Scheduling Processing Graphs of Gang Tasks on Heterogeneous Platforms

Shareef Ahmed, Denver Massey, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill
{shareef, denmas22, anderson}@cs.unc.edu

***Abstract*—Artificial-intelligence-powered real-time systems typically consist of numerous gang tasks, such as computations on graphics processing units (GPUs), that are interconnected by data-flow dependencies. Despite their relevance in many applications, scheduling processing graphs of gang tasks has received limited attention. This paper presents scheduling techniques and response-time analysis for such systems on heterogeneous computing platforms. Response-time bounds of a graph of gang tasks are presented when scheduled under a *work-conserving* or *semi-work-conserving* scheduler. Techniques to support multiple graphs using federated scheduling techniques are also presented. Experimental evaluations and a case study on a computer vision application are presented to demonstrate the effectiveness of the proposed approach.**

I. Introduction

The last few decades have witnessed the development of high-performance computing platforms, which have fueled the innovation of safety-critical real-time systems with autonomous features. These systems usually consist of numerous computation-heavy parallel tasks that require heterogeneous computing resources, such as multicore platforms augmented with hardware accelerators. Such parallel tasks often execute efficiently when multiple application threads are grouped into *gangs*. Moreover, tasks of autonomous systems frequently form processing graphs due to data-flow dependencies. Due to their significance, scheduling both gang-based systems and graph-based systems has garnered much attention in recent years [1], [20], [22], [28], [35], [38], [52].

Despite their significance, processing graphs consisting of gang tasks have received little attention. To the best of our knowledge, work on scheduling processing graphs of gang tasks has been limited to NVIDIA-specific graphics processing unit (GPU) scheduling [56], for which many intricate details are still being revealed [10]. Other work either considers chains (a special case of graphs) of gang tasks, called *bundled gang tasks* [47], [54], or avoids gang-specific analysis complexities by adopting a lock-based approach. The latter approach allows gang tasks' execution on non-CPU computing resources, such as GPUs, by a real-time locking protocol [7], [23], which may cause significant delays in GPU accesses [3].

In this paper, we consider scheduling processing graphs of gang tasks on heterogeneous platforms consisting of different types of *compute elements* (CEs) such as CPUs, GPUs, field programmable gate arrays (FPGAs), *etc.* We give a *federated-scheduling* approach to schedule multiple processing graphs by allocating each processing graph a set of processors on each CE. Today, such an approach can be realized on many hardware accelerators through the use of their compute-partitioning abilities [11], [13]. We give response-time bounds for each graph on its allocated processors under any *work-conserving* or *semi-work-conserving* scheduler (defined in Sec. III). The task model and scheduling approaches we consider generalize various systems. We describe two such examples below.

Work was supported by NSF grants CPS 2038960, CPS 2038855, CNS 2151829, and CPS 2333120.

Scheduling on multicore+GPU. Processing graphs that are scheduled on multicore platforms augmented with GPUs present a special case of our task model. For such systems, graph nodes that execute on GPUs are gang tasks, while nodes that execute on CPUs are sequential tasks (a special case of gang tasks). Moreover, common CPU scheduling approaches are work-conserving, whereas NVIDIA GPU scheduling is semi-work-conserving under certain assumptions [10].

Time partitioning in component-based systems. Hierarchical scheduling techniques can enable time partitioning among different software components of a component-based system. Under these techniques, the top-level scheduler allocates a gang-like set of processors to each component for certain time intervals. Since data-flow dependencies may be present between components, designing the top-level scheduler reduces to the problem of scheduling processing graphs of gang tasks.

Contributions. Our contributions are threefold:

First, we give response-time bounds for a *directed-acyclic-graph-*(DAG)-based task formed by precedence constraints among gang tasks on multiple CEs. Our response-time bound is valid for schedulers that are either work-conserving or semi-work-conserving.

Second, we show how to schedule multiple DAGs under federated scheduling by giving an integer linear program (ILP) to allocate processors of different CEs to each DAG.

Finally, we demonstrate the effectiveness of our approach through experimental studies. For DAG scheduling on multicore+GPU platforms, our approach outperforms locking-based methods by reducing response-time bounds by 63%. We also present a case study on multicore + GPU platforms to illustrate our approach in practice.

Organization. After covering needed background (Sec. II), we discuss considered scheduling algorithms in detail (Sec. III), provide techniques to account for *parallelism-induced* idleness for gang tasks that form DAGs (Sec. IV), give our response-time bound for a DAG (Sec. V), present techniques to support multiple DAGs (Sec. VI), present our experiments (Sec. VII),

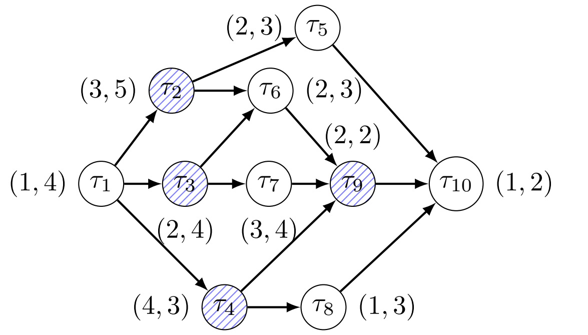


Fig. 1: A DAG G. Solid and hatched circles represent tasks allocated to two different CEs. Tuples circles represent (m_i, C_i).

TABLE I: Notation summary.

Symbol	Meaning	Symbol	Meaning
N	No. of DAGs	$r(\cdot)$	Release time
G	A DAG task	$f(\cdot)$	Finish time
V	Nodes of G	$pred(\cdot)$	Set of predecessors
E	Edges of G	$succ(\cdot)$	Set of successors
T	Period of G	$anc(\cdot)$	Set of ancestors
D	Rel. deadline of G	$desc(\cdot)$	Set of descendants
τ_i	i^{th} task of G	$R(\cdot)$	Response time
C_i	WCET of τ_i	m_i	τ_i's degree of parallelism
γ_i	Assigned CE of τ_i	$\mathcal{M}_p$	Processor count on p^{th} CE
J_j	j^{th} DAG job of G	$len(\lambda)$	$\sum_{\tau_i \in \lambda} C_i$
$\tau_{i,j}$	j^{th} job of τ_i	$vol(V')$	$\sum_{\tau_i \in V'} m_i C_i$
λ	path of G	V'_p	Tasks of V' on the p^{th} CE
Δ_i	Def. 1	λ^e	Envelope path
$I(\cdot)$	Def. 5	$dep(\tau_i)$	$anc(\tau_i) \cup desc(\tau_i)$
$\tau_i(V')$	Def. 6	$\Delta_i(V')$	Def. 6
$vi_k(V')$	Def. 9	$h(\cdot), g(\cdot)$	Def. 11
$F(\cdot,\cdot)$	Def. 8	$I^{diff}(\cdot)$	Def. 7

review prior work (Sec. VIII), and conclude (Sec. IX).

II. Preliminaries

We consider a task set Γ consisting of N DAG tasks $\{G^1, G^2, \ldots, G^N\}$. For ease of notation, DAG indices are used only when relevant. Each DAG task G releases a potentially infinite sequence of DAG jobs $J_1, J_2, \ldots$. The release and completion time of J_j are denoted by $r(J_j)$ and $f(J_j)$, respectively. DAG jobs of G are released sporadically with period T. DAG G has a constrained relative deadline $D \leq T$, *i.e.*, DAG job J_j must finish execution by time $r(J_j) + D$.

Each DAG G is represented as a tuple (V, E), where V and E are sets of nodes and directed edges, respectively. V consists of *gang* tasks $\{\tau_1, \tau_2, \ldots, \tau_{|V|}\}$. We assume tasks are indexed according to a topological ordering of G. Each gang task τ_i has a *worst-case execution time* (WCET) C_i and a *degree of parallelism* m_i that denotes the number of simultaneously available processors required to execute any job (instance) of τ_i. Thus, the *worst-case execution requirement* (WCER) of each job of τ_i can be represented by a rectangle of area $m_i \times C_i$ in a schedule. A directed edge from τ_i to τ_k represents a precedence constraint between the *predecessor* task τ_i and the *successor* task τ_k. The set of predecessors (resp., successors) of τ_i is denoted by $pred(\tau_i)$ (resp., $succ(\tau_i)$). We assume that each DAG G has a unique *source* task τ_1 with no incoming edge and a unique *sink* task $\tau_{|V|}$ with no outgoing edge.[1] The *utilization* of τ_i is $u_i = (C_i \times m_i)/T$. Note that u_i can exceed 1.0. The utilization of DAG task G is $U(G) = \sum_{i=1}^{|V|} u_i$. The *total utilization* of Γ is $U(\Gamma) = \sum_{G \in \Gamma} U(G)$.

DAGs in Γ are scheduled on μ *computational elements* (CEs). A CE might be a CPU or some specialized hardware accelerator. The k^{th} CE consists of a pool of $\mathcal{M}_k$ identical processors. Each task τ_i of a DAG G has a parameter $\gamma_i \in \{1, 2, \ldots, \mu\}$ that represents the CE on which τ_i executes.

Ex. 1. Fig. 1 shows a DAG of ten tasks on two CEs. Tasks τ_2, τ_3, τ_4, and τ_9 are assigned to one CE, while the remaining tasks execute on the other CE. τ_7's degree of parallelism is $m_7 = 3$ and its WCET is $C_7 = 4$. $\Diamond$

Each DAG job J_j is composed of the j^{th} job $\tau_{i,j}$ of each task τ_i. The release time and finish time of job $\tau_{i,j}$ are denoted by $r(\tau_{i,j})$ and $f(\tau_{i,j})$, respectively. The j^{th} job $\tau_{1,j}$ of the source task τ_1 is released when J_j is released, *i.e.*, $r(J_j) = r(\tau_{1,j})$. The j^{th} job of each non-source task is released once the j^{th} job of each of its predecessors finishes, *i.e.*, $r(\tau_{i,j}) = \max_{\tau_k \in pred(\tau_i)}\{f_{k,j}\}$. Job $\tau_{i,j}$ is *ready* to execute during $[r(\tau_{i,j}), f(\tau_{i,j}))$. DAG job J_j completes when $\tau_{|V|,j}$ completes. The *response time* of J_j is $R(J_j) = f(\tau_{|V|,j}) - r(\tau_{1,j})$. G's response time is $R(G) = \sup_j\{R(J_j)\}$.

A *path* $\lambda = \{v_1, v_2, \ldots, v_k\}$ is an ordered set of tasks of G (*i.e.*, $v_i \in V$ for each $1 \leq i \leq k$) such that $v_i \in pred(v_{i+1})$ holds. (We use the symbol v to simplify indexing nodes of λ.) When job indices are irrelevant, we also use λ to denote the ordered set of the j^{th} jobs of tasks in $\{v_1, v_2, \ldots, v_k\}$. A path is a *complete path* if it contains the source and sink nodes. We define the *length* of a path as follows:

$$len(\lambda) = \sum_{\tau_i \in \lambda} C_i. \tag{1}$$

If a path exists from τ_i to τ_k, then τ_i (resp., τ_k) is called an *ancestor* (resp., *descendant*) of τ_k (resp., τ_i). The j^{th} job of an ancestor (resp., descendant) task of τ_i is an ancestor (resp., descendant) job of $\tau_{i,j}$. The set of ancestors (resp., descendants) of τ_i is denoted as $anc(\tau_i)$ (resp., $desc(\tau_i)$). The set of ancestors (resp., descendants) of $\tau_{i,j}$ is denoted as $anc(\tau_{i,j})$ (resp., $desc(\tau_{i,j})$). We use $dep(\tau_i)$ (resp., $dep(\tau_{i,j})$) to denote $anc(\tau_i) \cup desc(\tau_i)$ (resp., $anc(\tau_{i,j}) \cup desc(\tau_{i,j})$). For any subset $V' \subseteq V$ of tasks, we define its *volume* as follows:

$$vol(V') = \sum_{\tau_i \in V'} m_i C_i. \tag{2}$$

Ex. 1 (Cont'd). In Fig. 1, task τ_6 has two predecessors τ_2 and τ_3. Tasks $\{\tau_1, \tau_3, \tau_7, \tau_9, \tau_{10}\}$ form a complete path with length 16. Task τ_6's ancestors (resp., descendants) are $anc(\tau_6) = \{\tau_1, \tau_2, \tau_3\}$ (resp., $desc(\tau_6) = \{\tau_9, \tau_{10}\}$). Finally, $vol(anc(\tau_6)) = 1 \cdot 4 + 3 \cdot 5 + 2 \cdot 4 = 27$. $\Diamond$

We summarize all introduced notation in Tbl. I.

Parallelism-induced idleness. When scheduling gang tasks, *parallelism-induced idleness* may occur [22]. A time instant t is parallelism-induced idle if there is an idle processor at time t and a job $\tau_{i,j}$ is pending but unscheduled at time t due to an insufficient number of available processors. For example,

[1] A DAG with multiple sources/sinks can be supported by adding a "virtual" source or sink with a WCET of zero.

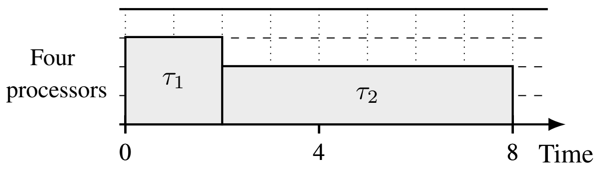


Fig. 2: Two independent gang tasks on four processors. Both tasks release a job at time 0.

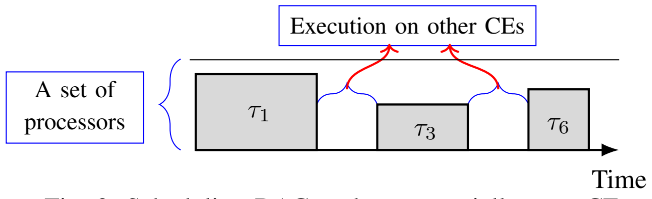


Fig. 3: Scheduling DAG nodes sequentially on a CE.

Fig. 2 shows two independent gang tasks τ_1 and τ_2 with $m_1 = 3$ and $m_2 = 2$ scheduled on four processors. During the time interval $[0, 2)$, there is an idle processor. Although τ_2 has a pending job during this interval, it cannot execute, as the number of available processors is less than m_2. Thus, there is parallelism-induced idleness during $[0, 2)$.

III. Scheduling

In this section, we describe the scheduling policies under which we give the response-time bounds in Sec. IV.

A. Federated Scheduling

We assume *federated scheduling* of DAGs in Γ, where each DAG G is allocated an exclusive set of processors from each CE. Let M_p denote the number of processors of the p^{th} CE assigned to G. Thus, all jobs of task τ_i with $\gamma_i = p$ are scheduled on the M_p processors of the p^{th} CE assigned to G. We require $M_p \geq \max_{\tau_i:\gamma_i=p}\{m_i\}$. M_p can be zero if no task of G requires the p^{th} CE.

Heavy vs. light DAGs. For scheduling DAGs of sequential tasks (*i.e.*, tasks with $m_i = 1$) on a single CE, federated scheduling approaches differentiate between *heavy* (DAGs with $\frac{\sum_{\tau_i \in V} C_i}{\min\{D,T\}} > 1$) and *light* (DAGs with $\frac{\sum_{\tau_i \in V} C_i}{\min\{D,T\}} \leq 1$) DAG tasks. Each heavy DAG requires parallel execution of its nodes, so it is allocated enough processors to meet its deadline. In contrast, all light DAGs share a set of processors, where they are scheduled as sequential tasks.

When DAG nodes are gang tasks, classifying the DAG as heavy or light is more nuanced, even on a single CE. The *density* of a gang task of G, $m_iC_i/\min\{D,T\}$, may exceed one. However, if $\sum_{\tau_i \in G} C_i \leq D$, then G can still be sequentially scheduled on a sufficient number of processors. Additionally, since the m_i values can vary across nodes in G, the processor requirements may differ within a single instance of G. Furthermore, if G requires multiple CEs, G's jobs may exhibit self-suspending behavior with respect to a CE.

Ex. 2. Assume that three nodes, τ_1, τ_3, and τ_6, are assigned to a CE. Fig. 3 shows a schedule of these nodes when they execute sequentially. All three nodes have different m_i values. The duration between the execution of τ_1 and τ_3, when other nodes execute on different CEs, can be regarded as self-suspension times on τ_1's CE. $\Diamond$

Thus, if some DAGs are scheduled sequentially on a shared set of processors, deriving their response-time bounds requires analyzing self-suspending *bundled* gang tasks. A bundled gang task consists of a chain of multiple rigid gang subtasks (a special case of our task model), where the m_i values of two subtasks may differ. Thus, execution in Fig. 3 can be viewed as bundled tasks with a self-suspension between two consecutive subtasks. Providing an analysis for such a task model is beyond the scope of this paper and is deferred to future work.

Algorithm 1 Work-conserving scheduling.

```
Variables:
    Ready($t$) : Set of ready jobs at time $t$
    Sched($t$) : Set of jobs to be scheduled at time $t$
1: procedure AN EXAMPLE WORK-CONSERVING SCHEDULING
2:     $M' \leftarrow M$
3:     Order jobs in Ready($t$) according to the scheduling policy
4:     for each $\tau_{i,j} \in$ Ready($t$) do
5:         if $m_i > M'$ then
6:             continue /* Use break for semi work-conserving */
7:         Sched($t$) $\leftarrow$ Sched($t$) $\cup \{\tau_{i,j}\}$
8:         $M' \leftarrow M' - m_i$
```

B. Scheduling DAGs on Allocated Processors

We consider *work-conserving* and *semi-work-conserving* approaches for scheduling each DAG on its allocated processors.

Work-conserving scheduling. For gang tasks, work-conserving schedulers do not allow a set of processors on a CE to remain idle if a ready gang job can use them. Specifically, under work-conserving schedulers, a job $\tau_{i,j}$ is ready but unscheduled at any time if and only if the number of idle processors of the γ_i^{th} CE is insufficient to schedule $\tau_{i,j}$. Thus, any work-conserving scheduler satisfies the following:

(WC) Under a work-conserving scheduler, among the processors of the p^{th} CE that are allocated to G, there are M'_p idle processors at time t if and only if, for each ready but unscheduled job $\tau_{i,j}$ with $\gamma_i = p$ at time t, $m_i > M'_p$ holds.

Algorithm 1 shows pseudocode for an example preemptive work-conserving scheduler. Note that non-preemptive schedulers can also be work-conserving. At any scheduling-decision point, Algorithm 1 iterates through all ready jobs to schedule as many jobs as possible in an order. If a job cannot fit on the available processors, the scheduler attempts to schedule the next job in the order (line 6).

Ex. 3. Fig. 4 shows a work-conserving schedule of a DAG job of G in Fig. 1 on two CEs consisting of four and six processors, respectively. At time 4, τ_1's job completes, causing the release of the jobs of τ_2, τ_3, and τ_4. At time 4, jobs of τ_2 and τ_3 are scheduled on the 2nd CE, but τ_4's job cannot be scheduled on the remaining one available processor. τ_3's job executes less than τ_3's WCET and completes at time 7. This causes τ_7 to release its job at time 7 on the 1st CE. At

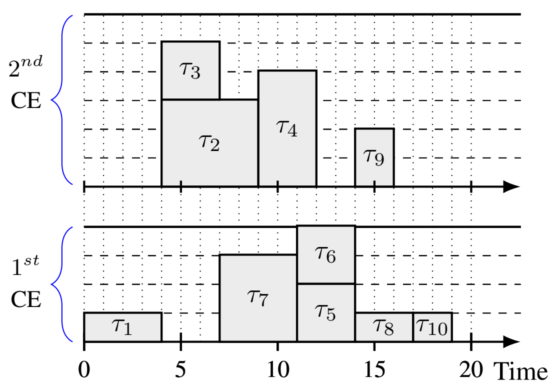


Fig. 4: A work-conserving schedule of G in Fig. 1.

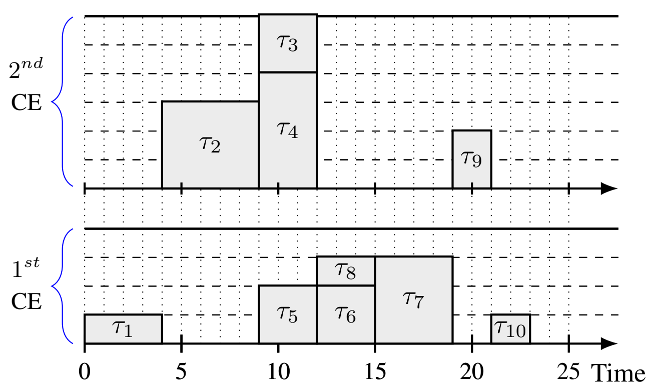


Fig. 5: A semi-work-conserving schedule of G in Fig. 1.

time 7, τ_4's job still cannot be scheduled on the three available processors of the 2^{nd} CE. $\Diamond$

Semi-work-conserving scheduling. A semi-work-conserving scheduler may allow some processors to remain idle even if an unscheduled ready job could fit there. However, in such a case, there must be another ready but unscheduled job that cannot fit on those processors. Specifically, under a semi-work-conserving scheduler, at any time, a job $\tau_{i,j}$ is ready but unscheduled if and only if the number of idle processors is insufficient to schedule a ready but unscheduled job $\tau_{k,\ell}$. Thus, the following holds.

(SC) Under a semi-work-conserving scheduler, among the processors of the p^{th} CE that are allocated to G, there are M'_p idle processors at time t if and only if there exists a ready but unscheduled job $\tau_{k,\ell}$ with $\gamma_k = p$ at time t for which $m_k > M'_p$ holds.

Thus, under semi-work-conserving schedulers, when a job $\tau_{i,j}$ with $\gamma_i = p$ is ready but unscheduled, the number of idle processors M'_p allocated to G on the p^{th} CE can be larger than m_i. However, there must be another job $\tau_{k,\ell}$ with $\gamma_k = p$ such that m_k is larger than M'_p. Algorithm 1 can be converted into a semi-work-conserving scheduling algorithm by replacing the statement **continue** at line 6 by **break**.

Ex. 4. Fig. 5 depicts a semi-work-conserving schedule of a DAG job of G in Fig. 1 on two CEs consisting of four and six processors, respectively. At time 4, τ_1's job completes, causing the release of the jobs of τ_2, τ_3, and τ_4. At time 4, τ_2's job is scheduled on the 2^{nd} CE. Assume that the scheduler attempts to schedule τ_4's job on the three available processors first. Since it cannot fit there, jobs for τ_3 and τ_4 are not scheduled. After τ_2's job is completed, τ_4's job is scheduled, and the only remaining ready job on the 2^{nd} CE is also scheduled. $\Diamond$

Semi-work-conserving scheduling in GPUs. When GPU-accessing tasks share the same address space, NVIDIA GPUs schedule tasks in a semi-work-conserving manner. Processors in NVIDIA GPUs are clustered into *streaming multiprocessors* (SMs). A CUDA-using[2] program launches a *kernel* to be executed on GPU. Each kernel consists of *blocks* of multiple threads that are co-scheduled on an SM. Blocks are schedulable entities on GPUs and can be abstracted as gang tasks. Note that all threads of a block must be scheduled on an SM, *i.e.*, its threads cannot be distributed to multiple SMs.

When a kernel is launched, it moves through a pipeline to enter into a *first-in-first-out execution engine* (EE) *queue*.[3] The blocks of the kernel at the head of the queue are scheduled on SMs. When all blocks at the head of the queue are scheduled, the kernel is removed from the queue, and the new head's blocks are scheduled until no more blocks can fit any SMs. Thus, if a block of the kernel at the head of the queue cannot fit on any SMs, no blocks of the non-head kernels are scheduled even if they can fit on remaining processors on an SM, satisfying (SC). Readers interested in the details of scheduling on NVIDIA GPUs are referred to [8], [10].

Note that we do not require modifying NVIDIA GPU scheduling. Rather, we require the construction of fine-grained processing graphs for analytical purposes. Such graphs require each node to be either a CPU- or GPU-only node. Furthermore, each GPU node represents a CUDA block. Thus, to construct such a graph, each kernel should be split into multiple nodes, each representing a block of the kernel. The number of such block nodes and their degrees of parallelism can be determined by NVIDIA's profiling tool `nvprof`. Furthermore, block WCET estimates can be obtained via measurements using the `globaltimer` performance-counter register [56].

IV. Parallelism-Induced Idleness

Before giving response-time bounds, we first give techniques to account for parallelism-induced idleness under both work-conserving and semi-work-conserving schedulers. We do so by determining, for each task τ_i of a DAG G, the maximum possible number of processors that may be idle on τ_i's CE when τ_i has a ready but unscheduled job. The number of such idle processors depends on other tasks of G on that CE, the number of processors allocated to G on that CE, and the scheduling algorithm. We begin by introducing the following notation to denote the maximum number of idle processors when a ready job cannot be scheduled. This was first introduced in [20] for scheduling independent gang tasks.

[2]Although other GPU-programming APIs exist, CUDA is commonly used in real-time systems.

[3]CUDA also provides CUDA *streams* that adds an additional queueing prior to EE queues. Using per-job streams, such queueing can be obviated [56].

Def. 1. *For each task τ_i of G, let Δ_i^W (resp., Δ_i^S) denote the maximum possible number of idle processors, among the M_{γ_i} processors allocated to G on the γ_i^{th} CE, when a job of τ_i is ready but unscheduled under any work-conserving (resp., semi-work-conserving) scheduler.*

Since Δ_i^W and Δ_i^S values depend only on G's tasks that execute on the γ_i^{th} CE, we introduce the following notation.

Def. 2. *For any set $V' \subseteq V$ of tasks, V'_p denotes the tasks in V' that execute on the p^{th} CE, i.e., $V'_p = \{\tau_i \in V' : \gamma_i = p\}$. Thus, V_p denotes all tasks of G that execute on the p^{th} CE.*

In the following two subsections, we show how to determine Δ_i^W and Δ_i^S values, respectively.

A. Work-Conserving Schedulers

By (WC), when τ_i has a ready but unscheduled job, the number of idle processors on τ_i's CE is at most $m_i - 1$. Thus, by Def. 1, $m_i - 1$ is a safe value for Δ_i^W. However, it is possible to further optimize the value of Δ_i^W. For independent sporadic tasks, Dong and Liu gave a dynamic programming algorithm to determine Δ_i^W by calculating the smallest number of occupied processors on the γ_i^{th} CE by jobs of a subset of other tasks, but leaving less than m_i available processors (thus, maximizing the number of idle processors) [20]. However, considering subsets of all tasks other than τ_i can be pessimistic for DAG tasks, as not all tasks can have ready jobs simultaneously with τ_i.

To accurately determine Δ_i^W values, we give an algorithm that considers only those sets of tasks that can have ready jobs simultaneously with τ_i. A set of tasks can simultaneously have ready jobs if no pairs of tasks in that set have ancestor-descendant relationships. To formally define such tasks, we introduce the notation $par(V')$ for any set $V' \subseteq V$, which is *true* if jobs of all tasks in V' can be ready simultaneously, and *false* otherwise:

$$par(V') = \bigwedge_{\tau_i \in V'} \left(\tau_i \notin \bigcup_{\tau_k \in V' \setminus \{\tau_i\}} dep(\tau_k) \right).$$

Using $par(V')$, we determine Δ_i^W as follows:

$$\Delta_i^W = \begin{cases} 0 & \text{if } \forall\ V' \subseteq V_{\gamma_i} : par(V' \cup \{\tau_i\}) :: \sum_{\tau_k \in V'} m_k \le M_{\gamma_i} - m_i \\ \max\{m' \in \{0, \cdots, m_i - 1\} : (\exists\ V' \subseteq V_{\gamma_i} \setminus \{\tau_i\} : par(V' \cup \{\tau_i\}) \wedge \sum_{\tau_j \in V'} m_j = M_{\gamma_i} - m')\} & \text{otherwise} \end{cases} \tag{3}$$

The first case in (3) sets Δ_i^W to zero, as no set V' of tasks satisfying $par(V')$ can occupy at least $M_{\gamma_i} - m_i$ processors. The second case sets Δ_i^W value by determining a subset of tasks V' satisfying $par(V' \cup \{\tau_i\})$, which occupy the smallest number of processors on τ_i's CE and leave less than m_i processors for a job τ_i.

Ex. 5. Consider the DAG G in Fig. 6 that is scheduled on a CE of ten processors. Only jobs of τ_2, τ_3, τ_5, and τ_6 can be ready when τ_4 has a ready job, as they are neither ancestors nor descendants of τ_4. However, since τ_3 is a predecessor of τ_6, jobs of τ_3 and τ_6 cannot be simultaneously ready. Thus, $par(\{\tau_3, \tau_4, \tau_6\})$ is *false*. In contrast, $par(\{\tau_2, \tau_4, \tau_5\})$ is *true*, as they can have ready jobs at the same time. Among all sets $V' \subset V \setminus \{\tau_4\}$ such that $par(V' \cup \{\tau_4\}) =$ *true*, tasks $\{\tau_2, \tau_3\}$ require the least number of processors, leaving less than $m_4 = 6$ processors for τ_4. Thus, by (3), $\Delta_4^W = 10 - m_2 - m_3 = 10 - 7 = 3$. ♢

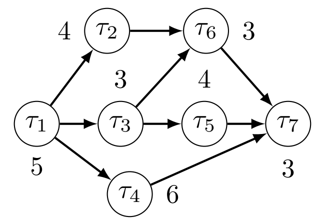


Fig. 6: A DAG. Numbers outside circles denote m_i values.

Computing Δ_i^W by (3). We now give a dynamic programming algorithm to compute Δ_i^W values according to the second case of (3). Note that the first case is applicable when there is no m' satisfying the second case. Thus, we only focus on the second case of (3). To compute Δ_i^W, we consider the set of tasks $V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\})$. We fill a two-dimensional dynamic-programming table with entries $\Delta_i^W(\tau_j, m)$ where $\tau_j \in V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\})$ and $m \in \{0, 1, \ldots, M_{\gamma_i}\}$. The entry $\Delta_i^W(\tau_j, m)$ stores a boolean value, which is *true* if τ_j and a subset of tasks in $V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\})$ with task indices at most $j - 1$ can execute in parallel and occupy exactly m processors in τ_i's CE, and *false* otherwise.

Ex. 5 (Cont'd). Consider the DAG in Fig. 6. For any m, $\Delta_4(\tau_3, m)$ depends on tasks in $V \setminus (dep(\tau_4) \cup \{\tau_4\})$ with task indices at most 3. Such tasks are $\{\tau_2, \tau_3\}$. $\Delta_4(\tau_3, m)$ is true for m values that can be occupied by only τ_3 or both τ_2 and τ_3. Thus, $\Delta_4(\tau_3, 5)$ and $\Delta_4(\tau_3, 9)$ are *true*, as five (resp., nine) processors can be occupied by τ_3 (resp., τ_2 and τ_3). ♢

The recurrence relation in (4) determines the $\Delta_i^W(\tau_j, m)$ values. The first case in (4) represents a base case; $\Delta_i^W(\tau_j, m_j)$ is *true* as τ_j can occupy m_j processors. The second case considers all tasks $\tau_j \in V_{\gamma_i}$ that cannot execute in parallel with any task $\tau_k \in V_{\gamma_i}$ with $k < j$ and sets the entries corresponding to $m \neq m_j$ as *false*. The final case considers all tasks with task indices smaller than j that are not τ_j's ancestors to determine the existence of a set of tasks occupying exactly $m - m_j$ processors. Note that case three precludes checking whether τ_k is a descendant of τ_j, as task indexing follows a topological ordering.

$$\Delta_i^W(\tau_j, m) = \begin{cases} true & \text{if } m = m_j \\ false & \text{if } m \neq m_j \wedge (\forall k < j : \tau_k \notin V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\}) \\ \bigvee_{k<j \wedge \tau_k \in V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\} \cup anc(\tau_j))} \Delta_i^W(\tau_k, m - m_j) & \text{otherwise} \end{cases} \tag{4}$$

Finally, to compute Δ_i^W, we determine the smallest m value, say m', larger than $M_{\gamma_i} - m_i$ for which a τ_j exists with $\Delta_i^W(\tau_j, m) = true$. We then set $\Delta_i^W = M_{\gamma_i} - m'$.

Running time. We can compute $dep(\tau_i)$ for all τ_i in $O(|V|^2)$ time [45]. Computing each entry of the dynamic programming table corresponding to the first case requires $O(1)$ time. Using pre-computed $dep(\tau_i)$, computing an entry corresponding to the second and third cases takes $O(|V|)$ time. Thus, the total running time to compute the dynamic programming table takes $O(|V|^2 M_{\gamma_i})$ time. Computing the Δ_i^W value from the table requires an additional $O(|V| M_{\gamma_i})$ time for scanning all entries in the table. Thus, the time complexity for computing Δ_i^W is $O(|V|^2 M_{\gamma_i})$. Finally, computing Δ_i^W values for all tasks requires $O(|V|^3 \max\{M_{\gamma_i}\})$ time.

B. Semi-Work-Conserving Schedulers

When a job of τ_i is ready but unscheduled under semi-work-conserving schedulers, the number of idle processors on τ_i's CE can exceed $m_i - 1$. This is because, by (SC), if a task τ_k with $m_k > m_i$ cannot be scheduled due to the unavailability of m_k processors, it may cause τ_i to be unscheduled too. Thus, in such a case, the number of idle processors is at most Δ_k^W, *i.e.*, the maximum possible number of idle processors when a job of τ_k cannot be scheduled under work-conserving schedulers. Therefore, Δ_i^S depends on Δ_k^W values of all tasks τ_k that can have ready jobs simultaneously with τ_i. Thus, we define Δ_i^S as follows:

$$\Delta_i^S = \max_{\tau_k \in V_{\gamma_i} \setminus dep(\tau_i)} \{\Delta_k^W\}. \qquad (5)$$

Proof of (5). Assume that the number of idle processors is M' at time t when a job of τ_i is ready but unscheduled. By (SC), there exists a task τ_k in $V_{\gamma_i} \setminus dep(\tau_i)$ with a ready but unscheduled job such that $m_k > M'$ holds. Since τ_k has a ready but unscheduled job, only jobs of tasks in $V_{\gamma_i} \setminus dep(\tau_k)$ can occupy processors on the γ_i^{th} CE. By (3), the maximum number of idle processors when tasks in $V_{\gamma_i} \setminus dep(\tau_k)$ occupy more than $M_{\gamma_i} - m_k$ processors of the γ_i^{th} CE is at most Δ_k^W. Thus, since $\tau_k \in V_{\gamma_i} \setminus dep(\tau_i)$, we have $M' \le \Delta_k^W \le \max_{\tau_\ell \in V_{\gamma_i} \setminus dep(\tau_i)} \{\Delta_\ell^W\}$, which satisfies (5). □

From computed Δ_k^W values, it requires an additional $O(|V_{\gamma_i}|)$ time to compute a Δ_i^S value, thus total $O(|V_{\gamma_i}|^2)$ time to compute such values for all nodes. Including computation times for Δ_k^W values, running time to compute all Δ_i^S is $O(|V|^3 \max_k\{M_k\} + |V|^2) = O(|V|^3 \max_k\{M_k\})$.

Ex. 6. Consider a semi-work-conserving schedule of the DAG G in Fig. 6 on ten processors. Consider Δ_5^S for τ_5. Only jobs of τ_2, τ_4, and τ_6 can be ready when τ_5 has a ready job. Thus, by (5), $\Delta_5^S = \max\{\Delta_2^W, \Delta_4^W, \Delta_6^W\}$. ♢

Δ_i^W and Δ_i^S values for GPUs. Recall that NVIDIA GPUs cluster their processors into SMs and a job of gang tasks must execute on a single SM. The above-mentioned approaches to compute Δ_i^W and Δ_i^S values can be applied for such a case by first determining such values assuming a single SM and then multiplying the values by the number of SMs allocated to the DAG. Thus, if c SMs, each containing M_p/c processors, are allocated to DAG G and $\Delta_{i,c}^W$ is the value computed by (3) assuming M_p/c processors, then $\Delta_i^W = c \cdot \Delta_{i,c}^W$.

V. Response-Time Bound

In this section, we give a response-time bound for a DAG G of gang tasks that are scheduled on μ CEs under a work-conserving or a semi-work-conserving scheduler. Since G has constrained deadlines, we consider a single DAG job to derive our response-time bound. For notational convenience, we omit job indices, *e.g.*, τ_i denotes both a task and its job. Our analysis technique is the same for work-conserving and semi-work-conserving schedulers. Specifically, replacing Δ_i^W by Δ_i^S from our response-time bound under work-conserving schedulers yields our response-time bound under semi-work-conserving schedulers. Thus, we give a response-time bound under an arbitrary schedule, as assumed in the following definition.

Def. 3. *Let $\mathcal{S}$ be a schedule of DAG job J on μ CEs where, for all p, M_p processors of the p^{th} CE are assigned to G. For each task τ_i, let $\Delta_i = \Delta_i^W$ (resp., $\Delta_i = \Delta_i^S$), if $\mathcal{S}$ is work-conserving (resp., semi-work-conserving).*

Note that, in $\mathcal{S}$, jobs of J may execute for less than their WCETs; our bound is also valid in such a case. Our analysis relies on an *envelope* path of J in $\mathcal{S}$, as defined below.

Def. 4. *In $\mathcal{S}$, a path of jobs $\{v_1, v_2, \cdots, v_k\}$ of J is an* envelope path *if and only if the following conditions hold.*
(i) $v_1 = \tau_1 \wedge v_k = \tau_n$,
(ii) $\forall\, i \in \{1, 2, \ldots, k-1\} : f(v_i) = r(v_{i+1})$,
(iii) $\forall\, i \in \{1, 2, \ldots, k-1\} : v_i \in pred(v_{i+1})$.
We denote an envelope path of J in $\mathcal{S}$ by λ^e.

Ex. 7. We can determine an envelope path by traversing the schedule backward (from sink to source). In Fig. 4, τ_{10} is released when τ_8 finishes. Similarly, τ_8 is released when τ_4 finishes. Iteratively doing this until τ_1 is reached, an envelope path in Fig. 4 is $\{\tau_1, \tau_4, \tau_8, \tau_{10}\}$. ♢

Note that there can be multiple envelope paths of J in $\mathcal{S}$. This can happen when multiple predecessor jobs of τ_i complete at the same time. For any task on the envelope path λ^e of J in $\mathcal{S}$, we have the following lemma.

Lemma 1. *Let τ_i be a job of λ^e. At any time $t \in [r(\tau_i), f(\tau_i))$, if τ_i, is not scheduled, then at least $M_{\gamma_i} - \Delta_i$ processors of the γ_i^{th} CE are busy in $\mathcal{S}$.*

Proof. Follows from Defs. 1 and 3. □

Let A^e be the union of all intervals when jobs of λ^e execute in $\mathcal{S}$. Also, let A^{ne} be the union of all intervals when no jobs of λ^e execute in $\mathcal{S}$. Thus, $A^e \cap A^{ne} = \emptyset$ holds, and we have

$$|A^e| + |A^{ne}| = f(\tau_n) - r(\tau_1) = R(J). \qquad (6)$$

Thus, response time $R(J)$ of job J can be upper bounded by upper bounding $|A^e|$ and $|A^{ne}|$. To upper bound $|A^{ne}|$, we define interfering workload for each task on a path.

Def. 5. *For any τ_i, we let $I(\tau_i) = V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\})$. For any set $V' \subseteq V$, we define $I(V') = \bigcup_{\tau_i \in V'} I(\tau_i)$.*

Ex. 7 (Cont'd). In Fig. 4, for envelope path $\lambda^e = \{\tau_1, \tau_4, \tau_8, \tau_{10}\}$, $A^e = [0,4) \cup [9,12) \cup [14,17) \cup [17,19)$, when jobs of λ^e executes. In contrast, $A^{ne} = [4,9) \cup [12,14)$. By Def. 5, $I(\tau_4) = \{\tau_2, \tau_3\}$. $\Diamond$

Lemma 2. *Let τ_i be a job of λ^e. For any time $t \in [r(\tau_i), f(\tau_i))$, if τ_i is not scheduled at time t, then jobs that are scheduled on the γ_i^{th} CE at time t are in $I(\tau_i)$.*

Proof. No jobs in $dep(\tau_i)$ are ready during $[r(\tau_i), f(\tau_i))$. Also, only jobs τ_k with $\gamma_k = \gamma_i$ can be scheduled on τ_i's CE. Thus, the lemma holds. $\square$

Next, we give an upper bound on $|A^{ne}|$. We begin by introducing some necessary notation.

Def. 6. *Let $\tau_k(V')$ be the task with the k^{th}-highest Δ_i value among the tasks of V', and $\Delta_k(V')$ is its Δ_i value. We assume ties are broken by task index.*

By Def. 6, for an ℓ-node path λ of G, $\{\tau_1(\lambda), \tau_2(\lambda), \ldots, \tau_\ell(\lambda)\}$ is an ordered set of tasks of λ in descending order of Δ_i values. Moreover, since λ_p denotes the set of tasks on λ that execute on the p^{th} CE (by Def. 2), $\tau_k(\lambda_p)$ is the task with the k^{th}-largest Δ_i value among all tasks on λ that execute on the p^{th} CE.

Def. 7. *For any set $V' \subseteq V$ of tasks, let $I_i^{cup}(V') = \bigcup_{j=1}^{i} I(\tau_j(V'))$ and $I_i^{diff}(V') = I(\tau_i(V')) \setminus I_{i-1}^{cup}(V')$.*

Thus, by Defs. 7 and 5, $I_i^{cup}(V')$ consists of all tasks that may interfere with any task in $\{\tau_1(V'), \ldots, \tau_i(V')\}$. In contrast, $I_i^{diff}(V')$ consists of tasks that may interfere with task $\tau_i(V')$ but not with any task in $\{\tau_1(V'), \ldots, \tau_{i-1}(V')\}$. Thus, $\cup_{j=1}^{i} I_j^{diff}(V')$ also consists of all tasks that may interfere with any task in $\{\tau_1(V'), \ldots, \tau_i(V')\}$. Therefore, we have

$$I_i^{cup}(V') = \bigcup_{j=1}^{i} I_j^{diff}(V'). \tag{7}$$

Moreover, for any $i \neq j$, $I_i^{diff}(V')$ and $I_j^{diff}(V')$ are disjoint:

$$\forall i \neq j : I_i^{diff}(V') \cap I_j^{diff}(V') = \emptyset. \tag{8}$$

Ex. 8. Assume that the DAG in Fig. 6 is scheduled by a work-conserving scheduler on a CE of ten processors. Let $V' = \{\tau_5, \tau_6\}$. According to (3), $\Delta_5 = 1$ and $\Delta_6 = 0$. Thus, by Def. 6, $\tau_1(V') = \tau_5$ and $\tau_2(V') = \tau_6$. By Def. 5, $I(\tau_5) = \{\tau_2, \tau_4, \tau_6\}$ and $I(\tau_6) = \{\tau_4, \tau_5\}$. Thus, by Def 7, $I_2^{cup}(V') = I(\tau_5) \cup I(\tau_6) = \{\tau_2, \tau_4, \tau_5, \tau_6\}$. By Def 7, $I_1^{diff}(V') = I(\tau_5) = \{\tau_2, \tau_4, \tau_6\}$ and $I_2^{diff}(V') = I(\tau_6) \setminus I(\tau_5) = \{\tau_5\}$. $\Diamond$

Def. 8. *For any subset of tasks V_p' assigned to the p^{th} CE (Def. 2) and $1 \leq j \leq |V_p'|$, we define $F(V_p', j)$ as follows:*

$$F(V_p', j) = \sum_{i=1}^{j} \frac{vol(I_i^{diff}(V_p'))}{M_p - \Delta_i(V_p')}. \tag{9}$$

In the following lemma, we upper bound $|A^{ne}|$ by considering all tasks in λ_p^e in decreasing order of their Δ_i values, *i.e.*, in the order: $\tau_1(\lambda_p^e), \ldots, \tau_{|\lambda_p^e|}(\lambda_p^e)$. Under such an ordering, we assume that a task that can interfere with multiple tasks on λ_p^e executes when the task with the largest Δ_i value is ready but unscheduled. Fig. 7 illustrates the idea: a task that can interfere with both $\tau_1(\lambda_p^e)$ and $\tau_2(\lambda_p^e)$ is scheduled when tasks of $I_1^{diff}(\lambda_p^e)$ are executing, *i.e.*, $I_1^{diff}(\lambda_p^e)$ contains that task.

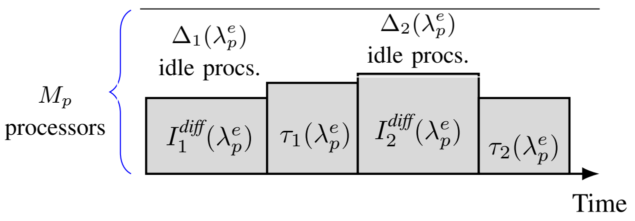


Fig. 7: Proof of Lem. 3.

Lemma 3. $|A^{ne}| \leq \sum_{p=1}^{\mu} F(\lambda_p^e, |\lambda_p^e|)$ *holds.*

Proof. For any job $\tau_i(\lambda_p^e) \in \lambda_p^e$ with $1 \leq p \leq \mu$, let $A_{i,p}^{ne}$ be the union of intervals when $\tau_i(\lambda_p^e)$ is ready but unscheduled. Note that $\tau_i(\lambda_p^e)$ follows Def. 6. Thus, $A_{i,p}^{ne} \subseteq A^{ne}$. By the definition of A^{ne}, we have $\bigcup_{p=1}^{\mu} \bigcup_{i=1}^{|\lambda_p^e|} A_{i,p}^{ne} = A^{ne}$. Moreover, by Def. 4, no two jobs of λ^e are ready at the same time. Thus, for any pair of jobs $\tau_i(\lambda_p^e)$ and $\tau_j(\lambda_q^e)$ on λ^e with $\tau_i(\lambda_p^e) \neq \tau_j(\lambda_q^e)$, $A_{i,p}^{ne} \cap A_{j,q}^{ne} = \emptyset$ holds. Therefore, we have

$$|A^{ne}| = \sum_{p=1}^{\mu} \sum_{i=1}^{|\lambda_p^e|} |A_{i,p}^{ne}|. \tag{10}$$

We now upper bound A^{ne} by upper bounding $\sum_{i=1}^{|\lambda_p^e|} |A_{i,p}^{ne}|$ in (10) for all $1 \leq p \leq \mu$. For any job τ_k on the p^{th} CE, let τ_k execute for $C_{k,i,p}$ time units during $A_{i,p}^{ne}$. By Lem. 2, jobs not in $I(\tau_i(\lambda_p^e))$ cannot execute on the p^{th} CE during $A_{i,p}^{ne}$. Thus, we have

$$\forall \tau_k \notin I(\tau_i(\lambda_p^e)) : \gamma_k = p :: C_{k,i,p} = 0. \tag{11}$$

Thus, the total execution on the p^{th} CE during $A_{i,p}^{ne}$ is $\sum_{\tau_k \in I(\tau_i(\lambda_p^e))} m_k C_{k,i,p}$. By Lem. 1 and Def. 6, at least $M_p - \Delta_i(\lambda_p^e)$ processors of the p^{th} CE are busy during $A_{i,p}^{ne}$. Hence, we can upper bound the length $|A_{i,p}^{ne}|$ as follows:

$$|A_{i,p}^{ne}| \leq \frac{\sum_{\tau_k \in I(\tau_i(\lambda_p^e))} m_k C_{k,i,p}}{M_p - \Delta_i(\lambda_p^e)}.$$

Therefore, we have

$$\sum_{i=1}^{|\lambda_p^e|} |A_{i,p}^{ne}| \leq \sum_{i=1}^{|\lambda_p^e|} \frac{\sum_{\tau_k \in I(\tau_i(\lambda_p^e))} m_k C_{k,i,p}}{M_p - \Delta_i(\lambda_p^e)}. \tag{12}$$

By Def. 5, $I(\lambda_p^e)$ denotes the set $\bigcup_{\tau_k \in \lambda_p^e} I(\tau_k)$. Using $I(\lambda_p^e)$, we can rearrange (12) as follows:

$$\sum_{i=1}^{|\lambda_p^e|} |A_{i,p}^{ne}| \leq \sum_{\tau_k \in I(\lambda_p^e)} \sum_{i=1}^{|\lambda_p^e|} \frac{m_k C_{k,i,p}}{M_p - \Delta_i(\lambda_p^e)}. \tag{13}$$

Now, consider a job $\tau_k \in I(\lambda_p^e)$. Let $sm(k)$ be the smallest i value such that $\tau_k \in I(\tau_i(\lambda_p^e))$, *i.e.*, $\forall j < sm(k), \tau_k \notin I(\tau_j(\lambda_p^e))$. By (11), for any $j < sm(k)$, τ_k does not execute during $A_{j,p}^{ne}$, *i.e.*, $C_{k,j,p} = 0$. Therefore, by (13), we have

$$\sum_{i=1}^{|\lambda^e|} |A_{i,p}^{ne}| \leq \sum_{\tau_k \in I(\lambda_p^e)} \sum_{i=sm(k)}^{|\lambda_p^e|} \frac{m_k C_{k,i,p}}{M_p - \Delta_i(\lambda_p^e)}$$

$$\leq \{\text{By Def. 6, } \Delta_i(\lambda_p^e) \geq \Delta_{i+1}(\lambda_p^e)\}$$
$$\sum_{\tau_k \in I(\lambda_p^e)} \sum_{i=sm(k)}^{|\lambda_p^e|} \frac{m_k C_{k,i,p}}{M_p - \Delta_{sm(k)}(\lambda_p^e)}$$
$$\leq \{\text{Since } \sum_{i=sm(k)}^{|\lambda_p^e|} C_{k,i,p} \leq C_k\}$$
$$\sum_{\tau_k \in I(\lambda_p^e)} \frac{m_k C_k}{M_p - \Delta_{sm(k)}(\lambda_p^e)}. \tag{14}$$

Now, by the definition of $sm(k)$, $sm(k) = i$ holds when τ_k is in $I(\tau_i(\lambda_p^e))$ but not in any $I(\tau_j(\lambda_p^e))$ with $j < i$. Thus, $sm(k) = i$ holds if and only if $\tau_k \in I(\tau_i(\lambda_p^e)) \setminus \bigcup_{j=1}^{i-1} I(\tau_j(\lambda_p^e)) = I_i^{\mathit{diff}}(\lambda_p^e)$ (by Def. 7). Thus, by (14) and (9),

$$\sum_{i=1}^{|\lambda_p^e|} |A_i^{ne}| \leq \sum_{i=1}^{|\lambda_p^e|} \frac{vol(I_i^{\mathit{diff}}(\lambda_p^e))}{M_p - \Delta_i(\lambda_p^e)} = F(\lambda_p^e, |\lambda_p^e|).$$

The lemma holds by applying the above inequality in (10). □

Applying Lem. 3 in (6), we have the following lemma.

Lemma 4. *In $\mathcal{S}$, the response time of J is bounded as follows: $R(J) \leq len(\lambda^e) + \sum_{p=1}^{\mu} F(\lambda_p^e, |\lambda_p^e|)$.*

Proof. By the definition of A^e, $|A^e| \leq len(\lambda^e)$ holds. Applying $|A^e| \leq len(\lambda^e)$ and Lem. 3 in (6), the lemma holds. □

Now, G's response time can be upper bounded by considering all complete paths as an envelope path in Lem. 4.

Theorem 1. *Let $\Lambda(G)$ be the set of all complete paths of G. G's response time is bounded as follows:*

$$R(G) \leq \max_{\lambda \in \Lambda(G)} \left(len(\lambda) + \sum_{p=1}^{\mu} F(\lambda_p, |\lambda_p|) \right) \tag{15}$$

Proof. Follows from Lem. 4. □

Unfortunately, even for the special case of sequential nodes (*i.e.*, $\Delta_i = 0$), computing the exact value of the right-hand-side of (15) is NP-hard in the strong sense [27, Thm. 4.2]. Moreover, the variation in the number of idle processors (Δ_i values) during different sub-intervals of A^{ne} (see Fig. 7) complicates the application of existing approaches to upper bound (15), as in [27], for the sequential case (where the denominator in (9) is always M_p). Thus, such approaches can focus only on maximizing the numerator of (9).

Upper bounding (15)**.** To upper bound (15), instead of the tasks of λ_p, we consider the tasks of V_p in order of decreasing Δ_i values. Thus, we consider tasks of V_p in the order: $\tau_1(V_p), \ldots, \tau_{|V_p|}(V_p)$. Since $\lambda_p \subseteq V_p$, considering the tasks of V_p in such an order assumes that more processors are idle during A^{ne}. This enables upper bounding $R(G)$ without determining the path λ that maximizes (15).

We now show how to upper bound $F(\lambda_p, |\lambda_p|)$ in (15) using the tasks of V_p in order of decreasing Δ_i values. Since $\lambda_p \subseteq V_p$, the volume of tasks that may interfere with tasks in λ_p (numerator in (9)) can be expressed as the volume of tasks that may interfere with a subset of tasks $\tau_1(V_p), \ldots, \tau_k(V_p)$. In Lem. 5, we determine such an equivalent interfering workload from a subset of tasks in V_p and divide them by the corresponding Δ_i values to upper bound $F(\lambda_p, j)$ for any j. We first give the following notation.

Def. 9. *For any set of tasks V', we denote by $vi_k(V')$ the volume of all tasks that may interfere with any tasks in $\{\tau_1(V'), \ldots, \tau_k(V')\}$. Thus, by Def. 7, $vi_k(V') = vol(I_k^{\mathit{cup}}(V'))$. We define $vi_0(V') = 0$. Furthermore, by* (7) *and* (8)*, $vi_k(V')$ satisfies the following:*

$$vi_k(V') = vol\left(\bigcup_{i=1}^{k} I_i^{\mathit{diff}}(V')\right) = \sum_{i=1}^{k} vol\left(I_i^{\mathit{diff}}(V')\right). \tag{16}$$

Note that, by Defs. 5 and 9, the volume of all tasks that may interfere with any task of V' is

$$vol(I(V')) = vi_{|V'|}(V'). \tag{17}$$

Lemma 5. *For any path λ, CE type p, and $j \leq |\lambda_p|$, let $1 \leq y(j) \leq |V_p|$ be the smallest integer so that $vi_j(\lambda_p) \leq vi_{y(j)}(V_p)$. Let $x(j) = vi_j(\lambda_p) - vi_{y(j)-1}(V_p)$. Then,*

$$F(\lambda_p, j) \leq F(V_p, y(j)-1) + \frac{x(j)}{M_p - \Delta_{h(V_p, y(j))}}. \tag{18}$$

Proof. Note that $vi_j(\lambda_p) = vi_{y(j)-1}(V_p) + x(j)$. Thus, the volume of tasks that interfere with $\{\tau_1(\lambda_p), \ldots, \tau_j(\lambda_p)\}$ is the same as the sum of $x(j)$ and the volume of tasks that interfere with $\{\tau_1(V_p), \ldots, \tau_{y(j)-1}(V_p)\}$.

We first consider the case where $vi_j(\lambda_p) = 0$. Then, by (16), $vol(I_i^{\mathit{diff}}(\lambda_p)) = 0$ for all $i \leq j$. Thus, by (9), $F(\lambda_p, j) = 0$, and the lemma trivially holds.

We now consider $vi_j(\lambda_p) > 0$. Since $y(j)$ is the smallest integer with $vi_j(\lambda_p) \leq vi_{y(j)}(V_p)$, we have

$$vi_{y(j)-1}(V_p) < vi_j(\lambda_p). \tag{19}$$

We start by showing the following:

(P) $\quad \Delta_{y(j)}(V_p) \geq \Delta_j(\lambda_p)$.

Assume that $\Delta_{y(j)}(V_p) < \Delta_j(\lambda_p)$. Thus, since $\lambda_p \subseteq V_p$, by Def. 6, each task in $\{\tau_1(\lambda_p), \ldots, \tau_j(\lambda_p)\}$ is also in $\{\tau_1(V_p), \ldots, \tau_{y(j)-1}(V_p)\}$. Therefore, by Defs. 5 and 7, $I_j^{\mathit{cup}}(\lambda_p) \subseteq I_{y(j)-1}^{\mathit{cup}}(V_p)$, which by (7) implies that

$$\bigcup_{i=1}^{j} I_i^{\mathit{diff}}(\lambda_p) \subseteq \bigcup_{i=1}^{y(j)-1} I_i^{\mathit{diff}}(V_p)$$
$$\xRightarrow{\text{By (16)}} vi_j(\lambda_p) \leq vi_{y(j)-1}(V_p).$$

This contradicts (19). Thus, (P) holds.

We now prove the lemma. We give a proof by induction on index j. Assume that the lemma holds for $j-1$:

$$F(\lambda_p, j-1) \leq F(V_p, y(j-1)-1) + \frac{x(j-1)}{M_p - \Delta_{y(j-1)}(V_p)}. \tag{20}$$

By (9), we have

$$F(\lambda_p, j) = F(\lambda_p, j-1) + \frac{vol(I_j^{\mathit{diff}}(\lambda_p))}{M_p - \Delta_j(\lambda_p)}$$
$$\leq \{\text{By (20) and (P)}\}$$

$$F(V_p, y(j-1)-1) + \frac{x(j-1)}{M_p - \Delta_{y(j-1)}(V_p)} + \frac{vol(I_j^{diff}(\lambda_p))}{M_p - \Delta_{y(j)}(V_p)}. \tag{21}$$

To prove the lemma, we now express $vol(I_j^{diff}(\lambda_p))$ in (21) using the volume of a subset of tasks in V_p. By (16), we have $vol(I_j^{diff}(\lambda_p)) = vi_j(\lambda_p) - vi_{j-1}(\lambda_p)$. Applying the definition of $y(\cdot)$ and $x(\cdot)$ in $vi_j(\lambda_p) - vi_{j-1}(\lambda_p)$, we have $vol(I_j^{diff}(\lambda_p)) = vi_{y(j)-1}(V_p) + x(j) - vi_{y(j-1)-1}(V_p) - x(j-1) = \sum_{i=y(j-1)}^{y(j)-1} vol(I_i^{diff}(V_p)) + x(j) - x(j-1)$. Dividing this equation by $M_p - \Delta_{y(j)}(V_p)$ yields

$$\begin{aligned}
&\frac{vol(I_j^{diff}(\lambda_p))}{M_p - \Delta_{y(j)}(V_p)} \\
&= \sum_{i=y(j-1)}^{y(j)-1} \frac{vol(I_i^{diff}(V_p))}{M_p - \Delta_{y(j)}(V_p)} + \frac{x(j) - x(j-1)}{M_p - \Delta_{y(j)}(V_p)} \\
&\leq \{\text{By Def. 6, for any } i \leq y(j), \Delta_i(V_p) \geq \Delta_{y(j)}(V_p)\} \\
&\quad \sum_{i=y(j-1)}^{y(j)-1} \frac{vol(I_i^{diff}(V_p))}{M_p - \Delta_i(V_p)} + \frac{x(j)}{M_p - \Delta_{y(j)}(V_p)} \\
&\quad - \frac{x(j-1)}{M_p - \Delta_{y(j-1)}(V_p)}.
\end{aligned}$$

Applying the above inequality in (21), we have

$$\begin{aligned}
F(\lambda_p, j) &\leq F(V_p, y(j-1)-1) + \sum_{i=y(j-1)}^{y(j)-1} \frac{vol(I_i^{diff}(V_p))}{M_p - \Delta_i(V_p)} \\
&\quad + \frac{x(j)}{M_p - \Delta_{y(j)}(V_p)} \\
&= \{\text{By (9)}\} \\
&= F(V_p, y(j)-1) + \frac{x(j)}{M_p - \Delta_{y(j)}(V_p)}.
\end{aligned}$$

This completes the proof of the lemma. □

Using (18) to upper bound $F(\lambda_p, |\lambda_p|)$ requires determining $y(|\lambda_p|)$ and $x(|\lambda_p|)$. By Lem. 18 and (21), determining such values requires determining $vi_{|\lambda_p|}(\lambda_p) = vol(I(\lambda_p))$. We upper bound $vol(I(\lambda_p))$ by determining a path $\lambda^{min(p)}$ for which the volume of tasks on the pth CE is the minimum. Since tasks on a path do not contribute to its interfering workload, for any λ, $vol(I(\lambda_p))$ does not exceed $vol(V_p) - vol(\lambda_p^{min(p)})$. Using the following definition, this is shown in Lem. 6.

Def. 10. *For any $1 \leq p \leq \mu$, let $\Lambda_p(G)$ be the set of all complete paths that have at least one task assigned to the pth CE. Let $\lambda^{\min(p)}$ be a path in $\Lambda_p(G)$ with the minimum $vol(\lambda_p^{\min(p)})$, i.e., $\lambda^{\min(p)} = \arg\min_{\lambda \in \Lambda_p(G)}(vol(\lambda_p))$.*

Lemma 6. *For any path λ and $1 \leq p \leq \mu$, $vol(I(\lambda_p)) \leq vol(V_p) - vol(\lambda_p^{\min(p)})$.*

Proof. If $|\lambda_p| = 0$, then $vol(I(\lambda_p)) = 0$, and the lemma trivially holds. Assuming $|\lambda_p| > 0$, by Def. 10, $vol(\lambda_p) \geq vol(\lambda_p^{min(p)})$. Since each task in $I(\lambda_p)$ executes on the pth CE, by Def. 5, $I(\lambda_p) \subseteq V_p$. However, by Def. 5, $I(\lambda_p)$ does not contain any task $\tau_i \in \lambda_p \subseteq V_p$. Thus, $vol(I(\lambda_p)) \leq vol(V_p) - vol(\lambda_p)$ holds. Since, $vol(\lambda_p) \geq vol(\lambda_p^{min(p)})$, we have $vol(I(\lambda_p)) \leq vol(V_p) - vol(\lambda_p^{\min(p)})$. □

We now define the two terms $h(p)$ and $g(p)$ that can be used, instead of $y(|\lambda_p|)$ and $x(|\lambda_p|)$, to upper bound $F(\lambda_p, |\lambda_p|)$. Using these values, we upper bound $F(\lambda_p, |\lambda_p|)$ in Lem. 7 by adding (potentially) more terms in (18).

Def. 11. *For any $1 \leq p \leq \mu$, let $1 \leq h(p) \leq |V_p|$ be the smallest integer with $vol(V_p) - vol(\lambda_p^{\min(p)}) \leq vi_{h(p)}(V_p)$ holds. Also, let $g(p) = vol(V_p) - vol(\lambda_p^{\min(p)}) - vi_{h(p)-1}(V_p)$.*

Lemma 7. *For any λ and $1 \leq p \leq \mu$, the following holds.*

$$F(\lambda_p, |\lambda_p|) \leq F(V_p, h(p)-1) + \frac{g(p)}{M_p - \Delta_{h(p)}(V_p)}$$

Proof. Let $y(|\lambda_p|)$ be the smallest integer so that $vi_{|\lambda_p|}(\lambda_p) \leq vi_{y(|\lambda_p|)}(V_p)$ holds. Let $x(|\lambda_p|) = vi_{|\lambda_p|}(\lambda_p) - vi_{y(|\lambda_p|)-1}(V_p)$. By Lem. 5, we have

$$F(\lambda_p, |\lambda_p|) \leq F(V_p, y(|\lambda_p|)-1) + \frac{x(|\lambda_p|)}{M_p - \Delta_{y(|\lambda_p|)}(V_p)}. \tag{22}$$

By (17), we have $vol(I(\lambda_p)) = vi_{|\lambda_p|}(\lambda_p)$. By Lem. 6, $vol(I(\lambda_p)) = vi_{|\lambda_p|}(\lambda_p) \leq vol(V_p) - vol(\lambda_p^{min(p)})$. Therefore, by the definition of $y(|\lambda_p|)$ and $h(p)$, we have $y(|\lambda_p|) \leq h(p)$. We now consider two cases.

Case 1. $y(|\lambda_p|) = h(p)$. Since $vol(I(\lambda_p)) = vi_{|\lambda_p|}(\lambda_p) \leq vol(V_p) - vol(\lambda_p^{min(p)})$, by the definition of $x(|\lambda_p|)$ and $g(p)$, we have $x(|\lambda_p|) \leq g(p)$. Thus, the lemma holds by replacing $y(|\lambda_p|)$ and $x(|\lambda_p|)$ with $h(p)$ and $g(p)$, respectively, in (22).

Case 2. $y(|\lambda_p|) < h(p)$. By the definition of $x(|\lambda_p|)$, $x(|\lambda_p|) = vi_{|\lambda_p|}(\lambda_p) - vi_{y(|\lambda_p|)-1}(V_p) \leq vi_{y(|\lambda_p|)}(V_p) - vi_{y(|\lambda_p|)-1}(V_p) = vol(I_{y(|\lambda_p|)}^{diff}(V_p))$. Replacing $x(|\lambda_p|)$ with $vol(I_{y(|\lambda_p|)}^{diff}(V_p))$ and adding additional non-negative terms in (22), we have

$$\begin{aligned}
F(\lambda_p, |\lambda_p|) &\leq F(V_p, y(|\lambda_p|)-1) + \frac{vol(I_{y(|\lambda_p|)}^{diff}(V_p))}{M_p - \Delta_{y(|\lambda_p|)}(V_p)} \\
&\quad + \sum_{i=y(|\lambda_p|)+1}^{h(p)-1} \frac{vol(I_i^{diff}(V_p))}{M_p - \Delta_i(V_p)} + \frac{g(p)}{M_p - \Delta_{h(p)}(V_p)} \\
&= F(V_p, h(p)-1) + \frac{g(p)}{M_p - \Delta_{h(p)}(V_p)}.
\end{aligned}$$

Thus, the lemma holds. □

Using Lem. 7, we have the following theorem.

Theorem 2. *G's response time is bounded as follows:*

$$R(G) \leq len(G) + \sum_{p=1}^{\mu} \left(F(V_p, h(p)-1) + \frac{g(p)}{M_p - \Delta_{h(p)}(V_p)} \right), \tag{23}$$

where $len(G)$ denotes length of the longest path of G.

Proof. The theorem follows from applying $len(G) = \max_{\lambda \in \Lambda(G)} len(\lambda)$ and Lem. 7 in Thm. 1. □

Running time. $len(G)$ and each $vol(\lambda_p^{\min(p)})$ can be computed in $O(V+E)$ time. For DAG, the set $I(\tau_i)$ for all tasks can be computed in $O(|V|^2)$ time. Using precomputed $I(\tau_i)$ sets, for each task, each I^{diff} set (to compute $F(V_p, h(p)-1)$) can be computed in $O(|V|)$ time. Since each task appears at most once in the response-time bound expression in (23), computing all numerators in (23) takes $O(|V|^2)$ time. Since computing all Δ_i values takes $O(|V|^3 \max_p\{M_p\})$ time, the total running time is $O(|V|^3 \max_p\{M_p\})$.

VI. Processor Allocation

In this section, we give an ILP to allocate processors among multiple DAGs. We consider DAGs $G^1, G^2, \cdots, G^N$. For all introduced notation, we use a superscript k to denote the corresponding term for the kth DAG G^k. We also assume that the pth CE has $\mathcal{M}_p$ processors.

For DAG G^k, let $R^k_{p,m}$ denote the value $\left(F(V^k_p, h^k(p)-1) + \frac{g^k(p)}{m - \Delta_{h^k(p)}(V^k_p)}\right)$ of (23) and $len(G^k)$ denote its longest-path length. Using this notation, the ILP is specified as follows.

Variables: For each pair of DAG G^k and pth CE, we define $\mathcal{M}_p$ variables $x^k_{p,1}, x^k_{p,2}, \cdots, x^k_{p,\mathcal{M}_p}$. $x^k_{p,m}$ is 1 if DAG G^k is assigned m processors on the pth CE, and 0 otherwise.

Constraint 1. For each pair of DAG and CE, exactly one $x^k_{p,m}$ is 1 (the DAG receives m processors on that CE):

$$\forall k \in \{1, \cdots, N\}, \forall p \in \{1, \cdots, \mu\} :: \sum_{m=1}^{\mathcal{M}_p} x^k_{p,m} = 1.$$

Constraint 2. The total number of allocated processors per CE is at most the number of processors that CE has:

$$\forall p \in \{1, \cdots, \mu\} :: \sum_{k=1}^{N} \sum_{m=1}^{\mathcal{M}_p} m \cdot x^k_{p,m} \leq \mathcal{M}_p.$$

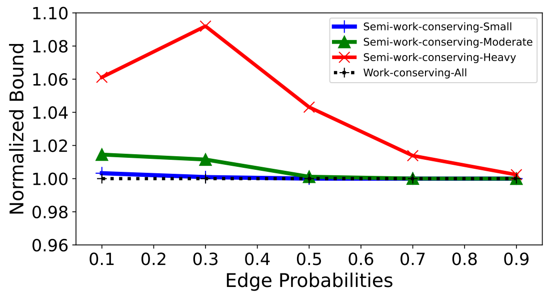
Constraint 3. On allocated processors, each DAG meets its deadline:

$$\forall k \in \{1, \cdots, N\} : len(G^k) + \sum_{p=1}^{\mu} \sum_{m=1}^{\mathcal{M}_p} R^k_{p,m} x^k_{p,m} \leq D^k.$$

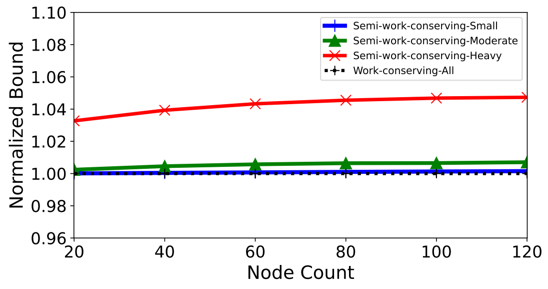
The ILP has $N \sum_{p=1}^{\mu} \mathcal{M}_p$ variables and $N\mu + N + \mu$ constraints. For systems with many heavy DAGs, the ILP can be solved efficiently, as seen in our experimental evaluation.

VII. Experiments

We now present the results of the experiments we conducted to evaluate the response-time bounds of our approach. First, we compared schedulability under work-conserving and semi-work-conserving scheduling for a DAG on an arbitrary number of processors. Second, for multicore+GPU platforms, we compared schedulability under semi-work-conserving scheduling on GPUs with traditional locking-based approaches [5]. Finally, we demonstrated the practicality of our approach via a case study on a multicore+GPU platform.



(a) Normalized bound vs. edge generation probabilities.



(b) Normalized bound vs. node count.

Fig. 8: Results of experiments on arbitrary number of CEs.

A. Experiments on Arbitrary Number of CEs

In this experiment, we compared the response-time bounds in Theorem 2 under work-conserving and semi-work-conserving schedulers. We generated DAGs following the *Erdős-Rényi method* [18]. The number of nodes per DAG was selected from $[20, 120]$. Each task's WCET was chosen from $[50, 100]$. For each pair of nodes (τ_i, τ_j) with $i < j$, an edge from τ_i to τ_j was added if a uniformly generated random number in $[0, 1]$ was at most a predefined *edge-generation probability*. We selected this probability value from $\{0.1, 0.3, 0.5, 0.7, 0.9\}$. A higher edge-generation probability makes DAGs more sequential. As in [48], additional edges were added to make each DAG weakly connected.

The number of CEs was randomly selected from $[2, 6]$. The number of processors per CE was selected from $\{8, 16, 24, 32\}$, which represent common values in real-world use cases [4], [36]. Each task was assigned to one of the CEs with uniform probabilities. We considered *small*, *moderate*, or *heavy* degrees of parallelism, for which m_i values were uniformly distributed in $[1, 0.2M_{\gamma_i}]$, $[1, 0.4M_{\gamma_i}]$, and $[1, 0.7M_{\gamma_i}]$, respectively, where M_{γ_i} is the number of processors on τ_i's CE. For each combination of edge-generation probabilities and degrees of parallelism, we generated 1,000 task sets.

We computed the average *normalized response-time bound*, which is the ratio between the response-time bound under semi-work-conserving and work-conserving scheduling. Thus, normalized response-time bounds less than 1.0 imply smaller response-time bounds under semi-work-conserving scheduling than under work-conserving scheduling. The normalized response-time bounds are plotted in Fig. 8.

Observation 1. *For small, moderate, and heavy degrees of parallelism, the average response-time bounds under semi-work-conserving scheduling were* $1.001\times$, $1.005\times$, *and* $1.04\times$

of those under work-conserving scheduling, respectively.

For heavy degrees of parallelism, semi-work-conserving scheduling caused larger response-time bounds compared to work-conserving scheduling. This is because the amount of wasted processing capacity (Δ_i values) due to each task can often be larger for semi-work-conserving scheduling (see (5)). For smaller edge-generation probabilities, the difference in response-time bounds between work-conserving and semi-work-conserving scheduling increased. (see Fig. 8(a)). This is because the interfering workload (the summation term in (23)) contributed more significantly to the response-time bounds. Increasing the number of nodes slightly increased the normalized response-time bounds (see Fig. 8(b)). For small degrees of parallelism, the response-time bounds under both work-conserving and semi-work-conserving scheduling were close, as Δ_i values under both scheduling were small. Note that semi-work-conserving scheduling becomes work-conserving when all tasks have $m_i = 1$.

B. Experiments on Multicore+GPU

In this experiment, we considered systems scheduled on multicore+GPU platforms. We compared our response-time bounds under semi-work-conserving scheduling with locking-based approaches. For the locking-based approach, we considered a recently proposed locking protocol, called the SMLP, which allows multiple jobs to access a GPU simultaneously by allocating SMs among them [5]. Under the SMLP, an upper bound on *priority-inversion* blocking[4] (pi-blocking) time can be derived under any *job-level fixed-priority* scheduling [5]. Using such a pi-blocking bound, DAG response-time bounds can be derived by inflating task WCETs and then applying any *suspension-oblivious* response-time analysis techniques.

For the locking-based approach, we considered two state-of-the-art response-time bounds: RM-HE [30] and WC-HE [28]. RM-HE considers *prioritized list scheduling* of a DAG of sequential tasks and supports multi-DAG systems by prioritizing different DAGs by the *rate-monotonic* algorithm. WC-HE applies under any work-conserving scheduler, where multiple DAGs are supported by federated scheduling techniques.

Single-DAG systems. To describe task generation, we use NVIDIA-GPU-specific terms. Our GPU-specific task parameter generation was inspired by prior work [5], [43], [53]. We considered platforms consisting of $\{8, 16, 24, 32\}$ processors and $\{16, 32, 48\}$ SMs, where each SM consists of 2,048 GPU threads. We first generated *coarse-grained* DAG tasks consisting of sequential CPU tasks by the same task-generation method given in Sec. VII-A, where we set $\mu = 1$ and $m_i = 1$ to generate only CPU tasks. We randomly selected some CPU tasks as *GPU-accessing* tasks. We considered *small* ([1–20]%), *moderate* ([20-50]%), and *heavy* ([50–80]%) ratios of GPU-accessing tasks. We generated GPU-access lengths according to the method in [5]. The maximum GPU-access lengths were selected uniformly from $[0.1C_i, 0.7C_i]$. By [5], a task's

[4]Under the locking-based approach, higher-priority jobs can suffer pi-blocking when they wait for GPU but a lower-priority job is scheduled.

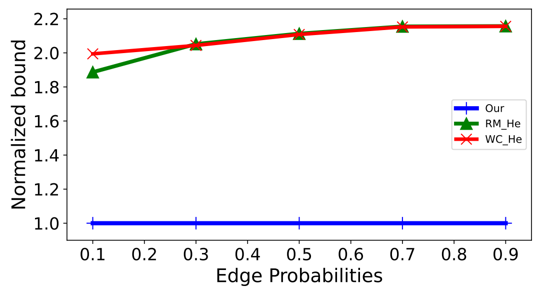


Fig. 9: Normalized bound vs. edge generation probabilities for *moderate* GPU-access ratio.

maximum GPU-access length occurs when the number of SMs allocated to the task is small. Similar to [5], for each GPU-accessing task τ_i, we selected a value ρ_i, not exceeding the number of total SMs, that represents the maximum number of SMs the task can utilize, *i.e.*, GPU-access lengths do not increase if more SMs are allocated.

To apply our approach, we then generated *fine-grained* DAGs by splitting the GPU-accessing tasks of *coarse-grained* DAGs. Each GPU-accessing task was split into two CPU tasks and multiple GPU blocks. Each *block* was a gang task, for which we selected *block sizes* (*i.e.*, m_i values) randomly from $\{126, 256, 512, 1028\}$. The number of blocks was determined so that increasing the number by one required more the ρ_i SMs. Finally, edges were added from one CPU task to all GPU blocks and from all GPU blocks to the other CPU task. For each combination of processor count, SM count, edge probabilities, and GPU-accessing task ratios, we generated 1,000 task sets. We compared our bound (OUR) under semi-work-conserving scheduling with RM-HE and WC-HE. Fig. 9 presents these three bounds *normalized* with respect to OUR.

Observation 2. *For small, moderate, and heavy GPU-accessing-task ratios, bounds under* RM-HE *(resp.,* WC-HE*) were, on average,* $1.24\times$, $2.07\times$, *and* $3.30\times$ *(resp.,* $1.25\times$, $2.09\times$, *and* $3.32\times$*), respectively, of those under* OUR.

For systems with many GPU-accessing tasks, OUR gave much smaller response-time bounds than RM-HE and WC-HE. Fig. 9 shows this by plotting normalized bounds with respect to edge-generation probabilities. The bounds of RM-HE and WC-HE were larger compared to OUR for larger edge-generation probabilities. This is because of pi-blocking-related inflation of WCETs under RM-HE and WC-HE. With large edge-generation probabilities, accumulated inflation of WCETs along the longest path of DAG became high.

Multi-DAG systems. In this section, we used the above single DAG-generation method iteratively to generate multiple DAGs. For each processor count in $\{8, 16, 24, 32\}$, we generated task systems with coarse-grained DAGs that have *normalized utilizations*, *i.e.*, sum of all DAG utilizations over processor count, from 0.1 to 1 with a step size of 0.1. Similar to [30], we chose DAG G's period uniformly from $[len(G), 6 \cdot len(G)]$, where $len(G)$ is its longest-path length. For each combination of processor count, SM count, edge probabilities, and GPU-accessing task ratios, we generated

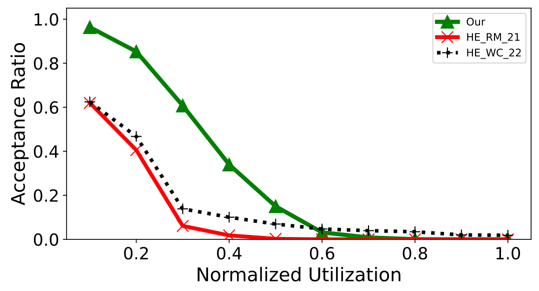


Fig. 10: Acceptance ratio vs. normalized utilizations for *light* GPU-access ratio.

1,000 task sets. For each combination, we determined the *acceptance ratio*, which gives the percentage of task systems that were schedulable under each of OUR, RM-HE, and WC-HE. Fig. 10 presents these acceptance ratios.

Observation 3. *For small, moderate, and heavy GPU-accessing-task ratios,* RM-HE *(resp.,* WC-HE*) scheduled* 14%*,* 19%*, and* 30% *(resp.,* 15%*,* 22%*, and* 48%*) of the systems compared to* OUR*, respectively.*

Similar to single-DAG experiments, WCET inflations caused fewer systems to be schedulable under RM-HE and WC-HE than OUR. However, as seen in Fig. 10, WC-HE scheduled some systems that OUR cannot. This is because light DAGs share processors under WC-HE, while our analysis requires allocating a dedicated processor to each light DAG.

C. Case Study on Multicore+GPU

For this case study, we employed the pedestrian-detection algorithm *Histogram of Oriented Gradients (HOG)* [19]. HOG computes gradients over each frame of a video feed via a series of CUDA kernels, forming a DAG of gang tasks with sequential CPU and parallel GPU computations. These experiments were conducted on a machine running a modified version of LITMUS^{RT} [14], [16], a Linux-based real-time kernel. The machine had a 3.5-GHz AMD Ryzen 9 3950X 16-Core Processor and one NVIDIA RTX 6000 Ada Generation GPU.

We considered both single- and multi-DAG scenarios. In both scenarios, we ran HOG under the locking-based approach using the OMLP protocol [9], [15] and under federated scheduling with the default semi work-conserving scheduler [10]. In the multi-DAG scenario, we ran four parallel HOG instances. We used `libsmctrl` [11] to partition GPU among the four HOG instances under federated scheduling. We measured response times of 1,000 DAG jobs, each processing a video frame at five image-scale levels. In the single-DAG scenario, under the locking-based approach and semi work-conserving scheduling, the average (resp., maximum) response time was 2.6ms (resp., 8.5ms) and 2.5ms (resp., 7.1ms), respectively. Thus, there was a 16.5% reduction in maximum response time under the semi work-conserving approach. For multiple DAGs, under the locking-based approach and semi work-conserving scheduling, the average (resp., maximum) response time was 9.9ms (resp., 15.9ms) and 143.0ms (resp., 31.5ms), respectively.

VIII. RELATED WORK

The literature on scheduling DAG tasks and gang tasks is quite vast. Below, we comment on some relevant works.

DAG scheduling. Most work on DAG scheduling considered nodes as sequential tasks and assumed homogeneous multiprocessor platforms. Graham gave a well-known response-time bound for DAGs under any work-conserving scheduler [26]. Recently proposed multi-path bounds improved Graham's bound by considering multiple paths of the DAGs [28], [52]. Some work also considered assigning node priorities to improve Graham's bound [17], [29], [30]. These bounds can be applied to multi-DAG systems using federated scheduling techniques [12]. Variants of federated scheduling exist that allow multiple DAGs to share a few processors [33], [34]. Other work on multi-DAG scheduling applied global and partitioned scheduling techniques and analyzed inter-DAG interference to derive response-time bounds [24], [39], [41], [44].

Jaffey first considered scheduling DAGs on heterogeneous platforms and gave an analysis based on envelope paths [32]. Later work improved this bound using a less pessimistic envelope path and interfering workload estimations [27], [31]. Lin *et al.* gave a type-aware federated scheduling algorithm on two-CE platforms that allows the sharing of processors of a CE among those DAGs that are light with respect to a CE [40]. Other work considered soft real-time DAGs on heterogeneous processors [55] and graph-restructuring techniques [49]. Work on scheduling DAGs on multicore+GPU either considered lock-based approaches [9] or soft real-time DAGs [56].

Gang scheduling. Most prior work on gang scheduling focused on independent gang tasks. Schedulability tests for preemptive gang scheduling are known for global [2], [20], [25], [35], [38], [38], [46] and variants of partitioned scheduling [50], [51]. Recent work proposed schedulability tests for non-preemptive gang scheduling [21], [37], [42]. Scheduling of bundled gang tasks was studied under global and partitioned fixed-priority scheduling [47], [54]. All the above-mentioned work considered independent or bundled gang tasks and specific schedulers that are special cases of graphs of gang tasks and work-conserving schedulers, respectively. To mitigate interference due to shared resources, scheduling one gang task at a time was studied [6], which leaves more processors idle than semi-work-conserving schedulers.

IX. CONCLUSION

In this work, we considered the scheduling of DAGs composed of gang tasks on heterogeneous processing platforms. We presented a polynomial-time response-time bound for such DAGs under any scheduler that is either work-conserving or semi-work-conserving. We have also given an ILP formulation to allocate processors among multiple DAGs. We have demonstrated the utility of our approach through schedulability studies and a case study on a multicore+GPU platform. In future work, we plan to devise techniques to share processors of different CEs among light DAGs. We also plan to devise response-time bounds for arbitrary-deadline DAGs.

References

[1] S. Ahmed and J. H. Anderson, "Exact Response-Time Bounds of Periodic DAG Tasks under Server-Based Global Scheduling," in *RTSS*, 2022, pp. 447–459.

[2] ——, "Soft Real-Time Gang Scheduling," in *RTSS*, 2023, pp. 331–343.

[3] ——, "Open Problem Resolved: The "Two" in Existing Multiprocessor PI-Blocking Bounds Is Fundamental," in *ECRTS*, 2024, pp. 11:1–11:21.

[4] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "An Empirical Survey-based Study into Industry Practice in Real-time Systems," in *RTSS*, 2020, pp. 3–11.

[5] S. W. Ali, Z. Tong, J. Goh, and J. H. Anderson, "Predictable GPU Sharing in Component-Based Real-Time Systems," in *ECRTS*, 2024, pp. 15:1–15:22.

[6] W. Ali and H. Yun, "RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems," in *RTAS*, 2019, pp. 143–155.

[7] T. Amert, S. Voronov, and J. Anderson, "OpenVX and real-time certification: The troublesome history," in *RTSS*, 2019, pp. 312–325.

[8] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *RTSS*, 2017, pp. 104–115.

[9] T. Amert, Z. Tong, S. Voronov, J. Bakita, F. D. Smith, and J. H. Anderson, "TimeWall: Enabling Time Partitioning for Real-Time Multicore+Accelerator Platforms," in *RTSS*, 2021, pp. 455–468.

[10] J. Bakita and J. H. Anderson, "Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management," in *RTAS*, 2024, pp. 294–305.

[11] ——, "Hardware Compute Partitioning on NVIDIA GPUs," in *RTAS*, 2023, pp. 54–66.

[12] S. Baruah, "The federated scheduling of constrained-deadline sporadic DAG task systems," in *DATE*, 2015, pp. 1323–1328.

[13] A. Biondi and G. Buttazzo, "Timing-aware FPGA partitioning for real-time applications under dynamic partial reconfiguration," in *AHS*, 2017, pp. 172–179.

[14] B. B. Brandenburg, "Scheduling and Locking in Multiprocessor Real-time Operating Systems," Ph.D. dissertation, 2011.

[15] B. B. Brandenburg and J. H. Anderson, "Optimality Results for Multiprocessor Real-Time Locking," in *RTSS*, 2010, pp. 49–60.

[16] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^{RT} : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *RTSS*, 2006, pp. 111–126.

[17] S. Chang, R. Bi, J. Sun, W. Liu, Q. Yu, Q. Deng, and Z. Gu, "Toward Minimum WCRT Bound for DAG Tasks Under Prioritized List Scheduling Algorithms," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 11, pp. 3874–3885, 2022.

[18] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *SIMUTools*, 2010, p. 60.

[19] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *CVPR*, vol. 1, 2005, pp. 886–893 vol. 1.

[20] Z. Dong and C. Liu, "Analysis techniques for supporting hard real-time sporadic gang task systems," *Real Time Syst.*, vol. 55, no. 3, pp. 641–666, 2019.

[21] ——, "A Utilization-based Test for Non-preemptive Gang Tasks on Multiprocessors," in *RTSS*, 2022, pp. 105–117.

[22] Z. Dong, K. Yang, N. Fisher, and C. Liu, "Tardiness Bounds for Sporadic Gang Tasks Under Preemptive Global EDF Scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 12, pp. 2867–2879, 2021.

[23] G. A. Elliott, K. Yang, and J. H. Anderson, "Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms," in *RTSS*, 2015, pp. 273–284.

[24] J. C. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic DAG tasks for global FP scheduling," in *RTNS*, 2017, pp. 28–37.

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

[26] R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM J. of Appl. Math.*, vol. 17, no. 2, pp. 416–429, 1969.

[27] M. Han, N. Guan, J. Sun, Q. He, Q. Deng, and W. Liu, "Response Time Bounds for Typed DAG Parallel Tasks on Heterogeneous Multi-Cores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 11, pp. 2567–2581, 2019.

[28] Q. He, N. Guan, M. Lv, X. Jiang, and W. Chang, "Bounding the Response Time of DAG Tasks Using Long Paths," in *RTSS*, 2022, pp. 474–486.

[29] Q. He, X. Jiang, N. Guan, and Z. Guo, "Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2283–2295, 2019.

[30] Q. He, M. Lv, and N. Guan, "Response Time Bounds for DAG Tasks with Arbitrary Intra-Task Priority Assignment," in *ECRTS*, 2021, pp. 8:1–8:21.

[31] Q. He, Y. Sun, M. Lv, and W. Liu, "Efficient Response Time Bound for Typed DAG Tasks," in *RTCSA*, 2023, pp. 226–231.

[32] J. M. Jaffe, "Bounds on the Scheduling of Typed Task Systems," *SIAM J. Comput.*, vol. 9, no. 3, pp. 541–551, 1980.

[33] X. Jiang, N. Guan, H. Liang, Y. Tang, L. Qiao, and W. Yi, "Virtually-Federated Scheduling of Parallel Real-Time Tasks," in *RTSS*, 2021, pp. 482–494.

[34] X. Jiang, N. Guan, X. Long, and W. Yi, "Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors," in *RTSS*, 2017, pp. 80–91.

[35] S. Kato and Y. Ishikawa, "Gang EDF Scheduling of Parallel Task Systems," in *RTSS*, 2009, pp. 459–468.

[36] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: enabling autonomous vehicles with embedded systems," in *ICCPS*, 2018, pp. 287–296.

[37] S. Lee, N. Guan, and J. Lee, "Design and Timing Guarantee for Non-Preemptive Gang Scheduling," in *RTSS*, 2022, pp. 132–144.

[38] S. Lee, S. Lee, and J. Lee, "Response Time Analysis for Real-Time Global Gang Scheduling," in *RTSS*, 2022, pp. 92–104.

[39] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global EDF scheduling for parallel real-time tasks," *Real-Time Systems*, vol. 51, no. 4, pp. 395–439, 2015.

[40] C. Lin, J. Shi, N. Ueter, M. Günzel, J. Reineke, and J. Chen, "Type-Aware Federated Scheduling for Typed DAG Tasks on Heterogeneous Multicore Platforms," *IEEE Trans. Computers*, vol. 72, no. 5, pp. 1286–1300, 2023.

[41] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional DAG tasks in multiprocessor systems," in *ECRTS*, 2015, pp. 211–221.

[42] G. Nelissen, J. M. i Igual, and M. Nasri, "Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks," in *ECRTS*, 2022, pp. 12:1–12:22.

[43] P. Patel, I. Baek, H. Kim, and R. Rajkumar, "Analytical Enhancements and Practical Insights for MPCP with Self-Suspensions," in *RTAS*, R. Pellizzoni, Ed., 2018, pp. 177–189.

[44] R. Pathan, P. Voudouris, and P. Stenström, "Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 915–928, 2018.

[45] P. Purdom, "A Transitive Closure Algorithm," *BIT*, vol. 10, pp. 76–94, 1970.

[46] P. Richard, J. Goossens, and S. Kato, "Comments on "Gang EDF Schedulability Analysis"," *CoRR*, vol. abs/1705.05798, 2017.

[47] V. Rispo, F. Aromolo, D. Casini, and A. Biondi, "Response-Time Analysis of Bundled Gang Tasks Under Partitioned FP Scheduling," *IEEE Transactions on Computers*, vol. 73, no. 11, pp. 2534–2547, 2024.

[48] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel Real-Time Scheduling of DAGs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3242–3252, 2014.

[49] M. A. Serrano and E. Quiñones, "Response-time analysis of DAG tasks supporting heterogeneous computing," in *DAC*, 2018, pp. 125:1–125:6.

[50] B. Sun, T. Kloda, and M. Caccamo, "Strict Partitioning for Sporadic Rigid Gang Tasks," in *RTAS*. IEEE, 2024, pp. 252–264.

[51] N. Ueter, M. Günzel, G. von der Brüggen, and J. Chen, "Hard Real-Time Stationary GANG-Scheduling," in *ECRTS*, 2021, pp. 10:1–10:19.

[52] ——, "Parallel Path Progression DAG Scheduling," *IEEE Trans. Computers*, vol. 72, no. 10, pp. 3002–3016, 2023.

[53] Y. Wang, C. Liu, D. Wong, and H. Kim, "GCAPS: GPU Context-Aware Preemptive Priority-Based Scheduling for Real-Time Tasks," in *ECRTS*, 2024, pp. 14:1–14:25.

[54] S. Wasly and R. Pellizzoni, "Bundled Scheduling of Parallel Real-Time Tasks," in *RTAS*, 2019, pp. 130–142.

[55] K. Yang, M. Yang, and J. H. Anderson, "Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms," in *RTNS*, 2016, pp. 349–358.

[56] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, "Making OpenVX Really "Real Time"," in *RTSS*, 2018, pp. 80–93.