Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors

Xu Jiang[1,2], Nan Guan[1], Xiang Long[2], Wang Yi[3]

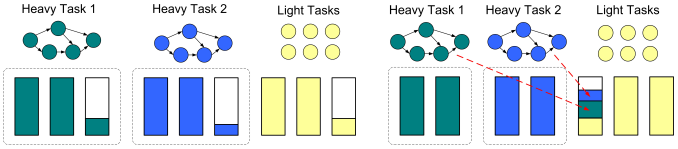
[1]The Hong Kong Polytechnic University, Hong Kong
[2]Beihang University, China
[3]Uppsala University, Sweden

***Abstract*—Federated scheduling is a promising approach to schedule parallel real-time tasks on multi-cores, where each heavy task exclusively executes on a number of dedicated processors, while light tasks are treated as sequential sporadic tasks and share the remaining processors. However, federated scheduling suffers resource waste since a heavy task with processing capacity requirement $x + \epsilon$ (where x is an integer and $0 < \epsilon < 1$) needs $x+1$ dedicated processors. In the extreme case, almost half of the processing capacity is wasted. In this paper we propose the semi-federate scheduling approach, which only grants x dedicated processors to a heavy task with processing capacity requirement $x+\epsilon$, and schedules the remaining ϵ part together with light tasks on shared processors. Experiments with randomly generated task sets show the semi-federated scheduling approach significantly outperforms not only federated scheduling, but also all existing approaches for scheduling parallel real-time tasks on multi-cores.**

I. Introduction

Multi-cores are more and more widely used in real-time systems, to meet their rapidly increasing requirements in performance and energy efficiency. The processing capacity of multi-cores is not a free lunch. Software must be properly parallelized to fully exploit the computation capacity of multi-core processors. Existing scheduling and analysis techniques for sequential real-time tasks are hard to migrate to the parallel workload setting. New scheduling and analysis techniques are required to deploy parallel real-time tasks on multi-cores.

A parallel real-time task is usually modeled as a Directed Acyclic Graph (DAG). Several scheduling algorithms have been proposed to schedule DAG tasks in recent years, among which *Federated Scheduling* [1] is a promising approach with both good real-time performance and high flexibility. In federated scheduling, DAG tasks are classified into *heavy* tasks (density > 1) and *light* tasks (density ≤ 1). Each heavy task exclusively executes on a subset of dedicated processors. Light tasks are treated as traditional sequential real-time tasks and share the remaining processors. Federated scheduling not only can schedule a large portion of DAG task systems that is not schedulable by other approaches, but also provides the best quantitative worst-case performance guarantee [1]. On the other hand, federated scheduling allows flexible workload specification as the underlying analysis techniques only require information about the critical path length and total workload of the DAG, and thus can be easily extended to more expressive models, such as DAG with conditional branching [2], [3].



(a) Federated scheduling. (b) Semi-federated scheduling.

Fig. 1. Illustration of federated scheduling and semi-federated scheduling.

However, federated scheduling may suffer significant resource waste, since each heavy task *exclusively* owns a subset of processors. For example, if a heavy task requires processing capacity $x + \epsilon$ (where x is an integer and $0 < \epsilon < 1$), then $\lceil x + \epsilon \rceil = x + 1$ dedicated processors are granted to it, as shown in Figure 1-(a). In the extreme case, almost half of the total processing capacity is wasted (when a DAG requires $1+\epsilon$ processing capacity and $\epsilon \to 0$).

In this work, we propose the *Semi-Federated Scheduling* approach to solve the above resource waste problem. In semi-federated scheduling, a DAG task requiring $x + \epsilon$ processing capacity is only granted x dedicated processors, and the remaining fractional part ϵ is scheduled together with the light tasks, as illustrated in Figure 1-(b).

The major challenge we face in realizing semi-federated scheduling is how to control and analyze the interference suffered by the fractional part, and its effect to the timing behavior of the entire heavy task. The fractional part of a heavy task is scheduled together with, and thus suffers interference from the light tasks and the fractional parts of other heavy tasks. Due to the intra-task dependencies inside a DAG, this interference is propagated to other parts of the DAG executed on the dedicated processors, and thus affects the timing behavior of the entire DAG task. Existing scheduling and analysis techniques for federated scheduling (based on the classical work in [4]) cannot handle such extra interference.

This paper addresses the above challenges and develops semi-federated scheduling algorithms in the following steps.

First, we study the problem of bounding the response time of an individual DAG executing on a *uniform* multiprocessor platform (where processors have different speeds). The results we obtained for this problem serve as the theoretical foundation of the semi-federated scheduling approach. Intuitively, we grant a portion (< 1) of the processing capacity of a processor

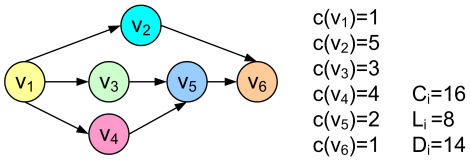


Fig. 2. A DAG task example.

to execute the fractional part of a DAG, which is similar to executing it on a slower processor.

Second, the above results are transferred to the realistic situation where the fractional parts of DAG tasks and the light tasks share several processors with unit speed. This is realized by executing the fractional parts via sequential *container tasks*, each of which has a *load bound*. A container task plays the role of a dedicated processor with a slower speed (equals the container task's load bound), and thus the above results can be applied to analyze the response time of the DAG task.

Finally, we propose two semi-federated scheduling algorithms based on the above framework. In the first algorithm, a DAG task requiring $x + \epsilon$ processing capacity is granted x dedicated processors and *one* container task with load bound ϵ, and all the container tasks and the light tasks are scheduled by *partitioned* EDF on the remaining processors. The second algorithm enhances the first one by allowing to divide the fractional part ϵ into *two* container tasks, which further improves resource utilization.

We conduct experiments with randomly generated workload, which show our semi-federated scheduling algorithms significantly improve schedulability over the state-of-the-art of, not only federated scheduling, but also the other types such as global scheduling and decomposition-based scheduling.

II. Preliminary

A. Task Model

We consider a task set τ consisting of n tasks $\{\tau_1, \tau_2, ..., \tau_n\}$, executed on m identical processors with unit speed. Each task is represented by a DAG, with a period T_i and a relative deadline D_i. We assume all tasks to have *constrained deadlines*, i.e., $D_i \leq T_i$. Each task is represented by a directed acyclic graph (DAG). A vertex v in the DAG has a WCET $c(v)$. Edges represent dependencies among vertices. A directed edge from vertex v to u means that u can only be executed after v is finished. In this case, v is a *predecessor* of u, and u is a *successor* of v. We say a vertex is *eligible* at some time point if all its predecessors in the current release have been finished and thus it can immediately execute if there are available processors. We assume each DAG has a unique head vertex (with no predecessor) and a unique tail vertex (with no successor). This assumption does not limit the expressiveness of our model since one can always add a dummy head/tail vertex to a DAG with multiple entry/exit points.

We use C_i to denote the total worst-case execution time of all vertices of DAG task τ_i and L_i to denote the sum of $c(v)$ of each vertex v along the longest chain (also called the critical path) of τ_i. The *utilization* of a DAG task τ_i is $\frac{C_i}{T_i}$, and its *density* is $\frac{C_i}{D_i}$. A DAG task is called a *heavy* task if it density is larger than 1, and a *light* task otherwise.

Figure 2 shows a DAG task with 6 vertices. We can compute $C_i = 16$ and $L_i = 8$ (the longest path is $\{v_1, v_4, v_5, v_6\}$). This is a heavy task since the density is $\frac{16}{14} > 1$.

B. Federated Scheduling

In federated scheduling [1], each heavy task exclusively executes on a subset of dedicated processors. Light tasks are treated as traditional sequential real-time tasks and share the remaining processors. As a heavy task exclusively owns several dedicated processors and its workload must be finished before the next release time (due to constrained deadlines), the response time of a heavy task can be bounded using the classical result for non-recurrent DAG tasks by Graham [4]:

$$R_i \leq L_i + \frac{C_i - L_i}{m_i} \tag{1}$$

where m_i is the number of dedicated processors granted to this heavy task. Therefore, by setting $R_i = D_i$, we can calculate the minimal amount of processing capacity required by this task to meet its deadline $\frac{C_i - L_i}{D_i - L_i}$, and the number of processors assigned to a heavy task τ_i is the minimal integer no smaller than $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$. The light tasks are treated as sequential sporadic tasks, and are scheduled on the remaining processors by traditional multiprocessor scheduling algorithms, such as global EDF [5] and partitioned EDF [6].

III. A Single DAG on Uniform Multiprocessors

In this section, we focus on the problem of bounding the response time of a single DAG task exclusively executing on a *uniform* multiprocessor platform, where processors in general have different speeds. The reason why we study the case of uniform multiprocessors is as follows. In the semi-federated scheduling, a heavy task may share processors with others. From this heavy task's point of view, it only owns a portion of the processing capacity of the shared processors. Therefore, to analyze semi-federated scheduling, we first need to solve the fundamental problem of how to bound the response time in the presence of portioned processing capacity. The results of this section serve as the theoretical foundation for semi-federated scheduling on *identical* multiprocessors in later sections (while they also can be directly used for federated scheduling on uniform multiprocessors as a byproduct of this paper).

A. Uniform Multiprocessor Platform

We assume a uniform multiprocessor platform of m processors, characterized by their normalized speeds $\{\delta_1, \delta_2, \cdots, \delta_m\}$. Without loss of generality, we assume the processors are sorted in non-increasing speed order ($\delta_x \geq \delta_{x+1}$) and the fastest processor has a unit speed i.e., $\delta_1 = 1$. In a time interval of length t, the amount of workload executed on a processor with speed δ_x is $t\delta_x$. Therefore, if the WCET of some workload on a unit speed processor is c, then its WCET becomes c/δ_x on a processor of speed δ_x.

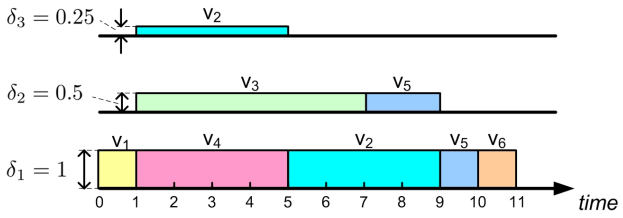


Fig. 3. A work-conserving scheduling sequence on uniform multiprocessors.

B. Work-Conserving Scheduling on Uniform Multiprocessors

On identical multiprocessors, a work-conserving scheduling algorithm never leaves any processor idle while there exists some eligible vertex. The response time bound for a DAG task in (1) is applicable to *any* work-conserving scheduling algorithm, regardless what particular strategy is used to assign the eligible vertices to available processors.

However, on uniform multiprocessors, the strategy to assign eligible vertices to processors may greatly affect the timing behavior of the task. Therefore, we extend the concept of work-conserving scheduling to uniform multiprocessors by enforcing execution on faster processors as much as possible [7]. More precisely, a scheduling algorithm is work-conserving on m uniform processors if it satisfies both of the following conditions:

1) No processor is idled when there are eligible vertices awaiting execution.
2) If at some time point there are fewer than m eligible vertices awaiting execution, then the eligible vertices are executed upon the fastest processors.

Figure 3 shows a possible scheduling sequence of the DAG task on three processors with speeds $\{1, 0.5, 0.25\}$. Vertex v_2 migrates to the fastest processor at time 5 and v_5 migrates to the fastest processor at time 9. These two extra migrations are the price paid for satisfying the second condition of work-conserving scheduling in above.

If we disallow the migration from slower processors to faster processors, there may be significant resource waste. In the worst case, a DAG task will execute its longest path on the lowest processor, which results in very large response time. In Appendix-A we discuss the response time bound and resource waste when the inter-processor migration is forbidden.

C. Response Time Bound

In the following we derive response time bounds for a single DAG task executing on a uniform multiprocessor platform under work-conserving scheduling. Although the task is recurrent, we only need to analyze its behavior in one release since the task has a constraint deadline. We first introduce the concept *uniformity* [7]:

Definition 1 (Uniformity)**.** *The uniformity of m processors with speeds $\{\delta_1, \cdots, \delta_m\}$ ($\delta_x \geq \delta_{x+1}$) is defined as*

$$\lambda \triangleq \max_{x=1}^{m} \left\{ \frac{S_m - S_x}{\delta_x} \right\} \tag{2}$$

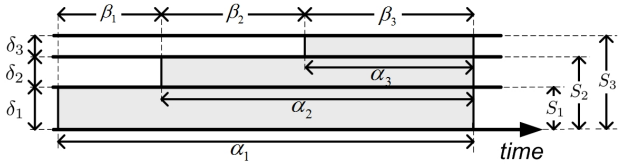


Fig. 4. Illustration of α_x and β_x.

where S_x is the sum of the speeds of the x fastest processors:

$$S_x \triangleq \sum_{j=1}^{x} \delta_j \tag{3}$$

Now we derive the response time upper bound:

Theorem 1. *The response time of a DAG task τ_i executing on m processors with speeds $\{\delta_1, \cdots, \delta_m\}$ is bounded by:*

$$R \leq \frac{C_i + \lambda L_i}{S_m} \tag{4}$$

where λ and S_m are defined in Definition 2.

Proof. For simplify of presentation, we assume that each vertex v executes exactly for its WCET $c(v)$[1]. Without loss of generality, we assume the task under analysis releases an instance at time 0, and thus finishes the current instance at time R. During the time window $[0, R]$, let α_x denote the total length of intervals during which the x^{th} processor (with speed δ_x) is busy. By the work-conserving scheduling rules in Section III-B, we know if the x^{th} processor is busy in a time interval then all faster processors (with index smaller than x) must also be busy. Therefore, we know $R = \alpha_1$. We define

$$\beta_x = \begin{cases} \alpha_x - \alpha_{x+1}, & 1 \leq x < m \\ \alpha_x, & x = m \end{cases}$$

Figure 4 illustrates the definition of α_x and β_x. So we can rewrite $R = \alpha_1$ as

$$R = \sum_{x=1}^{m} \beta_x \tag{5}$$

The total workload executed on all the processors in $[0, R]$ is $(\beta_1 S_1 + \cdots + \beta_m S_m)$, which equals the total worst-case execution time of the task:

$$C_i = \sum_{x=1}^{m} \beta_x S_x \tag{6}$$

Let π be an arbitrary path in the DAG starting from the head vertex and ending at the tail vertex. We use $\chi(\pi, \delta_x)$ to denote the total amount of workload executed for vertices along path π in all the time intervals during which both of the following conditions are satisfied:

- at least one processor is idle
- the slowest busy processor has speed δ_x.

[1]It is easy to show that the response time bound in (4) still holds if some vertices execute for shorter than its WCET.

The total length of such time intervals is β_x. Since at least one processor is idle, π must contain a vertex being executed in this time interval (since at any time point before R, there is at least one eligible vertex along any path). So we have

$$\chi(\pi, \delta_x) \geq \beta_x \delta_x$$
$$\Rightarrow \quad \sum_{x=1}^{m-1} \chi(\pi, \delta_x) \geq \sum_{x=1}^{m-1} \beta_x \delta_x \tag{7}$$

Let l_π denote the total workloadalong path π, so we know

$$\sum_{x=1}^{m-1} \chi(\pi, \delta_x) \leq l_\pi$$

Since L_i is the total workload of the longest path in the DAG, we know $l_\pi \leq L_i$. In summary, we have

$$\sum_{x=1}^{m-1} \chi(\pi, \delta_x) \leq L_i \tag{8}$$

Combining (7) and (8) gives

$$L_i \geq \sum_{x=1}^{m-1} \beta_x \delta_x \;\Rightarrow\; \lambda L_i \geq \sum_{x=1}^{m-1} \beta_x \delta_x \lambda \tag{9}$$

By the Definition of λ in (2) we know

$$\forall x : \frac{S_m - S_x}{\delta_x} \leq \lambda$$

Therefore, (9) can be rewritten as

$$\lambda L_i \geq \sum_{x=1}^{m-1} \beta_x (S_m - S_x)$$

and by applying (6) we get

$$C_i + \lambda L_i \geq \sum_{x=1}^{m-1} \beta_x (S_m - S_x) + \sum_{x=1}^{m} \beta_x S_x$$
$$\Leftrightarrow \quad C_i + \lambda L_i \geq \beta_m S_m + \sum_{x=1}^{m-1} \beta_x S_m$$
$$\Leftrightarrow \quad C_i + \lambda L_i \geq S_m \sum_{x=1}^{m} \beta_x$$

and by applying (5), the theorem is proved. □

When $\delta_1 = \cdots = \delta_m = 1$, we have $\lambda = m-1$ and $S_m = m$, so the bound in Theorem 1 perfectly degrades to (1) for the case of identical processors.

IV. Runtime Dispatcher of Each DAG

The conceptual uniform multiprocessor platform in last section imitates the resource obtained by a task when sharing processors with other tasks. A naive way to realize the conceptual uniform multiprocessors on our identical unit-speed multiprocessor platform is to use fairness-based scheduling, in which task switching is sufficiently frequent so that each task receives a fixed portion of processing capacity. However, this approach incurs extremely high context switch overheads which may not be acceptable in practice.

In the following, we introduce our method to realize the proportional sharing of processing capacity without frequent context switches. The key idea is to use a runtime *dispatcher* for each DAG task to encapsulate the execution on a conceptual processor with speed δ_p into a *container task* φ_p with a *load bound* δ_p. The dispatcher guarantees that the workload encapsulated into a container task does not exceed its load bound. These container tasks are scheduled using priority-based scheduling algorithms and their load bounds can be used to decide the schedulability.

As will be introduced in the next section, in our semi-federated scheduling algorithms, most of the container tasks used by a DAG task have a load bound 1, which correspond to the dedicated processors, and only a few of them have fractional load bounds (< 1). However, for simplicity of presentation, in this section we treat all container tasks identically, regardless whether the load bound is 1 or not.

Suppose we execute a DAG task through m container tasks $\{\varphi_1, \varphi_2, \cdots, \varphi_m\}$. Each of the container task is affiliated with the following information $\varphi_p = (\delta_p, d_p, exe_p)$:

- δ_p: the load bound of φ_p, which is a fixed value.
- d_p: the absolute deadline of φ_p, which varies at runtime.
- exe_p: the vertex currently executed by φ_p, which also varies at runtime

At each time instant, a container task is either *occupied* by some vertex or *empty*. If a container task is occupied by vertex v, i.e., $exe_p = v$, then this container task is responsible to execute the workload of v and the maximal workload executed by this container task executes before the absolute deadline d_p is $c(v)$. A vertex v may be divided into several parts, and the their total WCET equals $c(v)$, as will be discussed later when we introduce Algorithm 1. **Note that an occupied container task becomes empty when time reaches its absolute deadline**.

Algorithm 1 The dispatching algorithm (invoked at time t).

1: v = an arbitrary eligible vertex in S (S stores the set of vertices that have not been executed yet);
2: Remove v from S;
3: φ_p = the empty container task with the largest load bound;
4: d' = the earliest deadline of all occupied container tasks with load bound strictly larger than δ_p;
5: **if** (all container tasks are empty) $\vee$ $(d' > t + \frac{c(v)}{\delta_p})$ **then**
6: $\quad d_p = t + c(v)/\delta_p$
7: $\quad exe_p = v$
8: **else**
9: $\quad d_p = d'$
10: $\quad$ Split v into v' and v'' so that

$$c(v') = (d_p - t) \times \delta_p \text{ and } c(v'') = c(v) - c(v')$$

11: $\quad exe_p = v'$
12: $\quad$ Put v'' back to the head of S;
13: $\quad$ Add a precedence constraint from v' to v'';
14: **end if**

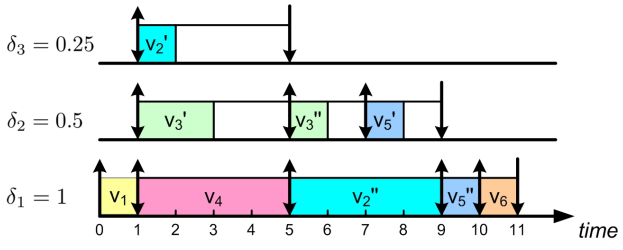


Fig. 5. A scheduling sequence on container tasks.

The pseudo-code of the dispatcher is shown in Algorithm 1. At runtime, the dispatcher is invoked when there exist both empty container tasks and eligible vertices. The target of the dispatcher is to assign (a part of) an eligible vertex to the *fastest* empty container task.

The absolute deadline d_p of a container task φ_p mimics the finishing time of a vertex if it is executed on a processor with the speed δ_p. When the container task starts to be occupied by a vertex v at time t, d_p is set to be $d_p = t + c(v)/\delta_p$. Therefore, we have the following property of Algorithm 1. First, the dispatcher guarantees the execution rate of a container task is consistent with the corresponding uniform processors:

Property 1. *If φ_p starts to be occupied by v from t_1 and becomes empty at t_2, the maximal workload executed by φ_p in $[t_1, t_2)$ is $(t_2 - t_1)\delta_p$.*

Another key point of Algorithm 1 is always keeping the container task with larger load bounds being occupied, which mimics the second work-conserving scheduling rule on uniform multiprocessors (workload is always executed on faster processors). This is done by checking the condition in line 5:

$$d' > t + c(v)/\delta_p \tag{10}$$

where d' is the earliest absolute deadline among all the container tasks currently being occupied and δ_p is load bound of the fastest empty container task which will be used now. If this condition does not hold, putting the entire v into φ_p may lead to the situation that a container task with a larger load bound becomes empty while φ_p is still occupied. This corresponds to the situation on uniform processors that a faster processor is idle while a slower processor is busy, which violates the second work-conserving scheduling rule. To solve this problem, in Algorithm 1, when condition (10) does not hold, v is split into two parts v' and v'', so that φ_p only executes the first part v', whose deadline exactly equals to the earliest absolute deadline of all faster container tasks (line 10). The remaining part v'' is put back to S and will be assigned in the future, and a precedence from v' to v'' is established to guarantee that v'' become eligible only if v' has finished. In summary, Algorithm 1 guarantees the following property:

Property 2. *The eligible vertices are always executed upon the container tasks with the largest load bounds.*

Figure 5 shows a possible scheduling sequence of the example DAG task in Figure 2 executed on three container tasks with load bounds $\delta_1 = 1$, $\delta_2 = 0.5$ and $\delta_3 = 0.25$. An upwards arrow represents an empty container task becoming occupied and a downwards arrow represents an occupied task becoming empty. Algorithm 1 is invoked whenever there exist both eligible vertices and empty container tasks. This scheduling sequence corresponds to the scheduling sequence of the same task on uniform processors with speeds $\delta_1 = 1$, $\delta_2 = 0.5$ and $\delta_3 = 0.25$ in Figure 3. We can see that the amount of workload executed between any two time points at which Algorithm 1 is invoked, is the same in both scheduling sequences. An step-by-step explanation of this example is given in Appendix-B.

In general, if each container task always finishes the workload of its assigned vertex before the corresponding deadline, the scheduling sequence resulted by Algorithm 1 on container tasks with load bounds $\{\delta_1, \cdots, \delta_m\}$ corresponds to a work-conserving scheduling sequence of the same DAG task on uniform multiprocessors with speeds $\{\delta_1, \cdots, \delta_m\}$. Therefore the response time bound in Theorem 1 can be applied to bound the response time of the DAG task executed on container tasks using Algorithm 1. By the above discussions, we can conclude the following theorem.

Theorem 2. *Suppose a DAG task τ_i executes on m container tasks with load bounds $\{\delta_1, \cdots, \delta_m\}$ and each container task always finishes its assigned workload before the corresponding absolute deadline, then the response time R of τ_i is upper bounded by:*

$$R \leq \frac{C_i + \lambda L_i}{S_m} \tag{11}$$

V. Semi-Federated Scheduling Algorithms

In this section, we propose two semi-federated scheduling algorithms based on container task and runtime dispatcher introduced in last section. In the first algorithm, a DAG task requiring $x + \epsilon$ processing capacity is granted x dedicated processors and *one* container task with load bound ϵ, and all the container tasks and the light tasks are scheduled by *partitioned* EDF on the remaining processors. The second algorithm enhances the first one by allowing to divide the fractional part ϵ into *two* container tasks, which further improves resource utilization.

A. The First Algorithm: SF[x+1]

By Theorem 2 we know a DAG task is schedulable if the load bounds $\{\delta_1, \cdots, \delta_m\}$ of the container tasks satisfy

$$\frac{C_i + \lambda L_i}{S_m} \leq D_i \tag{12}$$

where λ is the uniformity and S_m is the sum of $\{\delta_1, \cdots, \delta_m\}$, defined in Definition 1. There are difference choices of the container tasks to make a DAG task schedulable. In general, we want to make the DAG task to be schedulable with as little processing capacity as possible. The load bound of a container task actually represents its required processing capacity, and thus S_m represents the total processing capacity required by all the container tasks of a DAG task. In the following, we

will introduce how to choose the feasible container task set with the minimal S_m.

We first show that the total load bound of any container task set that can pass the condition (12) has a lower bound:

Definition 2. *The* minimal capacity requirement γ_i *of a DAG task* τ_i *is defined as:*

$$\gamma_i = \frac{C_i - L_i}{D_i - L_i} \tag{13}$$

Lemma 1. *A DAG task* τ_i *is scheduled on* m *container tasks with load bounds* $\{\delta_1, \delta_2, \cdots, \delta_m\}$*. If condition (12) is satisfied, then it must hold*

$$S_m \geq \gamma_i$$

Proof. Without loss of generality, we assume the container tasks are sorted in non-increasing order of their load bounds, i.e., $\delta_p \geq \delta_{p+1}$. By the definition of λ we have

$$\lambda \geq \frac{S_m - \delta_1}{\delta_1}$$

and since the load bounds are at most 1, i.e., $\delta_1 \leq 1$, we know

$$\lambda \geq S_m - 1$$

Applying this to (12) yields

$$\frac{C_i + (S_m - 1)L_i}{S_m} \leq D_i \Rightarrow S_m \geq \frac{C_i - L_i}{D_i - L_i}$$

so the lemma is proved. □

Next we show that the minimal capacity requirement is achieved by using only one container task with a fractional load bound (< 1) and x container tasks with load bound 1:

Lemma 2. *A DAG task* τ_i *is schedulable on* x *container tasks with load bound of* 1 *and one container task with load bound* δ*, where* $x = \lfloor \gamma_i \rfloor$ *and* $\delta = \gamma_i - \lfloor \gamma_i \rfloor$*.*

Proof. By the definition of λ, we get

$$\lambda = \max\left(\frac{\gamma_i - 1}{1}, \cdots, \frac{\gamma_i - \lfloor \gamma_i \rfloor}{1}\right) = \gamma_i - 1$$

and we know $S_m = x + \delta = \gamma_i$. So by (11) the response time of τ_i is bounded by

$$R \leq \frac{C_i + (\gamma_i - 1)L_i}{\gamma_i}$$

In order to prove τ_i is schedulable, it is sufficient to prove

$$\frac{C_i + (\gamma_i - 1)L_i}{\gamma_i} \leq D_i$$

which must be true by the definition of γ_i. □

In summary, by Lemma 1 and 2 we know using x container tasks with load bound 1 and *one* container task with a fractional load bound requires the minimal processing capacity, which motivates our first scheduling algorithm SF[x+1].

The pseudo-code of SF[x+1] is shown in Algorithm 2. The rules of SF[x+1] can be summarized as follows:

Algorithm 2 The first semi-federated algorithm: SF[x+1].

```
1: for each heavy task τ_i do
2:    γ_i = (C_i − L_i)/(D_i − L_i)
3:    if less than ⌊γ_i⌋ avaiable processors then
4:       return failure
5:    end if
6:    assign ⌊γ_i⌋ dedicated processors to τ_i
7:    create a container task with load bound γ_i − ⌊γ_i⌋ for τ_i
8: end for
9: Ω = the set of remaining processors
10: S = the set of container tasks ∪ the set of light tasks
11: if Sched(S, Ω) then return success else return failure
```

- Similar to the federated scheduling, SF[x+1] also classifies the DAG tasks into heavy tasks (density > 1) and light tasks (density ≤ 1).
- For each heavy task τ_i, we grant $\lfloor \gamma_i \rfloor$ dedicated processors and one container task with load bound $\gamma_i - \lfloor \gamma_i \rfloor$ to it where $\gamma_i = \frac{C_i - L_i}{D_i - L_i}$ (line 2 to 7). The algorithm declares a failure if some heavy tasks cannot get enough dedicated processors.
- After granting dedicated processors and container tasks to all heavy tasks, the remaining processors will be used to schedule the light tasks and container tasks. The function Sched(S, Ω) (in line 11) returns the the schedulability testing result of the task set consisting of light tasks and container tasks on processors in Ω.

Various multiprocessor scheduling algorithms can be used to schedule the light tasks and container tasks, such as global EDF and partitioned EDF. In this work, we choose to use partitioned EDF, and in particular with the Worst-Fit packing strategy [8], to schedule them.

More specificly, at design time, the light tasks and container tasks are partitioned to the processors in Ω. Tasks are partitioned in the non-increasing order of their load (the load of a light task τ_i equals its density C_i/D_i, and the load of a container task φ_p equals its load bound δ_p). At each step the processor with the minimal total load of currently assigned tasks is selected, as long the total load of the processor after accommodating this task still does not exceed 1. Sched(S, Ω) returns *true* if all tasks are partitioned to some processors, and returns *false* otherwise.

At runtime, the jobs of tasks partitioned to each processor are scheduled by EDF. Each light task behaves as a standard sporadic task. Each container task behaves as a GMF (general multi-frame) task [9]: when a container task φ_p starts to be occupied by a vertex v, φ_p releases a job with WCET $c(v)$ and an absolute deadline d_p calculated by Algorithm 1. Although a container task φ_p releases different types of jobs, its load is bounded by δ_p as the *density* of each of its jobs is δ_p.

Appendix-C presents an example to illustrate SF[x+1].

Recall that in the runtime dispatching, a vertex may be split into two parts, in order to guarantee a "faster" container task is never empty when a "slower" one is occupied. The following

theorem bounds the number of extra vertices created due to the splitting in SF[x+1].

Theorem 3. *Under* SF[x+1]*, the number of extra vertices created in each DAG task is bounded by the number of vertices in the original DAG.*

Proof. Let N be the number of vertices in the original DAG. According to Algorithm 1, a vertex will not be split if it is dispatched to a dedicated processor (i.e., a container task with load bound 1). The number of vertices executed on these dedicated processors is at most N. A vertex my be split when being dispatched to the container task with a fractional load bound, and upon each splitting, the deadline of the first part must align with some vertices on the dedicated processors, so the number of splitting is bounded by N. □

B. The Second Algorithm: SF[x+2]

In partitioned EDF, "larger" tasks in general lead to worse resource waste. The system schedulability can be improved if tasks can be divided into small parts. In SF[x+1], each heavy task is granted several dedicated processors and *one* container task with fractional load bound. The following examples shows we can actually divide this container task into *two* smaller ones without increasing the total processing capacity requirement.

Consider the DAG task in Figure 2, the minimal capacity requirement of which is

$$\gamma_i = \frac{C_i - L_i}{D_i - L_i} = \frac{16-8}{14-8} = \frac{4}{3}$$

Accordingly, SF[x+1] assigns one dedicated processor and one container task with load bound $\frac{1}{3}$ to this task.

Now we replace the container task with load bound $\frac{1}{3}$ by two container tasks with load bounds $\frac{1}{4}$ and $\frac{1}{12}$. After that, the total capacity requirement is unchanged since $\frac{1}{3} = \frac{1}{4} + \frac{1}{12}$, and the DAG task is still schedulable since the uniformity of both $\{1, \frac{1}{3}\}$ and $\{1, \frac{1}{4}, \frac{1}{12}\}$ is $\frac{1}{3}$.

However, in general dividing a container task into two may increase the uniformity. For example, if we divide the container task in the above example into two container tasks both with load bound $\frac{1}{6}$, the uniformity is increased to 1 and the DAG task is not schedulable. The following lemma gives the condition for dividing one container task into two without increasing the uniformity:

Lemma 3. *A heavy task τ_i with minimal capacity requirement γ_i is scheduled on $\lfloor \gamma_i \rfloor$ dedicated processors and two container tasks with load bounds δ' and δ'' s.t.*

$$\delta' + \delta'' = \gamma_i - \lfloor \gamma_i \rfloor$$

τ_i is schedulable if

$$\delta' \geq \max\left(\frac{\gamma_i - \lfloor \gamma_i \rfloor}{2}, \frac{\gamma_i - \lfloor \gamma_i \rfloor}{\gamma_i}\right) \tag{14}$$

Proof. By Theorem 2 we know the response time of τ_i is bounded by

$$R \leq \frac{C_i + \lambda L_i}{S_m} \tag{15}$$

Since $\delta' + \delta'' = \gamma_i - \lfloor \gamma_i \rfloor$ and $\delta' \geq (\gamma_i - \lfloor \gamma_i \rfloor)/2$, we know $\delta' \geq \delta''$. So we can calculate λ of $\lfloor \gamma_i \rfloor$ dedicated processors and two container tasks with load bounds δ' and δ'' by:

$$\begin{aligned}\lambda &= \max_{x=1}^{m}\left\{\frac{S_m - S_x}{\delta_x}\right\} \\ &= \max\left(\frac{\gamma_i - 1}{1}, \frac{\gamma_i - 2}{1}, \cdots, \frac{\gamma_i - \lfloor \gamma_i \rfloor}{1}, \frac{\delta''}{\delta'}, \frac{0}{\delta''}\right) \\ &= \max\left(\frac{\gamma_i - 1}{1}, \frac{\delta''}{\delta'}\right)\end{aligned} \tag{16}$$

By $\delta' + \delta'' = \gamma_i - \lfloor \gamma_i \rfloor$ and $\delta' \geq \frac{\gamma_i - \lfloor \gamma_i \rfloor}{\gamma_i}$ we get $\frac{\delta''}{\delta'} \leq \gamma_i - 1$. Applying this to (16) gives $\lambda = \gamma_i - 1$. Moreover, we know $S_m = \lfloor \gamma_i \rfloor + \delta' + \delta'' = \gamma_i$. Therefore, we have

$$R \leq \frac{C_i + \lambda L_i}{S_m} = \frac{C_i + (\gamma_i - 1)L_i}{\gamma_i}$$

and by the definition of γ_i in (13) we know

$$\frac{C_i + (\gamma_i - 1)L_i}{\gamma_i} = D_i$$

so we can conclude $R_i \leq D_i$, and thus τ_i is schedulable. □

Based on the above discussions, we propose the second federated scheduling algorithm SF[x+2]. The overall procedure of SF[x+2] is similar to SF[x+1]. The only difference is that SF[x+2] uses Sched$^*(S, \Omega)$ to replace Sched(S, Ω) in line 11 of Algorithm 2. The pseudo-code of Sched$^*(S, \Omega)$ is given in Algorithm 3. The inputs of Sched* are S, the set of sequential tasks (including the generated container tasks and the light tasks), and Ω, the remaining processors to be shared by these sequential tasks.

There are infinitely many choices to divide a container task into two under the condition of Lemma 3. Among these choices, on one simply dominates others, since the quality of a choice depends on how the tasks are partitioned to processors. In Sched$^*(S, \Omega)$, the container tasks are divided in an on-demand manner. Each container task φ_k of task τ_i, apart from its original load bound δ_k, is affiliated with a δ_k^*, representing the minimal load bound of the larger part if φ_k is divided into two parts. δ_k^* is calculated according to Lemma 3:

$$\delta_k^* = \max\left(\frac{\gamma_i - \lfloor \gamma_i \rfloor}{2}, \frac{\gamma_i - \lfloor \gamma_i \rfloor}{\gamma_i}\right) \tag{17}$$

For consistency, each light task τ_j is also affiliated with a δ_j^* which equals to its density $\delta_j = C_i/D_i$.

Sched$^*(S, \Omega)$ works in three steps:

1) It first partitions all the input container tasks and light tasks using the Worst-First packing strategy using their δ_k^* as the metrics. We use $\varphi(P_x)$ to denote the set of tasks have been assigned to processor P_x. If the sum of δ_k of all tasks in $\varphi(P_x)$ has exceeded 1, we stop assigning tasks to P_x and move it to the set Ψ.
2) The total δ_k of tasks on each processor P_x in Ψ is larger than 1, therefore some of tasks on P_x must be divided into two, and one of them should be assigned to other processors. On the other hand, the total δ_k^* of some tasks

on P_x is no larger than 1, which guarantees that we can divide tasks on P_x to reduce its total δ_k to 1. The function Scrape(P_x) divides container tasks on P_x and make the total load of P_x to be exactly 1 and returns the newly generated container tasks. The pseudo-code of Scrape(P_x) is shown in Algorithm 4.

3) Finally, Partition(S, Ω) partitions all the generated container tasks in step 2) to the processors remained in Ω using the Worst-Fit packing strategy. After the first step, the total load of tasks on processors remained in Ω is still smaller than 1, i.e., they still have remaining available capacity and potentially can accommodate more tasks. Partition(S, Ω) returns *true* if tasks in S can be successfully partitioned to processors remained in Ω, and returns *false* otherwise.

Algorithm 3 Sched*(S, Ω) in SF[x+2].

```
Sort elements in S in non-increasing order of their δ*_i
Ψ = ∅
for each sequential task φ_k (including both container tasks and light tasks) do
    P_x = a processor in Ω with the minimal Σ_{φ_i∈φ(P_x)} δ*_i and satisfying
                δ*_k + Σ_{φ_i∈φ(P_x)} δ*_i <= 1
    if P_x = NULL then return failure;
    φ(P_x) = φ(P_x) ∪ {φ_k}
    if Σ_{φ_i∈φ(P_x)} δ_i > 1 then move P_x from Ω to Ψ
end for
S = ∅
for each core P_x ∈ Ψ do
    S = S ∪ Scrape(P_x);
end for
if Partition(S, Ω) then return success else return failure
```

Algorithm 4 Scrape(P_x).

```
SS = ∅
w = Σ_{φ_k∈φ(P_x)} δ_k − 1
for each container task φ_k ∈ φ(P_x) do
    if δ_k − δ*_k > w then
        divide φ_k into φ'_k and φ''_k such that
                δ''_k = w ∧ δ'_k = δ_k − δ''_k
        put φ''_k in SS (φ'_k still assigned to P_x)
        return SS
    else
        divide φ_k into φ'_k and φ''_k such that
                δ'_k = δ*_k ∧ δ''_k = δ_k − δ*_k
        put φ''_k in SS (φ'_k still assigned to P_x)
        w = w − δ''_k
    end if
end for
```

Appendix-C includes an example to illustrate SF[x+2].

The number of extra vertices created by runtime dispatching of each DAG task in SF[x+2] is bounded as follows.

Theorem 4. *Under* SF[x+2]*, the number of extra vertices created in each DAG task is bounded by* $2N$*, where* N *is the number of vertices in the original DAG.*

The intuition of the proof is similar to that of Theorem 3. The difference is that SF[x+2] uses two container tasks, so the number of splitting is doubled in the worst-case. A complete proof of the theorem is provided in Appendix-D.

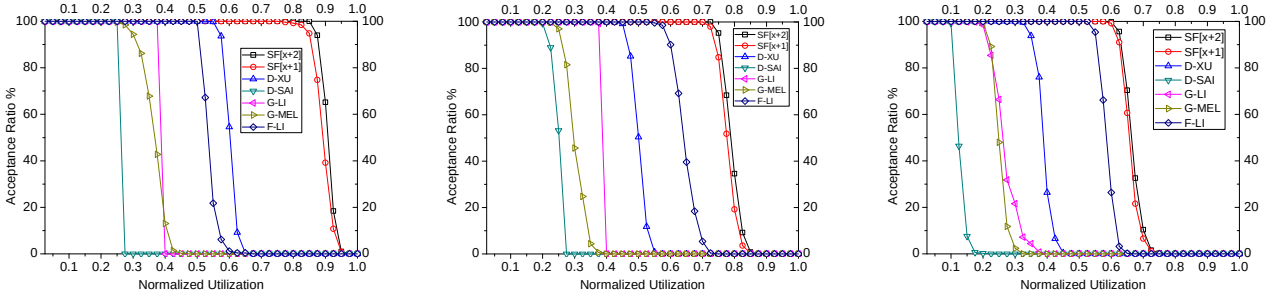
VI. Performance evaluations

In this section, we evaluate the performance of the proposed semi-federated algorithms. We compare the acceptance ratio of SF[x+1] and SF[x+2]with the state-of-the-art algorithms and analysis methods in all the three types of parallel real-time task scheduling algorithms:

- **Decomposition-based scheduling**: (i) The EDF-based scheduling and analysis techniques developed in [10], denoted by D-SAI. (ii) The EDF-based scheduling and analysis techniques in [11], denoted by D-XU.
- **Global scheduling**: (i) The schedulability test based on capacity augmentation bounds for global EDF scheduling in [1], denoted by G-LI. (ii) The schedulability test based on response time analysis for global EDF scheduling in [3], denoted by G-MEL. G-MEL was developed for a more general DAG model with conditional branching, but can be directly applied to the DAG model of this paper, which is a special case of [3].
- **Federated scheduling**: the schedulability test based on the processor allocation strategy in [1], denoted by F-LI.

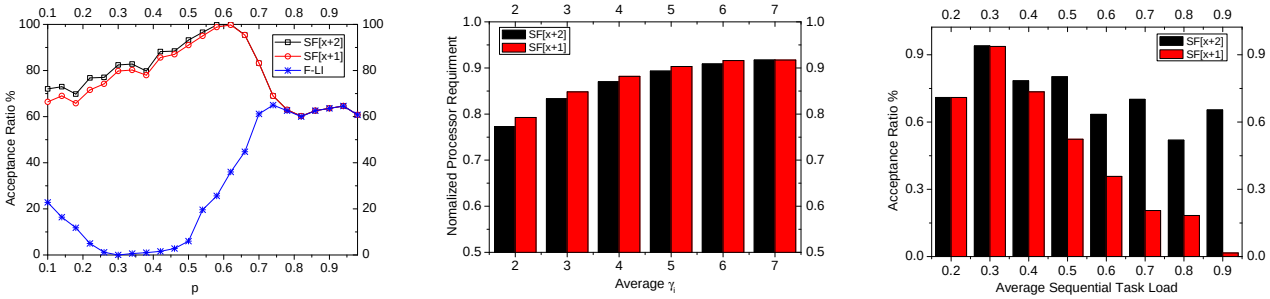
Other methods not included in our comparison are either theoretically dominated or shown to be significantly outperformed (with empirical evaluations) by one of the above methods.

The task sets are generated using the Erdös-Rényi method $G(n_i, p)$ [12]. For each task, the number of vertices is randomly chosen in the range $[50, 250]$ and the worst-case execution time of each vertex is randomly picked in the range $[50, 100]$, and a valid period is generated according to a similar method with [10]. The period T_i is set to be $(L_i + \frac{C_i}{0.4m*U}) * (1 + 0.25 * Gamma(2,1))$ where $Gamma(2,1)$ is a random value by using gamma distribution and U is the normalized utilization of the task set (total utilization divided by the number of processors m). In this way, we can: (i) make a valid period, (ii) generate a reasonable number of tasks when the processor number and total utilization of the task sets change. For each possible edge we generate a random value in the range $[0, 1]$ and add the edge to the graph only if the generated value is less than a predefined threshold p. In general the critical path of a DAG generated using the Erdös-Rényi method becomes longer as p increases, which makes the task more sequential. We compare the *acceptance ratio* of each method, which is the ratio between the number of task sets deemed to be schedulable by a method and the total number of task sets in the experiments (of a specific group). For each parameter configuration, we generate 10000 task sets.



(a) $m = 8$ (b) $m = 16$ (c) $m = 32$

Fig. 6. Comparing SF[x+1] and SF[x+2] with the state-of-the-art with different number of processors.



(a) Comparison with different p. (b) Comparison with different average sequtial task load. (c) Comparison with different average γ_i.

Fig. 7. Comparing SF[x+1] and SF[x+2] with federated scheduling in different dimensions.

Figure 6 compares the acceptance ratios of different methods with fixed $p = 0.1$ and different number of processors. In each figure, the experiment results are grouped by the normalized utilization task set (x-axis). We can see that our two semi-federated scheduling algorithms significantly outperform all the state-of-the-art methods in different categories.

Then we made in-depth comparison between federated scheduling (F-LI) and our semi-federated scheduling algorithms. Figure 7-(a) shows the acceptance ratio with $m = 16$ and different p values (x-axis). We can see that semi-federated scheduling significantly outperforms federated scheduling except when p is large, i.e., when tasks are very sequential. In the extreme case, when tasks are all sequential, both federated and semi-federated scheduling degrade to traditional multiprocessor scheduling of sequential tasks.

Figure 7-(b) compares the minimal number of processors required by the federated scheduling and semi-federated scheduling algorithms to make the task set schedulable. In these experiments we set $p = 0.1$. The experiment results are grouped by the average minimal capacity requirement γ_i of all heavy tasks in a task set. A value x on the x-axis represents range $(x - 1, x]$. The y-axis is the average ratio between the minimal number of processors required by SF[x+1](SF[x+2]) and the minimal number of processors required by F-LI, to make the task set schedulable. We can see the resource saving by SF[x+1](SF[x+2]) is more significant when γ_i is smaller.

Figure 7-(c) compares our two semi-federated scheduling algorithms, in which all task sets have a fixed total normalized utilization 0.9, and we set $m = 16$ and $p = 0.1$. The experiment results are grouped by the average load of the sequential tasks (container tasks with fractional load bounds and light tasks) participating the partitioning on the shared processors (i.e., tasks in S for Sched(S, Ω) and Sched*(S, Ω)). A value x on the x-axis represents range $(x - 0.1, x]$. As expected, when the task sizes are larger, the performance of SF[x+1] degrades. SF[x+2] maintains good performance with large tasks since dividing a large container task into two significantly improves resource utilization.

VII. Related Work

Early work on real-time scheduling of parallel tasks assume restricted constraints on task structures [13]–[22]. For example, a Gang EDF scheduling algorithm was proposed in [15] for moldable parallel tasks. The parallel synchronous task model was studied in [16]–[22]. Real-time scheduling algorithms for DAG tasks can be classified into three paradigms: (i) decomposition-based scheduling [10], [11], [23], [24], (ii) global scheduling (without decomposition) [3], [25], [26] , and (iii) federated scheduling [1], [27]–[29] .

The decomposition-based scheduling algorithms transform each DAG into several sequential sporadic sub-tasks and schedule them by traditional multiprocessor scheduling algorithms. In [10], a capacity augmentation bound of 4 was proved for global EDF. A schedulability test in [23] was provided to achieve a lower capacity augmentation bound in most cases, while in other cases above 4. In [24], a capacity augmentation

bound of $\frac{3+\sqrt{5}}{2}$ was proved for some special task sets. In [11], a decomposition strategy exploring the structure features of the DAG was proposed, which has capacity augmentation bound between 2 and 4, depending on the DAG structure.

For global scheduling (without decomposition), a resource augmentation bound of 2 was proved in [30] for a single DAG. In [25], [31], a resource augmentation bound of $2-1/m$ and a capacity augmentation bound of $4-2/m$ were proved under global EDF. A pseudo-polynomial time sufficient schedulability test was presented in [25], which later was generalized and dominated by [26] for constrained deadline DAGs. [31] proved the capacity augmentation bound $\frac{3+\sqrt{5}}{2}$ for global and 3.732 for global RM. In [32] a schedulability test for arbitrary deadline DAG was derived based on response-time analysis.

For federated scheduling, [1] proposed an algorithm for DAGs with implicit deadline which has a capacity augmentation bound of 2. Later, federated scheduling was generalized to constrained-deadline DAGs [27], arbitary-deadline DAGs [28] as well as DAGs with conditional branching [29].

The scheduling and analysis of sequential real-time tasks on *uniform* multiprocessors was studied in [7], [33], [34]. Recently, Yang and Anderson [35] investigated global EDF scheduling of npc-sporadic (no precedence constraints) tasks on uniform multiprocessor platform. This study was later extended to DAG-based task model on heterogeneous multiprocessors platform in [36] where a release-enforcer technique was used to transformed a DAG-based task into several npc-sporadic jobs thus eliminating the intra precedence constraints and provide analysis upon the response time.

VIII. CONCLUSIONS

We propose the semi-federate scheduling approach to solve the resource waste problem of federated scheduling. Experiments results show significantly performance improvements of our approach comparing with the state-of-the-art for scheduling parallel real-time tasks on multi-cores. In the next step, we will integrate our approach with the work-stealing strategy [37] to support hight resource utilization with both hard real-time and soft real-time tasks at the same time.

REFERENCES

[1] 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 *ECRTS*, 2014.

[2] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, "The global edf scheduling of systems of conditional sporadic dag tasks," in *ECRTS*, 2015.

[3] 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.

[4] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM journal on Applied Mathematics*, 1969.

[5] S. Baruah, "Techniques for multiprocessor global schedulability analysis," *RTSS*, 2007.

[6] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," *RTSS*, 2005.

[7] S. Funk, J. Goossens, and S. Baruah, "On-line scheduling on uniform multiprocessors," in *RTSS*, 2001.

[8] 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*, 1974.

[9] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, 1999.

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

[11] X. Jiang, X. Long, N. Guan, and H. Wan, "On the decomposition-based global edf scheduling of parallel real-time tasks," in *RTSS*, 2016.

[12] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *ICST*, 2010.

[13] G. Manimaran, C. S. R. Murthy, and K. Ramamritham, "A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems," *Real-Time Systems*, 1998.

[14] W. Y. Lee and L. Heejo, "Optimal scheduling for real-time parallel tasks," *IEICE transactions on information and systems*, 2006.

[15] S. Kato and Y. Ishikawa, "Gang edf scheduling of parallel task systems," in *RTSS*, 2009.

[16] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *RTSS*, 2010.

[17] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, 2013.

[18] 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 *ICCPS*, 2013.

[19] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *ECRTS*, 2012.

[20] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *RTNS*, 2014.

[21] B. Andersson and D. de Niz, "Analyzing global-edf for multiprocessor scheduling of parallel tasks," in *OPODIS*, 2012.

[22] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig, "Response-time analysis of parallel fork-join workloads with real-time constraints," in *ECRTS*, 2013.

[23] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, "Global edf scheduling of directed acyclic graphs on multiprocessor systems," in *RTNS*, 2013.

[24] M. Qamhieh, L. George, and S. Midonnet, "A stretching algorithm for parallel real-time dag tasks on multiprocessor systems," in *RTNS*, 2014.

[25] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic dag task model," in *ECRTS*, 2013.

[26] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic dag task systems," in *ECRTS*, 2014.

[27] ——, "The federated scheduling of constrained-deadline sporadic dag task systems," in *DATE*, 2015.

[28] ——, "Federated scheduling of sporadic dag task systems," in *IPDPS*, 2015.

[29] ——, "The federated scheduling of systems of conditional sporadic dag tasks," in *EMSOFT*, 2015.

[30] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*, 2012.

[31] J. Li, K. Agrawal, C. Lu, and C. Gill, "Outstanding paper award: Analysis of global edf for parallel tasks," in *ECRTS*, 2013.

[32] A. Parri, A. Biondi, and M. Marinoni, "Response time analysis for g-edf and g-dm scheduling of sporadic dag-tasks with arbitrary deadline," in *RTNS*, 2015.

[33] S. Funk and S. Baruah, "Characteristics of edf schedulability on uniform multiprocessors," *ECRTS*, 2003.

[34] S. Baruah and J. Goossens:, "The edf scheduling of sporadic task systems on uniform multiprocessors," *RTSS*, 2008.

[35] K. Yang and J. H. Anderson, "Optimal gedf-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors," in *ESTIMedia*, 2014.

[36] K. Yang, M. Yang, and J. H. Anderson, "Reducing response-time bounds for dag-based task systems on heterogeneous multicore platforms," in *RTNS*, 2016.

[37] J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill, and C. Lu, "Randomized work stealing for large scale soft real-time systems," in *RTSS*, 2016.

APPENDIX-A: RESPONSE TIME BOUNDS WITHOUT INTER-PROCESSOR MIGRATION

The work-conserving scheduling rules for uniform multiprocessors in Section III-B requires the vertices to migrate from slower processors to faster processors whenever possible. If such migration is forbidden, the resource may be significantly wasted and the response time can be much larger. We say a scheduling algorithm is *weakly work-conserving* if only the first work-conserving rule in Section III-B is satisfied and a vertex is not allowed to migrate from one processor to another. The response time of a DAG task under weakly conserving scheduling is bounded by the following theorem:

Theorem 5. *Give m uniform processors with speeds $\{\delta_1, \delta_2, ..., \delta_m\}$ (sorted in non-increasing order. The response time of a DAG task τ_i by a weakly work-conserving scheduling algorithm is bounded by*

$$R \leq \frac{L_i}{\delta_m} + \frac{C_i - L_i}{\sum_{i=1}^{m} \delta_i} \tag{18}$$

Proof. Without loss of generality, we assume the task under analysis releases an instance at time 0, and thus R is the time point when the currently release of π is finished. In the time window $[0, R]$, let α denote the total length of intervals during which the at least one processor is idle and β denote the total length of the intervals during which all processors are busy. Therefore, we know $R = \alpha + \beta$. Let π be an arbitrary path in the DAG starting from the head vertex and ending at the tail vertex. We use $\gamma(\pi)$ to denote the total amount of workload executed for vertices along path π in all the time intervals of α. The total work done in all time interval of β is $C_i - \gamma(\pi)$. Since at least one processor is idle in time intervals of α, π must contain a vertex being executed in these time intervals (since at any time point before R, there is at least one eligible vertex along any path) and δ_m is the speed of the slowest processor. Therefore, we know:

$$\alpha \leq \frac{\gamma(\pi)}{\delta_m}$$

As the total work been done in all time interval of β (where all processors are busy) is at most $C_i - \gamma(\pi)$, we have

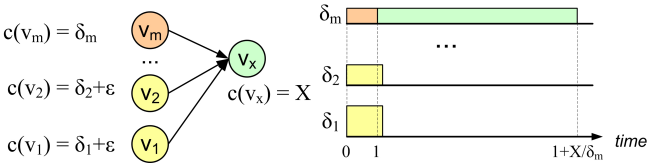
$$\begin{aligned} & \beta S_m \leq C_i - \gamma(\pi) \\ \Leftrightarrow \qquad & \beta \leq \frac{C - \gamma(\pi)}{S_m} \end{aligned}$$

Hence we have

$$R = \alpha + \beta \leq \frac{\gamma(\pi)}{\delta_m} + \frac{C_i - \gamma(\pi)}{S_m} \tag{19}$$

Let l_π denote the total workload (of all vertices) along path π, so we know $\gamma(\pi) \leq l_\pi$. Since L_i is the total workload of the longest path in the DAG, we know $l_\pi \leq L_i$, in summary we have $\gamma(\pi) \leq L_i$ and applying this to (19) concludes the lemma. □

Theorem 6. *The response time bound in (18) is tight for weakly work-conserving scheduling algorithms.*



(a) A DAG task example. (b) Scheduling sequence.

Fig. 8. Scheduling of a DAG task without migration.

Proof. The tightness is witnessed by the example in Figure 8. Let R' denote its actual response time of the task in the scheduling sequence in Figure 8-(b), and R'' the response time bound in 18. R'/R'' approaches 1 as X approaches infinity. □

As illustrated in Figure 8, if inter-processor migration is forbidden, the response time is almost the same as executing the critical path on the slowest processor, while the faster processors are all wasted.

APPENDIX-B: DETAILED EXPLANATION OF THE EXAMPLE IN FIGURE 5

- At time $t = 0$, the only eligible vertex v_1 starts to occupy the "fastest" container task φ_1, and its absolute deadline is set to be $d_1 = 0 + 1/1 = 1$.
- At time $t = 1$, φ_1 becomes empty, and v_2, v_3 and v_4 become eligible. Suppose we first select v_4 to execute on φ_1, with $d_1 = 1 + 4/1 = 5$. After that, Algorithm 1 is invoked again to assign a container task to the next eligible vertex v_3. If we encapsulate the entire v_3 into φ_2, then the resulting absolute deadline $1 + 3/0.5 = 7$ is later than the absolute deadline of a "faster" container task φ_1's deadline $d_1 = 5$. Therefore, we must split v_3 into v_3' and v_3'', so that putting v_3' into φ_2 results in the same absolute deadline as φ_1, and v_3'' is put back to S for further consideration. Next, Algorithm 1 is invoked again to assign the only eligible vertex v_2 to the remaining container task φ_3. Similarly, v_2 cannot be put into φ_3 entirely, and we split it into v_2' and v_2'' so that $d_3 = 5$.
- At time $t = 5$, all the container tasks reach their absolute deadlines and thus becomes empty, and currently only v_2'' and v_3'' are eligible. Suppose we first choose to put v_2'' into φ_1 with $d_1 = 5 + 4/1 = 9$, then put v_3'' into φ_2 with $d_2 = 5 + 1/0.5 = 7$, which is smaller than d_1.
- At time $t = 7$, φ_2 reaches its absolute deadline and thus become empty, and v_5 become eligible and should be put into φ_2 (φ_1 is still being occupied). v_5 also needs to be split into two parts $c(v_5') = c(v_5'') = 1$ to make $d_2 = d_1 = 9$.
- At time $t = 9$, both φ_1 and φ_2 become empty and v_5'' become eligible, which is put into φ_1 with $d_1 = 10$.
- At time 10, the execution of v_5'' on φ_1 is finished and the last vertex v_6 is put into the fastest container task φ_1.
- At time 11, the entire task is finished.

APPENDIX-C: ILLUSTRATION OF SF[x+1] AND SF[x+2]

We use the following example to illustrate SF[x+1]. Assume a task set consists of 4 DAG tasks, where the first three are heavy, with the minimal capacity requirements $\gamma_1 = 1.6$, $\gamma_2 = 1.6$ and $\gamma_3 = 1.5$, and one light task with density $\gamma_4 = 0.3$. If scheduled by standard federated scheduling, each of the three heavy tasks requires 2 dedicated processors, and in total 7 processor are needed. If scheduled by SF[x+1], each of the heavy task only requires one dedicated processors, and they generate three container tasks, with load bounds 0.6, 0.6 and 0.5. These three container tasks, together with the light tasks with density 0.3 is schedulable by partitioned EDF on 3 processors, so in total 6 processors are needed to schedule the task set using SF[x+1].

We use the same task set as above to illustrate SF[x+2]. Now we assume the tasks are scheduled on 5 processors. Since each heavy task is granted one dedicated processor, the container tasks and light task share 2 processors. The load bound of the three generated container tasks and the density of the light tasks are

$$\delta_1 = 0.6,\ \delta_2 = 0.6,\ \delta_3 = 0.5,\ \delta_4 = 0.3$$

We can compute δ_k^* for each task using (17):

$$\delta_1^* = \frac{3}{8},\ \delta_2^* = \frac{3}{8},\ \delta_3^* = \frac{1}{3},\ \delta_4^* = 0.3 \tag{20}$$

The algorithm Sched$^*(S, \Omega)$ works as follows:

1) φ_1 is assigned to an empty processor P_1.
2) φ_2 is assigned to the other empty processor P_2.
3) To assign φ_3, both processors are holding a task with the same load, so we choose any of them, say P_1, to accommodate φ_3. Since $\delta_1^*+\delta_3^* = 3/8+1/3 < 1$, we can assign φ_3 to P_1. After that, since $\delta_1+\delta_3 = 0.6+0.5 > 1$, P_1 is moved from Ω to Ψ.
4) To assign φ_4, since
$$\sum_{\varphi_i \in \varphi(P_1)} \delta_i^* = 3/8 + 1/3 \quad > \sum_{\varphi_i \in \varphi(P_2)} \delta_i^* = 3/8$$
we try to assign φ_4 to P_2. Since $\delta_2^*+\delta_4^* = 1/3+0.3 < 1$, we can assign φ_4 to P_2. After that, since $\delta_2 + \delta_4 = 0.6 + 0.3 < 1$, P_2 remains in Ω.
5) After assigning all the four tasks, only P_1 is in Ψ. So we execute Scrape(P_1). $w = \delta_1 + \delta_3 - 1 = 0.1$. Since $\delta_1 - \delta_1^* = 0.3 - 3/8 > 0.1$, so we divide φ_1 into $\varphi_1^{'}$ and $\varphi_1^{''}$ where $\delta_1^{''} = 0.1$ and $\delta_1^{'} = 0.6 - 0.1 = 0.5$, and put $\varphi_1^{''}$ in SS.
6) There is only one processor P_2 in Ω, since
$$\sum_{\varphi_i \in \varphi(P_1)} \delta_i + \delta_1^{''} = 0.6 + 0.3 + 0.1 = 1$$
we put $\varphi_1^{''}$ is put in P_2.

Therefore, the final result of Sched$^*(S, \Omega)$ is

$$P_1:\ \delta_1^{'} = \frac{1}{2},\quad \delta_3 = \frac{1}{2}$$
$$P_2:\ \delta_2 = \frac{3}{5},\quad \delta_4 = \frac{3}{10},\quad \delta_1^{''} = \frac{1}{10}$$

APPENDIX-D: PROOF OF THEOREM 4

Proof. Let a task execute on several dedicated processors and two fractional container tasks despite the unit containers with density of $\delta^{'}$ and $\delta^{''}$, $\delta^{'} \geq \delta^{''}$. By the proof of Theorem 3 we know the number of splitting occurred on the container task δ' is at most N. In the following we prove the number of splitting on the container task δ'' is also at most N.

We use A to denote the set of vertices (including the parts of the divided vertices) executed on dedicated processors, and use B to denote the set of vertices (parts) executed on container task δ' with a deadline different from any deadlines of vertices (parts) on the dedicated processors. If a vertex v is divided into two parts, v', executed on the container task δ', and v'', executed on dedicated processors. The migration of v must happens at a time point aligned with some deadline on the dedicated processors, so we know v' must not be in B. Moreover, according to Algorithm 1, the vertices assigned to dedicated processors will not migrate to other processors. Therefore, the total number of elements in $A \cup B$ is at most N. Therefore, the number of time points aligned with deadlines of vertices (parts) executed on the dedicated processors and container task δ' is bounded by N. Since a splitting on container task δ'' only occurs at time points aligned with deadlines of vertices (parts) executed on the dedicated processors and container task δ', we can conclude the number of splitting on container task δ'' is also bounded by N.

In summary, the total number of vertices splitting all the two container tasks is bounded by $2N$. Since the vertices assigned to dedicated processors will not migrate to other processors. Therefore, the total number of newly generated vertices is bounded by $2N$. □