Efficient Deterministic Federated Scheduling for Parallel Real-Time Tasks

Son Dinh, Christopher Gill, and Kunal Agrawal
Washington University in St. Louis
{sonndinh, cdgill, kunal}@wustl.edu

***Abstract*—Federated scheduling is a generalization of partitioned scheduling for parallel tasks on multiprocessors, and has been shown to be a competitive scheduling approach. However, federated scheduling may waste resources due to its dedicated allocation of processors to parallel tasks. In this work we introduce a novel algorithm for scheduling parallel tasks that require more than one processor to meet their deadlines (i.e., heavy tasks). The proposed algorithm computes a deterministic schedule for each heavy task based on its internal graph structure. It efficiently exploits the processors allocated to each task and thus reduces the number of processors required by the task. Experimental evaluation shows that our new federated scheduling algorithm significantly outperforms other state-of-the-art federated-based scheduling approaches, including semi-federated scheduling and reservation-based federated scheduling, that were developed to tackle resource waste in federated scheduling, and a stretching algorithm that also uses the tasks' graph structures.**

I. Introduction

The development of multicore processors enables applications with high computational demand to be deployed in modern real-time systems. Applications such as motion planning in autonomous vehicles [1], real-time hybrid structural simulation [2], and computer vision [3] require multiple processors simultaneously to meet their deadlines. In contrast to sequential tasks which only allow *inter-task parallelism*, such parallel applications also allow *intra-task parallelism* — a task can execute on more than one processor at the same time. With the prevalence of multicore processors and the available parallel programming languages and concurrency platforms such as OpenMP and Cilk Plus, ever more parallel applications are being deployed in modern real-time systems.

In addition to global and partitioned scheduling [4]–[11], federated scheduling is a promising approach for scheduling parallel tasks. Federated scheduling, originally proposed by Li et al. [12], classifies parallel tasks as *heavy tasks*, which must execute on multiple processors to meet their deadlines, and *light tasks*, which can execute sequentially and still meet their deadlines. Each heavy task is allocated a dedicated set of processors and scheduled exclusively on its processors. Light tasks are then scheduled sequentially on the remaining processors. Federated scheduling has been shown to be a promising approach for scheduling parallel tasks [12]–[15] due to its analytical properties and ease of implementation in practice [16]. Federated scheduling, however, may suffer from resource waste, as processors that are dedicated to a heavy task cannot be shared with other tasks, even if not fully exploited.

Recent work has attempted to address this resource waste problem, by either potentially reducing the number of processors allocated to heavy tasks [13] or increasing the ability for heavy tasks to share their processors with other tasks [17], [18]. In this work, we address the resource waste problem by reducing the number of processors exclusively allocated to heavy tasks. At runtime, each heavy task is executed based on a deterministic schedule computed offline by our algorithm. Light tasks are treated and scheduled as sequential tasks on the remaining processors as in [12]–[15].

The contributions of this work are as follows.

- We propose a novel algorithm to compute a deterministic schedule for each heavy task by using its internal graph structure along with basic parameters such as its deadline and its subtasks' execution times. The proposed algorithm efficiently exploits the processors allocated to each heavy task, hence reducing the number of processors required to schedule the task. We also present a new federated scheduling algorithm based on the proposed deterministic scheduling algorithm.
- We conduct an extensive evaluation of the performance of the proposed algorithm with randomly generated tasks. Experiment results show that the proposed algorithm significantly reduces the numbers of processors required by heavy tasks, and that our new federated scheduling algorithm outperforms the state-of-the-art federated-based scheduling approaches [13], [17], [18] and a stretching algorithm [19] by a large margin.

This paper is organized as follows. Sections II and III present related work and the considered task model respectively. Section IV presents our new deterministic scheduling algorithm for heavy tasks, and discusses its theoretical properties and how the computed schedules can be used at runtime. Sections V and VI present a performance evaluation of our deterministic and federated scheduling algorithms, and compare them with other state-of-the-art techniques. Finally, Section VII concludes our work.

II. Related Work

There are three common approaches for scheduling parallel tasks: global scheduling [4], [5], [8], [9], [12], [20], partitioned scheduling [10], [11], and federated-based scheduling [12]–[14], [17], [18]. Due to space limitation, we focus

978-1-7281-4403-0/20/$31.00 ©2020 IEEE

on federated-based scheduling in this section. In federated scheduling [12], parallel tasks are classified as heavy tasks (tasks with densities > 1.0) and light tasks (tasks with densities ≤ 1.0). Each heavy task is allocated a large enough number of dedicated processors so that worst-case releases of the task can meet their deadlines. Light tasks are then scheduled as sequential tasks on the remaining processors using an existing multiprocessor scheduling algorithm.

Federated scheduling, however, may over-provision heavy tasks, and thus may waste computational resources. To address this problem, Jiang et al. [17] and Ueter et al. [18] have proposed semi-federated scheduling and reservation-based federated scheduling, respectively. In semi-federated scheduling, a heavy task with a processing requirement of $x + \epsilon$, where $x \in \mathbb{Z}^+$ and $0 < \epsilon \in \mathbb{R} < 1$, is allocated x dedicated processors (instead of $x + 1$ as it would have been in [12]). The fractional part ϵ is scheduled together with light tasks. In reservation-based scheduling, parallel tasks are allocated dedicated reservation servers instead of dedicated processors. Reservation servers are then scheduled as sequential tasks using existing multiprocessor scheduling algorithms. We discuss these two approaches in detail in Section VI.

Baruah proposed a different method to reduce the number of processors allocated to heavy tasks [13], [14]. That approach allocates each heavy task a minimal number of processors on which Graham's list scheduling algorithm [21] can schedule the task successfully. We discuss this method in detail and compare our algorithm with it in Sections V and VI.

III. Task Model & Notation

We consider a task set of n real-time, sporadic tasks scheduled upon m identical processors. Each task τ_i is modeled using a tuple (G_i, D_i, T_i), where D_i and T_i are the relative deadline and minimum inter-arrival time (i.e., period) of τ_i, respectively. Task τ_i is represented by a *Directed Acyclic Graph* (DAG) $G_i = (V_i, E_i)$ in which $V_i \triangleq \{v_{i,1}, v_{i,2}, ..., v_{i,n_i}\}$ is the set of vertices and $E_i \subset (V_i \times V_i)$ is the set of directed edges of τ_i. Vertices are also called subtasks or nodes. Each vertex denotes a sequential execution unit of the task and each edge $(v_{i,p}, v_{i,q}) \in E_i$ denotes the precedence constraint between vertices $v_{i,p}$ and $v_{i,q}$ — $v_{i,p}$ must finish before $v_{i,q}$ may start its execution. A vertex with no incoming edges is called a source vertex, and a vertex with no outgoing edges is called a sink vertex. A sequence of vertices $(v_{i,k_j}, v_{i,k_{j+1}}, ..., v_{i,k_t})$, where v_{i,k_t} is a sink vertex and $(v_{i,k_p}, v_{i,k_{p+1}}) \in E_i$, $\forall j \leq p < t$, is a *path* starting from v_{i,k_j} of τ_i. The length of a path is the sum of the worst-case execution times (WCETs) of all subtasks along the path.

Task τ_i may release an infinite number of jobs and any two consecutive jobs of τ_i must be released at least T_i time units apart. We consider *constrained-deadline* tasks, i.e., $D_i \leq T_i$ for all τ_i. WCET of subtask $v_{i,j}$ is denoted by $C_{i,j}$. WCET of τ_i, denoted by C_i, is $C_i = \sum_{v_{i,j} \in V_i} C_{i,j}$. The WCET C_i is also called the *work* of τ_i. A path with the greatest length of τ_i is called a *critical-path* of τ_i. The *critical-path length* (or *span*) of τ_i is denoted by L_i. Figure 1 illustrates a task τ_i with

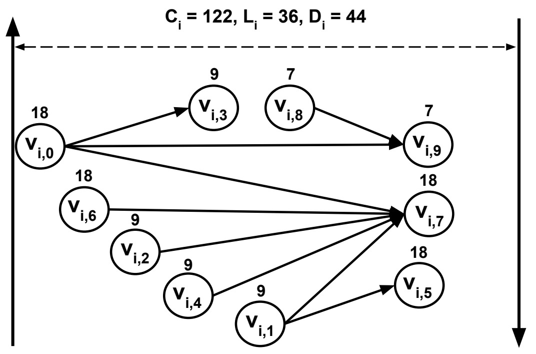


Fig. 1: An example DAG task.

10 vertices, $C_i = 122$, $L_i = 36$, and $D_i = 44$. The utilization and density of τ_i are denoted by $u_i = \frac{C_i}{T_i}$ and $\sigma_i = \frac{C_i}{D_i}$, respectively. τ_i is a heavy task if $\sigma_i > 1.0$ and a light task if $\sigma_i \leq 1.0$. We use n^h and n^l to denote the number of heavy and light tasks in a task set respectively. The total normalized utilization of a task set is denoted by $U = \frac{\sum_{\tau_i \in \tau} u_i}{m}$. The total normalized utilization of heavy and light tasks in a task set are denoted by U^h and U^l, respectively.

We use $dag(v_{i,j})$ to denote the subgraph of τ_i's DAG rooted at $v_{i,j}$, i.e., $dag(v_{i,j})$ contains $v_{i,j}$ and all of its descendants. Let $work(v_{i,j})$ denote the total work of $dag(v_{i,j})$: $work(v_{i,j}) = \sum_{v_{i,p} \in dag(v_{i,j})} C_{i,p}$. In Figure 1, $dag(v_{i,0})$ consists of $v_{i,0}$, $v_{i,3}$, $v_{i,7}$, $v_{i,9}$, and $work(v_{i,0}) = 52$. We use $\lambda_{i,j}$ to denote a longest path from $v_{i,j}$, and $len(\lambda_{i,j})$ to denote the length of $\lambda_{i,j}$. In other words, $\lambda_{i,j}$ is a critical-path of $dag(v_{i,j})$, and $len(\lambda_{i,j})$ is its critical-path length.

A subtask may be scheduled as multiple portions, i.e., it can be preempted and resumed multiple times. We call each such portion of a subtask a *fragment* of that subtask. The kth fragment of subtask $v_{i,j}$ is denoted as $v^k_{i,j}$. Fragment $v^k_{i,j}$ has execution time $C^k_{i,j}$. If $v_{i,j}$ is scheduled as $nfrags(v_{i,j})$ fragments, we have $\sum_{k=1}^{nfrags(v_{i,j})} C^k_{i,j} = C_{i,j}$. We extend the notation of subgraph, subgraph work, and critical-path of subgraph for fragments. In particular, we use $dag(v^k_{i,j})$ to denote the subgraph rooted at $v^k_{i,j}$. For instance, suppose $v_{i,0}$ in Figure 1 has executed 9 time units in its first fragment $v^0_{i,0}$ before it is preempted. Later, its second fragment $v^1_{i,0}$ is resumed. The subgraph $dag(v^1_{i,0})$ thus contains $v^1_{i,0}$ and $v_{i,3}$, $v_{i,7}$, $v_{i,9}$. We use $work(v^k_{i,j})$ to denote the total work of $dag(v^k_{i,j})$. In this example, $work(v^1_{i,0}) = 43$. A longest path starting from $v^k_{i,j}$ is denoted by $\lambda^k_{i,j}$ and its length is $len(\lambda^k_{i,j})$.

IV. A Deterministic Scheduling Algorithm

Much of the previous work on federated scheduling ignores the tasks' internal graph structures [12], [15], [18]. For instance, in [12] the number of processors allocated to τ_i is $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ and is independent of other details of τ_i's DAG. The advantage of this method is that it is simple and allows each task's DAG to vary in different releases as long as it satisfies the work and critical-path length bounds. However, this also means that this method must assume the worst-case DAG for every task when allocating processors to it. Hence, it risks over-provisioning the task, which leads to resource waste.

In [13], [14], the authors take the tasks' DAGs into account to some extent. In particular, they use Graham's list scheduling [21] to schedule a task on a given number of processors and only increase the number of processors allocated to the task one at a time if the schedule's makespan (i.e., the elapsed time from the start to the end of the schedule) is greater than the task's deadline. In other words, they indirectly take τ_i's DAG into consideration through the use of list scheduling to test whether τ_i can be scheduled successfully in each step.

Graham's list scheduling, however, was not originally designed for real-time systems. Instead, its objective is to schedule a task so that its completion time is minimized; there is no notion of deadline in list scheduling. In contrast, a real-time task's temporal correctness is satisfied as long as its jobs finish by their deadlines. In this section, we propose a new scheduling algorithm for heavy tasks which takes into account each task's DAG, deadline, and subtasks' WCETs to compute a deterministic schedule for the task. The goal of this algorithm is to exploit the processors allocated to heavy task as efficiently as possible, hence reducing the number of processors required.

A. DAG Scheduling Algorithm

Algorithm 1 shows the pseudocode for the proposed algorithm. In contrast to Graham's list scheduling, where subtasks are scheduled non-preemptively — once scheduled a subtask is not preempted until it completes — Algorithm 1 allows subtasks to be preempted and resumed at appropriate times. The computed schedule for τ_i consists of a chain of segments, each comprising fragments from different subtasks of τ_i executing in parallel (Figure 2). The algorithm starts by pre-processing τ_i's DAG (line 2). For each subtask $v_{i,j}$ it computes two parameters: (i) the length of a longest path originating from $v_{i,j}$, i.e., $len(\lambda_{i,j})$, and (ii) the work of the subgraph rooted at $v_{i,j}$, i.e., *work*$(v_{i,j})$. Algorithm 1 proceeds by allocating τ_i a minimal number of dedicated processors: $m_i = \lceil \frac{C_i}{D_i} \rceil$ (line 3). This is the smallest number of processors that can possibly schedule τ_i. The algorithm then iteratively increases the number of processors m_i allocated to τ_i, only when necessary (line 4). In each iteration, it computes a schedule for τ_i with the given m_i. If the computed schedule does not satisfy τ_i's deadline, it increases m_i by 1 and re-computes a schedule for τ_i. The algorithm terminates when a satisfying schedule for τ_i is obtained, or if there are not enough processors to schedule τ_i.

For a given m_i, the algorithm maintains a queue *readyQ* of ready fragments (line 12). At the beginning, fragments corresponding to the source subtasks are inserted into the queue. These fragments have execution times equal to the WCETs of the corresponding subtasks. For the DAG in Figure 1, fragments $v^0_{i,0}, v^0_{i,1}, v^0_{i,2}, v^0_{i,4}, v^0_{i,6}, v^0_{i,8}$ are inserted into *readyQ*; each has execution time equal to its subtask's WCET. The algorithm then incrementally constructs a schedule for τ_i (lines 13 - 48). In each iteration, it determines which fragments will be scheduled and for how long; all chosen fragments will

Algorithm 1 Scheduling Algorithm for Parallel DAG Tasks

```
1: procedure DAGSCHED($\tau_i$, $m$)
2:     Pre-process $\tau_i$'s DAG
3:     $m_i \leftarrow \lceil \frac{C_i}{D_i} \rceil$                    ▷ Assuming $m_i \le m$
4:     while SCHEDULECORE($\tau_i$, $m_i$) = False do
5:         if $m_i > m$ then Return False
6:         end if
7:     end while
8:     Return True and the schedule for $\tau_i$
9: end procedure
10: procedure SCHEDULECORE($\tau_i$, $m_i$)
11:     $currTime \leftarrow 0$
12:     $readyQ \leftarrow$ {fragments of the source subtasks}
13:     while readyQ $\neq \emptyset$ do
14:         $remainWork \leftarrow$ Total un-scheduled work at $currTime$
15:         $minCores \leftarrow \lceil \frac{remainWork}{D_i - currTime} \rceil$
16:         if $minCores > m_i$ then
17:             $m_i \leftarrow m_i + 1$
18:             Return False
19:         end if
20:         $\mathcal{S} \leftarrow \{v^k_{i,j} | v^k_{i,j} \in readyQ \wedge len(\lambda^k_{i,j}) = D_i - currTime\}$
21:         if $|\mathcal{S}| > m_i$ then
22:             $m_i \leftarrow m_i + 1$
23:             Return False
24:         end if
25:         $listFrags \leftarrow \mathcal{S}$    ▷ Fragments scheduled in the current iteration
26:         $readyQ \leftarrow readyQ \setminus \mathcal{S}$
27:         while $readyQ \neq \emptyset \wedge |listFrags| < m_i$ do
28:             Find $v^k_{i,j} \in readyQ$ with greatest $work(v^k_{i,j})$
29:             $listFrags \leftarrow listFrags \cup \{v^k_{i,j}\}$
30:             $readyQ \leftarrow readyQ \setminus \{v^k_{i,j}\}$
31:         end while
32:         $execTime \leftarrow \min_{v^k_{i,j} \in listFrags} C^k_{i,j}$
33:         Find $v^p_{u,w}$ s.t. $(v^p_{u,w} \in readyQ) \wedge (len(\lambda^p_{u,w}) \ge len(\lambda^k_{i,j}), \forall v^k_{i,j} \in readyQ)$
34:         if $v^p_{u,w} \neq \varnothing$ then
35:             $execTime \leftarrow \min\{execTime, D_i - currTime - len(\lambda^p_{u,w})\}$
36:         end if
37:         Find $v^q_{a,b}$ s.t. $(v^q_{a,b} \in readyQ) \wedge (work(v^q_{a,b}) \ge work(v^k_{i,j}), \forall v^k_{i,j} \in readyQ)$
38:         Find $v^r_{x,y}$ s.t. $(v^r_{x,y} \in listFrags) \wedge (v^r_{x,y} \notin \mathcal{S}) \wedge (work(v^r_{x,y}) \le work(v^k_{i,j}), \forall v^k_{i,j} \in listFrags \wedge v^k_{i,j} \notin \mathcal{S})$
39:         if $v^q_{a,b} \neq \varnothing \wedge v^r_{x,y} \neq \varnothing$ then
40:             $execTime \leftarrow \min\{execTime, work(v^r_{x,y}) - work(v^q_{a,b}) + 1\}$
41:         end if
42:         Split all $v^k_{i,j} \in listFrags$ that has $C^k_{i,j} > execTime$
43:         $\psi \leftarrow$ {remainder of the split fragments}
44:         $readyQ \leftarrow readyQ \cup \psi$
45:         $readyQ \leftarrow readyQ \cup$ {fragments of the newly enabled subtasks}
46:         Fragments in $listFrags$ are set up to run for $execTime$ time units
47:         $currTime \leftarrow currTime + execTime$
48:     end while
49:     Return True
50: end procedure
```

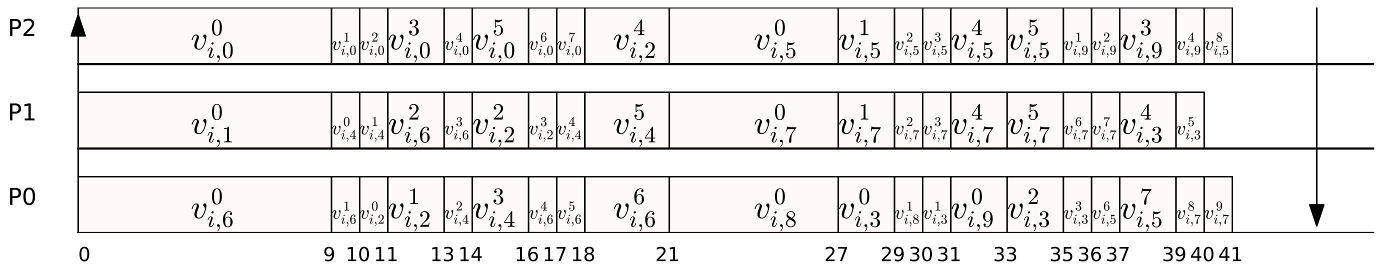


Fig. 2: The deterministic schedule computed by Algorithm 1 for the DAG in Figure 1.

be scheduled for the same duration. A variable *currTime* is used to denote the time up to that the schedule has been constructed (line 11).

Algorithm 1 allocates 1 more processor to τ_i if one of the following two cases happens. First, it computes an estimate for the minimum number of processors required to schedule the remaining work of τ_i: $\left\lceil \frac{remainWork}{D_i - currTime} \right\rceil$, where *remainWork* is the remaining work of τ_i that has not yet been scheduled at time $currTime$, and $D_i - currTime$ is the amount of time the algorithm has to schedule that remaining work (line 15). If this value is greater than the current m_i, i.e., it is impossible to schedule the remaining work using m_i processors, it increases m_i by 1 and re-computes the schedule for τ_i using the updated m_i (lines 16 - 19). Second, at the beginning of each iteration, the algorithm computes a set $\mathcal{S}$ of ready fragments for which the lengths of their longest paths are equal to the time remaining until τ_i's deadline: $\mathcal{S} \triangleq \{v_{i,j}^k \in readyQ | len(\lambda_{i,j}^k) = D_i - currTime\}$ (line 20). Each such fragment must execute immediately, for otherwise τ_i will miss its deadline since the path corresponding to a fragment that is not scheduled immediately will run past the deadline. If there are more fragments in $\mathcal{S}$ than m_i, the algorithm increases m_i by 1, since it is impossible to schedule all longest paths of the fragments in $\mathcal{S}$ to meet the deadline with the current m_i (lines 21 - 24).

If neither case happens, m_i is unchanged and the algorithm determines at most m_i ready fragments to schedule. Since all fragments in $\mathcal{S}$ must be scheduled immediately, the algorithm picks all of them to schedule in this iteration (line 25). We call the list of fragments chosen to be scheduled $listFrags$. For $m_i - |\mathcal{S}|$ processors left, it takes at most $m_i - |\mathcal{S}|$ ready fragments not in $\mathcal{S}$ and with the greatest subgraph work to add to $listFrags$ (lines 27 - 31). The intuition is that we want to prioritize the fragments with the greatest amounts of pending work, so that as much work will be enabled for the next iterations as possible; this gives a better chance of scheduling more work in parallel in the future.

The algorithm then computes the execution duration, denoted by *execTime*, for those fragments. If *execTime* is less than a chosen fragment $v_{i,j}^k$'s execution time $C_{i,j}^k$, the fragment is split into two smaller fragments. The first fragment, which is scheduled in this iteration, has execution time $C_{i,j}^k = execTime$. The second fragment, $v_{i,j}^{k+1}$, has execution time equal to the remaining WCET of $v_{i,j}$; it is then inserted back into *readyQ* for later iterations. The value of *execTime* is first determined by taking the minimum execution time among all chosen fragments (line 32). Two other factors are also considered when computing *execTime*. The intuition for them is that after the chosen fragments have been scheduled for some time, some ready, unchosen fragments may have greater subgraph work than the ones being scheduled. Similarly, some of them may have longest paths that need to execute immediately because their lengths become equal to the time remaining until the task's deadline. When that happens, the algorithm needs to choose a new set of fragments to schedule.

First, it picks from the unchosen ready fragments a fragment $v_{u,w}^p$ with the greatest longest-path length $len(\lambda_{u,w}^p)$ (line 33). The chosen fragments thus can execute for at most $D_i - currTime - len(\lambda_{u,w}^p)$ units before $v_{u,w}^p$ must be scheduled (line 35). If there is no such $v_{u,w}^p$, the algorithm ignores this factor. Second, the algorithm considers an unchosen fragment $v_{a,b}^q$ with the greatest subgraph work (line 37), and a chosen fragment $v_{x,y}^r$ with smallest subgraph work among all chosen fragments such that $v_{x,y}^r$ is not in $\mathcal{S}$ (line 38). The difference between *work*$(v_{x,y}^r)$ and *work*$(v_{a,b}^q)$ gives us the second factor (line 40). The reason is that after that amount of time, $v_{a,b}^q$ has its subgraph work equal to $v_{x,y}^r$'s and we should decide whether $v_{a,b}^q$ should be scheduled next. Specifically, the second factor is: *work*$(v_{x,y}^r)$ - *work*$(v_{a,b}^q)$ + 1. One time unit is added to break ties in case there are multiple fragments with the same subgraph work. If either $v_{a,b}^q$ or $v_{x,y}^r$ does not exist, we ignore this factor.

Now, *execTime* is computed by taking the minimum of the execution times of the chosen fragments and the values of the two factors. Each fragment with execution time greater than *execTime* is split; the remaining fragment after splitting is inserted back into *readyQ* as described above (lines 42 - 44). For fragments that complete, their corresponding subtasks are also finished, and the algorithm inserts the fragments corresponding to the enabled subtasks (those have become ready) into *readyQ* (line 45). The chosen fragments are then scheduled onto m_i processors for $execTime$ units (line 46).

Example for Algorithm 1. Figure 2 shows the schedule computed by Algorithm 1 for the DAG in Figure 1. The algorithm starts with $m_i = \lceil \frac{C_i}{D_i} \rceil = 3$ processors. At time 0, fragments $v_{i,0}^0, v_{i,1}^0, v_{i,6}^0$ are chosen to be scheduled since they have greatest subgraph work. They are then scheduled for $execTime = 9$ time units. At time 9, $v_{i,1}$ has completed,

$v_{i,5}$ is enabled and inserted into the ready queue as a fragment $v_{i,5}^0$ with $C_{i,5}^0 = C_{i,5} = 18$. The remainders of $v_{i,0}$ and $v_{i,6}$ are inserted into the ready queue as fragments $v_{i,0}^1$ with $C_{i,0}^1 = 9$, and $v_{i,6}^1$ with $C_{i,6}^1 = 9$. The algorithm keeps running until the ready queue is empty. For this task, Algorithm 1 only needs 3 processors, compared to $\lceil \frac{C_i - L_i}{D_i - L_i} \rceil = 11$ processors required by the algorithm in [12]. The proposed algorithm thus saves 8 processors for this task.

B. Correctness & Optimization

We now show that Algorithm 1 computes a valid schedule for τ_i in the sense that any schedule returned by the algorithm satisfies τ_i's deadline.

Lemma 1: For every iteration of the while-loop at line 13 of Algorithm 1, all ready fragments have their longest-path lengths less than or equal to $D_i - currTime$.

Proof: We prove by induction. For the first iteration, $currTime = 0$. Since $L_i \leq D_i$, there is no ready fragment with longest-path length greater than $D_i - currTime = D_i$.

Suppose that at iteration k, all ready fragments have their longest-path lengths less than or equal to $D_i - currTime$. We now prove that the lemma also holds for iteration $(k+1)$. Let $currTime_k$ and $currTime_{k+1}$ denote the values of $currTime$ at the beginning of iterations k and $(k+1)$, respectively. Let $execTime_k$ denote the execution length computed for the iteration k. At the beginning of iteration $(k+1)$, each fragment $v_{i,a}^p$ that was scheduled at iteration k either has completed, or its remaining $v_{i,a}^{p+1}$ has been inserted into the ready queue. For the latter case, $len(\lambda_{i,a}^{p+1}) = len(\lambda_{i,a}^p) - execTime_k \leq D_i - currTime_k - execTime_k = D_i - currTime_{k+1}$, since $len(\lambda_{i,a}^p) \leq D_i - currTime_k$. In the former case, for any subtask $v_{i,b}$ enabled by the completion of $v_{i,a}^p$, its first fragment $v_{i,b}^0$ has $len(\lambda_{i,b}^0) \leq len(\lambda_{i,a}^p) - execTime_k \leq D_i - currTime_k - execTime_k = D_i - currTime_{k+1}$. Hence, the lemma holds in both cases.

Let $v_{i,x}^q$ denote a fragment with greatest longest-path length among all ready fragments that were not chosen to be scheduled at iteration k. If there is no such fragment, then all ready fragments in iteration k were scheduled and the lemma holds as discussed above. Otherwise, we have $execTime_k \leq D_i - currTime_k - len(\lambda_{i,x}^q) \Leftrightarrow len(\lambda_{i,x}^q) \leq D_i - currTime_k - execTime_k \Leftrightarrow len(\lambda_{i,x}^q) \leq D_i - currTime_{k+1}$. Since every other fragment $v_{i,y}^r$ that was not scheduled at iteration k has $len(\lambda_{i,y}^r) \leq len(\lambda_{i,x}^q)$, the lemma holds for all ready fragments that were not scheduled at iteration k. ■

We now can prove that if Algorithm 1 returns a schedule for τ_i, then it is a valid schedule.

Theorem 2: If Algorithm 1 returns a schedule for τ_i, then this schedule satisfies τ_i's deadline.

Proof: Suppose the algorithm returns a schedule that misses τ_i's deadline. Then in the last iteration of the while-loop at line 13 before the algorithm terminates, there must be at least a ready fragment $v_{i,j}^k$ with $len(\lambda_{i,j}^k) > D_i - currTime$. This contradicts Lemma 1. ■

Theorem 2 in [12] proves that an implicit-deadline DAG task τ_i is schedulable on $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ processors using any work-conserving scheduler. This result also applies to constrained-deadline parallel tasks, as stated in the following lemma.

Lemma 3: A constrained-deadline parallel task allocated $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ dedicated processors is guaranteed to meet its deadline when scheduled using any work-conserving scheduler.

Proof: The proof is similar to the proof of Theorem 2 in [12]. We note that a parallel task with work C_i and critical-path length L_i will complete within D_i time units when scheduled by a work-conserving scheduler exclusively on $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ processors. Since the task is constrained-deadline ($D_i \leq T_i$), every job of the task finishes before the next job is released. Thus, all jobs of the task meet their deadlines. ■

Note that Algorithm 1 is a work-conserving algorithm since it does not leave any processor idle if there are some fragments ready to be executed. We now can bound the number of processors required by Algorithm 1 in the following theorem.

Theorem 4: The number of processors required by Algorithm 1 for task τ_i is at most $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$.

Proof: Suppose that Algorithm 1 returns a schedule for τ_i that requires $m_i^* > \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ processors. The algorithm starts with $m_i = \lceil \frac{C_i}{D_i} \rceil < \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ processors. In each subsequent call at line 4, it increases m_i by 1. Thus at some point, m_i is set to $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$. Since Algorithm 1 is work-conserving, it would have scheduled τ_i successfully using that number of processors (as proved in Lemma 3) and the returned m_i would have been $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil < m_i^*$. This contradicts the hypothesis. ■

In Sections V and VI, we show that Algorithm 1 performs much better than other federated-based scheduling algorithms, in terms of the number of processors allocated to each heavy task and the acceptance ratios of the generated task sets.

Time Complexity. In the pre-processing step, a variant of shortest paths algorithm presented in [22] (Section 24.2) can be used to compute the longest-path length of a subtask. That algorithm takes time $O(|V_i| + |E_i|)$. We can use breadth-first search (or depth-first search) to compute the subgraph work for each subtask with complexity of $O(|V_i| + |E_i|)$. Overall, the pre-processing step takes time $O(|V_i|(|V_i| + |E_i|))$. The while-loops at lines 4 and 13 run $O(m)$ and $O(D_i)$ iterations, respectively. We can implement $readyQ$ using a priority-queue with subgraph work values of ready fragments as keys. Computing the set $\mathcal{S}$ (line 20) takes $O(|V_i|log|V_i|)$ time by iteratively examining the fragments in $readyQ$. The while-loop at lines 27 - 31 takes $O(mlog|V_i|)$ time. Line 33 takes $O(|V_i|)$ time by simply examining the ready but unscheduled fragments, and computing the maximum longest-path length for them. Lines 42 - 45 take $O(|V_i|log|V_i|)$ time since there are at most $|V_i|$ ready fragments inserted into $readyQ$. Overall, Algorithm 1 takes $O(D_i mlog|V_i|(|V_i|+m) + |V_i|^2 + |V_i||E_i|)$ time, i..e, pseudo-polynomial time in the deadline of τ_i.

Optimizing and Using the Computed Schedule. A schedule computed by Algorithm 1 may contain multiple consecutive fragments of a subtask on the same processor. For instance, in Figure 2, all fragments of $v_{i,0}$ are scheduled consecutively on processor P2. Such consecutive fragments can be merged

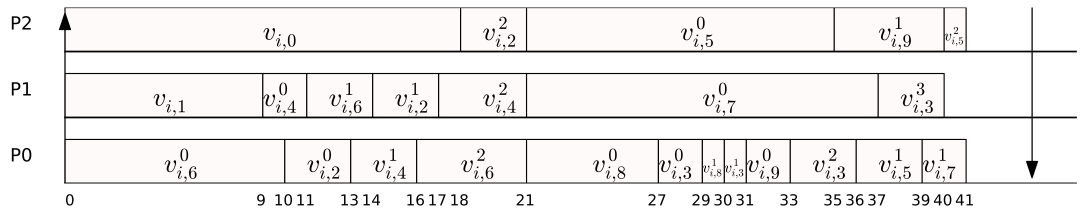


Fig. 3: The final schedule after fragments in Figure 2 are merged.

into a larger fragment or even into the original subtask, as in the case of $v_{i,0}$. Figure 3 shows the schedule computed by Algorithm 1 after merging. The final schedule computed offline for each task can then be stored in a lookup table, which can be referred to by the system at run time.

Even though the schedule computed for each task relies on the WCETs of its subtasks, in practice, a subtask may execute for less than its WCET. In this case, the runtime system can let the processors that host the remainder of the subtask busy waiting. (Note that the remainder of the subtask may comprise multiple fragments that are scheduled on different processors.) By doing so, we avoid the timing anomalies that are introduced by executing subtasks for less than their WCETs [21]. We note that in a practical system, such busy waiting intervals can be used more efficiently to co-schedule non-real-time workloads.

C. A Federated Scheduling Algorithm

Algorithm 1 can be incorporated into a federated scheduling algorithm. The pseudocode for our federated scheduling algorithm is shown in Algorithm 2. In this algorithm, heavy tasks are scheduled by Algorithm 1 on their dedicated processors, and light tasks are scheduled sequentially on the remaining processors using any existing multiprocessor scheduling algorithm. The algorithm returns failure if Algorithm 1 fails to schedule a heavy task (line 4) or if the light tasks cannot be scheduled on the remaining processors (line 9). Otherwise, the task set is scheduled successfully.

Algorithm 2 Federated Scheduling for DAG Tasks

```
 1: procedure FEDSCHED($\tau$, $m$)
 2:     $m_r \leftarrow m$
 3:     for Each heavy task $\tau_i$ do
 4:         if DAGSCHED($\tau_i$, $m_r$) = False then    ▷ Algo. 1
 5:             Return Failure
 6:         end if
 7:         $m_r \leftarrow m_r - m_i$    ▷ $\tau_i$ is allocated $m_i$ processors
 8:     end for
 9:     if Scheduling light tasks on $m_r$ processors fails then
10:         Return Failure
11:     end if
12:     Return Success
13: end procedure
```

V. Evaluation with Heavy Tasks

In this section, we compare the performance of Algorithm 1 (denoted by PRO) with the algorithms introduced by Baruah [13], [14] (denoted by BAR) and Li et al. [12] (denoted by LI). For a given DAG task, the schedule returned by BAR (and thus the number of processors required) depends on a specific order of subtasks in list scheduling. For the DAG task in Figure 1, BAR requires at least 4 processors to schedule the task, compared to 3 processors needed by Algorithm 1.

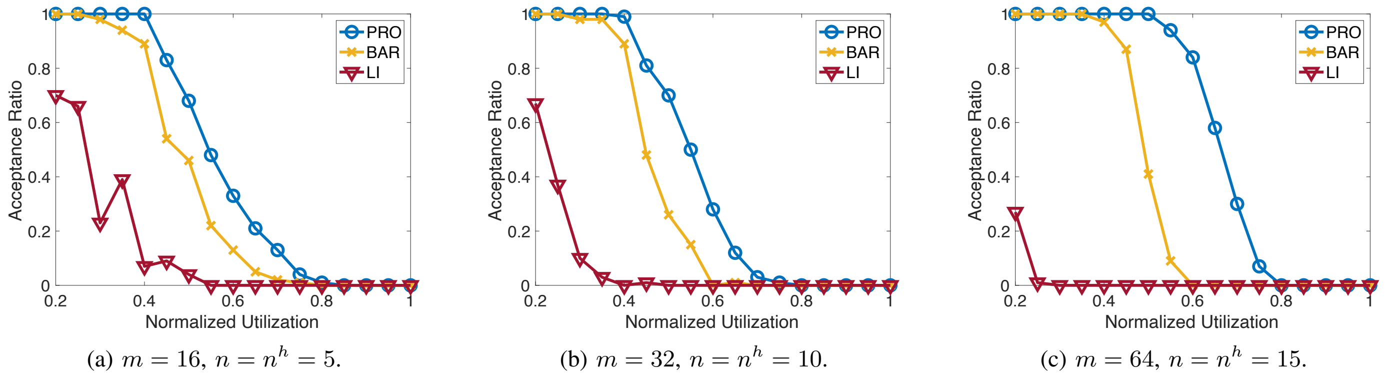
Experimental Setup. We used the Erdős-Rényi $G(n_i, p)$ method for generating DAGs [23]. In this method, n_i is the number of vertices and p is a probability threshold used to determine whether a directed edge between a pair of vertices is added. For each pair of vertices, a real number is drawn uniformly at random from $[0, 1]$ and if the number is less than p, an edge is added between the two vertices. If the obtained DAG is not connected, we add a minimal number of edges to make it weakly connected. For each task, the number of vertices was chosen uniformly at random in $[50, 250]$. The WCET of each vertex was drawn uniformly at random in $[50, 100]$. The relative deadline for τ_i was generated uniformly at random in $[L_i, C_i)$. Since we consider only heavy tasks in this section and tasks are constrained-deadline ($D_i \le T_i$), we do not need to generate periods — as long as each heavy task finishes by its deadline, the length of the period does not affect its schedulability. We varied the probability threshold p in $[0.1, 0.9]$ with a step of 0.1. For each setting, we generated 10,000 tasks and scheduled them using PRO and BAR algorithms. The numbers of processors required by PRO, BAR, and LI were recorded.

Results. Table I shows the percentages of the generated heavy tasks for which PRO needed fewer or more processors than BAR and LI. As we can see, PRO required fewer processors than BAR for $21\% - 48\%$ of the generated tasks. As p increases, the generated tasks become more sequential, and thus there is less room for the proposed algorithm to improve. There were still over 22% of the tasks for which at least 1 processor was saved when $p = 0.9$. Notably, there was no task for which PRO required more processors than BAR. PRO also significantly outperforms LI as $74\% - 92\%$ of the generated tasks needed fewer processors under PRO. Similar to the comparison with BAR, there was no task for which PRO required more processors than LI.

We also conducted experiments with task sets consisting of heavy tasks for systems with $m = \{16, 32, 64\}$ processors. For each value of m, we varied the normalized total utilization U in $[0.2, 1.0]$ with a step of 0.05. For $m = \{16, 32, 64\}$, each task set consisted of $n = \{5, 10, 15\}$ tasks, respectively. If U is too small to generate n heavy tasks, we generate smaller

TABLE I: Comparison to BAR and LI for Heavy DAG Tasks (Unit: %)

		Edge Probability Threshold p								
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Against BAR [13]	Fewer	43.55	42.81	40.6	44.47	48.28	39.67	28.35	21.73	22.14
	More	0	0	0	0	0	0	0	0	0
Against LI [12]	Fewer	83.93	82.03	80.25	76.91	75.21	74.45	76.26	92.04	84.84
	More	0	0	0	0	0	0	0	0	0



(a) $m = 16$, $n = n^h = 5$. (b) $m = 32$, $n = n^h = 10$. (c) $m = 64$, $n = n^h = 15$.

Fig. 4: Ratio of schedulable task sets for varying total utilization and number of processors.

numbers of tasks. For instance, when $m = 16$ and $U = 0.2$, we generated 2 tasks per task set. DAG tasks were generated using the Erdős-Rényi method as described above with p set to 0.2. We used the RandFixedSum algorithm [24] to generate individual utilizations for the tasks uniformly in $[1.1, U \times m]$. For each value of U, we generated 100 task sets and recorded the acceptance ratios of PRO, BAR, and LI. The results in Figure 4 show that PRO outperforms both BAR and LI. As m and n increase, the performance gap between PRO and BAR (and LI) increase. This is because as m and n increase, the chance that PRO can save processors for some tasks in each task set, and thus the chance that all tasks get sufficient processors, increases. Again there was no task for which PRO required more processors than BAR and LI.

VI. Evaluation Versus the State-of-the-Art

We now evaluate our federated scheduling algorithm against the state-of-the-art federated-based scheduling algorithms and a stretching algorithm for scheduling DAG task sets consisting of both heavy and light tasks.

A. Overview of the State-of-the-Art

In the semi-federated scheduling approach proposed by Jiang et al. [17], instead of allocating a heavy task τ_i with processing capacity requirement of $x + \epsilon = \frac{C_i - L_i}{D_i - L_i}$ (where $x = \lfloor \frac{C_i - L_i}{D_i - L_i} \rfloor$ and $0 \le \epsilon = \frac{C_i - L_i}{D_i - L_i} - x < 1$) $\lceil x + \epsilon \rceil$ processors as in [12], they allocate τ_i x processors. The remaining ϵ fraction is scheduled sequentially together with the light tasks on the remaining processors. The number of processors allocated to each heavy task is thus reduced by (at most) 1 compared to that of [12].

To realize this idea, a runtime dispatcher for each DAG determines when (and for how long) each subtask of the DAG is mapped to the computational fraction ϵ. They proposed two variants of the semi-federated approach which differ in the way the fraction ϵ is implemented. In the first variant, the fraction ϵ is encapsulated by a *container task* with a load equal to ϵ whereas in the second variant, it is encapsulated by two container tasks with the total load of ϵ. The container tasks are then scheduled sequentially together with the light tasks using P-EDF with Worst Fit packing [25].

In the reservation-based federated scheduling approach [18], instead of allocating each heavy task τ_i a set of dedicated processors, they allocate τ_i a set of dedicated *reservation servers* (each light task is assigned one server with a budget equal to its work). The obtained reservation servers are then scheduled using an existing multiprocessor scheduling algorithm for sequential tasks. Task τ_i is guaranteed to be schedulable if the total budget of its servers is $\ge C_i + L_i \cdot (m_i - 1)$, where m is the number of servers of τ_i (Equation 1 in [18]). They introduced two algorithms for assigning reservation servers to tasks, called R-MIN and R-EQUAL, which differ in the way they classify tasks as heavy or light. R-MIN classifies tasks in a similar fashion as [12], i.e., based on the tasks' densities, whereas R-EQUAL uses a common *stretch ratio* γ for all tasks to decide whether a task is heavy or not: τ_i is heavy if $C_i > \gamma L_i$. The authors also introduced an algorithm that dynamically adapts the number of servers assigned to a heavy task if one of its current servers fails to be partitioned.

In addition to those federated-based scheduling approaches, Qamhieh et al. [19] introduced an algorithm that take the tasks' graph structures into consideration. In particular, a critical path of each heavy task τ_i is stretched up to the task's deadline using work from subtasks not belonging to the critical path. As a result, τ_i is transformed into a master thread with an execution time equal to τ_i's deadline and a set of constrained-deadline threads. Each master thread is assigned a dedicated processor while all constrained-deadline threads and the light tasks are scheduled together on the remaining processors.

We choose to compare our algorithm with those algorithms since they have been shown to outperform other techniques, including the DAG decomposition algorithms [26], [27] and global scheduling [12], [20].

B. Experimental Evaluation

For all considered federated-based scheduling approaches, we used P-EDF for scheduling sequential tasks including light tasks, container tasks (in semi-federated scheduling), and reservation servers. We considered three packing heuristics for partitioning: Worst Fit (WF), Best Fit (BF), and First Fit (FF). For testing whether a sequential task can be assigned to a processor, two EDF schedulability tests were adopted. The first test, denoted by DEN, checks whether the sum of the task's load (or density) and the current total load of the processor is ≤ 1.0 or not. If it is, the task can be assigned to the processor. The second test, denoted by DBF, was introduced by Baruah et al. [28] and is based on an approximation to the demand bound function [29]. In this test, tasks are considered in non-decreasing order of their relative deadlines. In general, Best Fit packing combined with the DBF test produced the best results in our experiments.

In [18] the authors show that their approach performs best when R-MIN is used for assigning reservation servers together with BF packing and the DBF test. We hence included this variant in our figures and denote it RESV. For semi-federated scheduling, denoted by SEMI, we used WF packing together with the DEN test, which is similar to [17]. For the federated scheduling algorithm proposed by Baruah [13], denoted by BAR, the results for BF packing combined with the DBF test are reported since they performed the best for BAR. For the stretching algorithm proposed by Qamhieh el al. [19], we included the results for when G-EDF and P-EDF are used to schedule light tasks and the threads resulting after stretching. These two variants are denoted as STRG and STRP, respectively. In case of P-EDF, BF packing and the DBF test are used. For our federated scheduling algorithm, we include two variants: one for WF packing with the DEN test, and the other for BF packing with the DBF test. These two variants are denoted as PRO-WF-DEN and PRO-BF-DBF, respectively.

We applied the Erdős-Rényi method [23] to generate DAGs. For each task, the number of subtasks was uniformly chosen in $[50, 250]$. Each subtask has a WCET picked uniformly in $[50, 100]$ and the probability threshold p was set to 0.2. The work and span of each DAG were computed accordingly. For given values of m, U, U^h, n^h, n^l, we used the RandFixedSum algorithm [24] to generate individual utilizations uniformly in $[1.1, U^h \times m]$ for heavy tasks and in $[0.01, 0.9]$ for light tasks. Each task's period was computed from its work and utilization. For each setting, 500 task sets were generated and the ratios of schedulable task sets were recorded. We conducted experiments with $m = \{16, 32, 64\}$ processors.

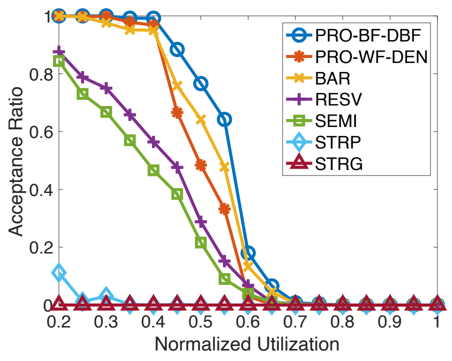
Varying total utilization: In Figure 5 we varied U in $[0.2, 1.0]$ with a step of 0.05. The ratio $\frac{U^h}{U}$ was set to 0.5, i.e., half of the total utilization is from heavy tasks. For $m = \{16, 32, 64\}$, we generated a maximum of $\{4, 7, 14\}$ heavy tasks, respectively (for small values of U we generated smaller numbers of heavy tasks accordingly). For light tasks, $n^l = \{20, 40, 80\}$ respectively. We observed that PRO-BF-DBF outperformed all other approaches. Especially, it outperformed SEMI, RESV, STRP, and STRG by a large margin. PRO-WF-DEN, which uses a similar combination of WF packing with DBF test as SEMI, also outperformed SEMI.

PRO outperforms SEMI because regardless of the processing capacity of a heavy task (the $x + \epsilon$ quantity), SEMI saves at most 1 processor for the task. SEMI's effectiveness thus reduces when the number of heavy tasks increases and/or heavy tasks get larger. In RESV, the ability of sharing processors between heavy and light tasks by using reservation servers comes at the cost of inflating budget. In particular, the total budget for τ_i must be $\geq C_i + L_i \cdot (m_i - 1) > C_i$, where m_i is the number of servers of τ_i. Similar to [18], we observe that RESV outperforms SEMI. STRP and STRG performed worst among all approaches. This is because when a task has a large number of subtasks, the stretching algorithm may generate many threads with short relative deadlines (compared to the DAG task's deadline). These threads may have high total density and thus are hard to schedule. Among these two variants, STRP performed better than STRG.

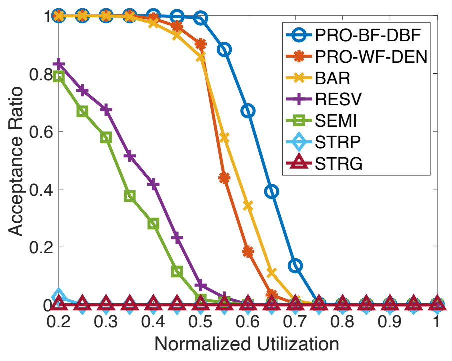
Surprisingly, BAR also outperformed RESV and SEMI (and STRP, STRG) significantly. This shows that federated scheduling is still very competitive if heavy tasks are scheduled efficiently. As m (and n) increases, the gap between group (PRO-BF-DBF, PRO-WF-DEN, BAR) and group (RESV, SEMI, STRP, STRG) expands. This is because as n and m increases, the number of processors saved by PRO-BF-DBF, PRO-WF-DEN, and BAR increases while the effectiveness of RESV and SEMI reduces due to over-provisioning. The gap between PRO-BF-DBF and BAR also increases for the same reason as was discussed in Section V. BF packing combined with the DBF test performed much better than the combination of WF packing with the DEN test. This is shown in the performance gap between PRO-BF-DBF and PRO-WF-DEN.

Varying load of heavy tasks: In Figure 6, for each m we kept U and varied $\frac{U^h}{U}$ in $[0.0, 1.0]$ with a step of 0.1. The maximum n^h in each task set is $\{4, 8, 14\}$ and the maximum n^l is $\{20, 40, 80\}$ for $m = \{16, 32, 64\}$, respectively.

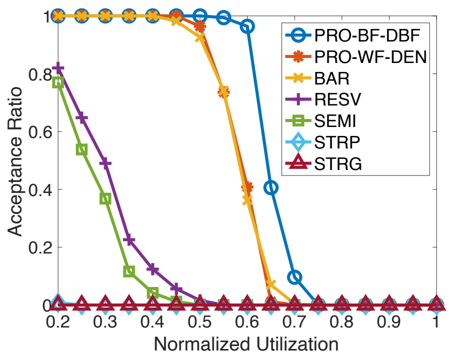
We observed a similar trend in this experiment. The fluctuation of PRO-BF-DBF, PRO-WF-DEN, and BAR is due to the integral processor allocation for heavy tasks. Consequently, there are cases when U^h increases but the heavy tasks do not require additional processors while U^l reduces and the light tasks become easier to schedule. In addition, as the proportion of heavy tasks increases, the task sets become harder to schedule and the performances of all approaches decrease.



(a) $m = 16$, $n^h \in [1, 4]$, $U^h = 0.5U$, $n^l = 20$.

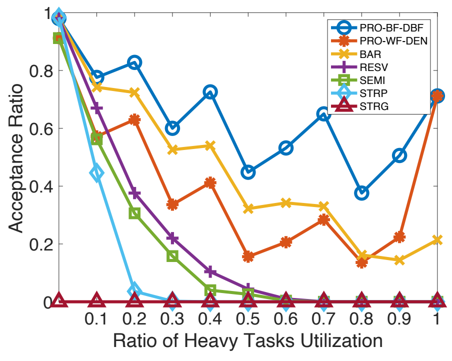


(b) $m = 32$, $n^h \in [2, 7]$, $U^h = 0.5U$, $n^l = 40$.

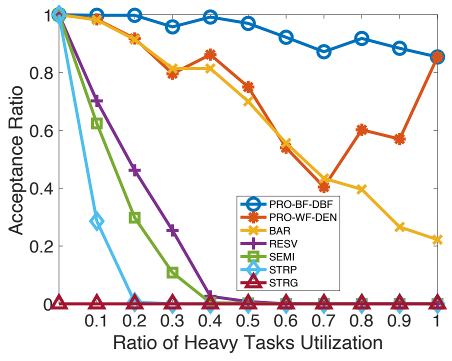


(c) $m = 64$, $n^h \in [4, 14]$, $U^h = 0.5U$, $n^l = 80$.

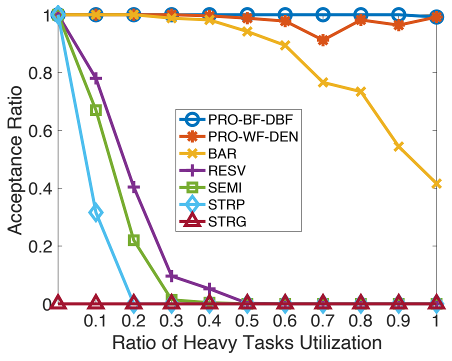
Fig. 5: Ratio of schedulable task sets for varying total utilization U.



(a) $m = 16$, $n^h \in [1, 4]$, $n^l = 20$, $U = 0.6$.

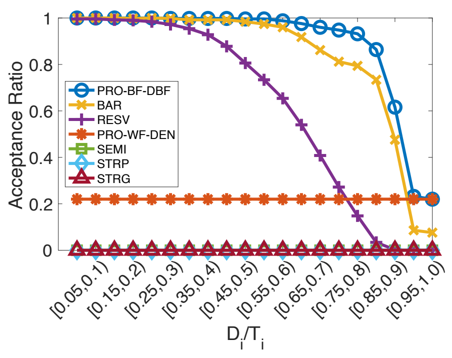


(b) $m = 32$, $n^h \in [1, 8]$, $n^l = 40$, $U = 0.55$.

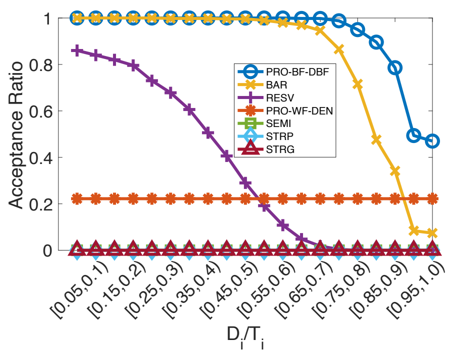


(c) $m = 64$, $n^h \in [2, 14]$, $n^l = 80$, $U = 0.5$.

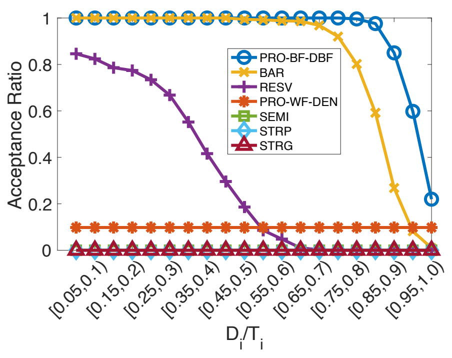
Fig. 6: Ratio of schedulable task sets for varying $\frac{U^h}{U}$.



(a) $m = 16$, $n^h = 4$, $n^l = 20$.



(b) $m = 32$, $n^h = 7$, $n^l = 40$.



(c) $m = 64$, $n^h = 14$, $n^l = 80$.

Fig. 7: Ratio of schedulable task sets for 0.9 normalized density, 0.5 normalized density for heavy tasks, and varying $\frac{D_i}{T_i}$.

However, RESV, SEMI and STRP degrade much faster than PRO-BF-DBF, PRO-WF-DEN, and BAR; and PRO-BF-DBF is the most stable approach. When $\frac{U^h}{U} = 0$, all tasks are light and STRP performed as well as other approaches. However, its performance degraded quickly as heavy tasks were included. Also, when $\frac{U^h}{U}$ increases, the performance gap between PRO-BF-DBF and BAR gets larger as Algorithm 1 schedules heavy tasks more efficiently.

Varying deadline and period ratio: Figure 7 shows the result for varying $\frac{D_i}{T_i}$. We generated 500 task sets with normalized density fixed to 0.9 for all values of m. In each task set, half of the total density was given to heavy tasks. For $m = \{16, 32, 64\}$, $n^h = \{4, 7, 14\}$ and $n^l = \{20, 40, 80\}$ respectively. We used the RandFixedSum algorithm [24] to generate individual densities for all tasks. D_i of each task was computed based on its density and work. The range for $\frac{D_i}{T_i}$ was varied in the set $\{[0.01, 0.05), [0.05, 0.1), ..., [0.95, 1.0)\}$. For a given range, $\frac{D_i}{T_i}$ was uniformly chosen in that range, and

T_i was computed accordingly based on D_i. As $\frac{D_i}{T_i}$ increases, the total utilization also increases and the task sets become harder to schedule. Again, PRO-BF-DBF outperformed other methods. RESV performed well for small $\frac{D_i}{T_i}$ but declined quickly when it increases. The performance of PRO-WF-DEN and SEMI did not change as $\frac{D_i}{T_i}$ increases since they use the DEN test for light tasks, which only depends on the densities (and thus the deadlines) of the tasks, and for heavy tasks the number of processors allocated to them is not affected.

VII. Conclusion

We have proposed a novel algorithm that computes a deterministic schedule for DAG tasks. The algorithm uses the graph structure of each task to efficiently schedule it on its dedicated processors. Our experiments show that the proposed algorithm significantly reduces the number of processors required by each heavy task. Our new federated scheduling algorithm is also shown to outperform the state-of-the-art federated-based scheduling algorithms [13], [17], [18] and a stretching scheduling algorithm [19] by a large margin. Our deterministic algorithm for scheduling heavy tasks can be combined with the work by Brandenburg et al. [30] for an overall efficient solution to scheduling DAG tasks. In [30], the authors show that near optimal schedulable utilization (over 99%) for sequential tasks on multiprocessors can be reached using techniques, such as semi-partitioned scheduling, reservations, period transformation, and their new heuristics for task placement. In this combination, Algorithm 1 can be used to schedule heavy tasks, and techniques in [30] can be applied to schedule light tasks on the remaining processors.

References

[1] J. Kim, H. Kim, K. Lakshmanan, and R. R. Rajkumar, “Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car,” in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. ACM, 2013, pp. 31–40.

[2] D. Ferry, G. Bunting, A. Maghareh, A. Prakash, S. Dyke, K. Agrawal, C. Gill, and C. Lu, “Real-time system support for hybrid structural simulation,” in *Proceedings of the 14th International Conference on Embedded Software*. ACM, 2014, p. 25.

[3] G. A. Elliott, K. Yang, and J. H. Anderson, “Supporting real-time computer vision workloads using OpenVX on multicore+ GPU platforms,” in *2015 IEEE Real-Time Systems Symposium*. IEEE, 2015, pp. 273–284.

[4] J. Li, K. Agrawal, C. Lu, and C. Gill, “Analysis of global EDF for parallel tasks,” in *ECRTS*, 2013.

[5] H. S. Chwa, J. Lee, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, “Global EDF schedulability analysis for parallel tasks on multi-core platforms,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1331–1345, 2016.

[6] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, “Global EDF scheduling of directed acyclic graphs on multiprocessor systems,” in *Proceedings of the 21st International conference on Real-Time Networks and Systems*. ACM, 2013, pp. 287–296.

[7] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, “Feasibility analysis in the sporadic DAG task model,” in *2013 25th Euromicro conference on real-time systems*. IEEE, 2013, pp. 225–233.

[8] S. Baruah, “Improved multiprocessor global schedulability analysis of sporadic DAG task systems,” in *2014 26th Euromicro conference on real-time systems*. IEEE, 2014, pp. 97–105.

[9] J. Fonseca, G. Nelissen, and V. Nélis, “Improved response time analysis of sporadic DAG tasks for global FP scheduling,” in *Proceedings of the 25th international conference on real-time networks and systems*. ACM, 2017, pp. 28–37.

[10] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, “Response time analysis of sporadic DAG tasks under partitioned scheduling,” in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2016, pp. 1–10.

[11] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, “Partitioned fixed-priority scheduling of parallel tasks without preemptions,” in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 421–433.

[12] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, “Analysis of federated and global scheduling for parallel real-time tasks,” in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE, 2014, pp. 85–96.

[13] S. Baruah, “The federated scheduling of constrained-deadline sporadic DAG task systems,” in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 1323–1328.

[14] ——, “Federated scheduling of sporadic DAG task systems,” in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 179–186.

[15] ——, “The federated scheduling of systems of conditional sporadic DAG tasks,” in *Proceedings of the 12th International Conference on Embedded Software*. IEEE Press, 2015, pp. 1–10.

[16] J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill, and C. Lu, “Randomized work stealing for large scale soft real-time systems,” in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 203–214.

[17] X. Jiang, N. Guan, X. Long, and W. Yi, “Semi-federated scheduling of parallel real-time tasks on multiprocessors,” in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 80–91.

[18] N. Ueter, G. von der Brüggen, J.-J. Chen, J. Li, and K. Agrawal, “Reservation-based federated scheduling for parallel real-time tasks,” in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 482–494.

[19] M. Qamhieh, L. George, and S. 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*. ACM, 2014, p. 13.

[20] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, “Response-time analysis of conditional DAG tasks in multiprocessor systems,” in *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 2015, pp. 211–221.

[21] R. L. Graham, “Bounds on multiprocessing timing anomalies,” *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[23] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, “Random graph generation for scheduling simulations,” in *Proceedings of the 3rd international ICST conference on simulation tools and techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010, p. 60.

[24] R. Stafford, “Random vectors with fixed sum,” http://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum, 2006.

[25] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, “Worst-case performance bounds for simple one-dimensional packing algorithms,” *SIAM Journal on computing*, vol. 3, no. 4, pp. 299–325, 1974.

[26] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. Gill, “Parallel real-time scheduling of DAGs,” *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[27] X. Jiang, X. Long, N. Guan, and H. Wan, “On the decomposition-based global EDF scheduling of parallel real-time tasks,” in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 237–246.

[28] S. Baruah and N. Fisher, “The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems,” *IEEE Transactions on Computers*, vol. 55, no. 7, pp. 918–923, 2006.

[29] S. K. Baruah, A. K. Mok, and L. E. Rosier, “Preemptively scheduling hard-real-time sporadic tasks on one processor,” in *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE, 1990, pp. 182–190.

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