AI Meets Real-Time: Addressing Real-World Complexities in Graph Response-Time Analysis*

Sergey Voronov, Stephen Tang, Tanya Amert, and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

***Abstract*—Artificial-intelligence algorithms are enabling ever more sophisticated autonomous features in safety-critical application domains. These algorithms can be quite complex—consisting of many tasks interconnected in processing graphs—and often must execute on complex heterogeneous hardware—typically multicore machines augmented with one or more hardware accelerators. To further complicate matters, these processing graphs often must be supported in contexts where a large system is broken into smaller components. With this confluence of factors, existing response-time analysis for processing graphs is not applicable. In this paper, such analysis is extended to address these complexities in systems where components are isolated via time partitioning. Additionally, graph restructuring methods are presented that enable response-time bounds to be reduced.**

***Index Terms*—Processing graphs, response-time bounds, hardware accelerators, component isolation**

I. Introduction

Today, we are witnessing ever more sophisticated autonomous features in safety-critical application domains, notably in the avionics and automotive industries. These features are being realized via an evolving repertoire of artificial-intelligence (AI) algorithms that realize capabilities such as visual perception and decision-making. These algorithms often have complex dataflow dependencies expressible using processing graphs that can be computationally intensive, requiring the usage of hardware accelerators (HACs), such as graphics processing units (GPUs), field-programmable gate arrays (FPGAs), *etc.* To further complicate matters, multiple AI functions typically must be integrated on a common multicore+HAC platform to save on size, weight, power, and cost; these functions are often developed separately, by different developers or even companies. Such a separation is helpful, as smaller components are easier to specify, implement, and understand.

Unfortunately, the confluence of the three factors alluded to above—**(i)** complex graph-based workloads, **(ii)** complex heterogeneous hardware, and **(iii)** the need to integrate separately developed components—is creating challenges for real-time certification. These challenges were recently highlighted at an Industry Panel at RTAS 2021 [1], where the *lack of real-time analysis that fully addresses these challenges* was called out as a key stumbling block.

This paper is directed at producing such analysis. Our specific contributions lie in extending prior graph-based response-time analysis so that accelerators can be accommodated, and separately developed graph-based components can be isolated via time partitioning. We elaborate on these contributions below, after first introducing some relevant concepts and reviewing prior efforts on analyzing real-time processing graphs.

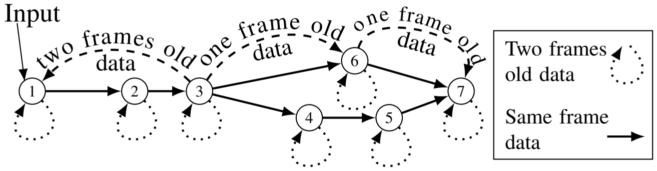


Fig. 1: An example AI algorithm that processes video frames.

Graph model. AI algorithms are often depicted as graphs in which nodes represent computations and edges indicate data dependencies. For example, consider the special case of an AI-based computer-vision (CV) algorithm (which we use here to illustrate key concepts) as shown in Fig. 1. This CV algorithm includes data dependencies from the current video frame (solid edges in Fig. 1) and may also include dependencies from previous frames (dotted/dashed edges); such a combination is necessary to determine object motion, for example.

To exploit the parallelism inherent to such graphs, real-time task models must also utilize a graph-based structure. This structure may support instance-level self-parallelism, in which multiple invocations of the same graph may execute concurrently (*i.e.*, parallel processing of frames). However, most prior work on graph scheduling does not properly handle such parallelism; instead, it either precludes instance-level self-parallelism [11], [19], [51], [55], [68], [73] or arbitrarily forces dependencies to the prior frame only [57], [90], [91].

Additionally, graph-based tasks may have historical data dependencies (*i.e.*, cycles in the graph). However, prior work mostly ignores such inter-instance node-level dependencies entirely [12], [42], [55], [66], [85], [92].

Accelerator usage. Accesses to HACs tend to be non-preemptive; for example, even when possible, GPU preemptions typically introduce prohibitively high overhead. Thus, HAC usage must be carefully managed if multiple workloads share a platform. Non-preemptive accesses complicate response-time analysis, as HAC blocking by one application can increase response times for other applications, or even break temporal isolation if not handled correctly. Some prior work has considered HAC accesses by graph-based task systems, but not when shared between components that are

*This work was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

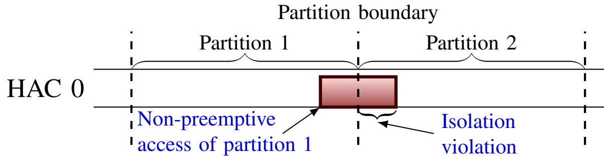


Fig. 2: Isolation violation example.

temporally isolated from each other.

Component isolation. On a conventional uniprocessor, time and space partitioning can be applied to fully isolate system components from one another, enabling a separation of concerns that is crucial for providing certification authorities with high confidence in a system's correctness. In avionics, for example, ARINC 653 [67] provides real-time operating system design specifications for hardware sharing. In ARINC 653, system applications execute within *time slices*; a time slice is a time interval during which an application is ensured sole access to a given set of processing resources.

Time-partitioned systems can be scheduled hierarchically [3]: a top-level scheduler schedules time slices, and for each time slice, a lower-level scheduler allocates processing resources to the assigned workload. As shown in Fig. 2, a non-preemptive access to a HAC shared by two components may violate temporal isolation if the access crosses the boundary between time slices. However, prior approaches that considered the isolation of graph-based systems (*e.g.*, [25], [86]) have not addressed the challenge posed by multiprocessor platforms equipped with non-preemptive HACs.

Summary of prior work. A summary of prior approaches that address some of challenges (i)–(iii) is presented in Tbl. I. To our knowledge, no prior work has considered all three challenges, all of which are addressed herein. Also, we show how to reduce graph response-time bounds by modifying a graph's structure, an issue also unaddressed by prior work.

Contributions. Our focus in this paper is real-time analysis of graph-based AI applications on time-partitioned multiprocessor+HAC platforms. Our contributions are fourfold.

First, we present an approach for ensuring that non-preemptive HAC accesses do not cross time-slice boundaries. This approach can add delays in accessing HACs; we derive upper bounds on these delays.

Second, we present analysis for partially available CPUs and HACs, as in our time-sliced setting. A series of transformations enables us to leverage prior graph response-time analysis that assumes a fully available platform of identical CPUs.

Third, in some AI use cases, HACs (especially GPUs) can be highly contended. We show that prior HAC-arbitration approaches can result in overly conservative schedulability analysis in this case, and present a new approach for handling such accesses that can greatly improve schedulability.

Finally, we propose heuristics to reduce graph response-time bounds by merging graph nodes. In experiments that we conducted, these heuristics improved response-time bounds by 30-35%. To our knowledge, we are the first to consider node merging as a means to reduce graph response-time bounds.

	Graph Model		**Accelerators Usage**	**Component Isolation**	**Response-Time Reductions via Graph Refinement**
	Instance-level Self-parallelism	**Node-level Dependencies Among Instances**			
[75]	No	Obviated[1]	Partially[2]	No	No
[91]	Yes	No	Yes	No	No
[92]	Yes	No	Yes	No	No
[90]	Partially[3]	Yes	Yes	No	No
[7]	Yes	Yes	Yes	No	No
[81]	Yes	No	No	Partially[4]	No
[25]	No	Obviated[1]	No	Yes	No
[86][5]	No	Obviated[1]	No	Yes	No
[62]	Sequential Tasks		Yes	Partially[6]	No
This Paper	Yes	Yes	Yes	Yes	Yes

[1] Fine-grained dependencies are replaced with per-instance dependency (due to implicit/constrained deadlines).
[2] Graph has only one accelerated node.
[3] Limited by prior instance dependencies.
[4] Different reservation type (one interval).
[5] Task model considers communication costs among nodes.
[6] CPU can be occupied by only one component, no isolation over HACs.

TABLE I: Comparison of papers that consider at least two of graphs, accelerators, and component isolation.

Organization. In the rest of this paper, we overview our bound-computation process (Sec. II), provide needed background (Sec. III), present our four contributions (Secs. IV–VII) and a related experimental validation of our approach (Sec. VIII), discuss implications for real-world AI graphs (Sec. IX) and related work (Sec. X), and conclude (Sec. XI).

II. The Transformation Process

Our primary goal is to provide response-time bounds for a single component whose workload is a collection of graphs with HAC accesses. To this end, we apply a series of transformations to the workload and the processing supply, as depicted in Fig. 3; we focus on a single graph, but the same steps can be applied to a set of graphs simultaneously.

The five main steps transform a graph into a set of individual tasks (Steps 1 and 2, covered in Sec. III), abstract HAC requests (Step 3, Sec. III, IV), and transform the processing supply and workload to leverage existing response-time analysis on identical multiprocessors [7] (Steps 4 and 5, Sec. V).

Two optional steps can additionally be used to reduce response-time bounds via factoring out tasks with the longest HAC requests (Step 6, Sec. VI) and modifying a DAG's structure (Step 7, Sec VII).

III. Working with Graph-Based Task Systems

Prior work on response-time analysis for graph-based task systems has shown how to convert a graph containing both CPU and HAC computations into a set of CPU-only tasks [7], [57] (Steps 1–2 in Fig. 3). We illustrate this process using the graph in Fig. 4a. Each graph node denotes a recurrent computation. *Instances* (or *jobs*) of these nodes are released with a period common to all nodes in the graph. Edges and their weights denote dependencies between nodes, *e.g.*, an instance

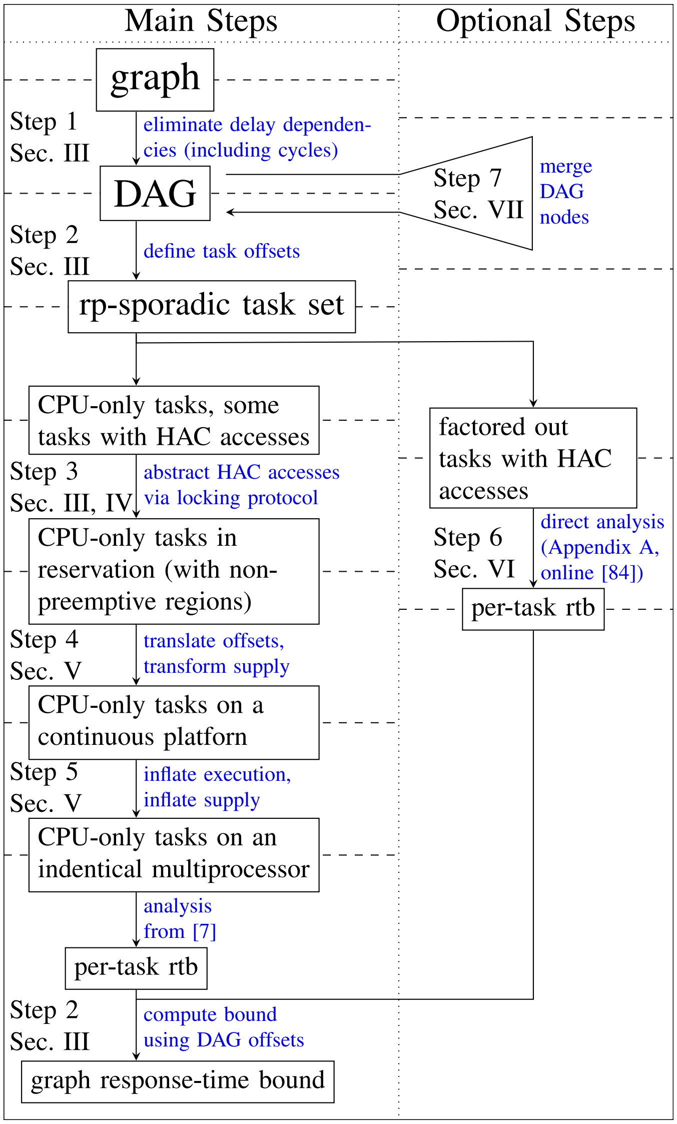
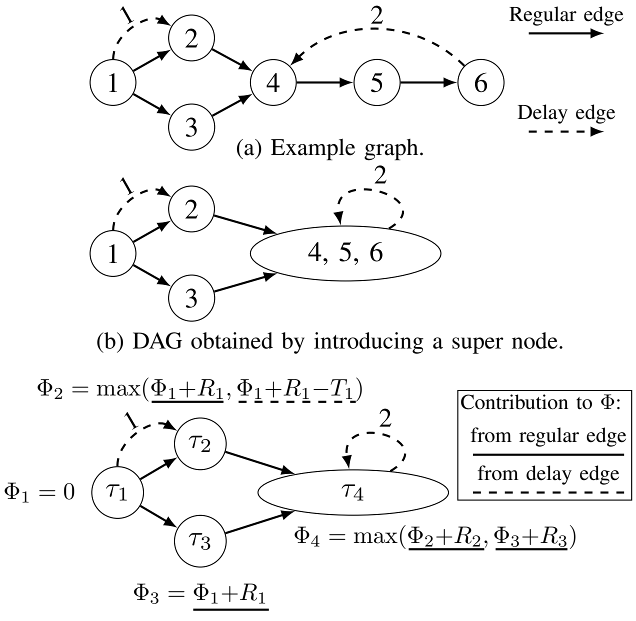

Main Steps
Optional Steps
graph
Step 1
Sec. III
eliminate delay dependencies (including cycles)
DAG
Step 7
Sec. VII
merge DAG nodes
Step 2
Sec. III
define task offsets
rp-sporadic task set
CPU-only tasks, some tasks with HAC accesses
factored out tasks with HAC accesses
Step 3
Sec. III, IV
abstract HAC accesses via locking protocol
CPU-only tasks in reservation (with non-preemptive regions)
Step 6
Sec. VI
direct analysis (Appendix A, online [84])
per-task rtb
Step 4
Sec. V
translate offsets, transform supply
CPU-only tasks on a continuous platform
Step 5
Sec. V
inflate execution, inflate supply
CPU-only tasks on an indentical multiprocessor
analysis from [7]
per-task rtb
Step 2
Sec. III
compute bound using DAG offsets
graph response-time bound

Fig. 3: The response-time-bound (rtb) computation process.

of node 4 requires the output of node 6 from two instances ago. Omitted weights denote intra-instance dependencies.

Step 1: Graph to DAG. The first step of the transformation produces a DAG by removing backward delay edges (dashed lines in Fig. 4a from higher-indexed to lower-indexed nodes). Backward delay edges (*e.g.*, from node 6 to node 4) may form cycles, and can be eliminated by combining the nodes in a cycle into a single "super node" [7], [57]. Forward delay edges (*e.g.*, from node 1 to node 2) are addressed using offsets similarly to non-delay edges (solid lines), as discussed below.

The resulting DAG in our example is depicted in Fig. 4b. For simplicity, we omit task self-dependencies in Fig. 4b (dotted lines in Fig. 1); these dependencies are used later to define tasks. The cycle comprised of nodes 4, 5, and 6 has been merged into a single super node.[1]



(a) Example graph.

(b) DAG obtained by introducing a super node.

(c) Task offset computation.

Fig. 4: Graph-to-tasks transformation example.

Step 2: DAG to tasks. The next step of the transformation converts the DAG to a set of independent tasks. Each node in Fig. 4b is made into a task in Fig. 4c. Dependencies between tasks are removed by the introduction of *release offsets*. With suitably sized release offsets, by the time any job of interest J is released, all prior jobs that J depends on (via edges such as in Fig. 4b, including both solid and forward delay edges) will have already completed. Thus no job's execution is delayed by dependencies, so it is safe to analyze this task system with the assumption that tasks are independent.

Release offsets are computed by initially computing response-time bounds R_i for each task τ_i assuming independent tasks ("per-task rtb" nodes in Fig. 3 represent the collection of R_i values). Then, the offset Φ_i of each task τ_i is calculated recursively as the maximum of the values $\Phi_j + R_j$ for any task τ_j that τ_i depends on (some additional details about forward delay edges are omitted here; see [7] for details). The response-time bound of a node τ_i in the DAG is given by $\Phi_i + R_i$. For example, τ_4 in Fig. 4c has $\Phi_4 = (\Phi_2 + R_2, \Phi_3 + R_3)$, as τ_4 depends on τ_2 and τ_3, and the response-time bound for τ_4 is given by $\Phi_4 + R_4$. In [7], these steps are carried out assuming global earliest-deadline-first (G-EDF) scheduling, which we also assume in this paper.

After the per-task response-time bounds computation, an end-to-end response-time bound for one graph can be simply computed as the largest response-time bound for any of its tasks (see the last step in Fig. 3).

Enabling intra-task parallelism via the rp-sporadic task model. While the above process is sufficient for transforming to the standard sporadic task model, much potential parallelism in the graph is lost under this model. Such parallelism can be recaptured by instead transforming to the *rp-sporadic task model* (rp stands for restricted parallelism), which augments

[1] Each node invocation is sequential, so the super node's self loop represents a trivial cycle.

the sporadic task model by assigning to each task τ_i a parallelism level P_i denoting how many jobs of τ_i may execute simultaneously. Parallelism is limited by DAG node self-dependencies (dotted lines in Fig. 1) and delay edges, particularly backward delay edges such as between τ_4 and itself in Fig. 4b (recall that forward delay dependencies are satisfied by release offsets). The weight of this edge is 2, *i.e.*, τ_4 requires output from two instances ago, so two consecutive jobs of τ_4 may execute concurrently; thus, $P_4 = 2$.

Enabling intra-task parallelism as in the rp-sporadic task model may be required for other reasons as well. For example, a super node (which forces several of the original tasks to execute sequentially) could easily over-utilize a single processor, in which case intra-task parallelism must be allowed for it.

Early releasing. Using release offsets may increase observed response times. This negative impact can be mitigated by allowing jobs to be *early released* [32]: a job J is eligible for scheduling as soon as all jobs it depends on have completed, even if this occurs before J's actual release time, as long as its scheduling priority remains unchanged. An example of early releasing can be found in Fig. 5 (job $J_{1,5}$ is scheduled before its actual release). Early releasing does not violate schedulability guarantees under G-EDF-based scheduling [32].

Step 3: Tasks to CPU-only tasks. To transform to CPU-only tasks, all HAC execution must be abstracted as CPU execution (Step 3 in Fig. 3). This was done in [7] via a locking protocol by analytically treating time spent waiting for and executing on a HAC as additional CPU execution time. We compute the total waiting time in our setting in Lemma 1 (in Sec. IV). Once the graph is fully transformed into a set of independent rp-sporadic tasks with only CPU execution, the existing response-time analysis [7] can be applied to derive an end-to-end response-time bound for the entire graph.

Problems. Four fundamental problems arise when attempting to apply the transformation process summarized above to time-partitioned AI components:

Problem 1: HAC accesses, which are typically non-preemptive, can overrun time-slice boundaries.

Problem 2: The considered response-time analysis is not applicable when hardware resources are partially available to a component because of time partitioning.

Problem 3: In AI use cases, some HACs (particularly GPUs) can be highly contended, causing blocking bounds (which are analytically translated into CPU execution time) to become so large that over-utilization results, making acceptable response-time bounds impossible to obtain.

Problem 4: Response-time bounds, which scale with release offsets, may be prohibitively large when dependencies form long paths; this effect is exacerbated by time slicing.

We address these problems in turn next in Secs. IV–VII.

IV. Preventing HAC Time-Slice Overruns

Given that the first two steps of the transformation process discussed in the prior section maps a graph to a set of rp-sporadic tasks, it suffices to limit our attention to such tasks for now. (We return to graphs later in Sec. VII when discussing graph-restructuring methods to reduce graph response-time bounds.) Given this focus, a formal treatment of the rp-sporadic task model is needed. For ease of reading, a table of notation (Tbl. III) is provided in an online appendix [84].

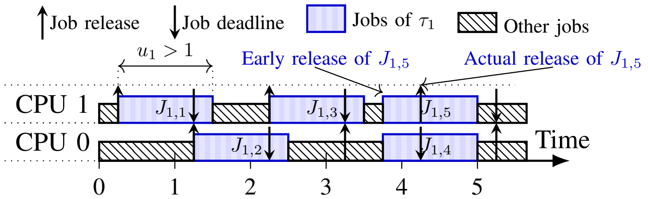


Fig. 5: An rp-sporadic task τ_1 schedule ($u_1 = 1.25$, $P_1 = 2$).

The rp-sporadic task model, formally defined. We denote the *period* (and relative deadline) of task τ_i as T_i, τ_i's *worst-case execution time* (WCET) as C_i, and its *utilization* as $u_i = C_i/T_i$. We denote the j^{th} job of task τ_i as $J_{i,j}$ and its *release time* as $r_{i,j}$. Thus, its (absolute) *deadline* is at time $r_{i,j} + T_i$. Note that deadlines in this paper are only used to prioritize jobs by schedulers such as G-EDF, and are not required upper bounds on jobs' completion times.

As in [7], we associate with each task τ_i a *parallelization level* P_i: up to P_i jobs of τ_i may execute concurrently, and job $J_{i,j}$ (if $j > P_i$) cannot execute until job $J_{i,j-P_i}$ completes. For example, in Fig. 5, two jobs of task τ_1 with $P_1 = 2$ execute simultaneously at several time instants. We allow early releasing of jobs of rp-sporadic tasks (*e.g.*, $J_{1,5}$ in Fig. 5).

We consider a set of rp-sporadic tasks τ to be *schedulable* if each task has a bounded response time; τ is schedulable if and only if $\forall \tau_i : u_i \leq P_i$, and $\sum_{\tau_i \in \tau} u_i$ does not exceed the total number of CPUs [7].

Partial-supply model. Given our focus on time-partitioned AI computations, we consider a system divided into sets of rp-sporadic tasks, called *components*. When considering time-partitioned platforms, we look to certification processes used in industry, as exemplified by the current ARINC 653 time-slicing approach as used in avionics. Because complicated partitioning strategies are unlikely to be adopted in practice, we consider simple periodic time slicing.

We abstract the idea of time slicing by ensuring that each component Γ is granted exclusive periodic access to a set Υ of computing resources by defining a *periodic component reservation* (PCR) for Γ (similar to the Single Time Slot Periodic Partition model by Mok and Chen [60] and the multiprocessor periodic resource (MPR) model by Shin *et al.* [76]). We assume that Υ contains M unit-speed identical CPUs and (for simplicity) a single non-preemptive HAC (we discuss extending to multiple HACs per component later in this section). The PCR for component Γ is defined as a three-tuple (Θ, Π, Υ), denoting that Γ receives exclusive access to the computing resources in Υ within continuous intervals of Θ time units that begin every Π time units ($\Theta \leq \Pi$). We assume that the choice of Θ and Π per component is given; optimizing such choices is outside the scope of this paper.

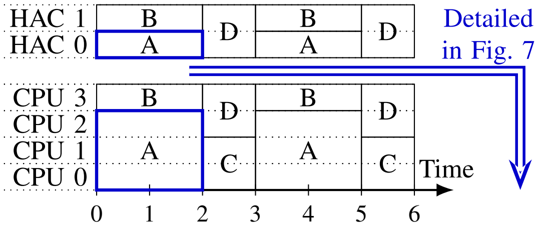


Fig. 6: Time-sliced schedule (rectangles are component slices).

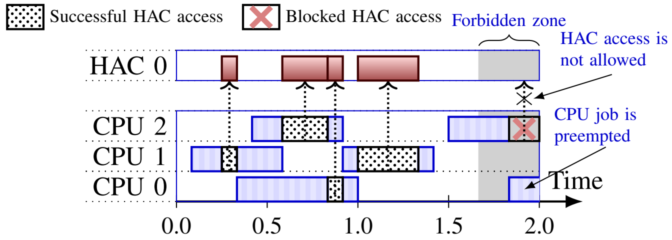


Fig. 7: Forbidden zone example (component A of Fig. 6).

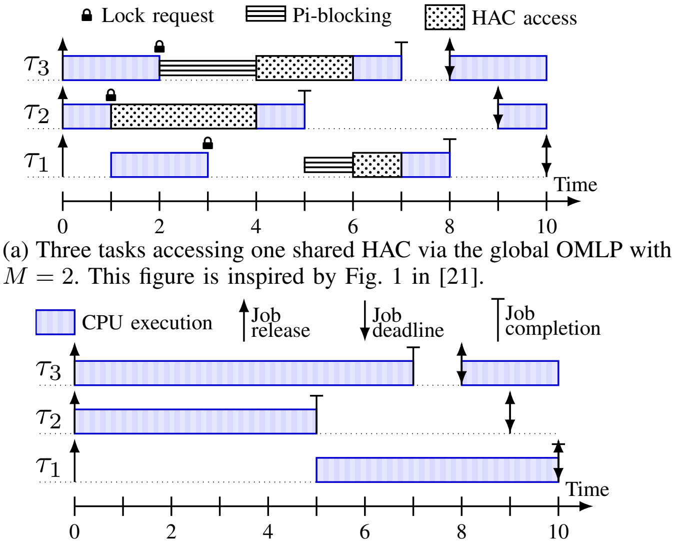
As an example, Fig. 6 shows the first few time slices for four components on a platform with four CPUs and two HACs. In this example, Component A is specified by $(2, 3, \{\text{CPU } 0, \text{CPU } 1, \text{CPU } 2, \text{HAC } 0\})$.

Forbidden zones. We henceforth assume Γ denotes a component with PCR (Θ, Π, Υ) as defined above. If a HAC access within Γ crosses a time-slice boundary, then the next component to execute does not have exclusive access to this HAC. This behavior breaks the assumed resource model.

Fortunately, a concept from prior work known as a *forbidden zone* [46] can be applied to solve this problem.[2] A forbidden zone for a given HAC access in Γ is a region of time in which that access may not be initiated, as it may cross Γ's next time-slice boundary; the zone length is thus the worst-case duration of that access. Note that accelerator usage by other components has no impact on Γ's forbidden-zone lengths. The use of forbidden zones in Γ requires that no accelerator access in Γ takes more than Θ time units.

The forbidden-zone idea is illustrated in Fig. 7, which shows execution details of Component A from Fig. 6 within the time slice $[0, 2)$. The forbidden zone is shown in grey prior to the time-slice boundary at time 2. Note that a HAC access initiated on CPU 2 is rejected in this zone, while ordinary CPU execution (such as on CPU 0) is allowed.

Step 3: Blocking analysis of a HAC access. Blocking analysis for locking protocols can be augmented to consider forbidden zones. In this paper, we use the suspension-based global OMLP [21] to arbitrate HAC accesses within each component, though alternate locking protocols could be used. Under the global OMLP, a job waiting to acquire a HAC suspends (freeing a CPU for other jobs) and is added to the global OMLP's queues—both a priority queue and an M-element FIFO queue are used, with HAC requests at the head of the priority queue feeding into the FIFO queue. The request at the head of the FIFO queue is allowed HAC access.



(a) Three tasks accessing one shared HAC via the global OMLP with $M = 2$. This figure is inspired by Fig. 1 in [21].

(b) The same task system from inset (a) with pi-blocking and accelerator-execution time modeled as CPU execution time.

Fig. 8: Illustration of suspension-oblivious analysis under the global OMLP.

▷ **Def. 1.** *Let B_{max} be the longest HAC-access duration in Γ.* ◁

Given our aim of extending the prior response-time analysis overviewed in Sec. III, and that analysis's focus on G-EDF, we assume G-EDF scheduling within each component (it is "global" within a component). This choice of scheduler motivates our interest in the global OMLP because it has optimal priority-inversion blocking (pi-blocking) under suspension-oblivious analysis, which is the suspension-accounting method usually used under G-EDF. Under suspension-oblivious analysis, a job in Γ is only pi-blocked if it is one of the M highest-priority active jobs but is not scheduled.

For example, in Fig. 8a, job $J_{1,1}$ is blocked during the entire interval $[3, 6)$, but is only pi-blocked in the interval $[5, 6)$, as only then is $J_{1,1}$ one of the $M = 2$ highest-priority active jobs. In checking schedulability, both pi-blocking times and HAC execution times are analytically viewed as preemptive CPU execution time, as depicted in Fig. 8b. Note that this execution-time inflation may not include all task suspension time, but only that occurring while a task is actually pi-blocked.

▷ **Def. 2.** *Let $X \triangleq (2M - 1)B_{max}$ be the maximum duration of blocking induced by the global OMLP on an identical multiprocessor without time partitioning [22].* ◁

We now calculate the total HAC-related blocking by accounting for zone-related blocking.

Lemma 1. *The total pi-blocking introduced by the management of non-preemptive HAC accesses is at most $X + \lceil (X + B_{max}) \,/\, (\Theta - B_{max}) \rceil \cdot B_{max}$ time units for each lock request by a task in Component Γ.*

Proof. In each time slice of length Θ, because B_{max} is the worst-case duration of a forbidden zone, at least $\Theta - B_{max}$ time units are available for a job to initiate an access to

[2] A similar idea was later applied in a uniprocessor component-based setting [14], but that work did not target graphs as considered in this paper.

a HAC. In executing this work, and while the access is unfinished, additional blocking of up to B_{max} time units may be incurred for each time-slice boundary crossed. Thus, we upper bound the number of such boundaries that may be crossed between the initiation of the request and the completion of the access. The worst case occurs when the request is initiated right before a time-slice boundary, in which case $\lceil (X + B_{max}) \,/\, (\Theta - B_{max}) \rceil$ boundaries are crossed. □

Skipping ahead. If the next HAC access of the job at the head of the OMLP's FIFO queue is requested within its forbidden zone, then we can allow other access requests to "skip ahead" of that access until the beginning of the next time slice. This corresponds to the *Skip Protocol* proposed previously [46]. With skipping, the total blocking bound of an access of length B can be reduced to $X + \lceil (X + B) \,/\, (\Theta - B) \rceil B$.

Inflating execution times. To convert (analytically) to a CPU-only workload, as in Sec. III, we must add to the WCET of each task in Γ the maximum blocking duration specified in Lemma 1 and the duration of the HAC access itself for each such access. This WCET inflation may cause task utilizations to exceed 1.0, which is an additional motivation to use the rp-sporadic task model instead of the standard sporadic model.

Extending to multiple HACs. Extending to multiple HACs per component is straightforward. Each HAC is governed by its own OMLP lock within the component. The analysis must be modified such that B_{max} and X are defined per HAC.

If a set of k HACs is interchangeable (any HAC in the set can service a request), then the k HACs can instead be managed by a k-exclusion locking protocol (which allows up to k "lock holders"). This roughly divides X by a factor of k in the analysis. In practice, however, certain HACs (like GPUs in some use cases) may be unreasonable to view as interchangeable due to high costs associated with moving task state between HACs.

V. Computing Response Times under Time-Slicing

In this section, we provide response-time analysis for tasks in Γ (*i.e.*, CPU tasks with WCETs inflated using Lemma 1). We seek to leverage existing response-time analysis for rp-sporadic tasks on multiprocessor platforms for which (unlike our considered platform) all CPUs are fully available [7]; our results are applicable for any such analysis. This per-task analysis is used to compute task offsets and thereby obtain a graph response-time bound (the last step in Fig. 3).

As depicted in Fig. 3, Steps 4 and 5 define the sequence of processing-supply transformations, starting with the supply given by Γ's PCR and ending with one corresponding to a fully available platform. Step 4 transforms our reservation to a reservation with the same schedule and continuous supply. Step 5 inflates task execution times to apply the analysis of [7]. We perform these transformations such that no job's response time decreases, and we track task utilization changes as we transform. It is important to track utilizations, because as noted in Sec. III, the rp-sporadic task model requires $u_i \leq P_i$ for each task τ_i. To prevent over-utilization on an M-CPU unit-speed platform, we also require that the total utilization of all tasks in Γ, denoted as U, satisfies $U \leq M$.

Step 4: Transform PCR to a continuous supply. Firstly, we transform the PCR parameterized by Θ, Π, and M to a continuous processing supply without changing the total processing capacity supplied over long time intervals (*e.g.*, the hyperperiod of all system reservations).

▷ **Def. 3.** *Let Φ_{res} be the start of the first time slice of the reservation of Γ. As the reservation is periodic, its time slices are $[\Phi_{res} + i\Pi, \Phi_{res} + i\Pi + \Theta)$ for $i \in \{0, 1, 2, ...\}$ and $\Phi_{res} + \Theta \leq \Pi$.* ◁

Def. 3 allows us to describe the transformed CPU platform.

▷ **Def. 4.** *Define a new platform with M CPUs, each with speed Θ/Π, that begins supplying processing time at time Φ_{res}. We call this platform the* reduced-speed platform. ◁

The total supply provided to Γ by the initial and reduced-speed platforms is the same over any time interval of length $i\Pi$ for $i \in \{0, 1, 2, ...\}$ that starts after Φ_{res}. The processing supply provided by both platforms is depicted in Fig. 9a.

Step 4: Transform releases and deadlines. Although both platforms deliver equal processing supply in the long run, the change in how processing supply is provided changes the schedule significantly. In particular, job completion times may differ greatly. For example, consider a task in Γ that releases a job in the interval after one time slice completes but before the next begins. On the initial platform, such a job must wait until the next slice to be considered for execution, whereas on the reduced-speed platform, it would execute immediately if there are free CPUs. To avoid this issue, we transform job releases and deadlines of the tasks in Γ.

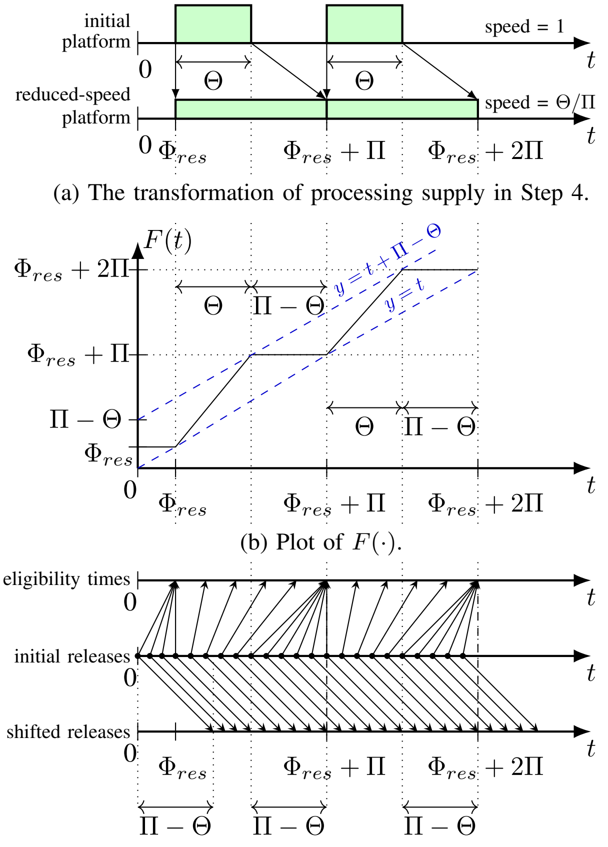
▷ **Def. 5.** *Define the piecewise-linear function $F(\cdot)$, plotted in Fig. 9b (solid line), as follows:*

$$F(t) = \begin{cases} \Phi_{res}, & \text{if } t \in [0, \Phi_{res}), \\ \Phi_{res} + i\Pi + z\Pi/\Theta, & \text{if } t \in [\Phi_{res} + i\Pi, \\ & \qquad \Phi_{res} + i\Pi + \Theta), \\ \Phi_{res} + (i+1)\Pi, & \text{if } t \in [\Phi_{res} + i\Pi + \Theta, \\ & \qquad \Phi_{res} + (i+1)\Pi), \end{cases}$$

where $z = t - \Phi_{res} - i\Pi$. ◁

To circumvent the issue of inconsistent allocations noted above, we shift all job releases and deadlines on the reduced-speed platform into the future by $\Pi - \Theta$ time units compared to the initial platform. We also allow the early releasing of jobs on the reduced-speed platform. Specifically, letting $r_{i,j}$ be the release time of a job on the initial platform, we define its release time on the reduced-speed platform to be $(r_{i,j} + \Pi - \Theta)$ and its eligibility time to be $F(r_{i,j})$ (we explain below why this is "early"), as illustrated in Fig. 9c.

To preserve the scheduling order, we break deadline ties the same way as before the transformation. Thus, the priority order of jobs does not change after this transformation. To explore



(a) The transformation of processing supply in Step 4.

(b) Plot of $F(\cdot)$.

(c) The transformation of releases and eligibility times.

Fig. 9: Transformation clarification figures.

how schedules on the initial and reduced-speed platforms are now connected, we first need bounds on $F(\cdot)$.

Lemma 2. $t \le F(t) \le t + (\Pi - \Theta)$.

Proof. Fig. 9b illustrates the lemma statement. First, consider $t \in [0, \Phi_{res})$. By Def. 5, $t < F(t)$. Furthermore, because the first reservation slice is $[\Phi_{res}, \Phi_{res} + \Theta)$, and the reservation is periodic, $\Phi_{res} + \Theta \le \Pi$. Thus, $F(t) = \Phi_{res} \le \Pi - \Theta \le t + \Pi - \Theta$.

Second, consider $t \in [\Phi_{res} + i\Pi, \Phi_{res} + i\Pi + \Theta)$ for some integer i, and $z = t - \Phi_{res} - i\Pi$. In this case, by Def. 5,

$$\begin{aligned} t &= \Phi_{res} + i\Pi + z \\ &\le \Phi_{res} + i\Pi + z \cdot \Pi/\Theta && \{= F(t)\} \\ &\quad \{\text{because } \Theta \le \Pi \text{ and } z \ge 0\} \\ &= \Phi_{res} + i\Pi + z + z \cdot (\Pi/\Theta - 1) \\ &= t + z \cdot (\Pi/\Theta - 1) \\ &\le t + \Theta \cdot (\Pi/\Theta - 1) && \{\text{because } z \le \Theta\} \\ &= t + (\Pi - \Theta). \end{aligned}$$

Finally, if $t \in [\Phi_{res} + i\Pi + \Theta, \Phi_{res} + (i+1)\Pi)$, then by Def. 5, $F(t) = \Phi_{res} + (i+1)\Pi$, so $t < F(t)$. Additionally, $F(t) = \Phi_{res} + i\Pi + \Pi + (\Theta - \Theta) \le t + (\Pi - \Theta)$, as $t \ge \Phi_{res} + i\Pi + \Theta$. □

By Lemma 2, on the reduced-speed platform, a job with a release at time $r_{i,j}$ can be scheduled at time $F(r_{i,j})$ due to its defined eligibility time, while its actual release happens at time $r_{i,j} + \Pi - \Theta \ge F(r_{i,j})$.

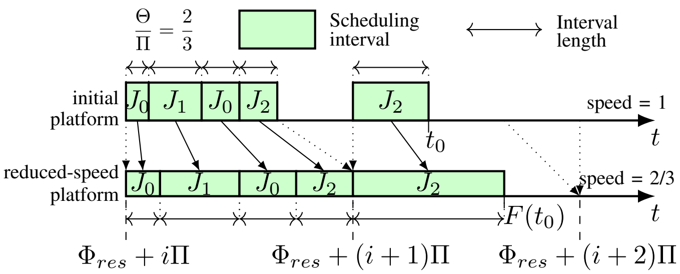


Fig. 10: Lemma 3 intuition.

▷ **Def. 6.** *Let S_{in} be the schedule of Γ on the initial platform defined by PCR (Θ, Π, Υ). Let S_{tr} denote the corresponding schedule on the reduced-speed platform, with job releases and deadlines adjusted as above.* ◁

The next lemma gives the schedule correspondence we seek.

Lemma 3. *A job is scheduled at time t in S_{in} if and only if it is scheduled at time $F(t)$ in S_{tr}.*

Proof. Fig. 10 illustrates the proof. Assume, for the purpose of contradiction, that the lemma does not hold, *i.e.*, that there exists a time t such that the sets of scheduled jobs in S_{in} at t and S_{tr} at $F(t)$ differ. Let t_0 be the first such time instant.

By the definition of Φ_{res}, no jobs are scheduled in either S_{in} or S_{tr} within $[0, \Phi_{res})$, so $t_0 > 0$. By the definition of t_0, any scheduling interval of a job within $[0, t_0)$ in S_{in} is transformed into a scheduling interval of the same job in S_{tr} by $F(\cdot)$. By Def. 5, any such interval of length h is transformed into a scheduling interval of length $h \cdot \Pi/\Theta$, scheduled on a CPU with speed Θ/Π, because a job can be scheduled in S_{in} only during active reservation slices (see Fig. 10). This results in the same total amount of completed work, h, for both the initial and transformed intervals. Thus, the amount of completed work for each job in S_{in} within $[0, t_0)$ is identical to the amount of completed work for the same job in S_{tr} within $[0, F(t_0))$. As the eligibility times of jobs are also transformed from the actual releases with $F(\cdot)$, the sets of uncompleted jobs at t_0 in S_{in} and at $F(t_0)$ in S_{tr} are identical. The transformation process does not affect the relative order of deadlines, so the set of scheduled jobs is the same in S_{in} at t_0 and in S_{tr} at $F(t_0)$, which contradicts the definition of t_0. □

Because $F(\cdot)$ directly transforms S_{in} into S_{tr}, we can bound the response time of an initial job in S_{in} via the response time of its transformed job in S_{tr}.

Theorem 1. *If a job has a response time of R_{tr} in S_{tr}, then its response time R_{in} in S_{in} is at most $R_{tr} + \Pi - \Theta$.*

Proof. Let $r_{i,j}$ be release time of the job in S_{in}, and let $f_{i,j}$ be its completion in S_{in}. Then $R_{in} = f_{i,j} - r_{i,j}$. By Lemma 3, all of this job's scheduling intervals in S_{in} are transformed via $F(\cdot)$ into scheduling intervals in S_{tr}. Thus, the job is completed at time $F(f_{i,j})$ in S_{tr}, and by definition of the transformed release time, $R_{tr} = F(f_{i,j}) - (r_{i,j} + \Pi - \Theta)$. By

Lemma 2, $F(f_{i,j}) \geq f_{i,j}$, so $R_{tr} \geq f_{i,j} - (r_{i,j} + \Pi - \Theta) = R_{in} - (\Pi + \Theta)$. Thus, $R_{in} \leq R_{tr} + \Pi - \Theta$. □

Step 4 does not affect task utilizations, so thus far, no changes to schedulability conditions are required. However, such changes are inevitable because the initial reservation restricts processing supply. These changes occur in the final step, discussed next.

Step 5: From reduced-speed supply to identical multiprocessor. On the reduced-speed platform, each CPU has speed Θ/Π. However, we can easily rescale these speeds to 1.0 by multiplying by the factor Π/Θ, which requires correspondingly multiplying each C_i by Π/Θ (thus, the utilization of task τ_i on the identical multiprocessor platform is defined as $u'_i = u_i \cdot \Pi/\Theta$). This step allows us to completely abstract the reservation and consider the scheduling of Γ on an identical multiprocessor platform with M CPUs with WCETs that have been inflated to account for HAC accesses.

By Theorem 1, Step 4 preserves the system schedulability. Step 5 preserves the schedule, so it preserves schedulability too. Thus, the schedulability conditions for Γ before Steps 4 and 5 can be derived from those of the system after Step 5: $U' \leq M$, and $\forall i \; u'_i \leq P_i$ [7], where $U' = \Pi/\Theta \cdot U$ is the modified system's utilization.

Lemma 4. *The schedulability conditions of* Γ *before Steps 4 and 5 are* $U \leq \Theta/\Pi \cdot M$ *and* $u_i \leq \Theta/\Pi \cdot P_i$.

We now consider the scheduling of tasks in Γ on an identical multiprocessor (recall that there are M CPUs available to Γ) with WCETs and releases altered as described above. Let $C'_i = \Pi/\Theta \cdot C_i$ be the altered WCET for task $\tau_i \in \Gamma$. We leverage results for rp-sporadic tasks on uniform multiprocessor+accelerator platforms [7].

▷ **Def. 7.** *We call a task* $\tau_i \in \Gamma$ p-restricted *(i.e., parallelism is restricted) if* $P_i < M$. *Let* U'_{res} *be the sum of the* $M-1$ *largest altered utilizations of p-restricted tasks in* Γ. *Let* C'_{res} *be the sum of the* $M-1$ *largest altered WCETs of p-restricted tasks in* Γ. ◁

If $\forall i, P_i \geq \alpha$ for some integer $\alpha \geq 2$, then the above terms can be reduced: $M-1$ can be replaced by $\lfloor (M-1)/\alpha \rfloor$ in the definitions of both C'_{res} and U'_{res}. As seen in the bound stated next, these reductions imply that increasing parallelism decreases response-time bounds (which is intuitive).

Lemma 5 (follows from Corollary 1 of [7])**.** *The response time of any task* $\tau_i \in \Gamma$ *is bounded by* $x + T_i + C'_i$, *where*

$$x = \frac{(M-1)C'_{max} + 2C'_{res}}{M - U'_{res}},$$

and C'_{max} *is the largest altered WCET of any task in* Γ.

The bound in Lemma 5 can be reduced by computing x iteratively (see Theorem 1 of [7]), or by computing per-task x_i values (via compliant-vector analysis [37], [39]). We stress that Lemma 5 merely provides one method for computing response-time bounds; any response-time analysis for sporadic tasks on an M-CPU identical multiprocessor may be used.

VI. Dealing with Highly Contended HACs

Unfortunately, a single long HAC access can cause high blocking times (by Def. 2 and Lemma 1). These blocking times are used to inflate tasks' execution times, so even a short access can be inflated by a relatively large value,[3] which produces pessimistic response-time bounds (or makes the system unschedulable). In this section, we describe an optional step that may reduce these bounds (Step 6 in Fig. 3).

Example 1. *Consider a system with a single component* Γ *comprised of 15 tasks with continuous access to eight CPUs and a single HAC. Assume that each task performs 1 time unit of CPU execution and has a period of 30 time units (thus, from the CPU point of view, the system has utilization* 0.5*).*

Additionally, assume that eight of the tasks must also access the HAC once per job for 2 time units. Each HAC-accessing task has its WCET increased by $X = (2 \cdot 8 - 1)2 = 30$ *under the global* OMLP. *Thus, each HAC-accessing task's utilization increases by* $30/30 = 1$, *leading to a total utilization increase of* $(8 \cdot 1) = 8$. *The increase in total utilization due to blocking alone consumes the capacity of all eight allocated CPUs!*

Step 6: Factoring out HACs. Total utilization explodes in Ex. 1 because the increase in utilization caused by lock-related blocking scales with the number of HAC-accessing tasks *multiplied* by the number of CPUs. This would be exacerbated if supply was not continuous under Γ's PCR due to additional blocking from forbidden zones. This encourages *factoring out* the problematic HAC and its associated tasks into a subsystem with the smallest possible CPU count, namely one. Tasks accessing that HAC are then scheduled non-preemptively and busy-wait on the dedicated CPU during their accesses, ensuring that jobs access the HAC immediately upon request (thus removing the need for a locking protocol).

Example 1 (cont'd)**.** *Suppose instead that the seven CPU-only tasks are scheduled on seven CPUs and the eight HAC-accessing tasks are scheduled exclusively on a dedicated CPU. Factoring out the HAC results in* $1 + 2 = 3$ *time units of execution per job and a utilization of* $3/30$ *per task (no blocking inflation is needed), totaling to* $8(3/30) = 24/30$ *for all HAC-accessing tasks. The seven CPU-only tasks can clearly be scheduled on the seven remaining CPUs. The dedicated CPU has greater capacity (1.0) than the HAC-accessing tasks (total utilization of* $24/30$*).*

Of course, not all systems will have only one HAC as in Ex. 1. If, after factoring out a single HAC, total utilization remains high due to accesses to other HACs, then we can simply continue to factor out HACs onto other dedicated CPUs until the remaining subsystem becomes schedulable.

Factoring assumptions. Being able to factor out HACs is predicated upon the assumption that all tasks that access a HAC fit on a single CPU. This is not a strong assumption for some AI applications, such as deep learning (*e.g.*, [18]), that

[3] Note that the Skip Protocol [46] improves only the forbidden-zone-related blocking time in Lemma 1 and does not affect the lock-related blocking X.

tend to be dominated by GPU workloads and require negligible CPU execution in comparison. Furthermore, if a HAC's tasks over-utilize a dedicated CPU and their CPU execution is negligible compared to their HAC execution, then the HAC must already be executing nearly continuously. This implies that HAC capacity is the limiting factor for schedulability, in which case adding more CPUs is ineffective.

We have also assumed that each task accesses at most one HAC (if not, tasks would need to migrate between different dedicated CPUs, requiring us to reintroduce locking to manage accesses). This assumption can be removed by allowing multiple HACs to share the same dedicated CPU (assuming their corresponding tasks fit on a single CPU). Because only one HAC is accessed by a task on a shared dedicated CPU at a time (dedicated CPUs schedule non-preemptively), the HACs that share the dedicated CPU are effectively combined into a single *logical HAC* with worst-case access time equal to the largest time to access any of the combined HACs.

Response-time analysis is needed for the tasks on dedicated CPUs. When the system is time-sliced (unlike in Ex. 1), forbidden zones of HACs must then be accounted for. By analytically pretending that the CPU is also unavailable when the HAC is in its forbidden zone, the response-time analysis for a HAC and its dedicated CPU in a PCR reduces to that of a single partially available CPU, for which existing analysis can be applied [32]. This existing analysis (which targets global scheduling) is pessimistic for dedicated CPUs, so we provide our own response-time analysis. This analysis is presented in an online appendix [84].

Choosing HACs to factor out. When factoring out HACs, some capacity will inevitably be lost due to partitioning tasks onto dedicated CPUs, making the choice of which HACs to factor out a trade-off. We assume that the number of HACs is small enough that considering every possibility of factoring out HACs is feasible (note that for any two HACs accessed by the same task, the fact that that task cannot be managed both by the OMLP and via a dedicated CPU means that either both or neither HAC must be factored out). Our proposed approach iterates through every possible factoring and chooses the factoring of HACs that best satisfies a given metric (*e.g.*, lowest maximum or average response-time bound).

In the discussion above, we have implicitly assumed that the precise HACs a task accesses are fixed and known (as noted earlier, using any of a set of HACs is sometimes not practical due to costs associated with moving task state among HACs). If HAC accesses are not fixed and known, then the full space of possible assignments of tasks to HACs must be considered. We do not consider this possibility here due to space constraints (while we cannot consider every possible nuance in detail here, the discussion above should make the idea of "factoring out" HACs intuitively clear). Generally, tasks that make long accesses should be made to access factored-out HACs.

The intuition behind preferentially assigning factored-out HACs to tasks with long accesses is that removing long accesses to HACs that are managed by the global OMLP reduces B_{max} (Def. 1), which in turn reduces X (Def. 2), thereby reducing blocking under the global OMLP (Lemma 1). As such blocking inflates jobs execution times, reducing blocking should reduce the total utilization.

Although our discussion here has focused on using the global OMLP, the principles above apply regardless of the choice of locking protocol.

VII. Reducing Response-Time Bounds

In graph-based scheduling, graph nodes are normally taken to be the schedulable entities, *i.e.*, the scheduler focuses on prioritizing and scheduling the tasks that define these nodes. However, as shown next, end-to-end response-time bounds can often be reduced by merging certain nodes to create new schedulable entities; this is the optional Step 7 in Fig. 3.

Task/node merging has been considered before, but mostly for reasons orthogonal to our work, such as to reduce task-to-task communication costs [8], [40], [95], enable task clustering [15], [31], [34], [87], or reduce energy consumption [70], [72]. Also, as noted in Sec. III, prior work [7], [90] used node merging to eliminate cycles. In fact, with respect to response-time bounds and schedulability, the conventional wisdom seems to favor *task splitting*, not merging [23], [30], [38], [41], [49], [50], [52], [78].

Recall from Sec. III (see Step 2) that the graph-scheduling approach of release offsets results in end-to-end response-time bounds proportional to the longest path in the DAG. In this section, we show that such bounds can be reduced via restructuring such graphs by merging certain nodes.

Redefining schedulable entities through node merging. We illustrate our approach with a simple example, which does not involve HACs or time-partitioning but still illustrates key concepts. Consider a component Γ that executes on a four-core platform without HACs and consists of one graph without delay dependencies, as depicted in Fig. 11. Assume that this DAG has a period of 15, a parallelization level of one, and its tasks have execution times of $\{3, 1, 2, 4, 5\}$ time units.

As shown in Tbl. II, we can compute an end-to-end DAG response time-bound of 122.8 time units using Lemma 5. This same table allows us to infer individual task response-time bounds (which are of the form $x + C_i + T_i$) from the x value given. This response-time bound calculation is impacted by the offset-based approach introduced earlier in Sec. III.

Now consider the same graph, but with tasks τ_3 and τ_4 considered as a single schedulable entity (colored gray in Fig. 11). We call this new task τ_{new}. Note that τ_{new} has an execution time of 6 time units, so C_{max}, C_{res}, and U_{res} (in Lemma 5) are changed. The end-to-end bound for this new graph is given in Tbl. II; x is increased (and hence per-task bounds increased), but the end-to-end response time of the graph is 104 time units, a 15% reduction.

Step 7: Graph node merging. The main source of pessimism in the end-to-end bound is the general looseness of the per-task response-time bounds. Most papers that derive response-time bounds like we use here [7], [33], [37], [39], [80], [89] develop a bound of at least $C_i + T_i + x_i$ for task τ_i with $x_i > 0$ (x_i

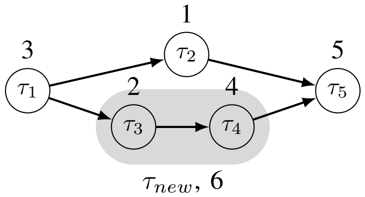


Fig. 11: Scheduling entities of the graph.

	C_{max}	C_{res}	U_{res}	x	end-to-end r.t.
Unmodified DAG	5	12	0.8	12.2	122.8
$\{\tau_3, \tau_4\} \to \tau_{new}$	6	14	0.93	15	104

TABLE II: Response-time bounds for the graph in Fig. 11.

may be constant for all tasks). Denote the *length* of a path in a DAG as the sum of the per-node response-time bounds over the path. Then, merging two nodes in the longest path reduces the end-to-end DAG response-time bound by approximately T_i time units if other paths have lesser lengths and per-node bounds do not change significantly due to merging.

From the discussion above it may seem desirable to merge many nodes into one (*e.g.*, merge all nodes of the DAG into one node). However, the ordinary sporadic task model limits merging opportunities: for the system to remain schedulable, the total utilization of a merged node must be at most 1.0. Fortunately, the rp-sporadic task model eases utilization restrictions: the total utilization of a merged node is limited by the graph parallelization level.

Connections between merging and bin-packing. While node merging can decrease end-to-end response-time bounds, the choice of which nodes to merge to minimize the longest path is complex. We illustrate this complexity with an example. In this example, we assume that per-task response-time bounds do not change significantly after merging nodes relative to the length of the longest path (*i.e.*, a path of two nodes always has smaller length than a path of three nodes).

Consider a set of rp-sporadic tasks $\tau = \{\tau_0, \tau_{i_1}, ..., \tau_{i_s}, \tau_{j_1}, ..., \tau_{j_q}\}$ with parallelization level P, where task τ_0 has utilization equal to P, and tasks $\{\tau_{j_1}, ...\tau_{j_q}\}$ have utilizations of $\varepsilon \to 0$. The DAG specifying their dependencies is shown in Fig. 12.

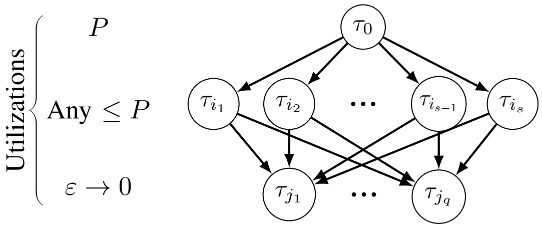


Fig. 12: Shortest path elimination as bin packing.

Note that the longest path in the graph contains three nodes. Also note that τ_0 cannot be merged with any other node, because of its utilization constraint. Thus, the shortest possible path length after node merging is two. To achieve the longest path of length two, each node of $\{\tau_{i_1}, ..., \tau_{i_s}\}$ should be merged with one node of $\{\tau_{j_1}, ...\tau_{j_q}\}$. The only constraints on which nodes may be merged are utilization constraints, and thus, the minimization of the shortest path in this graph is equivalent to the bin-packing problem with j_q bins of size $P - \varepsilon$.

Heuristics. The common way of solving a problem that likely has no polynomial-time solution is by using heuristics. Each heuristic chooses a valid pair of nodes to merge, and is applied repeatedly until no new pair is chosen (*i.e.*, response-time bound R would not decrease further). We experimentally show that these heuristics can enable significant reductions in graph response-time bounds. As our systems contain several DAGs, we use the largest response-time bound of these DAGs as the system's response-time bound.

Our first heuristic computes the longest path in any graph of the system, weighing each of the z system nodes by their individual response-time bound R_i. As we consider DAGs, this step takes $O(z)$ time. From the longest path, this heuristic randomly chooses two consecutive nodes. We call this heuristic *SinglePathRT*.

The second heuristic, *BestPair*, considers each of the $O(z^2)$ pairs of nodes and computes R assuming they are merged (including all $O(z)$ nodes on any path between them); this heuristic chooses a valid pair that minimizes R.

We also consider the simpler version of BestPair, *ElementaryPair*. It considers only nodes connected by a direct edge that do not have any other path between them.

Experimental setup. To determine the efficacy of our heuristics, we generated synthetic graph task systems consisting of several connected DAGs. First, for each DAG, we allocated a number of nodes from the total nodes in the system. Second, we generated a random tree over these nodes to make the DAG connected. Finally, the remaining edges were generated using the Erdős-Rényi model [36] (with constant probability). Each edge was made a dependency by making the lower-indexed task a predecessor of the higher-indexed task (thereby guaranteeing that dependencies form DAGs).

Tasks' parameters were generated independently. P_i for each task τ_i in each subset was sampled uniformly from $[2, 3, 4]$. T_i was sampled uniformly from the interval of possible periods $[10, 50]$. The utilization of nodes was uniformly chosen by the Dirichlet-Rescale algorithm [44].

Evaluation metric. To evaluate these heuristics, we use two metrics: the *share of improved graphs (SIG)* and the *relative end-to-end bound improvement (RBI)*. SIG is the percentage of DAGs that see a response-time bound improvement. RBI estimates how good the improvement is. For a DAG D, it is defined as $(R_{in} - R_m)/R_{in}$, where R_{in} is D's initial end-to-end bound, and R_m is D's bound after merging nodes. To compute response-time bounds, we used a tighter version of Lemma 5 (Theorem 1 of [7]).

CPU-only graph workloads. Our first experiment compares the heuristics on the same graphs with CPU-only nodes without reservations. The goal of the experiment is to show that the node merging approach works for the most common case and is not tied to the usage of HACs or time partitioning (as it improves the offset-based response-time bound computation).

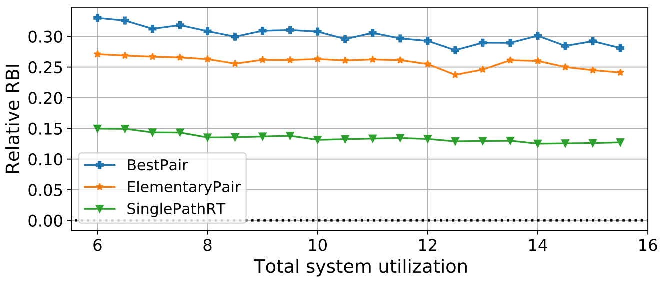


Fig. 13: Comparison of heuristics with CPU-only workloads.

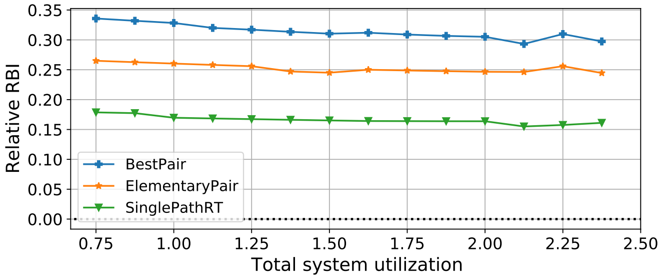


Fig. 14: Comparison of heuristics with CPU/HAC workloads within PCR.

We generated graph task systems with five connected components and 100 nodes total on a platform with 16 CPUs. The total utilization of these systems varies from 6 to 15.5. We generated 60 tasks systems for each choice of total utilization.

Fig. 13 shows RBI trends for this experiment. BestPair is the best heuristic (with a relative RBI of around 30%), with ElementaryPair being second best. Note that, for all heuristics, RBI trends downward as system utilization increases. This trend is due to fewer options for merging at higher utilization.

CPU+HAC workloads within PCRs. In the second experiment, we evaluated the above heuristics in the presence of HAC nodes within PCRs, and sought to demonstrate that introducing HACs and PCRs does not compromise the efficacy of heuristics. We assumed that all HAC accesses are arbitrated using a locking protocol (as described in Sec. IV).

We generated graph task systems with five connected components and 100 total nodes for a PCR with six CPUs and two HACs ($\Theta = 40, \Pi = 80$). Each node is a HAC-only node with a probability of 0.35. As the total CPU capacity of this PCR is 3.0, no system with utilization higher than 3.0 can be schedulable. Moreover, some nodes are HAC-only: the execution times of these nodes must be inflated to reflect lock-related blocking. Thus, most of the generated systems with utilizations of more than 2.4 were unschedulable (thus, they are not presented). The total utilization of the considered systems varies from 0.75 to 2.4. We generated 60 task systems for each choice of total utilization.

Fig. 14 shows RBI trends for this experiment (with results similar to the previous experiment). BestPair is again the best heuristic, with ElementaryPair being second best. Note that, for all heuristics, RBI trends downward as system utilization increases. This trend is again due to fewer options for merging at higher utilizations.

Efficacy vs. run time. In both experiments, heuristics with higher efficacy had higher run time. For a connected DAG with z nodes and E edges, the number of merge attempts per step is proportional to z^2 for BestPair, to E for ElementaryPair, and to z for SinglePathRT. Thus, BestPair should be used generally unless the number of nodes makes its run time infeasible, in which case a heuristic with lower efficacy can be used (*e.g.*, ElementaryPair has comparable results to BestPair)

As these experiments show, our node-merging heuristics have value in systems with and without HACs or time-partitioning.

VIII. Experimental Validation

A full experimental evaluation of the real-time graph-based analysis presented in this paper must answer three questions:

Question 1: How long are forbidden zones in practice?

Question 2: What is the cost of providing isolation between HAC-using components?

Question 3: What reservation time-slice lengths are appropriate for a given task system?

Answering these questions requires a full operating-system-based implementation, which is beyond the scope of this paper. These questions are answered in detail in a companion paper [6], in which we introduce TimeWall, a time-partitioning framework that supports scheduling graph-based task systems within reservations; TimeWall is implemented within the 5.4.0-rc7 LITMUS$^{\mathrm{RT}}$ kernel [20], [26].

In seeking to answer Questions 1–3, our experimental evaluation of TimeWall revealed some surprising timing edge cases for GPU-using workloads, necessitating further explorations:

Question 4: How should HAC accesses be budgeted in the event of high worst-case access durations?

We now summarize the answers to these questions using the key results of our evaluation; full details can be found in the companion paper [6].

Q1: Forbidden-zone lengths. To determine how long forbidden zones can be in practice, we measured durations of data copies and computations for a GPU-using pedestrian-detection application. We found that CPU- and GPU-based measurements differed beyond the 99.95^{th} percentile; worst-case CPU-based measurements were two orders of magnitude higher than those taken on the GPU. Our investigation into these edge cases revealed NVIDIA's proprietary CUDA API as the culprit. As we could not remove the source of the edge cases, we added a GPU budgeting mechanism to TimeWall to prevent misbehaving jobs from performing additional work.

Q2: Cost of isolation. In TimeWall, forbidden-zone enforcement increases lock overheads. Our measured lock overheads were comparable to the original global OMLP ($5.4\ \mu$s per request), with an additional $0.9\ \mu$s per GPU access. We also found that using TimeWall instead of the original global OMLP (without support for forbidden zones) resulted in slightly higher median and lower worst-case response times.

Q3: Time-slice lengths. Recall from Sec. IV that we require $\Theta \geq B_{max}$. However, long time slices can result in high response times due to the interval of length $\Pi - \Theta$ when jobs are released but not scheduled (*i.e.*, when other components' jobs are scheduled). We explored this trade-off via synthetic GPU-using tasks, assuming $\Theta/\Pi = 0.5$, and found that the lowest response times occurred when $\Theta > C_i$ and $\Pi \leq T_i$.

Q4: Budgeting HAC accesses. Increasing the budget for HAC accesses reduces the number of accesses that exceed that budget. Budgeting at the 99.95^{th} percentile of CPU-based measurements results in a theoretical frame-drop rate (*i.e.*, the proportion of jobs during which an access duration exceeded its budget) of 3.8%, but allows for shorter forbidden zones (closer to the GPU-measured worst cases).

IX. Discussion

In this paper, we are motivated by graph-based AI algorithms, of which neural networks (NNs) currently serve as the most popular class. These NNs (VGG [77], ResNet [45], Inception [79], *etc.*) are generally assumed to use one or several GPUs, utilizing all available capacity, and are represented by large multi-node graphs. Unfortunately, the high latency and low bandwidth of CPU-GPU memory transfers make it inefficient to offload nodes to the CPU, so an optimal scheduler should consider an invocation of a typical NN as a single GPU job (and the NN itself as a single GPU graph node).

However, real applications may be comprised of many additional nodes, including CPU ones (*e.g.*, data processing or sequential computations), as well as other HAC-using tasks, which together form the graph we consider. An example of such an application was mentioned in the RTAS 2021 Industry Panel [1] (at time 59:00).

X. Related Work

Prior work on real-time graph models has mostly focused on the DAG model, which differs from our model by considering different timing constraints (*e.g.*, hard real-time with implicit/constrained deadlines) [11], [19], [68], [69], [73], [94], ignoring inter-instance graph dependencies (*e.g.*, hard real-time with arbitrary deadlines) [12], [42], [51], [55], [66], [85], arbitrarily forcing these dependencies to the prior instance only [57], [90]–[92], or by considering a different graph task model (*e.g.*, conditional DAGs) [13], [43], [48], [50], [59], [66]. Some other work on graph scheduling considered different schedulers [12], [49], [56], [61], [71], [74], [83] or objectives (*e.g.*, minimizing makespan or energy consumption) [15], [16], [47], [85], [88].

Other real-time graph models include the synchronous dataflow graph model (SDF) and digraph task model. These models can be used under our approach with minor analysis modifications (*e.g.*, SDF graphs can be converted into DAGs or sporadic tasks [4], [9], [10], [58], [63]).

GPUs and other HACs. GPUs are perhaps the most popular type of HAC, as they are commonly used to accelerate AI applications. Of the prior work that considers GPU or HAC accesses in graph-based task systems, most use locking protocols to arbitrate accesses (*e.g.*, [7], [62], [90]), whereas others rely on the underlying driver's scheduling policies (*e.g.*, [92]). GPU-access arbitration has typically either used a real-time locking protocol (*e.g.*, [35], [53], [82]) or relied on modifications to the GPU driver to reorder work [27], [54]. Alternatively, some work has sought to understand the scheduling rules employed by the GPU drivers themselves through micro-benchmarking experiments [5], [64], [65].

Isolation of system components. Real-time isolation of system components is typically done using the notion of virtual processors (or reservations) with guaranteed capacity. We consider a simple PCR model that requires capacity to be supplied over a continuous time interval; more general models (*e.g.*, MPR [76] or service functions [29]) cannot be easily used instead because of HAC non-preemptivity issues. However, the prior work with these models uses a simpler (and less general) task model (*e.g.*, standard sporadic task) without accelerator accesses [2], [17], [24].

Work that considers graph models and component isolation [25], [28], [86], [93] does not address the complexities related to the usage of non-preemptive accelerators, such as the potential for component-isolation violations. Nemati *et al.* [62] considered accelerator accesses in component-based systems, but they required that a given CPU be dedicated to a single component, and thus did not consider a model in which a component has exclusive access to a HAC.

XI. Conclusion

We have extended prior work on computing response-time bounds for graph-based accelerator-using computations so that such bounds can be computed in systems of time-partitioned components. In doing so, we have explored various nuances that arise when attempting to support accelerator accesses efficiently without compromising inter-component time isolation. Time partitioning can increase response-time bounds. To ameliorate this issue, we have presented node-merging heuristics that restructure graphs. To our knowledge, this is the first work to consider node merging as a means for reducing response times. These heuristics are of interest even in the absence of time partitioning.

This paper was motivated by the desire to efficiently support AI workloads on multicore+HAC platforms. The techniques we have proposed have been applied in related work that focuses on experimental trade-offs involving such workloads [6]. That work shows that the concepts we have discussed (the rp-sporadic task model, lock-based accelerator arbitration, forbidden zones, *etc.*) can be practically applied on real hardware.

Many avenues for further work exist. For example, it would be interesting to consider resource models that are more flexible than the PCR model. Conditional graphs, which are often used in AI applications, also warrant consideration. Finally, many trade-offs exist when allocating time partitions to the components of a system. We plan to investigate these trade-offs both theoretically and experimentally.

References

[1] Panel Discussion – RTOS for Autonomous Machines, 27th IEEE Real-Time and Embedded Technology and Applications Symposium. www.youtube.com/watch?v=VYUWwUHMcj4, May 2021.

[2] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the linux kernel. *ACM SIGBED Review*, 2019.

[3] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert Ian Davis. An empirical survey-based study into industry practice in real-time systems. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*, 2020.

[4] Hazem Ismail Ali, Benny Akesson, and Luis Miguel Pinho. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015.

[5] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, 2017.

[6] Tanya Amert, Zelin Tong, Sergey Voronov, Joshua Bakita, F Donelson Smith, and James H Anderson. Timewall: Enabling time partitioning for real-time multicore+accelerator platforms. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, 2021.

[7] Tanya Amert, Sergey Voronov, and James H Anderson. OpenVX and real-time certification: The troublesome history. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, 2019.

[8] Peter Aronsson and Peter Fritzson. Task merging and replication using graph rewriting. In *Proceedings of the 10th International Workshop on Compilers for Parallel Computers*, 2003.

[9] Mohamed A Bamakhrama and Todor Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the 9th ACM International Conference on Embedded Software*, 2011.

[10] Mohamed A Bamakhrama and Todor Stefanov. Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the 8th IEEE International Conference on Hardware/Software Codesign and System Synthesis*, 2012.

[11] Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systemss*, 2014.

[12] Sanjoy Baruah. The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the 19th Design, Automation and Test in Europe Conference and Exhibition*, 2015.

[13] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, 2015.

[14] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Nolin. SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM and IEEE International Conference on Embedded Software*, 2007.

[15] Ashikahmed Bhuiyan, Zhishan Guo, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient real-time scheduling of DAG tasks. *ACM Transactions on Embedded Computing Systems*, 17(5):1–25, 2018.

[16] Ashikahmed Bhuiyan, Di Liu, Aamir Khan, Abusayeed Saifullah, Nan Guan, and Zhishan Guo. Energy-efficient parallel real-time scheduling on clustered multi-core. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2097–2111, 2020.

[17] Enrico Bini, Marko Bertogna, and Sanjoy Baruah. Virtual multiprocessor platforms: Specification and use. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, 2009.

[18] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[19] 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*, 2013.

[20] Björn B Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2011.

[21] Björn B Brandenburg and James H Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, 2010.

[22] Björn B Brandenburg and James H Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342, 2013.

[23] Björn B Brandenburg and Mahircan Gül. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, 2016.

[24] Artem Burmyakov, Enrico Bini, and Eduardo Tovar. Compositional multiprocessor scheduling: the GMPR interface. *Real-Time Systems*, 50(3):342–376, 2014.

[25] Giorgio Buttazzo, Enrico Bini, and Yifan Wu. Partitioning real-time applications over multicore reservations. *IEEE Transactions on Industrial Informatics*, 7(2):302–315, 2011.

[26] John Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, 2006.

[27] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for GPU with preemption support. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, 2018.

[28] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, 2019.

[29] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the 7th Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[30] Jian-Jia Chen and Cong Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Proceedings of the 35th IEEE Real-Time Systems Symposium*, 2014.

[31] Klaus Danne and Marco Platzner. Periodic real-time scheduling for FPGA computers. In *Proceedings of the 3rd International Workshop on Intelligent Solutions in Embedded Systems*, 2005.

[32] UmaMaheswari C Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2006.

[33] UmaMaheswari C Devi and James H Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, 2005.

[34] Ahmed Ebaid, Reda Ammar, Sanguthevar Rajasekaran, and Tahany Fergany. Task clustering & scheduling with duplication using recursive critical path approach (RCPA). In *Proceedings of the 10th IEEE International Symposium on Signal Processing and Information Technology*, 2010.

[35] Glenn A Elliott, Bryan Ward, and James H Anderson. GPUSync: a framework for real-time GPU management. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, 2013.

[36] P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[37] Jeremy P Erickson and James H Anderson. Response time bounds for G-EDF without intra-task precedence constraints. In *Proceedings of the 15th International Conference On Principles Of Distributed Systems*, 2011.

[38] Jeremy P Erickson and James H Anderson. Reducing tardiness under global scheduling by splitting jobs. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, 2013.

[39] Jeremy P Erickson, Nan Guan, and Sanjoy Baruah. Tardiness bounds for global EDF with deadlines different from periods. In *Proceedings of the 14th International Conference On Principles Of Distributed Systems*, 2010.

[40] Hamid R Faragardi, Björn Lisper, Kristian Sandström, and Thomas Nolte. An efficient scheduling of AUTOSAR runnables to minimize communication cost in multi-core systems. In *Proceedings of the 7th International Symposium on Telecommunications*, 2014.

[41] David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A real-time scheduling service for parallel tasks. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013.

[42] 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.

[43] José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luís Miguel Pinho. A multi-DAG model for real-time parallel applications with conditional execution. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015.

[44] David Griffin, Iain Bate, and Robert I Davis. Generating utilization vectors for the systematic evaluation of schedulability tests. In *Proceedings of the 41th IEEE Real-Time Systems Symposium*, 2020.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[46] Philip Holman and James H Anderson. Locking under Pfair scheduling. *ACM Transactions on Computer Systems*, 24(2):140–174, 2006.

[47] Biao Hu, Zhengcai Cao, and Mengchu Zhou. Energy-minimized scheduling of real-time parallel workflows on heterogeneous distributed computing systems. *IEEE Transactions on Services Computing*, 2021.

[48] Xu Jiang, Nan Guan, Di Liu, and Weichen Liu. Analyzing G-EDF scheduling for parallel real-time tasks with arbitrary deadlines. In *Proceedings of the 23rd Design, Automation and Test in Europe Conference and Exhibition*, 2019.

[49] 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*, 2017.

[50] Xu Jiang, Xiang Long, Nan Guan, and Han Wan. On the decomposition-based global EDF scheduling of parallel real-time tasks. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, 2016.

[51] Xu Jiang, Jinghao Sun, Yue Tang, and Nan Guan. Utilization-tensity bound for real-time DAG tasks under global EDF scheduling. *IEEE Transactions on Computers*, 69(1):39–50, 2019.

[52] Augusto Santos Júnior, George Lima, Konstantinos Bletsas, and Shinpei Kato. Multiprocessor real-time scheduling with a few migrating tasks. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, 2013.

[53] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Raj Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, 2011.

[54] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 22th USENIX Annual Technical Conference*, 2011.

[55] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Analysis of global EDF for parallel tasks. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, 2013.

[56] 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*, 2014.

[57] Cong Liu and James H Anderson. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, 2010.

[58] Di Liu, Jelena Spasic, Jiali T Zhai, Todor Stefanov, and Gang Chen. Resource optimization for CSDF-modeled streaming applications with latency constraints. In *Proceedings of the 18th Design, Automation & Test in Europe Conference & Exhibition*, 2014.

[59] 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*, 2015.

[60] Aloysius K Mok, Xiang Feng, and Deji Chen. Resource partition for real-time systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, 2001.

[61] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, 2019.

[62] Farhang Nemati, Moris Behnam, and Thomas Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, 2011.

[63] Sobhan Niknam, Peng Wang, and Todor Stefanov. Hard real-time scheduling of streaming applications modeled as cyclic CSDF graphs. In *Proceedings of the 23th Design, Automation & Test in Europe Conference & Exhibition*, 2019.

[64] Ignacio S Olmedo, Nicola Capodieci, Jorge L Martinez, Andrea Marongiu, and Marko Bertogna. Dissecting the CUDA scheduling hierarchy: A performance and predictability perspective. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2020.

[65] Nathan Otterness and James H Anderson. Exploring AMD GPU scheduling details by experimenting with "worst practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, 2021.

[66] Andrea Parri, Alessandro Biondi, and Mauro Marinoni. Response time analysis for G-EDF and G-DM scheduling of sporadic DAG-tasks with arbitrary deadline. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, 2015.

[67] Paul J Prisaznuk. ARINC 653 role in integrated modular avionics (IMA). In *Proceedings of the 27th IEEE/AIAA Digital Avionics Systems Conference*, 2008.

[68] Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, 2013.

[69] Manar Qamhieh, Laurent George, and Serge Midonnet. A stretching algorithm for parallel real-time DAG tasks on multiprocessor systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, 2014.

[70] Qinru Qiu, Shaobo Liu, and Qing Wu. Task merging for dynamic power management of cyclic applications in real-time multiprocessor systems. In *Proceedings of the 24th International Conference on Computer Design*, 2006.

[71] Shenyuan Ren, Ligang He, Junyu Li, Chao Chen, Zhuoer Gu, and Zhiyan Chen. Scheduling DAG applications for time sharing systems. In *Proceedings of the 21st International Conference on Algorithms and Architectures for Parallel Processing*, 2018.

[72] Peng Rong and Massoud Pedram. Energy-aware task scheduling and dynamic voltage scaling in a real-time system. *Journal of Low Power Electronics*, 4(1):1–10, 2008.

[73] Abusayeed Saifullah, David Ferry, Chenyang Lu, and Christopher Gill. Real-time scheduling of parallel tasks under a general DAG model. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[74] Maria A Serrano, Alessandra Melani, Marko Bertogna, and Eduardo Quinones. Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In *Proceedings of the 20th Design, Automation & Test in Europe Conference & Exhibition*, 2016.

[75] Maria A Serrano and Eduardo Quinones. Response-time analysis of DAG tasks supporting heterogeneous computing. In *Proceedings of the 55th Annual Design Automation Conference*, 2018.

[76] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[77] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[78] Muhammad R Soliman and Rodolfo Pellizzoni. PREM-based optimal task segmentation under fixed priority scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, 2019.

[79] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[80] Guangmo Tong and Cong Liu. Supporting soft real-time sporadic task systems on uniform heterogeneous multiprocessors with no utilization loss. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2740–2752, 2015.

[81] 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*, 2018.

[82] Uri Verner, Avi Mendelson, and Assaf Schuster. Scheduling periodic real-time communication in multi-GPU systems. In *Proceedings of the 23rd International Conference on Computer Communication and Networks*, 2014.

[83] Micaela Verucchi, Mirco Theile, Marco Caccamo, and Marko Bertogna. Latency-aware generation of single-rate DAGs from multi-rate task sets. In *Proceedings of the IEEE 26th Real-Time and Embedded Technology and Applications Symposium*, 2020.

[84] Sergey Voronov, Stephen Tang, Tanya Amert, and James H Anderson. AI meets real-time: Addressing real-world complexities in graph response-time analysis (extended version with appendix), https://jamesanderson.web.unc.edu/papers/, 2021.

[85] Kankan Wang, Xu Jiang, Nan Guan, Di Liu, Weichen Liu, and Qingxu Deng. Real-time scheduling of DAG tasks with arbitrary deadlines. *ACM*

Transactions on Design Automation of Electronic Systems, 24(6):1–22, 2019.

[86] Yifan Wu, Zhigang Gao, and Guojun Dai. Deadline and activation time assignment for partitioned real-time application on multiprocessor reservations. *Journal of Systems Architecture*, 60(3):247–257, 2014.

[87] Yinglong Xia, Viktor K Prasanna, and James H Li. Hierarchical scheduling of DAG structured computations on manycore processors with dynamic thread grouping. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 2010.

[88] Guoqi Xie, Xiongren Xiao, Renfa Li, and Keqin Li. Schedule length minimization of parallel applications with energy consumption constraints using heuristics on heterogeneous distributed systems. *Concurrency and Computation: Practice and Experience*, 29(16):402, 2017.

[89] Kecheng Yang and James H Anderson. On the soft real-time optimality of global EDF on uniform multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, 2017.

[90] Kecheng Yang, Glenn A Elliott, and James H Anderson. Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 23th International Conference on Real-Time Networks and Systems*, 2015.

[91] Kecheng Yang, Ming Yang, and James H Anderson. Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, 2016.

[92] Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H Anderson, F Donelson Smith, and Shige Wang. Making OpenVX really "real time". In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, 2018.

[93] Tao Yang, Qingxu Deng, and Lei Sun. Building real-time parallel task systems on multi-cores: A hierarchical scheduling approach. *Journal of Systems Architecture*, 92:1–11, 2019.

[94] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*, 2020.

[95] Naqin Zhou, Deyu Qi, Xinyang Wang, and Zhishuo Zheng. A static task scheduling algorithm for heterogeneous systems based on merging tasks and critical tasks. *Journal of Computational Methods in Sciences and Engineering*, 17(4):715–732, 2017.

TABLE III: Notation.

Symbol	Meaning
HAC	Hardware accelerator
τ_i	i^{th} task
$J_{i,j}$	j^{th} job of τ_i
R_i	Response-time bound of τ_i (ignoring dependencies)
Φ_i	Release offset of τ_i
P_i	Parallelism level of τ_i
T_i	Period of τ_i
C_i	WCET of τ_i
u_i	Utilization of τ_i
$r_{i,j}$	Release time of $J_{i,j}$
Γ	Component of interest
PCR	Periodic component reservation
Υ	Computing resources of a PCR
M	Number of CPUs in Υ
Π	Period of PCR
Θ	Execution duration of PCR (every Π time units)
B_{max}	Longest HAC-access duration (Def. 1)
OMLP	$O(M)$ multiprocessor locking protocol [21]
X	Blocking induced by OMLP (Def. 2)
U	Total utilization of tasks in Γ
Φ_{res}	Start of first time slice of Γ (Def. 3)
$F(\cdot)$	Release time transformation function (Def. 5)
S_{in}	Initial schedule (Def. 6)
S_{tr}	Transformed schedule (Def. 6)
u_i'	Utilization of τ_i on identical multiprocessor platform
U'	Total utilization on identical multiprocessor platform
C_i'	WCET of τ_i on identical multiprocessor platform
U_{res}'	Sum of $M-1$ largest u_i' of p-restricted tasks (Def. 7)
C_{res}'	Sum of $M-1$ largest C_i' of p-restricted tasks (Def. 7)
C_{max}'	Largest C_i'
x	Component of response-time bound (Lemma 5)
R	End-to-end response time of graph system
z	Number of nodes in graph system
E	Number of edges in graph system
SIG	Share of improved graphs
RBI	Relative end-to-end bound improvement
$\mathcal{H}$	HAC of interest
$\mathcal{C}$	Dedicated CPU of $\mathcal{H}$
τ_g	Tasks accessing $\mathcal{H}$
$B_{max}^{\mathcal{H}}$	Duration of longest access of $\mathcal{H}$
C_{max}	Largest WCET of task in τ_g
V	Longest duration $\mathcal{H}$ is unavailable (Def. 9)
U_g	Total utilization of τ_g (Def. 10)
S_g	Long-run availability rate of $\mathcal{H}$ (Def. 11)

APPENDIX

RESPONSE-TIME ANALYSIS FOR FACTORED-OUT HACS

We assume non-preemptive earliest-deadline-first (NP-EDF) scheduling on the dedicated CPUs of HACs. We use EDF-based scheduling because our response-time analysis is not affected by early releasing (unlike other schedulers such as FIFO). Recall that the dedicated CPU for a HAC schedules the group of tasks that access said HAC. Under the notion of NP-EDF used here, we only forbid tasks in said group from preempting each other; in-progress jobs are still preempted at the end of time-slice boundaries (the use of forbidden zones guarantees that in-progress execution at a time-slice boundary is preemptible CPU work). Tasks busy-wait on dedicated CPUs while accessing HACs. Thus, their WCETs must be inflated by their worst-case HAC access times.

Consider HAC $\mathcal{H}$ and its dedicated CPU $\mathcal{C}$.

▷ **Def. 8.** *Let τ_g be $\mathcal{H}$'s group of tasks, $B_{max}^{\mathcal{H}}$ be the duration of the longest HAC access made by any job of a task in τ_g, and C_{max} be the longest WCET of a task in τ_g.* ◁

Any HAC request by a job of a task in τ_g should be immediately satisfied on $\mathcal{H}$ because scheduling on $\mathcal{C}$ is non-preemptive and tasks busy-wait while accessing $\mathcal{H}$. Thus, there is no contention for $\mathcal{H}$, as any job that accesses it must have first occupied $\mathcal{C}$. As HAC accesses are abstracted as non-preemptive sections on $\mathcal{C}$, we analyze the scheduling of $\mathcal{C}$.

Analysis is complicated by the fact that $\mathcal{H}$ may not be available while $\mathcal{C}$ is available due to forbidden zones. We abstract away from this complication by treating the dedicated CPU-HAC pair $\mathcal{C}$ and $\mathcal{H}$ as a single resource.

▷ **Def. 9.** *Let $V = \Pi - \Theta + B_{max}^{\mathcal{H}}$. V denotes the longest duration of time when $\mathcal{H}$ is unavailable within a reservation period Π.* ◁

For the purpose of abstracting $\mathcal{C}$ and $\mathcal{H}$ as a single resource, we assume that $\mathcal{C}$, like $\mathcal{H}$, may be unavailable for at most V time units per Π time units. Thus, from an analysis perspective, $\mathcal{C}$ and $\mathcal{H}$ are available for execution and execute tasks at the same times (*i.e.*, a job occupies $\mathcal{C}$ during its HAC access). This analytically wastes CPU capacity when $\mathcal{C}$ is available while $\mathcal{H}$ is in its forbidden zone, but this is not particularly damaging for AI algorithms that make extensive use of HACs.

Consider a job of interest $J_{k,l}$ scheduled on $\mathcal{C}$ with release time $r_{k,l}$, absolute deadline t_d, and completion time t_f. Our goal is to provide an upper bound on the response time of $J_{k,l}$, given by $t_f - r_{k,l} = t_f - t_d + T_k$. We prove our bound by contradiction in Theorem 2 after proving bounds on the amount of work that can delay $J_{k,l}$ (Lemma 6) and on the amount of work completed by time t_f (Lemma 7).

Denote the set of all jobs of tasks of τ_g with priority higher than $J_{k,l}$ (plus $J_{k,l}$) as $\Psi \cup \{J_{k,l}\}$. By the definition of NP-EDF, all jobs in $\Psi \cup \{J_{k,l}\}$ are released before t_d. Thus, outside of one potential lower-priority job, discussed later, only jobs of $\Psi \cup \{J_{k,l}\}$ (and intervals outside of reservations) may delay the execution of $J_{k,l}$. We say that a job is *ready* at time t if it is released at or before t and not completed at t.

Let t' be the last time instant before t_d when $\mathcal{C}$ is idle[4] during τ_g's reservation time slice (or 0 if no such time instant exists). Let t'' be the start of the last time interval before t_d when a job not in $\Psi \cup \{J_{k,l}\}$ is scheduled on $\mathcal{C}$ (or the beginning of τ_g's first reservation if no such interval exists). Let $t_0 = \max(t', t'')$. Note that no jobs in $\Psi \cup \{J_{k,l}\}$ are ready at t_0 (otherwise one of them would be scheduled). Thus, as $J_{k,l}$ is ready at $r_{k,l}$,

$$t_0 \leq r_{k,l} \tag{1}$$

The interval $[t_0, t_f]$ is illustrated in Fig. 15 and is referenced in the following proofs. We compute an upper bound on t_f with the following sufficient condition for $J_{k,l}$ to have been completed: if the total workload of all jobs that can

[4] A more mathematically rigorous definition of t' would be "the last time instant before t_d when $\mathcal{C}$ is idle at t'^{-} ...".

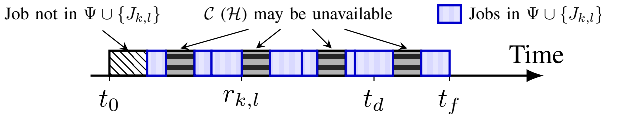

Job not in $\Psi \cup \{J_{k,l}\}$
$\mathcal{C}$ $(\mathcal{H})$ may be unavailable
Jobs in $\Psi \cup \{J_{k,l}\}$
Time
t_0
$r_{k,l}$
t_d
t_f

Fig. 15: Important proof points.

be scheduled within interval $[t_0, t_f]$ is less than the provided capacity, then $J_{k,l}$ is completed within the interval $[t_0, t_f]$.

▷ **Def. 10.** *Let U_g be the total utilization of tasks in τ_g (as a job of a task in τ_g occupies $\mathcal{C}$ during its HAC requests, U_g accounts for both CPU and HAC execution).* ◁

Lemma 6. *The total workload of jobs in $\Psi \cup \{J_{k,l}\}$ that can be scheduled within $[t_0, t_f]$ is at most $(t_d - t_0)U_g$.*

Proof. By the definition of t_0 there were no ready jobs in $\Psi \cup \{J_{k,l}\}$ just before[5] t_0. By the definition of NP-EDF, no jobs of $\Psi \cup \{J_{k,l}\}$ have deadline later than t_d. Thus, all jobs in $\Psi \cup \{J_{k,l}\}$ that execute within $[t_0, t_f]$ have release and deadline within $[t_0, t_d]$.

Each task $\tau_i \in \tau_g$ releases at most $\lfloor (t_d - t_0)/T_i \rfloor$ jobs that execute within $[t_0, t_f]$. Thus, the total workload of jobs in $\Psi \cup \{J_{k,l}\}$ over $[t_0, t_f]$ is the total workload of jobs in $\Psi \cup \{J_{k,l}\}$ released within $[t_0, t_d]$:

$$\sum_{\tau_i \in \tau_g} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor C_i \leq \sum_{\tau_i \in \tau_g} \frac{t_d - t_0}{T_i} C_i = (t_d - t_0)U_g. \quad \square$$

Now we estimate the total available capacity of $\mathcal{C}$ within $[t_0, t_f]$ spent on jobs in $\Psi \cup \{J_{k,l}\}$. The total capacity of $\mathcal{C}$ within this interval is $t_f - t_0$. By the definition of t_0, because the scheduler is NP-EDF, the first job executed in this interval may not be in $\Psi \cup \{J_{k,l}\}$ (as is the case when $t_0 = t''$), which induces at most C_{max} of capacity loss (see the grey execution in Fig. 15). As we work within a periodic reservation, $\mathcal{C}$ may not be available for $\lceil (t_f - t_0)/\Pi \rceil$ intervals of length V (striped intervals in Fig. 15). Thus, the guaranteed available capacity of $\mathcal{C}$ for jobs in $\Psi \cup \{J_{k,l}\}$ is at least

$$(t_f - t_0) - \left\lceil \frac{t_f - t_0}{\Pi} \right\rceil V - C_{max}.$$

▷ **Def. 11.** *Let $S_g = 1 - V/\Pi$ be the long-run rate for which $\mathcal{H}$ and $\mathcal{C}$ are both available.* ◁

With $\left\lceil \frac{t_f - t_0}{\Pi} \right\rceil \leq \frac{t_f - t_0}{\Pi} + 1$ and the definition of S_g, we get the following lemma.

Lemma 7. *The available capacity of $\mathcal{C}$ for jobs in $\Psi \cup \{J_{k,l}\}$ within $[t_0, t_f]$ is at least $(t_f - t_0)S_g - V - C_{max}$.*

Note that we require $S_g \geq U_g$, as otherwise $\mathcal{C}$ is over-utilized. Now we can bound the response time of $J_{k,l}$.

Theorem 2. *If $S_g \geq U_g$, then the response time of any job of τ_k on $\mathcal{C}$ is at most*

$$T_k + \frac{V + C_{max} - T_k(S_g - U_g)}{S_g}. \tag{2}$$

Proof. We prove by contradiction. Suppose otherwise that $t_f - r_{k,l}$ exceeds (2). Consider the interval $[t_0, t_f]$.

In the following derivations, we derive necessary and sufficient conditions on t_f such that the guaranteed capacity $(t_f - t_0)S_g - V - C_{max}$ (Lemma 7) for jobs of $\Psi \cup \{J_{k,l}\}$ over $[t_0, t_f]$ exceeds the upper bound on the workload of jobs in $\Psi \cup \{J_{k,l}\}$ in this interval, $(t_d - t_0)U_g$ (Lemma 6).

$$\begin{aligned}
&(t_d - t_0)U_g &&\leq (t_f - t_0)S_g - V - C_{max}\\
\Leftrightarrow &V + C_{max} + (t_d - t_0)U_g &&\leq (t_f - t_0)S_g\\
\Leftrightarrow &V + C_{max} + (t_d - t_0)U_g &&\leq (t_f - t_d + t_d - t_0)S_g\\
\Leftrightarrow &V + C_{max} + (t_d - t_0)(U_g - S_g) &&\leq (t_f - t_d)S_g\\
\Leftrightarrow &\frac{V + C_{max} + (t_d - t_0)(U_g - S_g)}{S_g} &&\leq t_f - t_d\\
\Leftrightarrow &\frac{V + C_{max} - (t_d - t_0)(S_g - U_g)}{S_g} &&\leq t_f - r_{k,l} - T_k
\end{aligned} \tag{3}$$

Note that $t_f - r_{k,l}$ is the response time of $J_{k,l}$. By (1), $t_0 \leq r_{k,l}$. Thus, $t_d - t_0 \geq t_d - r_{k,l} = T_k$. By Def. 10, $S_g - U_g \geq 0$. Thus, $-(t_d - t_0)(S_g - U_g) \leq -T_k(S_g - U_g)$. By (3), $t_f - r_{k,l}$ exceeding (2) is a sufficient condition for the capacity provided by $\mathcal{C}$ to jobs of $\Psi \cup \{J_{k,l}\}$ to exceed the workload required by this job set over $[t_0, t_f]$. Thus, $\mathcal{C}$ must have been idle during a reservation in $[t_0, t_f]$.

However, for $\mathcal{C}$ to have been idle in $[t_0, t_f]$, job $J_{k,l}$, which was ready at t_0, must have completed prior to the end of this interval. This contradicts the definition of t_f, meaning the response time of $J_{k,l}$ must be at most (2).

Note that (2) does not depend on job number, so it applies to any job of any task $\tau_k \in \tau_g$. □

[5] A more mathematically rigorous definition of "just before t_0" would be "at t_0^-".