in m slots using the worst-fit strategy, that is, each group is assigned to the slot with the lowest total utilization. The groups are redefined in a way that execution group are merged if they are in the same slot. After the group merging, the response time bound from Theorem 1 can be used to check if the execution groups allow a feasible schedule of the DAG. The merging procedure is summarized in Algorithm 1.

After the execution group merging, the number of execution groups is identical to the number of available processors m and our analysis from Section IV can be applied. The runtime of the algorithm is $O(|\Gamma| \cdot \log |\Gamma|)$ to sort the groups according to their utilization and $O(|\Gamma| \cdot m)$ for the partition, where m is the number of available processors.

VI. Adding Edges to Improve WCRT

In Section IV, we already introduced the response-time analysis for a given DAG task with execution group constraints. However, the quantification of the delay I_d in Lemma 3 is pessimistic since it needs to account for all different sequentializations of potentially-parallel subtasks. As a result, $\Delta(\pi_e) = \sum_{i=1}^{p} \Delta(v_i)$ in Equation (9) is overly pessimistic if there are several parallel subtasks. To mitigate this problem, we propose a sequentialization approach to reduce this pessimism and to tighten the analytical guarantees from Section IV further. To that end, we first motivate our approach intuitively and then present our solution. The improvement is evaluated in Section VIII.

To respect the execution group constraints in the scheduling mechanism, subjobs of the same execution group have to be executed sequentially. However, since the sequence of the subjobs of the same execution group is not clear, all scenarios must be considered in the analysis. For example, in Figure 1 there is one execution group Γ_1 with three subtasks v_4, v_5 and v_8. We know that v_8 will only be released after v_4 and v_5 have finished. Therefore, there is no delay from the execution group, i.e., $\Delta(v_8) = 0$. For the subjobs of v_4 and v_5 there might be delay from the other subjob, respectively. That is, v_5 may be delayed by the length of v_4 ($\Delta(v_5) = 2$) and conversely ($\Delta(v_4) = 1$). Since it is not clear which of these two subtasks is executed first, both delays must be considered, although it is obvious that for this simple example only one subjob can be delayed and therefore the total delay is in fact at most $\max\{1, 2\} = 2$.

Our solution to avoid this over-approximation is to artificially sequentialize subtasks of the same execution group by adding additional edges between execution group subjobs to the graph $G = (V, E)$. Please note that while adding edges to the graph changes the structure, the graph can still fulfill the same functionality as before since the original constraints are still respected. Rather, adding additional edges enforces a certain sequentialization behavior of the scheduler.

To decide where to add edges to the graph, we use the following procedure, as summarized in Algorithm 2. For each subtask $v \in V$, the longest path π_v from the source to

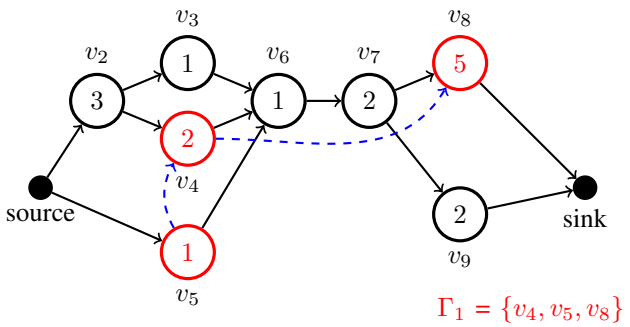


Fig. 3: The DAG task from Figure 1 with additional edges (dashed blue arrows) to improve the analytical guarantees.

v is calculated. Afterwards, the execution group is ordered according to $\text{vol}(\pi_v) - \text{vol}(v)$, where ties are broken arbitrarily. This ordering represents a likely sequence of the execution group subtasks of the DAG task. That is, if $\text{vol}(\pi_v) - \text{vol}(v)$ is shorter, then less workload must be done before the subtask v can be released, and therefore the subjob of that subtask will likely be executed first anyway. Finally, we achieve the sequentialization of each execution group Γ_j by adding edges (v_i, v_{i+1}) between two subsequent subtasks v_i and v_{i+1} of the ordered Γ_j.

For the example, in Figure 1, the execution group is ordered as (v_5, v_4, v_8) because:

- $\text{vol}(\pi_{v_4}) - \text{vol}(v_4) = 3$
- $\text{vol}(\pi_{v_5}) - \text{vol}(v_5) = 0$
- $\text{vol}(\pi_{v_8}) - \text{vol}(v_8) = 3 + 2 + 1 + 2 = 8$

Therefore, the edges (v_5, v_4) and (v_4, v_8) are added to the graph. The updated graph is illustrated in Figure 3. Please note that the edge (v_4, v_8) is redundant and can safely be removed again because there is already a path from v_4 to v_8.

To keep the DAG task well-defined, it must be ensured that after this procedure there are still no cycles in the graph.

Lemma 4. *After adding edges to the DAG task described by DAG $G = (V, E)$ according to Algorithm 2, the graph G remains acyclic.*

Proof. First, we order V according to $\text{vol}(\pi_v) - \text{vol}(v)$. If there are two vertices $v, w \in V$ with $\text{vol}(\pi_v) - \text{vol}(v) = \text{vol}(\pi_w) - \text{vol}(w)$, we either follow the sorting of Γ_j if both v and w are in Γ_j, or we decide the sorting arbitrarily otherwise.

Any edge $e = (v, w) \in E$ can either follow the ordering of V, that is v appears before w in V, or it does not follow the ordering of V. If e follows the ordering of V, then we say it is in '$\rightarrow$'-direction. Otherwise, we say it is in '$\leftarrow$'-direction.

The proof is divided in two parts:

a) In the original G there are no edges in '$\leftarrow$'-direction.
b) Adding edges in '$\rightarrow$'-direction keeps the graph acyclic.

a) In the original G there are no edges in '$\leftarrow$'-direction: Otherwise, there exists an edge $e = (v, w) \in E$ with

$$\text{vol}(\pi_v) - \text{vol}(v) \geq \text{vol}(\pi_w) - \text{vol}(w). \tag{17}$$

Because edge e exists, we can also define a path from source to w as $\tilde{\pi} := (\pi_v, w)$. Since π_w is the longest path from source to w, we have $\text{vol}(\pi_w) \geq \text{vol}(\tilde{\pi})$. Hence,

$$\begin{aligned} \text{vol}(\pi_v) - \text{vol}(v) &\geq \text{vol}(\pi_w) - \text{vol}(w) && (18) \\ &\geq \text{vol}(\tilde{\pi}) - \text{vol}(w) = \text{vol}(\pi_v) && (19) \end{aligned}$$

and we achieve a contradiction $\text{vol}(v) \leq 0$ to the assumption that $\text{vol}(v) > 0$ for all $v \in V$.

b) Adding edges in '$\rightarrow$'-direction keeps the graph acyclic: We prove this by contradiction and assume that after adding edges in '$\rightarrow$'-direction there exists a cycle $(v_1, \ldots, v_\xi)$ with $v_1 = v_\xi$. Since v_1 and v_ξ are at the same position in V, there must be at least one edge (v_i, v_{i+1}) in '$\leftarrow$'-direction. However, there are no edges in '$\leftarrow$'-direction in the original G by a), and also after adding edges in '$\rightarrow$'-direction. This leads to the contradiction.

Edges that are added by Algorithm 2 are always in '$\rightarrow$'-direction due to the definition of the total ordering of V. Hence, the graph G is still acyclic after Algorithm 2 according to b). □

After the procedure described by Algorithm 2, the subtasks of each execution group Γ_j are fully sequentialized and there are no parallel subtasks anymore, i.e., $paral(v) = \emptyset$ for all $v \in \Gamma_j$. Hence, the delay I_d has measure $\lambda(I_d) = 0$. This speeds up and tightens the response time analysis, as evaluated in Section VIII. We note that this procedure does not achieve a dominating behavior if the longest path increases by adding edges. However, evaluation results in Figure 4 showed that it does not have significant impact on the performance ratio.

VII. Implementation

This section details the implementation for the scheduling mechanism of DAG tasks with group execution constraints (as illustrated in Figure 2).

Task and Group Identification. Each vertex (or subjob) in the EG-DAG task contains associated group information. This can either be Γ_j for a grouped or $\Gamma^\perp$ for an ungrouped vertex.

Job Queues and Management. A *Global Wait Queue* is introduced to store all subjobs with pending precedence constraints. Additionally, n ready queues, labeled from RQ_1 to RQ_n, are designated for each execution group. They are applied to store ready subjobs from their associated groups. For ungrouped subjobs, another queue, $RQ^\perp$, is instantiated.

Job Scheduling Mechanism. Once all predecessors of a subjob v_i conclude their execution, the subjob is dequeued from the global wait queue. Depending on its group affiliation:

- For *Grouped Subjobs* ($v_i \in \Gamma_j$), they are enqueued into the corresponding RQ_j. A processor-group mapping table ensures each processor's exclusivity to one execution group. If v_i is the only subjob of an empty RQ_j, the processor-group mapping table is consulted. If RQ_j does not have a processor association, it binds with a currently unassigned processor. In situations where no processor remains idle, RQ_j associates with a processor

executing ungrouped subjobs. Herein, the new grouped subjob preempts any executing ungrouped subjob.
- For *Ungrouped Subjobs* ($v_i \in \Gamma^{\perp}$), they are enqueued into the $RQ^{\perp}$. An unoccupied processor, given its specific ready queue is also vacant, can execute a job from $RQ^{\perp}$. However, the lower priority of ungrouped subjobs means any incoming grouped subjob can preempt them.

Preemption and Priority Handling. Within the scheduling mechanism, grouped subjobs naturally hold higher priority. When processors are executing ungrouped subjobs, any incoming grouped subjob can preempt their execution. This mechanism not only optimizes computational resource utilization but also guarantees task continuity and ensures data consistency, especially for grouped subjobs.

To the best of our knowledge, most research-oriented RTOSes and open-source RTOSes, such as $LITMUS^{RT}$ [5], [8], RTEMS, and FreeRTOS, do not natively support the DAG task model. Implementing the DAG task model within an RTOS is beyond the scope of this work. Consequently, the most suitable approach to implement our newly proposed EG-DAG model is to simulate the scheduling behavior of each sub-job, and to utilize a table-driven scheduler, a feature that is, for instance, supported by $LITMUS^{RT}$. In order to avoid multiprocessor timing anomalies, the execution time of each subjob has to be forced to its WCET, i.e., no early completion is allowed.

VIII. Evaluation

In this section, we present a detailed experimental evaluation to assess the effectiveness of our proposed worst-case response-time analysis introduced in Section IV. We compare our analysis with the state-of-the-art DAG response-time analyses without execution groups. Additionally, we evaluate the performance enhancements achieved by our improvements, as delineated in Section VI.

A. Environment Configurations

We generated 100 task sets, each with a total utilization of 640%. The number of DAG tasks n was selected uniformly at random from $[16,\ 32]$. We applied the Dirichlet-Rescale (DRS) algorithm [22] to determine the utilization for each of the n tasks. Task periods T_i were uniformly and randomly chosen from the set $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$, a selection inspired by common practice in automotive systems [24], [29], [42], [45]. We assumed implicit deadlines for all tasks, with each task's relative deadline D_i matching its period T_i.

To generate a DAG, we first randomly selected the number of subtasks from $[10,\ 100]$, and applied the DRS to generate the utilization for each subtask. The $G(n, p)$ algorithm [14] was used to generate the edges between subtasks, with probability $p_e \in \{[10\%,\ 30\%],\ [40\%,\ 60\%],\ [70\%,\ 90\%]\}$, where p_e is the probability of an edge $(v_j, v_k) \in E$ for any pair of subtaks (v_j, v_k) with $j < k$. For each DAG task, the probability of a subtask being grouped was $p_g \in \{50\%, 70\%, 90\%\}$. The number of groups for each task set was randomly selected from the ranges $n_g \in \{[4,\ 16], [16,\ 32]\}$. For each grouped subtask, the exact group allocation, i.e., the group id, was uniformly and randomly generated. We executed the generated task sets on either $\{4,\ 8,\ 16\}$ processors and recorded the makespan for each task.

In the evaluation, we refer to the proposed approaches as **EG-DAG** for the original analysis and **EG-IMP** for the improved version in Section VI. We compared our approach to the following algorithms, which do not consider group execution constraints:

- **HE**: Single path approach represented by He et al. [26].
- **PPP**: Parallel Path Progression DAG Scheduling approach proposed by Ueter et al. [44].
- **FED**: Federated scheduling from Li et al. [32].
- **LB**: The lower bound of the makespan, which is defined as the maximum between the WCET of the task over the available processor and the length of its critical path, i.e., $\max\{vol(\tau)/M,\ vol(\pi^*)\}$.

We evaluated the makespan of different approaches based on the performance ratio, defined as the makespan of the dedicated DAG scheduling algorithm divided by the lower bound, i.e., LB.

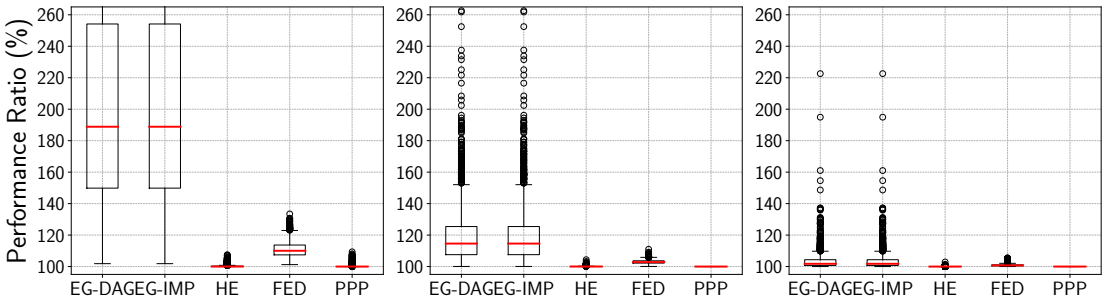
B. Evaluation Results and Discussions

We evaluated all 54 combinations under different settings. Due to the similarity of the performance, only a subset of the results is presented in Figure 4.

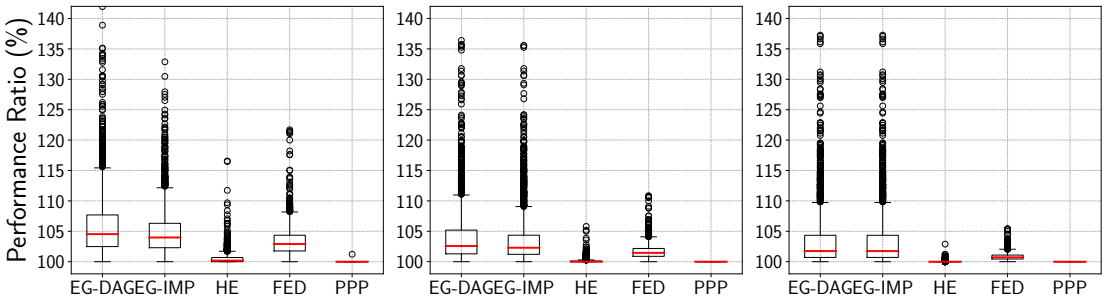
In general, our proposed approach for EG-DAG is comparable with the approaches that have not taken communication overheads into consideration when either (i) the probability of edges between subtasks is relatively high; (ii) the number of available processors is relatively large; or (iii) the probability that a subtask being grouped is relatively high. We analyzed the effect of the three parameters individually by changing:

1) $\boldsymbol{p_e \in \{[10\%, 30\%],\ [40\%, 60\%],\ [70\%,\ 90\%]\}}$ (Figure 4a): For a fixed number of available processors $M = 16$, a fixed number of groups $n_g \in [4,\ 16]$, and a fixed probability that a subtask is grouped $p_g = 70\%$, increasing the probability of edges between subtasks can significantly improve the performance of our new proposed approach for EG-DAG.
2) $\boldsymbol{M \in \{4,\ 8,\ 16\}}$ (Figure 4b): By increasing M, while keeping a fixed probability of edges between subtasks $p_e \in [70\%,\ 90\%]$, a fixed number of groups $n_g \in [4,\ 16]$, and a fixed probability that a subtask is grouped $p_g = 70\%$, the performance of the proposed approach was significantly improved.
3) $\boldsymbol{p_g \in \{50\%, 70\%, 90\%\}}$ (Figure 4c): Increasing the probability that a subtask is grouped, while keeping the number of available processors $M = 16$, the probability of edges between subtasks $p_e \in [70\%,\ 90\%]$, and the number of groups $n_g \in [4,\ 16]$, enhanced the performance of the proposed approach.

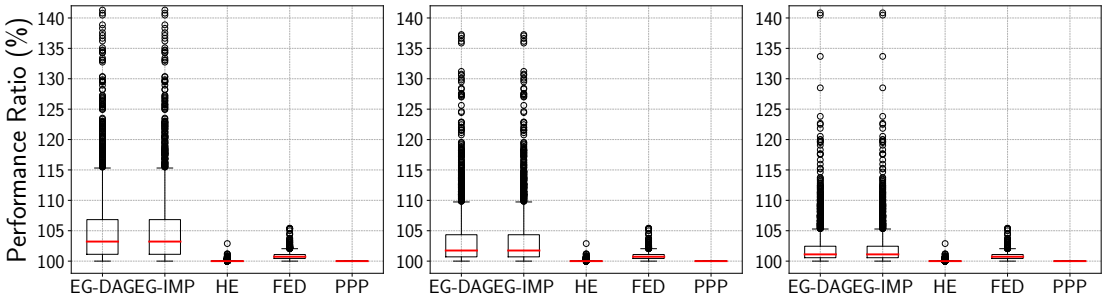
Just considering Figure 4, while the performance of EG-DAG is comparable, the other approaches usually perform slightly better. However, a loss compared to the other methods



(a) Increasing the probability of edges between subtasks, i.e., $p_e \in \{10\% - 30\%,\ 40\% - 60\%,\ 70\% - 90\%\}$.



(b) Increasing the number of available processors, i.e., $M \in \{4, 8, 16\}$.



(c) Increasing the probability that a subtask is grouped, i.e., $p_g \in \{50\%,\ 70\%,\ 90\%\}$.

Fig. 4: Comparison of different approaches with different configurations.

was expected as the additional group execution constraints must be respected. Recall that these group constraints ensure that groups of tasks with a high intra-group communication load are assigned to the same core, thus drastically reducing the communication overhead. This communication overhead is not considered by the other approaches.

In most configurations the median performance loss of our approach, compared to the lower bound, is less than 5%. Therefore, the evaluation results substantiate that our analysis accommodates communication overheads and additional constraints with only a marginal degradation in performance. Once non-negligible communication overheads are considered, such as 5% of the WCET for each task, our newly proposed approaches can outperform all other approaches in most of the evaluated configurations.

Additionally, the improved version of the proposed approach outperforms the original in most evaluated configurations. However, when the probability of edges between subtasks is relatively high, i.e., $p_e \in [70\%,\ 90\%]$, and the number of available processors is also relatively large, i.e., $M = 16$, the performance difference becomes negligible. In the aforementioned configurations, these subtasks have already been connected in most of the cases, adding additional edges does not affect the structure of the considered DAG task significantly. Therefore, the makespan does not change significantly as well.

IX. Extension of the Analysis to Scenario with Multiple DAGs

In typical systems, not all m processors are exclusively allocated to a single task. Therefore, in this section we discuss how to extend our analysis to the general case with n DAG tasks $\tau_1, \ldots, \tau_n$. To achieve this, we have to schedule the n tasks such that it is ensured that the subtasks with the execution constraints execute on the same processor.

The most straightforward approach is to partition the n tasks on m processors by assigning $m_i \geq 1$ processors for task τ_i exclusively while ensuring that $\sum_{i=1}^{n} m_i \leq m$. However, this approach only works if sufficient cores are available, and especially performs very poorly if certain tasks utilize only a

fraction of their dedicated processor capacity. Therefore, we might want to deploy a less restrictive mechanism.

We observe that the following two properties are needed to apply our scheduling mechanism and the related analysis: 1) During execution of a task τ_i, a certain amount of m_i processors is available exclusively for τ_i, and 2) the processors that are used by task τ_i do not change until its job is completed.

We utilize hierarchical scheduling, a layered scheduling approach, in which so-called reservation systems are scheduled on the physical processors based on the higher-level scheduling strategy. These reservations then serve the assigned actual tasks whenever executed on the physical processors based on the lower-level scheduling strategy. Specifically, we consider rigid gang scheduling on the higher level as it was shown to have performance benefits compared to non-gang scheduling [16], [27], and the proposed scheduling mechanism from Section III on the lower level.

Gang scheduling achieves 1) directly. In gang scheduling, for each DAG task τ_i, a dedicated gang reservation is scheduled on m_i dedicated processors in parallel. In our case, m_i is rigid, i.e., it is fixed and predefined offline. Inside each reservation, the DAG tasks can be scheduled using our proposed scheduling mechanism from Section III. There is already a rich literature on gang scheduling [2], [13], [20], [21], [28], [31], [34], [43].

To ensure 2), we must forbid task migration during runtime. That is, we must either restrict the gang scheduling algorithm in the time-domain (i.e., non-preemptive scheduling) or in the space-domain (i.e., stationary gang scheduling). A restriction in the time domain is achieved by non-preemptive scheduling where each job finishes its execution before the next job can access the processors. With non-preemptive scheduling, there is no need for task migration. A restriction in the space domain can be realized by stationary gang scheduling. That is, the m_i processors used for task τ_i are predefined. We refer to the literature for state-of-the art response time analyses, e.g., in [31] or [43], respectively.

The inherent characteristics of various gang scheduling algorithms significantly influence the schedulability performance of a task set, particularly when different makespan analysis algorithms are applied on a specific number of available processors. Therefore, in our evaluation, we focus on assessing the makespan of different approaches across varying numbers of available processors for each task. This approach diverges from evaluating the performance of the entire task set using different gang scheduling algorithms. Our objective is to compare the performance of each makespan analysis algorithm for a single DAG task in a more isolated manner.

X. Related Work

In practice, one common example of DAG tasks with group execution constraints are applications following the AUTOSAR standard that are deployed on multiprocessor platforms. Each task is composed of several runnables with data dependencies; these runnables are similar to the subtasks in this work. In order to minimize the communication overheads, a lot of strategies have been proposed and studied. Saidi et al. [36] presented an Integer Linear Programming (ILP) formulation for mapping AUTOSAR runnables to a multi-core architecture, with the aim to minimize inter-core communication and to balance the workload on available cores. Gupta et al. [23] proposed an approach for mapping runnables to different cores, meeting the corresponding timing and precedence constraints while considering hardware dependency and inter-core communication cost. Faragardi et al. [15] proposed a feedback-based solution framework for component-based embedded software on a multi-core processor, subject to reduce both the inter-core communication cost and the waiting time caused by the synchronization between dependent transactions. However, there results consider the minimization of inter-core communication overheads rather than taking the optimization of schedule algorithms for given mappings of runnables with release precedence constraints into consideration.

On the other hand, the scheduling of DAG tasks with release precedence constraints on heterogeneous/homogeneous multiprocessor platforms has received a lot of attention over the years. Li et al. [32] proposed federated scheduling, where the intra-task interference on the critical path is upper-bounded by the workload of subjobs not belonging to the critical path divided by the number of processors. The corresponding response time analysis requires no information about the internal structure of the DAG except for the total volume and the length of the critical path. Sun et al. [38], [39] focused on OpenMP task systems, where *tied* tasks must be executed on the same thread. For an OpenMP task system, the task is defined as a set of sequentially executed subtasks; hence, the model is less general than the DAG task model where the order of the subjobs is based on precedence constraints. Tessler et al. [41] developed a cache-aware BUNDLE-scheduling algorithm for federated scheduling of sporadic DAG task sets, which focuses on cache affinities and allows for trade-offs. He et al. [26] proposed a priority assignment policy for each subtask of a DAG as well as a response time bound for DAG tasks with arbitrary priority assignment. Ueter et al. [44] proposed a parallel path progression scheduling algorithm with two distinct subtask priorities, which allows to quantify the parallel execution of a user chosen collection of complete paths in the response time analysis. Tessler et al. [40] considered cache-aware co-located scheduling for fork-join tasks, aiming to improve schedulability by carefully scheduling threads to share cached values. However, all these results do not consider the communication overheads of subtasks with data dependencies in DAG task systems, where execution groups are regarded as a strict requirement.

XI. Conclusion

In this paper, we have enhanced the DAG task model by introducing the concept of execution groups requiring designated subtasks to be executed on the same processor. This refinement aims to minimize communication overhead within multi-core systems. To facilitate this model, we have proposed a dedicated scheduling mechanism that ensures the

coordinated execution of subtasks within these groups while providing a comprehensive response time analysis.

To the best of our knowledge, our work represents the first effort to introduce a scheduling mechanism tailored for DAG tasks with execution groups. Our evaluation results demonstrate the competitiveness of our approach, even after refining the DAG task model to accommodate execution groups, compared to existing approaches that do not explicitly consider communication overhead and group execution constraints. These results also suggest that our approach can offer superior schedulability guarantees, especially when accounting for communication overhead, such as a modest fraction (e.g., 5%) of the corresponding task's Worst-Case Execution Time.

Acknowledgement

This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170). This work has been supported by Deutsche Forschungsgemeinschaft (DFG), as part of Sus-Aware (project No. 398602212).

References

[1] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Haupenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR*, volume 8052 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2013.

[2] W. Ali and H. Yun. Rt-gang: Real-time gang scheduling framework for safety-critical systems. In *RTAS*, pages 143–155. IEEE, 2019.

[3] S. Baruah. Federated scheduling of sporadic DAG task systems. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 179–186, 2015.

[4] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility analysis in the sporadic dag task model. In *ECRTS*, pages 225–233, 2013.

[5] B. B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, University of North Carolina, Chapel Hill, USA, 2011.

[6] J. M. Calandrino and J. H. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *20th Euromicro Conference on Real-Time Systems, ECRTS*, pages 299–308. IEEE Computer Society, 2008.

[7] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *21st Euromicro Conference on Real-Time Systems, ECRTS*, pages 194–204. IEEE Computer Society, 2009.

[8] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmusˆrt : A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, pages 111–126. IEEE Computer Society, 2006.

[9] D. Casini, A. Biondi, G. Nelissen, and G. C. Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *RTSS*, pages 421–433. IEEE Computer Society, 2018.

[10] D. Casini, A. Biondi, G. Nelissen, and G. C. Buttazzo. A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling. In *RTAS*, pages 239–252. IEEE, 2020.

[11] R. Cole and V. Ramachandran. Analysis of randomized work stealing with false sharing. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pages 985–998. IEEE Computer Society, 2013.

[12] Z. Dong and C. Liu. Work-in-progress: New analysis techniques for supporting hard real-time sporadic dag task systems on multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 151–154, 2018.

[13] Z. Dong and C. Liu. A utilization-based test for non-preemptive gang tasks on multiprocessors. In *RTSS*, pages 105–117. IEEE, 2022.

[14] P. Erdös. On random graphs I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[15] H. R. Faragardi, B. Lisper, K. Sandström, and T. Nolte. A resource efficient framework to run automotive embedded software on multi-core ecus. *Journal of Systems and Software*, 139:64–83, 2018.

[16] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *J. Parallel Distributed Comput.*, 16(4):306–318, 1992.

[17] J. Feljan and J. Carlson. The impact of intra-core and inter-core task communication on architectural analysis of multicore embedded systems. In *The Eighth International Conference on Software Engineering Advances, ICSEA*, pages 402–407, 2013.

[18] J. Fonseca, G. Nelissen, and V. 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*, pages 28–37, 2017.

[19] J. C. Fonseca, G. Nelissen, V. Nélis, and L. M. Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *11th IEEE Symposium on Industrial Embedded Systems, SIES*, pages 290–299, 2016.

[20] J. Goossens and V. Berten. Gang FTP scheduling of periodic and parallel rigid real-time tasks. *CoRR*, abs/1006.2617, 2010.

[21] J. Goossens and P. Richard. Optimal scheduling of periodic gang tasks. *Leibniz Trans. Embed. Syst.*, 3(1):04:1–04:18, 2016.

[22] D. Griffin, I. Bate, and R. I. Davis. Generating utilization vectors for the systematic evaluation of schedulability tests. In *41st IEEE Real-Time Systems Symposium, RTSS*, pages 76–88. IEEE, 2020.

[23] P. Gupta, N. P. Singh, and G. Srinivasan. An efficient approach for mapping autosar runnables in multi-core automotive systems to minimize communication cost. In *2019 Innovations in Power and Advanced Computing Technologies (i-PACT)*, volume 1, pages 1–4, 2019.

[24] 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*, volume 76 of *LIPIcs*, pages 10:1–10:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[25] Q. He, x. jiang, N. Guan, and Z. Guo. Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2283–2295, 2019.

[26] Q. He, M. Lv, and N. Guan. Response time bounds for DAG tasks with arbitrary intra-task priority assignment. In *33rd Euromicro Conference on Real-Time Systems, ECRTS 2021, July 5-9, 2021, Virtual Conference*, volume 196 of *LIPIcs*, pages 8:1–8:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[27] M. A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 54. ACM, 1997.

[28] S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *IEEE Real-Time Systems Symposium, RTSS*, pages 459–468, 2009.

[29] 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.

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

[31] S. Lee, N. Guan, and J. Lee. Design and timing guarantee for non-preemptive gang scheduling. In *RTSS*, pages 132–144. IEEE, 2022.

[32] J. Li, 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. IEEE Computer Society, 2014.

[33] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *Proceedings of the 2015 27th Euromicro Conference on Real-Time Systems*, 2015.

[34] G. Nelissen, J. M. i Igual, and M. Nasri. Response-time analysis for non-preemptive periodic moldable gang tasks. In *ECRTS*, volume 231 of *LIPIcs*, pages 12:1–12:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[35] P. Radojkovic, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla. On the evaluation of the impact of shared resources in

multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, 2012.

[36] S. E. Saidi, S. Cotard, K. Chaaban, and K. Marteil. An ILP approach for mapping AUTOSAR runnables on multi-core architectures. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO@HiPEAC*, pages 6:1–6:8. ACM, 2015.

[37] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-Core Real-Time Scheduling for Generalized Parallel Task Models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 217–226, 2011.

[38] J. Sun, N. Guan, X. Wang, C. Jin, and Y. Chi. Real-time scheduling and analysis of synchronous openmp task systems with tied tasks. In *DAC*, page 94. ACM, 2019.

[39] 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. IEEE Computer Society, 2017.

[40] C. Tessler, P. P. Modekurthy, N. Fisher, A. Saifullah, and A. Murphy. Co-located parallel scheduling of threads to optimize cache sharing. In *IEEE Real-Time Systems Symposium, RTSS*, pages 251–264. IEEE, 2023.

[41] C. Tessler, V. P. Modekurthy, N. Fisher, and A. Saifullah. Bringing inter-thread cache benefits to federated scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 281–295. IEEE, 2020.

[42] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein. System-level timing feasibility test for cyber-physical automotive systems. In *11th IEEE Symposium on Industrial Embedded Systems, SIES 2016, Krakow, Poland, May 23-25, 2016*, pages 121–130. IEEE, 2016.

[43] N. Ueter, M. Günzel, G. von der Brüggen, and J. Chen. Hard real-time stationary gang-scheduling. In *ECRTS*, volume 196 of *LIPIcs*, pages 10:1–10:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[44] N. Ueter, M. Günzel, G. von der Brüggen, and J. Chen. Parallel path progression DAG scheduling. *IEEE Transactions on Computers*, pages 1–15, 2023.

[45] 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*, pages 108–117, 2017.

[46] J. Xiao, S. Altmeyer, and A. D. Pimentel. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *2017 IEEE Real-Time Systems Symposium, RTSS*, pages 199–208. IEEE Computer Society, 2017.

[47] M. Xu, L. T. X. Phan, H. Choi, and I. Lee. Analysis and implementation of glenergy saving for mixed-criticalityobal preemptive fixed-priority scheduling with dynamic cache allocation. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 123–134. IEEE Computer Society, 2016.

[48] H. Yun. Evaluating the isolation effect of cache partitioning on cots multicore platforms. In *OSPERT*, 2015.

[49] S. Zhao, X. Dai, I. Bate, A. Burns, and W. Chang. DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 128–140. IEEE, 2020.

[50] M. Zini, G. Cicero, D. Casini, and A. Biondi. Profiling and controlling i/o-related memory contention in COTS heterogeneous platforms. *Softw. Pract. Exp.*, 52(5):1095–1113, 2022.