A *Generalized Packing Server* for Scheduling Task Graphs on Multiple Resources

Shen Li *, Xiaoxiao Wang *, Shaohan Hu *, Yiran Zhao *, Shiguang Wang *, Lu Su †, Tarek Abdelzaher *
**University of Illinois at Urbana-Champaign*, †*State University of New York at Buffalo*
{*shenli3, xwang104, shu17, zhao97, swang83, zaher*}@*illinois.edu, lusu@buffalo.edu*

Abstract*—This paper presents the *generalized packing server*. It reduces the problem of scheduling tasks with precedence constraints on multiple processing units to the problem of scheduling independent tasks. The work generalizes our previous contribution made in the specific context of scheduling Map/Reduce workflows. The results apply to the generalized parallel task model, introduced in recent literature to denote tasks described by workflow graphs, where some subtasks may be executed in parallel subject to precedence constraints. Recent literature developed schedulability bounds for the generalized parallel tasks on multiprocessors. The generalized packing server, described in this paper, is a run-time mechanism that packs tasks into server budgets (in a manner that respects precedence constraints) allowing the budgets to be viewed as independent tasks by the underlying scheduler. Consequently, any schedulability results derived for the *independent task model* on multiprocessors become applicable to *generalized parallel tasks*. The catch is that the sum of capacities of server budgets exceeds by a certain ratio the sum of execution times of the original generalized parallel tasks. Hence, a scaling factor is derived that converts bounds for independent tasks into corresponding bounds for generalized parallel tasks. The factor applies to any work-conserving scheduling policy in both the global and partitioned multiprocessor scheduling models. We show that the new schedulability bounds obtained for the generalized parallel task model, using the aforementioned conversion, improve in several cases upon the best known bounds in current literature. Hence, the packing server is shown to improve the schedulability of generalized parallel tasks. Evaluation results confirm this observation.

I. Introduction

This paper introduces the *generalized packing server*, which allows converting generalized parallel tasks into a corresponding set of independent tasks for the underlying scheduler. Hence, any schedulability results, previously derived for independent tasks on multiprocessors become applicable to analyze the schedulability of generalized parallel tasks.

The generalized parallel task model was studied extensively in recent literature [1–8]. It refers to tasks that are described by workflow graphs, where some subtasks can execute in parallel subject to precedence constraints. The new server packs computation time of these tasks into budgets, while respecting precedence constraints. The resulting budgets, however, can be treated by the underlying scheduler as *independent tasks*. To ensure that the packing server can always successfully pack the original task set into budgets, we show that the sum of the budgets should be larger by a certain factor than the sum of the original computation times. As a result, a conversion factor is derived between schedulability bounds derived in prior literature for independent tasks on multiprocessors (that therefore apply to the budgets) and the corresponding bounds that apply to the schedulability of generalized parallel tasks. The new bounds derived using the aforementioned conversion approach for the generalized parallel tasks are shown to be better in many cases than the corresponding best-known bounds for this task model.

The paper extends our previous work [9] that specifically addresses the scheduling problem in the context Map/Reduce systems. The previous approach required that the packing server simulate the exact future task execution timeline by the underlying scheduler, in order to determine the exact time intervals when one of the budgets will be scheduled on one of the resources in the future. This need for simulation made the previous approach of limited applicability. In particular, it applied well to Map/Reduce scheduling, where schedulers are heavy-weight application-layer entities burdened by layers of cloud middleware (making the added overhead of the aforementioned simulation negligible). However, in the context of embedded computing, schedulers must be lighweight and efficient. The generalized packing server, described in this paper, no longer requires simulation. Instead, it can use *any work-conserving scheduling policy* to pack tasks into server budgets. Hence, results derived for this server are much more broadly applicable.

The rest of this paper is organized as follows. Section II presents the model of generalized parallel tasks, and introduces the high-level idea of generalized packing servers. Then, technical details and theoretical guarantees are elaborated in Section III. Section IV shows evaluation results. We survey related work in Section V. Finally, Section VI summarizes the paper.

II. Overview

We first review the task model considered in this paper, in Section II-A. We then present, in Section II-B, our main result and explain the high-level intuition behind the generalized packing server. A detailed proof will follow in subsequent sections.

A. Generalized Parallel Task Model

The paper schedules generalized parallel tasks on multiprocessors composed of a number of identical processing elements (or cores). A job (i.e., an instance of occurrence) of a generalized parallel task is composed of multiple segments that together form a Directed Acyclic Graph (DAG), where nodes represent segments and edges represent precedence constraints. Each segment may be composed of mutiple parallel threads. No segment can start until all threads of all its predecessor segments are finished.

The generalized parallel task model extends a previous parallel task model described in literature, where a job consists of multiple segments, chained together into a pipeline. Each segment may contain multiple parallel threads. Precedence constraints ensure that no segments can start before all threads of its predecessor segments are finished.

More formally, in this paper, we consider a set of n generalized parallel tasks, denoted by $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, running on a platform of m cores. Each task τ_i contains a DAG of s_i segments, where the j^{th} segment consists of m_i^j threads, each with the worst case execution time (WCET) c_i^j. Different threads in the same segment may execute on different cores in parallel. A thread can be preempted, and then resumed on the same or a different core. The j^{th} segment can only start after all its predecessor segments on the DAG have finished. Every T_i time units, the task τ_i spawns a generalized parallel job with relative deadline D_i. Jobs from the same task are considered independent. The utilization u_i of task τ_i is the total amount of computation over its deadline (*i.e.*, $\frac{\sum_j m_i^j c_i^j}{D_i}$).

Let L_i denote the critical path length of task τ_i. Note that, in a parallel job, where segments form a pipeline, $L_i = \sum_j c_i^j$, summed over all segments. In a generalized parallel job, the critical path is the longest execution path in the DAG. Hence, $L_i = \max_{path_k \in DAG} \sum_{j \in path_k} c_i^j$. We define the stretch of a generalized parallel task, τ_i, denoted by φ_i, as the ratio of its relative deadline D_i over its critical path length L_i. Let φ denote the minimum stretch of all tasks (*i.e.*, $\varphi = \min\{\varphi_i | \forall i\}$). Define β as a tunable parameter that controls the upper bound of the maximum individual task utilization $u_{\max}$ in converted independent tasks, such that $u_{\max} \leq \frac{1}{\beta}$.

For simplicity, we use the term *parallel tasks* interchangeably with *pipelines*, and use the term *generalized parallel tasks* interchangeably with (DAG) *workflows*.

B. The Generalized Packing Server

To understand the intuition behind the generalized packing server, it is good to remember the literature on scheduling of aperiodic tasks. Since aperiodic tasks were harder to analyze, due to the intrinsic uncertainly regarding the time duration between two consecutive job releases, real-time scheduling literature introduced the idea of aperiodic task servers [10–12], such that aperiodic tasks can be scheduled inside these servers, while the servers themselves can be invoked periodically. Hence, scheduling and analysis techniques for periodic tasks could be extended to mixtures of periodic and aperiodic tasks.

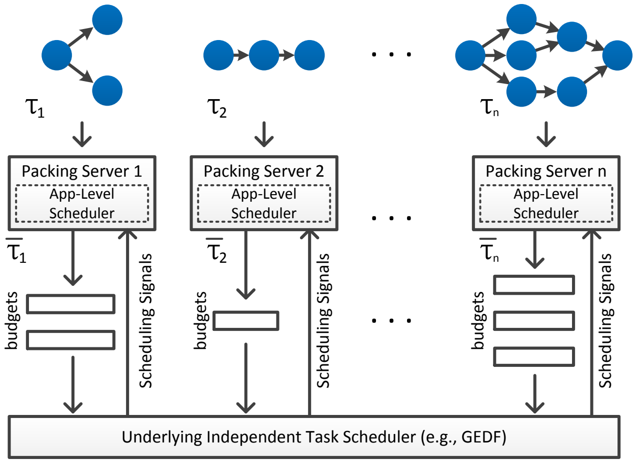


Figure 1: Packing Server Overview

The generalized packing server shares the same motivation and high-level approach with the above literature. Recent work has introduced the generalized parallel task model, drawing an increasing amount of attention to its expressiveness and broad applications [4–7]. Generalized parallel tasks are more difficult to schedule and analyze than independent tasks, due to the existence of precedence constraints. Can one design servers that offer parallel budgets, such that generalized parallel tasks can be scheduled inside those budgets, whereas the budgets themselves apprear as independent tasks to the underlying scheduler? This paper answers the aforementioned question in the affirmative. Moreover, it shows that the server *guarantees* that all generalized parallel tasks meet their deadlines as long as the independent budgets do.

In our paper, each generalized parallel task is given its own generalized packing server given by a set of budgets. We show that if these budgets are appropriately sized (to be slightly larger, in total, than the sum of execution times of the threads in the task in question), then a work-conserving scheduling policy will *always succeed* at fitting all threads of a task within the server budgets, while respecting precedence constraints, *regardless* of how these budgets are scheduled by the underlying multiprocessor scheduler. Hence, the underlying scheduler can treat these budgets as *independent* and simply ensure that the indepenent budgets meet the overall deadline of the task. This is the standard scheduling problem for independent tasks on multiprocessors. As a result, we can schedule (and analyze schedulability of) generalized parallel tasks through packing servers using existing techniques for independent tasks.

The conversion from the generalized parallel task model to independent tasks (budgets) introduces a penalty, since the sum of budgets must be larger than the computational demand of the original tasks. We quantify this penalty as the ratio between the total utilization of the original precedence-constrained task set and the resulting set of independent tasks (or, rather, budgets). We show that this ratio is equal to

$\frac{\varphi-\beta}{\varphi}$, where β and φ are as defined in Section II-A. Hence, if the utilization bound for scheduling of independent tasks by the underlying multiprocessor scheduling policy, is U_B, then the generalized parallel task set is schedulable when its utilization does not exceed $U_B \cdot \frac{\varphi-\beta}{\varphi}$. The observation allows us to map bounds for independent tasks to those for generalized parallel tasks, essentially discovering new schedulability bounds for the latter in several cases.

Figure 1 illustrates the high-level idea of the generalized packing server. The underlying scheduler runs some independent task scheduling algorithm, such as Global EDF (GEDF) [13] or EDF First Fit (EDF-FF) [14]. When the underlying scheduler executes a budget, much like the case with aperiodic task servers, the application-level scheduler of the corresponding packing server executes threads of the original workflow task within that budget using a simple work-conserving scheduling policy. The policy respects precedence constraints by assigning threads whose predecessors are finished (in some priority order) to the cores. This scheduler always succeeds at fitting all threads into the budgets, as long as the budgets are sized as described in this paper.

III. Generalized Packing Server

In this section, we first present generalized packing servers for pipelines, and then, extend it for workflows by transforming workflows into pipelines.

A pipeline can be converted into a set of identical and independent tasks using two key operations, *packing* and *inflating*. Section III-A describes the packing operation that converts a pipeline into identical budgets. Then, Section III-B introduces the inflating operation that decouples budgets into independent tasks. Based on the packing and inflating operations, we develop the conversion bound $\frac{\varphi-\beta}{\varphi}$, where β is a tunable parameter that controls the upper bound of the converted independent task utilization, such that $u_{max} \leq \frac{1}{\beta}$. The analysis also shows that all work-conserving application-level schedulers can achieve this conversion bound. In Section III-C, we further prove that no application-level scheduler can improve this conversion bound. Finally, we describe how workflows can be transformed into pipelines preserving the same conversion bound in Section III-D.

A. The Packing Operation

The packing operation packs a pipeline τ_i into a given number (x_i) of identical budgets, such that $x_i \leq \max\{m_i^j | \forall j\}$. In the next section, we will describe how to determine the value of x_i.

The intuition is that, as packing servers execute threads in budgets, using a smaller number of budgets reduces the parallelism during the execution of a pipeline, which helps avoid unnecessary resource contentions. After packing τ_i into x_i budgets, if $m_i^j \geq x_i$, segment j is called a *large segment*. Otherwise, it is a *small segment*. The packing operation concentrates the total WCET of each large segment into x_i identical budget portions, and adds $(x_i - m_i^j)$ virtual budget portions to each small segment, where *budget portions* denote the portions of budgets allocated to a specific segment. After that, every segment consists of an equal number (x_i) of (virtual) budget portions. The packing server then concatenates each (virtual) budget portion with another (virtual) budget portion in the successor segment, resulting in x_i identical budgets. Denote, $\overline{c}_i^j$ as the size of identical budget portions allocated to segment i, which can be computed as:

$$\overline{c}_i^j = \frac{\max\left\{x_i, m_i^j\right\} \cdot c_i^j}{x_i}. \quad (1)$$

Notation	Description
τ_i	pipeline i
D_i	deadline of τ_i
L_i	critical path length of τ_i
u_i	utilization of τ_i
$\overline{u}_i$	total utilization of budgets converted from τ_i
φ_i	D_i over critical path length of τ_i (stretch)
β	reverse of max individual budget utilization
m_i^j	# of threads in segment j of τ_i
c_i^j	length of thread in segment j of τ_i
D_i'	D_i over β
$\widehat{m}_i$	budget concurrency
$\overline{c}_i$	budget size after packing
$\overline{c}_i^j$	budget portion size of segment j after packing
$\widehat{c}_i$	budget size after inflating
$\widehat{c}_i^j$	budget portion size of segment j after inflating

Figure 2 (a)-(b) show an example. Solid blue rectangles represent threads, and black rectangle frames represent budgets. The entire input pipeline packs into $x_i = 4$ budgets. The first segment is a small segment as $m_1^1 = 3 < 4$, and the second segment is large segment as $m_1^2 = 5 > 4$. A virtual budget portion of size 6 is introduced into the first segment to make all budgets identical. The second segment packs into a 4 budget portions each with size $c_1^2 = \frac{\max\{4,5\} \times 8}{4} = 10$;

Worst-Case Budget Schedule Analysis

Even though all budgets converted from the same pipeline are identical as they share the same size and deadline, they cannot be trivially considered as independent if the packing server only enforces the application-level scheduler to be work-conserving. Because, the packing operation alone cannot guarantee that a segment can fit into its corresponding budget portion schedules. For example, Figure 3 depicts a valid schedule of the pipeline and budgets shown in Figure 2 (a) and (b). The black frames represent the budget schedule of the first segment, and the grey frames the budget schedule of the second segment. In this example, we focus on the first segment (the small segment), omitting irrelevant threads. During the execution, the parallelism of budget schedule equals 1 in the first 6 time units, which allows the first thread to finish. Then, all remaining budgets of the segment

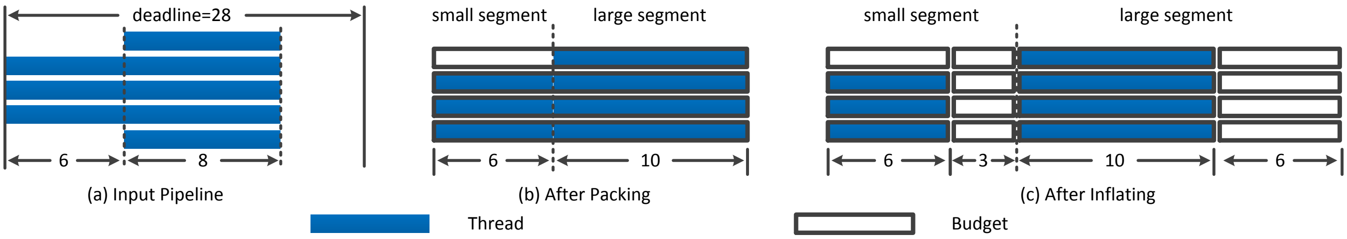


Figure 2: A Packing Server Example

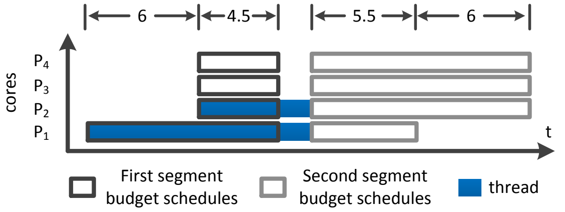

6
4.5
5.5
6
cores
P4
P3
P2
P1
t
First segment budget schedules
Second segment budget schedules
thread

Figure 3: Valid Budget Schedules

schedule to four cores in the next 4.5 time units. As a result, the second and the third threads cannot completely fit into the budget portion schedules.

Using PFair-like [15, 16] policies or simulations [9] in the application-level scheduler can help to fit segments into budget schedules, but the overhead induced by exorbitant preemptions and simulations are luxuries that many systems cannot afford. In order to get rid of excessive preemptions and simulations, we propose to further enlarge budgets after the packing operation to guarantee schedulability. As enlarging budget size introduces a larger amount of virtual budget, which negatively impacts the conversion bound, it is desired to keep the budget enlargement as small as possible.

The minimum enlargement depends on budget schedules. For example, if all budgets execute sequentially in the underlying scheduler, the packing server no longer needs to introduce any enlargement, as the threads can run one by one in budget schedules. In another example as elucidated in Figure 3, each budget needs to be enlarged by at least 1.5 time units. In order to guarantee schedulability, we have to analyze the worst-case budget schedules that force the packing server to introduce the maximum amount of virtual budget.

To understand the worst-case schedule, define $\mathcal{I}_i^j$ as the maximum amount [1] of idle budget in the worst-case, where a portion of a budget is considered idle if there is no threads running in it. Then, we prove Lemma 1.

Lemma 1. *Let $\mathcal{I}_i^j$ denote the maximum amount of idle budget in any schedule of segment j in pipeline i. For any work-conserving algorithm, enlarging each budget by $\mathcal{I}_i^j - \left(x_i\bar{c}_i^j - m_i^j c_i^j\right)$ guarantees that all threads of the segment can fit into any budget schedule.*

Proof: After the packing operation, the amount of budget allocated to segment j of pipeline i is $x_i\bar{c}_i^j$. Enlarging it by $\mathcal{I}_i^j - \left(x_i\bar{c}_i^j - m_i^j c_i^j\right)$ results in $\mathcal{I}_i^j + m_i^j c_i^j$ amount of budget. As $\mathcal{I}_i^j$ denotes the maximum amount of idle budget, it follows that at least $\left(\mathcal{I}_i^j + m_i^j c_i^j\right) - \mathcal{I}_i^j = m_i^j c_i^j$ amount of budget is occupied by thread executions. Because the segment has $m_i^j c_i^j$ thread execution requirement in total, it is guaranteed that all threads in the segment can fit into the budget schedule. ■

According to Lemma 1, the minimum budget enlargement can be calculated by deriving the maximum amount of idle budget. The worst-case budget schedule is the one that forces the maximum amount of idle budget. We consider an adversary that manipulates the budget schedule to force a work-conserving algorithm to leave the maximum amount of budget running idle. We allow the adversary to introduce an unlimited amount of budget into the segment execution time interval. The only constraint is that the parallelism of the budget schedule has to subject to the maximum budget concurrency x_i.

Lemma 2. *In the budget schedule derived using any work-conserving algorithm, the maximum amount of idle budget of segment j in pipeline i is upper bounded by $(x_i - 1)c_i^j$.*

Proof: Let y denote the number of unfinished threads. As for a work-conserving scheduler, there will be no idle budget when $y \geq x_i$.

When $y < x_i$, at most $(x_i - y)$ budgets could run idle. Therefore, the total amount of idle budget accumulates at a speed no faster than $(x_i - y)$, which is upper bounded by $(x_i - 1)$.

We define a time interval as idle if there is at least one budget running idle in all time instances throughout the interval. The idle budget in different time instances does not have to be the same one. According to the property of work-conserving scheduling algorithms, it must be the case that all y remaining threads are executing in idle time intervals. Therefore, the total length of idle time intervals is upper bounded by c_i^j. Otherwise, there must be some thread in the segment whose thread length is larger than c_i^j, which contradicts the definition of a pipeline segment.

In summary, the amount of idle budget can accumulate at the maximum speed of $(x_i - 1)$ with the maximum duration of c_i^j, resulting in at most $(x_i - 1)\,c_i^j$ amounts of idle budget. ■

[1] We use the *amount of budget* to refer to the total length of budgets in a schedule, which is different from the *number of budgets* that represents the number of individual budget in the schedule.

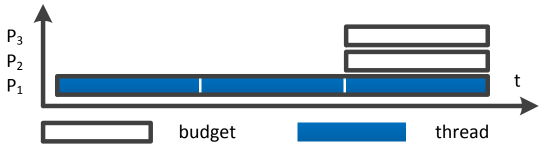


Figure 4: Budget Schedule Examples

The example depicted in Figure 4 shows a segment execution example where there is $(x_i-1)c_i^j$ amount of idle budget. Hence, the bound proven in Lemma 2 is tight.

B. The Inflating Operation

After the packing operation converts a pipeline into identical budgets, the inflating operation further decouples the budgets into independent sequential tasks by enlarging budgets a certain amount such that all threads can fit into the worst-case budget schedule. According to Lemma 2, the packing server guarantees schedulability if the amount of budget allocated to segment j of pipeline i is $(x_i-1)c_i^j$ larger than the total WCET of the segment. As the packing operation has already introduced $\left(\max\{m_i^j, x_i\} - m_i^j\right) c_i^j$ extra amount of budget, the minimum sufficient budget enlargement for segment j is:

$$\frac{(x_i-1)c_i^j - \left(\max\{m_i^j, x_i\} - m_i^j\right) c_i^j}{x_i}. \tag{2}$$

Hence, after the inflating each segment by the amount shown in Equation (2), the pipeline is guaranteed to fit into its budget schedules. Consequently, the budgets can be treated as independent by the underlying scheduler, and the schedulability of those budgets immediately implies the schedulability of the original task graphs. Therefore, by developing the lower bound (the conversion bound) of total task graph utilization ($\sum_i u_i$) over total independent task utilization ($\sum_i \overline{u}_i$), packing servers can bridge independent task utilization bounds to generalized parallel tasks.

As the total amount of virtual budget is $\sum_j (x_i-1)c_i^j$ after applying the packing and inflating operations, it is desired to use the minimum x_i to reduce of the total utilization of converted independent tasks. The constraint is that the result budget size has to be smaller than the deadline D_i. The budget size after the inflating operation is:

$$\begin{aligned} \widehat{c}_i &= \frac{\sum_j m_i^j c_i^j + (x_i-1)\sum_j c_i^j}{x_i} \\ &= \frac{1}{x_i}\sum_j \left(m_i^j - 1\right) c_i^j + \sum_j c_i^j, \end{aligned} \tag{3}$$

Equation (3) helps the packing operation determine the value of x_i for τ_i. According to Equation (3), the budget size $\widehat{c}_i$ decreases monotonically with the increase of budget concurrency x_i. Hence, we can find the minimum possible concurrency $x_i = \widehat{m}_i$ that satisfies the deadline $D'_i = \frac{D_i}{\beta}$ using binary search. With techniques to determine both the concurrency $\widehat{m}_i$ and size $\widehat{c}_i$ of budgets, we can now prove the utilization bound for generalized packing servers.

Theorem 1. *If an independent task scheduler $\mathcal{A}$ achieves a utilization bound of U_B, the generalized packing server over $\mathcal{A}$ as the underlying scheduler guarantees that $U_B \cdot \frac{\varphi-\beta}{\varphi}$ is a valid utilization bound for generalized parallel tasks.*

Proof: Based on the definition of $\widehat{m}_i$, we have that using one less concurrency $(\widehat{m}_i - 1)$ violates deadline:

$$\sum_j \frac{m_i^j c_i^j + (\widehat{m}_i - 1)c_i^j}{\widehat{m}_i - 1} = \sum_j \frac{m_i^j c_i^j}{\widehat{m}_i - 1} + \sum_j c_i^j \geq D'_i \tag{4}$$

According the definition of D'_i, we have:

$$\sum_j \frac{m_i^j c_i^j}{\widehat{m}_i - 1} \geq D'_i - \sum_j c_i^j = (\frac{\varphi_i}{\beta} - 1)\sum_j c_i^j \tag{5}$$

Then, we can represent the lower bound of the total execution time of the original pipeline as:

$$\sum_j m_i^j c_i^j \geq (\widehat{m}_i - 1)(\frac{\varphi_i}{\beta} - 1)\sum_j c_i^j \tag{6}$$

As the amount of virtual budget in each segment is upper bounded by $(\widehat{m}_i - 1)\, c_i^j$, we can compute the upper bound ratio of virtual budget over original task graph total WCET

$$\begin{aligned} \frac{\overline{u}_i - u_i}{u_i} &= \frac{\sum_j (\widehat{m}_i - 1)\, c_i^j}{\sum_j m_i^j c_i^j} \\ &\leq \frac{(\widehat{m}_i - 1)\sum_j c_i^j}{(\frac{\varphi_i}{\beta} - 1)\sum_j c_i^j (\widehat{m}_i - 1)} \qquad \text{Inequality (6)} \\ &= \frac{\beta}{\varphi_i - \beta} \end{aligned} \tag{7}$$

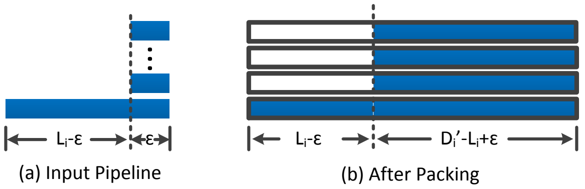
Finally we have:

$$\sum_i u_i \geq \frac{\varphi - \beta}{\varphi}\sum_i \overline{u}_i, \qquad \varphi = \min\{\varphi_i | \forall i\} \tag{8}$$

■

This conversion bound matches the one derived in literature [9]. However, compared to literature [9], we completely remove the requirement of simulations in the underlying scheduler, promoting the packing server technique to a broader spectrum of applications.

Figure 2 depicts an example of packing and inflating a pipeline. The input pipeline i contains two segments, where the first segment consists of $m_i^1 = 3$ threads with length $c_i^1 = 6$, and the second segment consists of $m_i^2 = 5$ threads with length $c_i^2 = 8$. The deadline D_i is 28. According to Equation (3), we can calculate that the minimum budget concurrency without violating the deadline to be $\widehat{m}_i = 4$. Therefore, after conducting the packing operation, $\overline{c}_i^1 = c_i^1 = 6$, as $m_i^1 < \widehat{m}_i$. The five threads of the second segment concentrate into four, each with length $\overline{c}_i^2 = \frac{5\times 8}{4} = 10$. Now, we calculate the amount $\left(\widehat{c}_i^j - \overline{c}_i^j\right)$ that budgets should enlarge for each segment j.



(a) Input Pipeline

(b) After Packing

Figure 5: Bound Tightness

- For the first segment:

$$\begin{aligned}\widehat{c}_i^1 - \overline{c}_i^1 &= \frac{m_i^1 c_i^1 + (\widehat{m}_i - 1)\, c_i^1}{\widehat{m}_i} - \overline{c}_i^1 \\ &= \frac{3 \times 6 + (4-1) \times 6}{4} - 6 = 9 - 6 = 3.\end{aligned}$$

- For the second segment:

$$\begin{aligned}\widehat{c}_i^2 - \overline{c}_i^2 &= \frac{m_i^2 c_i^2 + (\widehat{m}_i - 1)\, c_i^2}{\widehat{m}_i} - \overline{c}_i^2 \\ &= \frac{5 \times 8 + (4-1) \times 8}{4} - 10 = 16 - 10 = 6.\end{aligned}$$

A merit of the technique is that the application-level scheduler can be very simple and efficient. As it is guaranteed that segments can fit into any budget schedule, the application-level scheduler can schedule an arbitrary thread when a budget starts execution. One can imagine a simple implementation that all threads of the current segment are organized into a queue. The application-level scheduler picks and runs the head thread from the queue when a budget executes, and preempts and appends the thread to the end of the queue when that budget gets preempted by the underlying scheduler.

C. Bound Tightness

As described above, the conversion bound $\frac{\varphi-\beta}{\varphi}$ applies to all work-conserving application-level schedulers. One natural question to ask is whether the conversion bound is tight. For example, when proving the conversion bound, we do not make use of proactive preemptions at all, where *Proactive Preemption* is the thread preemption issued by the application-level scheduler when the corresponding budget is not preempted by the underlying scheduler. Hence, it is meaningful to know whether the conversion bound can be further improved if the application-level scheduler is allowed to proactively preempt thread executions following some judicious algorithms.

Intuitively, proactive preemptions may help reduce the amount of budget enlargement. For example, by enforcing a PFair-like scheduling policy in the application-level scheduler, packing servers no longer need the inflating operation to enlarge budgets. However, it turns out that the bound is still tight even if the application-level scheduler uses unlimited proactive preemptions.

Consider a pipeline of two segments as depicted in Figure 5 (a). The first segment contains a single thread of length c_i^1. The second segment consists of $\frac{(D_i'-c_i^1)x}{\epsilon}$ threads of length ϵ. After packing the pipeline into x budgets, we compute:

$$\begin{aligned}\frac{\overline{u}_i - u_i}{u_i} &= \frac{(x-1)c_i^1}{(D_i' - c_i^1)\, x + c_i^1} \\ &\approx \frac{(x-1)L_i}{\left(\frac{\varphi_i}{\beta} - 1\right) L_i x + L_i} \quad \text{as } L_i = c_i^1 + \epsilon \approx c_i^1 \\ &\approx \frac{\beta}{\varphi_i - \beta} \qquad \text{when } x \text{ is large} \qquad (9)\end{aligned}$$

This example shows that the packing operation alone may introduce $u_i \cdot \frac{\beta}{\varphi-\beta}$ amount of virtual budget. Hence, the bound shown in (8) is the optimal conversion bound no matter what algorithm is used to fit threads into budget schedules.

D. Transforming Workflows into Pipelines

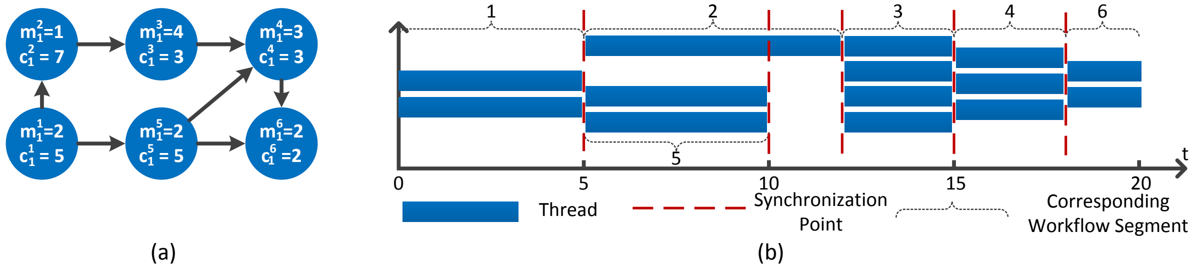
This section further generalizes the same conversion bound $\frac{\varphi-\beta}{\varphi}$ to workflows by transforming a workflow into a pipeline. Our goal is to develop a strategy that leads to the highest conversion bound. As shown in previous sections, the conversion bound is $\frac{\varphi-\beta}{\varphi}$, which improves with the decrease of the pipeline critical path length, as $\varphi = \min\{\frac{D_i}{L_i} | \forall i\}$. This inspires us to use a transformation algorithm that minimizes the critical path length in the transformed pipeline. The pipeline length is lower bounded by the critical path length L_i of the input workflow. Therefore, conversion bound is maximized if the transformation algorithm successfully curbs the pipeline critical path length at L_i.

The workflow transformation algorithm developed in literature [9] can help to achieve this objective. The algorithm uses a virtual time axis, and allows each segment in the workflow to start execution at the earliest possible time. To keep the result as a valid pipeline, a synchronization time point is inserted at the end of each workflow segment, which may potentially break threads of some other segments into parts. Figure 6 shows an example. The original workflow is depicted in (a) and the resulting pipeline is shown in (b). As segments are of different lengths, a segment in the original workflow may break into multiple segments in the pipeline. A pipeline segment may consist of parts from multiple workflow segments. For example, workflow segment 2 and segment 5 both follow segment 1. Hence, these two segments may start at the same time. However, as segment 5 finishes sooner than segment 2, its synchronization time point breaks segment 2 into two pipeline segments.

This algorithm transforms a workflow task set into a pipeline task set without increasing its utilization, and at the same time preserves the same critical path length L_i. Therefore, the same utilization bound $U_B \cdot \frac{\varphi-\beta}{\varphi}$ also applies to workflows.

IV. Evaluation

In the evaluation section, we compare six scheduling algorithms:



(a)

(b)

Figure 6: An example of transforming a workflow into a pipeline

- **Packing/GEDF**: The generalized packing server over GEDF as the underlying scheduler.
- **Packing/EDF-FF**: The generalized packing server over EDF First Fit (EDF-FF) [14] as the underlying scheduler.
- **Sim/GEDF**: The simulation-based packing server over GEDF as the underlying scheduler. The simulation-based packing server, first proposed in [9] for Map/Reduce applications, shares a similar structure as the generalized packing server, except that it requires the underlying scheduler to simulate budget executions up to D_{max} time units ahead, where D_{max} is the largest deadline of all workflows.
- **Sim/EDF-FF**: The simulation-based packing server over EDF First Fit (EDF-FF) [14] as the underlying scheduler.
- **GEDF**: Global EDF is a well-known and widely used scheduler. It assigns the highest priority to the workflow with the earliest absolute deadline. Its best-known utilization bound is $\frac{2}{3+\sqrt{5}} \approx 38.2\%$ [7].
- **Federated**: Federated scheduling is the state-of-the-art generalized parallel task scheduling algorithm [7]. It achieves the highest-known utilization bound of 50% for generalized parallel tasks, when the stretch φ surpasses 2. The algorithm partitions tasks based on their utilizations. For every task τ_i with utilization at least one (*i.e.*, $u_i \geq 1$), the algorithm allocates $\lceil \frac{C_i - L_i}{D_i - L_i} \rceil$ dedicated cores to τ_i, where $C_i = \sum_j m_i^j c_i^j$. All other lower-utilization tasks share the remaining cores.

The simulations emulate 50 cores. In all figures, the shown utilization is divided by the number of cores in the system to normalize it to a per-core value.

The DAG generation program generates DAGs using parameters that include the degree of parallelism of each segment, the number of hops on the critical path, the segment length, the number of segments in each DAG, and the tightness of deadline. Detailed DAG generation policies will be described close to corresponding experiments.

A large family of DAGs are generated first for each experiment. When load is increased on the horizontal axis, more DAGs are drawn from the set. If there are unfinished threads when a DAG reaches its deadline, all remaining threads are preempted and discarded, and the simulation records the deadline-violation event. The experiments enforce no admission controls.

A. *Computing the Optimal Budget Size*

We begin by computing the optimal packing server budget size (or equivalently, the optimal value of β). Generalized packing servers may use any work-conserving independent task scheduling policy in the underlying scheduler to achieve a utilization bound of $U_B \cdot \frac{\varphi - \beta}{\varphi}$ for generalized parallel tasks, where U_B is the utilization bound of the underlying independent task scheduler. The value of β affects the utilization bound in two competing directions: 1) The underlying scheduling algorithms usually favor smaller individual task utilization, and hence larger β; 2) The conversion bound prefers smaller β. Therefore, it is desired to find an optimal β that balances the two. For GEDF the optimal β maximizes:

$$U_B \cdot \frac{\varphi - \beta}{\varphi} = \frac{(\varphi - \beta)(m\beta - m + 1)}{m\varphi\beta}. \tag{10}$$

By taking derivatives with respect to β, and setting the derivative to 0, the highest utilization bound for the packing server over GEDF is achieved at:

$$\beta = \sqrt{\frac{\varphi(m-1)}{m}}. \tag{11}$$

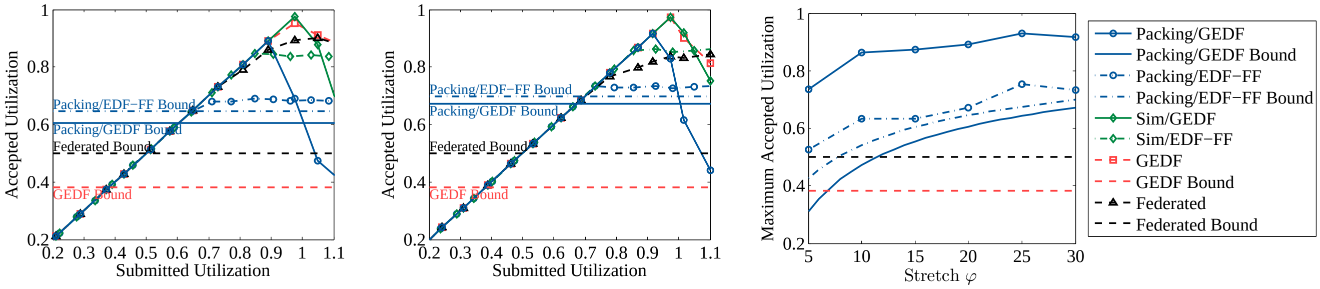
Similarly, one can show that the highest utilization bound for the packing server over EDF-FF is achieved at:

$$\beta = \sqrt{\frac{(\varphi + 1)(m-1)}{m}} - 1. \tag{12}$$

The following experiments use the optimal β values to configure the packing server according to the underlying scheduling policy, unless otherwise stated.

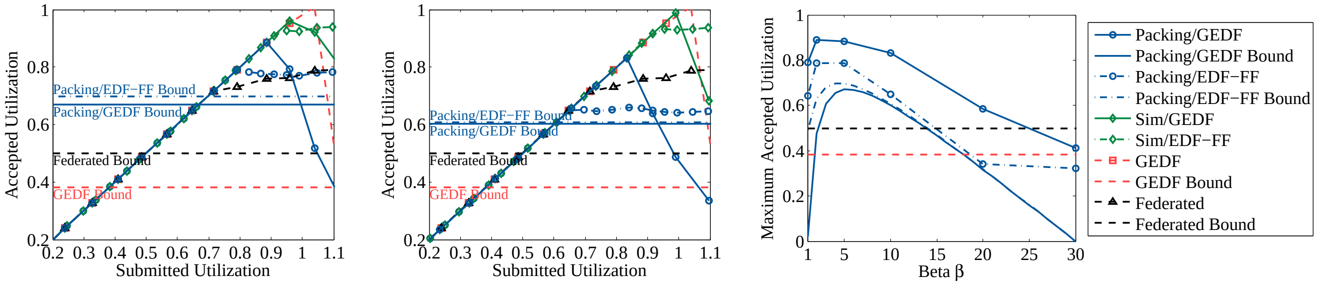
B. *Effect of φ*

This section evaluates how the key parameter φ in the conversion bound affects the performance of the generalized packing servers. During the DAG generation process, the program first generates a random topology. Then, the deadline of the DAG is set to the product of its critical path length L_i and the given stretch φ (*i.e.*, $D_i = L_i \cdot \varphi$). In this case, varying the parameter φ also affects the task utilization, which we shall show later in Section IV-D is an important parameter for the *Federated* scheduling policy. Tuning two key parameters in the same experiment would cause confusion about which one is the true driving factor of observed results. Therefore, we compensate by adding more



(a) $\varphi = 20$ (b) $\varphi = 30$ (c) Summary

Figure 7: Varying Stretch φ



(a) $\beta = 5$ (b) $\beta = 10$ (c) Summary

Figure 8: Varying Budget Utilization Cap

threads into segments to keep individual task utilizations around the same value.

We vary the value of φ from 5 to 30 with step length 5, and compute optimal β for each different φ using Equation (11) and (12). The results show that, although packing server does not dominate other algorithms for every φ value, it beats the best-known generalized parallel task utilization bound when φ is larger than 8. Moreover, to the best of our knowledge, Packing/EDF-FF introduces the first known bound of 70% for EDF-FF algorithm on generalized parallel tasks with $\varphi = 30$.

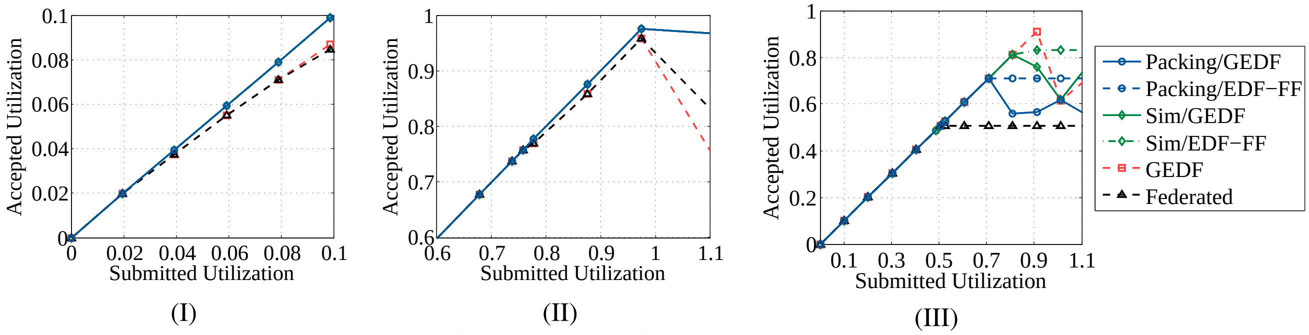
Figure 7 (a) depicts the $\varphi = 20$ case, where the utilization bounds for Packing/EDF-ff and Packing/GEDF are 64% and 60% respectively, beating the best-known bound of 50%. The generalized packing servers over EDF-FF and GEDF start to miss deadlines when the submitted utilization surpasses 68% and 90% respectively. Sim/EDF-FF accepts up to 83% utilization, and Sim/GEDF successfully accepts a task set with 97% utilization. It can be seen that although simulation-based and generalized packing servers share the same theoretical utilization bound, sim/EDF-FF and sim/GEDF outperform their corresponding generalized packing servers empirically. The reason is that, a common worst-case workflow topology as shown in Figure 5 curbs the utilization bounds of both algorithms, leading to the same theoretical utilization bound. However, randomly generated DAGs quite unlikely form the worst-case topology, which allows simulation-based packing servers to avoid budget enlargements caused by the inflating operations. When φ is set to 30 as shown in Figure 7 (b), the theoretical schedulability utilization bound for packing servers over EDF-FF and GEDF further improve to 70% and 67% respectively.

The curves using GEDF-related schedulers dip at higher utilization to a value below the bound. This is because no admission control is used. In the absence of admission control, EDF suffers a domino effect when the system gets overloaded causing a sharp decline in the utilization of tasks that actually meet deadlines. If admission control was used, all tasks would meet their deadlines as is clear from the point on the x-axis where the submitted utilization is equal to the bound.

Figure 7 (c) summarizes, in general, how the parameter φ affects the maximum observed accepted utilization. The curves without markers indicate the theoretical utilization bound for Packing/EDF-FF and Packing/GEDF respectively. For randomly generated DAGs, both generalized and simulation-based packing servers accept higher utilizations at larger φ values, following the same trend shown by the theoretical bound. Additionally, comparing the bound computed to the empirically obtained maximum schedulable utilization, we can conclude that Packing/EDF-FF leads to the least degree of pessimism when scheduling randomly generated workflows.

C. *Effect of β*

The other key parameter in the conversion bound is β. Although the optimal β can be derived using Equations 11 and 12, it is still important to know how generalized packing servers behave when using different β values. Figure 8 plots



(I) (II) (III)

Figure 9: Unfavorable DAGs for GEDF and Federated Scheduling

how β influences accepted utilizations, with β varying from 1 to 30 and φ fixed at 30. The optimal β values are 4.56 and 5.47 for Packing/EDF-FF and Packing/GEDF respectively. In Figure 8 (a) where $\beta = 5$, Packing/EDF-FF and Packing/GEDF both achieve considerably high utilization bounds, as β is close to the optimal value. Empirically, they also accepts more than 80% utilization. The accepted utilization of Packing/EDF-FF dramatically drops to 65% when β increases to 10, as shown in Figure 8 (b). The results confirm that β fundamentally impacts accepted utilizations. As summarized in Figure 8 (c), there is a clear trend of ascending followed by descending when β increases from 1 to 30. The pinnacle in the bound curve conforms to the optimal beta value of 4.56 and 5.42. Again, Figure 8 shows that the utilization bound for Packing/EDF-FF achieves the least degree of pessimism.

D. Effect of DAG Topology

In experiments using randomly generated DAGs, GEDF performs well, and accepts more than 90% utilization in most cases. However, there are also cases where the generalized packing servers significantly outperforms GEDF and Federated scheduling policies. Consider three sets of implicit-deadline parallel tasks.

- Task set (**I**): Task τ_1 consists of one segment of $m_1^1 = 100$ threads, each with length $c_1^1 = 1$. Task τ_2 contains a single thread of length $c_2^1 = 100$. The deadlines for the two workflows are $D_1 = 101$, and $D_2 = 102$ respectively.
- Task set (**II**): Compared to Task set **I**, the only difference is that c_2^1 shrinks to 25.
- Task set (**III**): Task τ_1 consists of a single segment, with $c_1^1 = 3$, and $m_1^1 = 27$. The deadline D_1 is set to 80.

In the above three task sets, τ_1 can be copied an arbitrary number of times to achieve a desired task set utilization.

Task sets (I) and (II) represent unfavorable DAGs for the GEDF scheduler. Figure 9 (I) plots the simulation results for task set (I). In this case, the GEDF scheduler starts to miss deadline at a utilization of 4%, while the both generalized packing server and simulation-based packing servers accept up to 99% task set utilizations. In this task set, τ_2 contains a long thread of length $c_2^1 = 100$, and subjects to a slightly larger deadline of $D_2 = 102$, compared to other tasks. So, GEDF schedules other tasks prior to τ_2, forcing τ_2 to miss its deadline at a very low task set utilization. Packing servers handle these input tasks well due to the result of packing. By setting $\beta = 1$, every task is packed into a single budget, allowing τ_2 to enjoy its execution opportunity when the task set utilization is below 99%. Task set (I) is out of the scope considered in literature [7], as the stretch φ is smaller than $\frac{3+\sqrt{5}}{2}$. Task set (II) present tasks with larger stretches where the $\frac{2}{3+\sqrt{5}} \approx 38.2\%$ utilization bound is applicable. Figure 9 (II) elucidates the result. GEDF and Federated schedulers misses deadlines when task utilization reaches 76%, whereas packing servers accepts all tasks when the total utilization stays below 97%.

Task sets (III) is unfavorable for the Federated scheduler. The simulation results are presented in Figure 9 (III). The accepted utilization of the Federated scheduling policy stays at 50%, which is its theoretical schedulability utilization bound. It is because, under this configuration, $\frac{C_i - L_i}{D_i - L_i} = \frac{81-3}{80-3} = \frac{78}{77}$ is slightly larger than 1, forcing the Federated scheduler to allocate 2 cores to each task. As a result, half of the resources in the platform are wasted. From this perspective, Federated scheduling algorithm prefers higher individual task utilizations. Generalized packing servers accept 72% task set utilization, and simulation-based packing servers accept 81% utilization. GEDF does not miss any deadline when the task set utilization is smaller than 90%.

These measurements convey that no single scheduler dominates all others. Therefore, systems without prior knowledge of the DAG topology can only turn to the theoretically utilization bound for schedulability guarantees.

V. Related Work

The generalized parallel task scheduling problem has been recently studied on multiprocessor platforms. Baruah *et al.* [5] prove that EDF can achieve a 2X speedup bound for a single recurrent workflow. Saifullah *et al.* [4] propose to arrange a workflow into stages, and then the workflow's deadline is split and assigned to each stage. If some optimal algorithm can successfully schedule the original workflow, their solution is guaranteed to satisfy the same deadline with

4X (speedup bound) speed processors. When the workflow is restricted to a fork-join model [17, 18], Lakshmanan *et al.* [19] improve the speedup bound to 3.42. Li *et al.* [6] develop a capacity augmentation bound of $4 - \frac{2}{m}$ for workflows, which immediately leads to a simple and effective schedulability test. More recently, Li *et al.* [7] improve the capacity augmentation bound to 2 using the federated scheduling algorithm for implicit deadline workflows. They then prove the same result for stochastic parallel tasks [20]. Analysis from literature [1] shows a speedup bound for the federated scheduling algorithm to be $\left(4 - \frac{2}{m}\right)$ when using arbitrary-deadline sporadic DAG models. Fonseca *et al.* [2] further introduce conditional executions into the generalized parallel task models to capture the conditional constructs, such as *if-then-else* clauses. Baruah *et al.* [3] later prove a speedup factor of $\left(2 - \frac{1}{m}\right)$ for the conditional sporadic generalized parallel tasks. These efforts study the generalized parallel task scheduling problem by directly analyzing its behavior on multiprocessor platforms.

Compared to above approaches, packing servers take a different path to solve the generalized parallel task scheduling problem. Namely, they offer a mechanism for converting the original task set into independent tasks. Unlike other conversion-based approaches, in packing servers the resulting independent tasks (server budgets) are subject to the same original end-to-end deadlines. Since the deadlines are not broken into per-segment deadlines, no artificial deadline constraints are introduced, which improves schedulability. Our prior work [9] analyzes the workflow scheduling problem in MapReduce applications [21], where the MapReduce workload and system architecture allow the packing server to conduct simulation in the underlying scheduler. However, simulations are a luxury that many embedded system schedulers cannot afford, due to excessive overhead. In this paper, we propose the generalized packing server that no longer requires simulations, and at the same time achieves the same theoretically schedulability utilization bound as the simulation-based packing servers. With the generalized packing server techniques, the generalized parallel task scheduling problem will continuously benefit from the rich body of existing and future advances of independent task scheduling algorithms and analysis.

VI. Conclusion

In this paper, we propose and evaluate the *generalized packing server* for task graphs on multiple resource platforms. Generalized packing servers allow utilization bounds for independent tasks to be applied to generalized parallel tasks, with a scaling factor $\frac{\varphi-\beta}{\varphi}$, where φ is the deadline to critical path length ratio and β is a tunable parameter that controls the maximum utilization of the corresponding independent budgets ($u_{max} \leq \frac{1}{\beta}$). The scaling factor is called the conversion bound. More specifically, if the underlying independent task scheduler achieves a utilization bound of U_B, then generalized packing servers guarantee that $U_B \cdot \frac{\varphi-\beta}{\varphi}$ is a valid utilization bound for generalized parallel tasks. For task sets with large φ, the new bound beats the best-known utilization bound of generalized parallel tasks (50%). For example, when the underlying scheduler uses the EDF-FF policy, generalized packing servers may improve the utilization bound to 70% on $\varphi \geq 30$ task sets. Our evaluation empirically confirms the validity of derived utilization bounds and evaluate their degree of pessimism.

References

[1] S. Baruah, "federated scheduling of sporadic dag task systems," in *IEEE IPDPS*, 2015.

[2] J. C. Fonseca, V. Nélis, G. Raraviand, and L. M. Pinho, "A multi-dag model for real-time parallel applications with conditional execution," in *ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2015.

[3] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, "federated scheduling of sporadic dag task systems," in *ECRTS*, 2015.

[4] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *IEEE RTSS*, 2011.

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

[6] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global edf for parallel tasks," in *IEEE ECRTS*, 2013.

[7] 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.

[8] G. Liu, Y. Lu, S. Wang, and Z. Gu, "Partitioned multiprocessor scheduling of mixed-criticality parallel jobs," in *IEEE RTCSA*, 2014.

[9] S. Li, S. Hu, and T. Abdelzaher, "The packing server for real-time scheduling of mapreduce workflows," in *IEEE RTAS*, 2015.

[10] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *The Journal of Real-Time Systems*, vol. 1, pp. 27–60, 1989.

[11] B. Sprunt and L. Sha and J. Lehoczky, "Enhanced aperiodic responsiveness in hard real-time environment," in *IEEE RTSS*, 1987.

[12] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, 1995.

[13] T. P. Baker, "A comparison of global and partitioned edf schedulability tests for multiprocessors," International Conference on Real-Time Networks and Systems (RTNS), Tech. Rep., 2005.

[14] J. M. López, M. García, J. L. Díaz, and D. F. García, "Worst-case utilization bound for edf scheduling on real-time multiprocessor systems," in *ECRTS*, 2000.

[15] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *ACM STOC*, 1993.

[16] S. Baruah, J. Gehrke, and C. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *IEEE IPDPS*, 1995.

[17] C. Maia, L. Nogueira, L. M. Pinho, and M. Bertogna., "Response-time analysis of fork/join tasks in multiprocessor systems," in *ECRTS*, 2013.

[18] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2014.

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

[20] J. Li, K. Agrawal, C. Gill, and C. Lu, "Federated scheduling for stochastic parallel real-time tasks," in *RTCSA*, 2014.

[21] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace, "Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop cluster," in *IEEE ICDCS*, 2014.