Report Number: WUCSE-2014-44

2014

Capacity Augmentation Bound of Federated Scheduling for Parallel DAG Tasks

Jing Li, Abusayeed Saifullah, Kunal Agrawal, and Christopher Gill

We present a novel federated scheduling approach for parallel real-time tasks under a general directed acyclic graph (DAG) model. We provide a capacity augmentation bound of 2 for hard real-time scheduling; here we use the worst-case execution time and critical-path length of tasks to determine schedulability. This is the best known capacity augmentation bound for parallel tasks. By constructing example task sets, we further show that the lower bound on capacity augmentation of federated scheduling is also 2 for any m > 2. Hence, the gap is closed and bound 2 is a strict bound for federated scheduling. The... **Read complete abstract on page 2.**

Capacity Augmentation Bound of Federated Scheduling for Parallel DAG Tasks

Jing Li, Abusayeed Saifullah, Kunal Agrawal, and Christopher Gill

Complete Abstract:

We present a novel federated scheduling approach for parallel real-time tasks under a general directed acyclic graph (DAG) model. We provide a capacity augmentation bound of 2 for hard real-time scheduling; here we use the worst-case execution time and critical-path length of tasks to determine schedulability. This is the best known capacity augmentation bound for parallel tasks. By constructing example task sets, we further show that the lower bound on capacity augmentation of federated scheduling is also 2 for any m > 2. Hence, the gap is closed and bound 2 is a strict bound for federated scheduling. The federated scheduling algorithm is also a schedulability test that often admits task sets with utilization much greater than 50%m.

2014-44

Capacity Augmentation Bound of Federated Scheduling for Parallel DAG Tasks

Authors: Jing Li, Abusayeed Saifullah,
Kunal Agrawal, Christopher Gill, and Chenyang Lu

Corresponding Author: li.jing@wustl.edu

Abstract: We present a novel federated scheduling approach for parallel real-time tasks under a general directed acyclic graph (DAG) model. We provide a capacity augmentation bound of 2 for hard real-time scheduling; here we use the worst-case execution time and critical-path length of tasks to determine schedulability. This is the best known capacity augmentation bound for parallel tasks. By constructing example task sets, we further show that the lower bound on capacity augmentation of federated scheduling is also 2 for any m > 2. Hence, the gap is closed and bound 2 is a strict bound for federated scheduling. The federated scheduling algorithm is also a schedulability test that often admits task sets with utilization much greater than 50%m.

Type of Report: Other

Capacity Augmentation Bound of Federated Scheduling for Parallel DAG Tasks

Jing Li, Abusayeed Saifullah,
Kunal Agrawal, Christopher Gill, and Chenyang Lu

Department of Computer Science and Engineering
Washington University in St. Louis
{li.jing, saifullah}@go.wustl.edu, {kunal, cdgill, lu}@cse.wustl.edu

Abstract

We present a novel ***federated scheduling approach*** *for parallel real-time tasks under a general directed acyclic graph (DAG) model. We provide a* ***capacity augmentation bound*** *of* 2 *for hard real-time scheduling; here we use the worst-case execution time and critical-path length of tasks to determine schedulability. This is the best known capacity augmentation bound for parallel tasks. By constructing example task sets, we also show that the lower bound on capacity augmentation of federated scheduling is also* 2 *for any* $m > 2$*. Hence, the gap is closed and bound* 2 *is a strict bound for federated scheduling. The federated scheduling algorithm is also a schedulability test that often admits task sets with utilization much greater than* $50\%m$*.*

***Index Terms*—real-time scheduling, parallel scheduling, federated scheduling, capacity augmentation bound**

I. Introduction

In the last decade, multicore processors have become ubiquitous and there has been extensive work on how to exploit these parallel machines for real-time tasks. In the real-time systems community, there has been extensive research on scheduling task sets with *inter-task parallelism* — where each task in the task set is a sequential program. In this case, increasing the number of cores allows us to increase the number of tasks in the task set. However, since each task can only use one core at a time, the computational requirement of a single task is still limited by the capacity of a single core. Recently, there has been some interest in design and analysis of scheduling strategies for task sets with *intra-task parallelism* (in addition to inter-task parallelism), where individual tasks are parallel programs and can potentially utilize more than one core in parallel. These models enable tasks with higher execution demands and tighter deadlines, such as those used in autonomous vehicles [28], video surveillance, computer vision, radar tracking and real-time hybrid testing [25]

In this paper, we consider the general *directed acyclic graph (DAG)* model. We analyze three different scheduling strategies: global EDF, global rate-monotonic scheduling and a proposed federated scheduling. We prove that all three strategies provide strong performance guarantees, in the form of *capacity augmentation bounds*, for scheduling parallel DAG tasks with implicit deadlines.

One can generally derive two types of performance bounds for real-time schedulers. The traditional bound is called ***resource augmentation bound*** (also called *processor speed-up factor*). A scheduler $\mathcal{S}$ provides a resource augmentation bound of $b \geq 1$ if it can successfully schedule any task set τ on m cores of speed b as long as the ideal scheduler can schedule τ on m cores of speed 1. A resource augmentation bound provides a good notion of how close a scheduler is to the optimal schedule, but it has a drawback. Note that the *ideal scheduler* is only a hypothetical scheduler, meaning that it always finds a feasible schedule if one exists. Unfortunately, Fisher et al. [23] proved that optimal online multiprocessor scheduling of sporadic task systems is impossible. Since, since we often cannot tell whether the ideal scheduler can schedule a given task set on unit-speed cores, a resource augmentation bound may not provide a schedulability test.

Another bound that is commonly used for sequential tasks is a ***utilization bound***. A scheduler $\mathcal{S}$ provides a utilization bound of b if it can successfully schedule any task set which has total utilization at most m/b on m cores.[1] A utilization bound provides more information than a resource augmentation bound; any scheduler that guarantees a utilization bound of b automatically guarantees a resource augmentation bound of b as well. In addition, it acts as a very simple schedulability test in itself, since the

[1]A utilization bound is often stated in terms of $1/b$; we adopt this notation in order to be consistent.

total utilization of the task set can be calculated in linear time and compared to m/b. Finally, a utilization bound gives an indication of how much load a system can handle; allowing us to estimate how much over-provisioning may be necessary when designing a platform. Unfortunately, it is often impossible to prove a utilization bound for parallel systems due to Dhall's effect; often, we can construct pathological task sets with utilization arbitrarily close to 1, but which cannot be scheduled on m cores.

Li et al. [31] defined a concept of ***capacity augmentation bound*** which is similar to the utilization bound, but adds a new condition. A scheduler $\mathcal{S}$ provides a capacity augmentation bound of b if it can schedule any task set τ which satisfies the following two conditions: (1) the total utilization of τ is at most m/b, and (2) the worst-case critical-path length of each task L_i (execution time of the task on an infinite number of cores)[2] is at most $1/b$th fraction of its deadline. A capacity augmentation bound is quite similar to a utilization bound: It also provides more information than a resource augmentation bound does; any scheduler that guarantees a capacity augmentation bound of b automatically guarantees a resource augmentation bound of b as well. It also acts as a very simple schedulability test. Finally, it can also provide the estimation of load a system is expected to handle.

There has been some recent research on proving both resource augmentation bounds and capacity augmentation bounds for various scheduling strategies for parallel tasks. This work falls in two categories. In *decomposition-based strategies*, the parallel task is decomposed into a set of sequential tasks and they are scheduled using existing strategies for scheduling sequential tasks on multiprocessors. In general, decomposition-based strategies require explicit knowledge of the structure of the DAG off-line in order to apply decomposition. In non-decomposition based strategies, the program can unfold dynamically since no offline knowledge is required.

For decomposed strategy, most prior work considers *synchronous tasks* (subcategory of general DAGs) with implicit deadlines. Lakshmanan et al. [29] proved a capacity augmentation bound of 3.42 for partitioned fixed-priority scheduling for a restricted category of synchronous tasks[3] by decomposing tasks and scheduling the decomposed tasks using a under decomposed deadline monotonic scheduling. Saifullah et al. [40] provide a different decomposition strategy for general parallel synchronous tasks and prove a capacity augmentation bound of 4 when the decomposed tasks are scheduled using global EDF and 5 when they are scheduled using partitioned DM. Kim et al. [28] provide another decomposition strategy and prove a capacity augmentation bound of 3.73 using global deadline monotonic scheduling. In the respective papers, these results are stated as resource augmentation bounds, but they are in fact the stronger capacity augmentation bounds. Nelisson et al. [36] proved a resource augmentation bound of 2 for general synchronous tasks.

[2]critical-path length of a sequential task is equal to its execution time

[3]Fork-join task model in their terminology

For non-decomposition strategies, researchers have studied primarily global earliest deadline first (***G-EDF***) and global rate-monotonic (***G-RM***). Andersson and Niz [4] show that G-EDF provides resource augmentation bound of 2 for synchronous tasks with constrained deadlines. Both Li et. al [31] and Bonifaci et. al [13] concurrently showed that G-EDF provides a resource augmentation bound of 2 for general DAG tasks with arbitrary deadlines. In their paper, Bonifaci et al. also proved that G-RM provides a resource augmentation bound of 3 for parallel DAG tasks with arbitrary deadlines; Li et. al also provide a capacity augmentation bound of 4 for G-EDF for task sets with implicit deadlines.

In summary, the best known capacity augmentation bound for implicit deadlines tasks are 4 for DAG tasks using global EDF, and 3.73 for parallel synchronous tasks using decomposition combined with global DM. The contributions of this paper are as follows:

1) We propose a novel ***federated scheduling strategy***. Each *high-utilization task* (utilization > 1) is allocated a dedicated cluster (set) of cores. A multiprocessor scheduling algorithm is used to schedule all *low-utilization tasks*, each of which is run sequentially, on a shared cluster composed of the remaining cores. Federated scheduling can be seem as an extension of partitioned scheduling for parallel tasks.
2) We prove that federated scheduling has the best known capacity augmentation bound 2 for *any scheduler* for parallel DAGs. By constructing example task sets, we further show that the lower bound on capacity augmentation of federated scheduling is also 2 for any $m > 2$. Hence, the gap is closed and bound 2 is strict.
3) The federated scheduling algorithm is also a schedulability test that often admits task sets with utilization much greater than $50\%m$. If the algorithm *admits* a task set — returns a valid core allocation for all tasks, then the task set is schedulable, otherwise it is not.

The paper is organized as follows. Section II defines the DAG model for parallel tasks and provides some definitions. Section III presents our federated scheduling algorithm and proves the augmentation bound. Section IV discusses related work and Section V concludes this paper.

II. System Model

We now present the details of the DAG task model for parallel tasks and some additional definitions.

Task Model This paper considers a given set τ of n independent sporadic real-time tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. A task τ_i represents an infinite sequence of arrivals and executions of task instances (or also called jobs). We consider the *sporadic task model* [9, 35] where, for a task τ_i, the *minimum inter-arrival time or period* T_i represents the time between consecutive arrivals of task instances, and the *relative deadline* D_i represents the temporal constraint for executing the job. If a task instance of τ_i arrives at time t, the execution of this instance must be finished no later than the *absolute deadline* $t + D_i$ and the release of the next instance of task τ_i must be no earlier than t plus the minimum inter-arrival time, i..e, $t + T_i$. In this paper, we consider *implicit deadline tasks* where each task τ_i's relative deadline D_i is equal to its minimum inter-arrival time T_i; that is, $T_i = D_i$.

Each task $\tau_i \in \tau$ is a parallel task; we consider a general model for parallel tasks, namely the *DAG* model. Each task is characterized by its execution pattern, defined by a directed acyclic graph (DAG). Each node (subtask) in the DAG represents a sequence of instructions (a thread) and each edge represents dependency between nodes. A node (subtask) is *ready* to be executed when all its predecessors have been executed. Throughout this paper, as it is not necessary to build the analysis based on specific structures of the execution pattern, only two parameters related to the execution pattern of task τ_i are defined:

- *total execution time (or work)* C_i of task τ_i: This is the summation of the worst-case execution times of all the subtasks of task τ_i.
- *critical-path length* L_i of task τ_i: This is the length of the critical-path in the given DAG, in which each node is characterized by the worst-case execution time of the corresponding subtask of task τ_i; critical-path length is the worst-case execution time of the task on an infinite number of cores.

Given a DAG, obtaining work C_i and critical-path length L_i [41, pages 661-666] can both be done in linear time.

For brevity, the *utilization* $\frac{C_i}{T_i} = \frac{C_i}{D_i}$ of task τ_i is denoted by u_i for implicit deadlines. The total utilization of the task set is $U_{\sum} = \sum_{\tau_i \in \tau} u_i$. Moreover, let the *critical-path utilization* of task τ_i, denoted as Δ_i, be $\frac{L_i}{T_i} = \frac{L_i}{D_i}$. Also, let $\Delta_{\max}$ be the maximum critical-path utilization of task set τ, i.e., $\Delta_{\max} = \max_{\tau_i \in \tau} \Delta_i$. Finally, we also define V_i as $\Delta_{\max} \cdot D_i$.

This paper considers scheduling a task set on a uniform multicore system consisting of m identical cores.

Utilization-Based Schedulability Test In this paper, we analyze schedulers in terms of their capacity augmentation bounds. The formal definition is presented here:

Definition 1. *Given a task set τ with total utilization of $U_{\sum}$, a scheduling algorithm $\mathcal{S}$ with* ***capacity augmentation bound*** *b can always schedule this task set on m cores of speed b as long as τ satisfies the following conditions on unit speed cores.*

$$\textit{Utilization does not exceed total cores, } \sum_{\tau_i \in \tau} u_i \le m \quad (1)$$

$$\textit{For each task } \tau_i \in \tau, \textit{the critical path } L_i \le D_i \quad (2)$$

Since no scheduler can schedule a task set τ on m unit speed cores unless Conditions (1) and (2) are met, capacity augmentation bound automatically leads to a resource augmentation bound. This definition can be equivalently stated (without reference to the speedup factor) as follows: Condition (1) says that the total utilization $U_{\sum}$ is at most m/b and Condition (2) says that the critical-path length of each task is at most $1/b$ of its relative deadline, that is, $\Delta_{\max} \le 1/b$. Therefore, in order to check if a task set is schedulable we only need to know the total task set utilization, and the maximum critical-path utilization. Note that a scheduler with a smaller b is better than another with a larger b, since when $b = 1$ $\mathcal{S}$ is an optimal scheduler.

III. Federated Scheduling

In this section, we present the federated scheduling algorithm that provide hard real-time guarantees to parallel task sets with implicit deadlines and prove that it provides a capacity augmentation bound of 2 on a machine with m uniform cores. Federated scheduling can be seem as an extension of partitioned scheduling for parallel tasks.

A. Federated Scheduling Algorithm

Given a task set τ, the *federated scheduling algorithm* works as follows: First, tasks are divided into two disjoint sets: τ_{high} contains all ***high-utilization tasks*** — tasks with worst-case utilization at least one ($u_i \ge 1$), and τ_{low} contains all the remaining ***low-utilization tasks***. Consider a high-utilization task τ_i with worst-case execution time C_i, worst-case critical-path length L_i, and deadline D_i (which is equal to its period T_i). We assign n_i dedicated cores to τ_i, where n_i is

$$n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \quad (3)$$

We use $n_{\text{high}} = \sum_{\tau_i \in \tau_{\text{high}}} n_i$ to denote the total number of cores assigned to high-utilization tasks τ_{high}. We assign the remaining cores to all low-utilization tasks τ_{low}, denoted as $n_{\text{low}} = m - n_{\text{high}}$. The federated scheduling algorithm *admits* the task set τ, if n_{low} is non-negative and $n_{\text{low}} \ge 2 \sum_{\tau_i \in \tau_{\text{low}}} u_i$.

After a valid core allocation, the *runtime scheduling* proceeds as follows: (1) Any greedy (work-conserving) parallel scheduler can be used to schedule a high-utilization task τ_i on its assigned n_i cores. Informally, a ***greedy scheduler*** is one that never keeps a core idle if some node is ready to execute. (2) Low-utilization tasks are treated and executed as though they are sequential tasks and any multiprocessor scheduling algorithm with a capacity augmentation bound of at most 2 can be used to schedule all the low-utilization tasks on the allocated n_{low} cores.

Since most existing partitioned multiprocessor schedulability tests have a utilization bound of 50% and hence a capacity augmentation bound of 2. Therefore, in principle, we can use these partitioned multiprocessor scheduling algorithm to schedule them on the n_{low} processors, such as partitioned EDF [33], or various rate-monotonic schedulers [3]. The important observation is that we can safely treat low-tilization tasks as sequential tasks since $C_i \leq D_i$ and parallel execution is not required to meet their deadlines.[4]

B. Capacity Augmentation Bound 2 of Federated Scheduling

Theorem 1. *The federated scheduling algorithm has a capacity augmentation bound of* 2.

To prove Theorem 1, we consider a task set τ that satisfies Conditions (1) and (2) from Definition 1 for b =2. Then, we (1) state the relatively obvious Lemma 1; (2) prove that a high utilization task τ_i meets its deadline when assigned n_i cores; and (3) show that n_{low} is non-negative and satisfies $n_{\text{low}} \geq b\sum_{\tau_i \in \tau_{\text{low}}} u_i$ and therefore all low utilization tasks in τ will meet deadlines when scheduled using any multiprocessor scheduling strategy with utilization bound no less than b (i.e. can afford total task set utilization of $m/b = 50\% m$). These three steps complete the proof.

Lemma 1. *A task set τ is classified into disjoint subsets $s_1, s_2, ..., s_k$, and each subset is assigned a dedicated cluster of cores with size $n_1, n_2, ..., n_k$ respectively, such that $\sum_i n_i \leq m$. If each subset s_j is schedulable on its n_j cores using some scheduling algorithm $\mathcal{S}_j$ (possibly different for each subset), then the whole task set is guaranteed to be schedulable on m cores.*

High-Utilization Tasks Are Schedulable Assume that a machine's execution time is divided into discrete quanta called *steps*. During each step a core can be either idle or performing one unit of work. We say a step is *complete* if no core is idle during that step, and otherwise we say it is *incomplete*. A *greedy* scheduler never keeps a cores idle if there is ready work available. Then, for a greedy scheduler on n_i cores, the following two straightforward lemmas are derived in [31].

Lemma 2. [Li13] *Consider a greedy scheduler running on n_i cores for t time steps. If the total number of incomplete steps during this period is t^*, the total work F^t done during these time steps is at least $F^t \geq n_i t - (n_i - 1)t^*$.*

Lemma 3. [Li13] *If a job of task τ_i is executed by a greedy scheduler, then every incomplete step reduces the remaining critical-path length of the job by 1.*

From Lemmas 2 and 3, we can establish Theorem 2.

Theorem 2. *If an implicit-deadline deterministic parallel task τ_i is assigned $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ dedicated cores, then all its jobs can meet their deadline when using a greedy scheduler.*

Proof: For contradiction, assume that some job of a high-utilization task τ_i misses its deadline when scheduled on n_i cores by a greedy scheduler. Therefore, during the D_i time steps between the release of this job and its deadline, there are fewer than L_i incomplete steps; otherwise, by Lemma 3, the job would have completed. Therefore, by Lemma 2, the scheduler must have finished at least $n_i D_i - (n_i - 1)L_i$ work.

$$\begin{aligned} & n_i D_i - (n_i - 1)L_i = n_i(D_i - L_i) + L_i \\ = & \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil (D_i - L_i) + L_i \\ \geq & \frac{C_i - L_i}{D_i - L_i}(D_i - L_i) + L_i = C_i \end{aligned}$$

Since the job has at most C_i, it must have finished in D_i steps, leading to a contradiction. □

Low-Utilization Tasks are schedulable We first calculate a lower bound on n_{low}, the number of total cores assigned to low-utilization tasks, when a task set τ that satisfies Conditions (1) and (2) of Definition 1 for $b = 2$ is scheduled using federated scheduling strategy.

Lemma 4. *The number of cores assigned to low-utilization tasks is at least $n_{low} \geq 2\sum_{low} u_i$.*

Proof: As defined in Section II, the critical path utilization $\delta_i = \frac{L_i}{D_i}$. Here, for the brevity of the proof, we denote $\sigma_i = \frac{1}{\delta_i} = \frac{D_i}{L_i}$. It is obvious that $D_i = \sigma_i L_i$ and hence $C_i = D_i u_i = \sigma_i u_i L_i$. Therefore,

$$n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil = \left\lceil \frac{\sigma_i u_i L_i - L_i}{\sigma_i L_i - L_i} \right\rceil = \left\lceil \frac{\sigma_i u_i - 1}{\sigma_i - 1} \right\rceil$$

Since each task τ_i in task set τ satisfies the Condition (2) of Definition 1 for $b = 2$; therefore, the critical-path length of each task is at most $1/b$ of its relative deadline, that is, $\Delta_i \leq 1/b \implies \sigma_i \geq b = 2$.

By the definition of high-utilization task τ_i, we have $1 \leq u_i$. Together with $\sigma_i \geq 2$, the following formula is always non-negative:

$$0 \leq \frac{(u_i - 1)(\sigma_i - 2)}{\sigma_i - 1}$$

[4]Even if these tasks are expressed as parallel programs, it is easy to enforce correct sequential execution of parallel tasks — any topological ordered execution of the nodes of the dag is a valid sequential execution.

From the definition of ceiling, we can derive

$$\begin{aligned} n_i &= \left\lceil \frac{\sigma_i u_i - 1}{\sigma_i - 1} \right\rceil \\ &< \frac{\sigma_i u_i - 1}{\sigma_i - 1} + 1 = \frac{\sigma_i u_i + \sigma_i - 2}{\sigma_i - 1} \\ &\leq \frac{\sigma_i u_i + \sigma_i - 2}{\sigma_i - 1} + \frac{(u_i - 1)(\sigma_i - 2)}{\sigma_i - 1} \\ &= \frac{\sigma_i u_i + \sigma_i - 2 + \sigma_i u_i - 2u_i - \sigma_i + 2}{\sigma_i - 1} \\ &= \frac{2\sigma_i u_i - 2u_i}{\sigma_i - 1} = \frac{2u_i(\sigma_i - 1)}{\sigma_i - 1} = 2u_i \\ &= bu_i \end{aligned}$$

In summary, for each high-utilization task, $n_i < bu_i$. So their sum τ_{high} satisfies $n_{\text{high}} = \sum_{\text{high}} n_i < b \sum_{\text{high}} u_i$. Since the task set also satisfies Condition (1), we have

$$n_{\text{low}} = m - n_{\text{high}} > b \sum_{\text{all}} u_i - b \sum_{\text{high}} u_i = b \sum_{\text{low}} u_i$$

So the number of remaining cores allocated to low-utilization tasks is at least $n_{\text{low}} > 2 \sum_{\text{low}} u_i$. □

Corollary 1. *For task sets satisfying Conditions (1) and (2), a multiprocessor scheduler with utilization bound of at least* 50% *can schedule all the low-utilization tasks sequentially on the remaining* n_{low} *cores.*

Proof: Low-utilization tasks are allocated n_{low} cores, and from Lemma 4 we know that the total utilization of the low utilization tasks is less than $n_{\text{low}}/b = 50\% n_{\text{low}}$. Therefore, any multiprocessor scheduling algorithm that provides a utilization bound of 2 (i.e. can schedule any task set with total worst-case utilization ratio no more than 50%) can schedule it. □

As mentioned in Section IV, many multiprocessor scheduling algorithms provide a utilization bound of 2 (i.e. 50%) to sequential tasks. That is, given n_{low} cores, they can schedule any task set with a total worst-case utilization up to $0.5 n_{\text{low}}$. Therefore, all these multiprocessor schedulers can be used to schedule low utilization tasks by enforcing their sequential execution. For example, federated algorithm can use partitioned EDF or partitioned RM for τ_{low} and τ_{low} will meet all deadlines.

C. Schedulability Analysis

The capacity augmentation bound of 2 for federated scheduling functions functions as a simple schedulability test, since we can safely admit task sets that satisfy $\sum u_i \leq m/2$ and $L_i \leq D_i/2$ for each task τ_i, but this test is often pessimistic, especially for tasks with high parallelism.

More importantly, note that the federated scheduling algorithm described in Section III-A can also be directly used as a (polynomial-time) schedulabily test: given a task set, after assigning cores to each high-utilization task using our algorithm, if the remaining cores are sufficient for all low-utilization tasks, then the task set is schedulable and we can admit it without deadline miss. This schedulability test admits a strict subset of tasks admitted by the bound, so in practice it aften admits many task sets with utilization greater than $50\% m$.

D. Lower Bound on Capacity Augmentation of Federated Scheduling

Here, we show that the capacity augmentation bound of 2 of federated scheduling is tight by constructing an example to show that the lower bound on capacity augmentation of federated scheduling is also 2.

Given a system with cores of speed $b = \frac{2}{1+\epsilon} < 2$, where $0 < \epsilon < 1$ is an arbitrarily small positive number, then the speed of the cores is arbitrarily close to 2. Consider such a system with $m = 2 + i$ cores, where i is a positive integer, we construct a task set τ with a single parallel task τ_1 with high-utilization $u_1 = 1 + 0.5i$. We further assume that its critical-path length utilization is $\delta_1 = 1/\sigma_1 = (1+\epsilon)/2$. Therefore, the deadline of task τ_1 can be represented as $D_1 = \sigma_1 L_1 = 2L_1/(1+\epsilon)$ and its total work is $C_1 = u_1 D_1 = (1 + 0.5i)2L_1/(1+\epsilon)$.

We can see that converted from system with speed of $b = 2/(1+\epsilon)$, the two conditions from Definition 1 are both satisfied on unit speed cores:

$$\begin{aligned} &\text{Condition (1), } u_1 = 1 + 0.5i \leq m/b \\ &\qquad = (2+i)(1+\epsilon)/2 = (1+0.5i)(1+\epsilon) \\ &\text{Condition (2), } L_1 \leq D_1/b = \frac{2L_1/(1+\epsilon)}{2/(1+\epsilon)} = L_1 \end{aligned}$$

Hence, by the definition of a capacity augmentation bound, if federated scheduling could have a capacity augmentation bound of $b = 2/(1+\epsilon) < 2$, then this constructed task set should be schedulable under federated scheduling algorithm.

However, as we calculate the number of cores needed for this single high-utilization task τ_1 to be schedulable using federated scheduling algirthm in Section III-A

$$\begin{aligned} n_1 &= \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil = \left\lceil \frac{(1+0.5i)2L_1/(1+\epsilon) - L_i}{2L_1/(1+\epsilon) - L_i} \right\rceil \\ &= \left\lceil \frac{(1+0.5i)2 - (1+\epsilon)}{2 - (1+\epsilon)} \right\rceil = \left\lceil \frac{1 + i - \epsilon}{1 - \epsilon} \right\rceil \\ &= \left\lceil \frac{(1+i)(1-\epsilon) + \epsilon i}{1 - \epsilon} \right\rceil = \left\lceil (1+i) + \frac{\epsilon i}{1-\epsilon} \right\rceil \\ &> (1+i) = m \qquad \text{(since } 0 < \epsilon < 1\text{)} \end{aligned}$$

we can see that the number of cores required for the schedulability of task τ_1 is larger than the total number of available cores m. Therefore, this task set is unschedulable under federated scheduling algorithm with the speed-up of $b = 2/(1+\epsilon) < 2$, which contradicts the assumption.

As for any speed-up $1 < b < 2$ we can construct above task set on a system with $m > 2$ cores that is unschedulable using federated scheduling, we can conclude that the lower bound on capacity augmentation of federated scheduling is at least 2. Note that this lower bound is true for all multicore systems with different numbers of cores (larger than 2). Since we have shown that the upper bound on capacity augmentation of federated scheduling is also 2, we have closed the gap between the lower and upper bound. Therefore, the capacity augmentation bound of federated scheduling is strictly 2.

IV. Related Work

In this section, we review closely related work on real-time scheduling, concentrating primarily on scheduling task sets with parallel tasks.

Real-time multiprocessor scheduling considers scheduling sequential tasks on computers with multiple processors or cores and has been studied extensively (see [10, 19] for a survey). In addition, platforms such as Litmus^{RT} [14, 16] have been designed to support these task sets. Here, we review a few relevant theoretical results. Researchers have proven both resource augmentation bounds, utilization bounds and capacity augmentation bounds. The best known utilization bound for global EDF for sequential tasks on a multiprocessor is 2 (traditionally stated as $1/b = 50\%$)[7]; therefore, global EDF trivially provides a resource and capacity augmentation bound of 2 as well. Partitioned EDF and versions partitioned static priority schedulers also provide a utilization bound of 2 [3, 33]. Global RM provides a capacity augmentation bound of 3 [2] to implicit deadline tasks.

For parallel real-time tasks, most early work considered intra-task parallelism of limited task models such as *malleable tasks* [18, 27, 30] and *moldable tasks* [34]. Kato et al. [27] studied the Gang EDF scheduling of moldable parallel task systems.

Researchers have since considered more realistic task models that represent programs that are typically generated by commonly used general purpose parallel programming languages such as Cilk family [12, 26], OpenMP [37], and Intel's Thread Building Blocks [39]. These languages and libraries generally support primitives such as parallel-for loops and fork/join or spawn/sync in order to expose parallelism within the programs. Using these constructs in various combinations generates tasks whose structure can be represented with different types of DAGs.

Tasks with one particular structure, namely *parallel synchronous tasks*, have been studied more than others in the real-time community. These tasks are generated if only we use only parallel-for loops to generate parallelism. Lakshmanan et al. [29] proved a (capacity) augmentation bound of 3.42 for a restricted synchronous task model which is generated when we restrict each parallel-for loop in a task to have the same number of iterations. General synchronous tasks (with no restriction on the number of iterations in the parallel-for loops), have also been studied [4, 28, 36, 40]. (More details on these results were presented in Section I) Chwa et al. [17] provide a response time analysis.

If we do not restrict the primitives used to parallel-for loops, we get a more general task model — most easily represented by a general directed acyclic graph. A resource augmentation bound of $2 - \frac{1}{m}$ for G-EDF was proved for a single DAG with arbitrary deadlines [8] and for multiple DAGs [13, 31]. A capacity augmentation bound of $4 - \frac{2}{m}$ was proved in [31] for tasks with for implicit deadlines. Liu and Anderson [32] provide a response time analysis for G-EDF.

There has been significant work on scheduling parallel systems in the non-real time context [5, 6, 20–22, 38]. In this context, the goal is generally to maximize throughput; tasks have no deadlines or periods. Various provably good scheduling strategies, such as list scheduling [15, 24] and work-stealing [11] have been designed. In addition, many platforms have been built based on these results: examples include parallel languages and runtime systems, such as the Cilk family [12, 26], OpenMP [37], and Intel's Thread Building Blocks [39]. While multiple tasks on a single platform have been considered in the context of fairness in resource allocation [1], none of this work considers real-time constraints.

V. Conclusion

This paper presents a novel federated approach for scheduling parallel real-time tasks (for both deterministic and stochastic task models). For hard-real time tasks, this strategy provides the best known theoretical capacity augmentation bound of 2. The federated scheduling strategy is promising due to its simplicity since it separately schedules high-utilization tasks on dedicated cores and low-utilization cores on shared cores; therefore, one can potentially use out-of-the-box schedulers in a prototype implementation.

References

[1] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu. "Adaptive work-stealing with parallelism feedback". In: *ACM Trans. Comput. Syst.* 26 (3 2008), pp. 112–120.

[2] B. Andersson, S. Baruah, and J. Jonsson. "Static-priority scheduling on multiprocessors". In: *Real Time Systems Symposium*. 2001, pp. 193–202.

[3] B. Andersson and J. Jonsson. "The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%". In: *Euromicro Conference on Real Time Systems*. 2003, pp. 33–40.

[4] B. Andersson and D. de Niz. “Analyzing Global-EDF for Multiprocessor Scheduling of Parallel Tasks”. In: *Principles of Distributed Systems*. 2012, pp. 16–30.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. “Thread Scheduling for Multiprogrammed Multiprocessors”. In: *SPAA ’98*.

[6] N. Bansal, K. Dhamdhere, J. Konemann, and A. Sinha. “Non-clairvoyant Scheduling for Minimizing Mean Slowdown”. In: *Algorithmica* 40.4 (2004), pp. 305–318.

[7] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. “Improved Multiprocessor Global Schedulability Analysis”. In: *Real-Time Syst.* 46.1 (Sept. 2010), pp. 3–24. ISSN: 0922-6443.

[8] S. Baruah, V. Bonifaciy, A. Marchetti-Spaccamelaz, L. Stougiex, and A. Wiese. “A generalized parallel task model for recurrent real-time processes”. In: *Real Time Systems Symposium*. 2012.

[9] S. K. Baruah, A. K. Mok, and L. E. Rosier. “Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor”. In: *IEEE Real-Time Systems Symposium*. 1990, pp. 182–190.

[10] M. Bertogna and S. Baruah. “Tests for global EDF schedulability analysis”. In: *Journal of System Architecture* 57.5 (2011), pp. 487–497.

[11] R. D. Blumofe and C. E. Leiserson. “Scheduling multithreaded computations by work stealing”. In: *Journal of the ACM* 46.5 (1999), pp. 720–748.

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. “Cilk: An Efficient Multithreaded Runtime System”. In: *PPoPP*. Santa Barbara, California, 1995, pp. 207–216.

[13] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. “Feasibility Analysis in the Sporadic DAG Task Model”. In: *ECRTS*. 2013, pp. 225–233.

[14] B. B. Brandenburg, A. D. Block, J. M. Calandrino, U. Devi, H. Leontyev, and J. H. Anderson. *LITMUS RT: A Status Report*. 2007.

[15] R. P. Brent. “The Parallel Evaluation of General Arithmetic Expressions”. In: *Journal of the ACM* (1974), pp. 201–206.

[16] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. “LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers”. In: *Proceedings of the 27th IEEE International Real-Time Systems Symposium*. RTSS ’06. 2006, pp. 111–126. ISBN: 0-7695-2761-2.

[17] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin. “Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms”. In: *ECRTS ’13*.

[18] S. Collette, L. Cucu, and J. Goossens. “Integrating job parallelism in real-time scheduling theory”. In: *Information Processing Letters* 106.5 (2008), pp. 180–187.

[19] R. I. Davis and A. Burns. “A survey of hard real-time scheduling for multiprocessor systems”. In: *ACM Computing Surveys* 43 (4 2011), 35:1–44.

[20] X. Deng, N. Gu, T. Brecht, and K. Lu. “Preemptive Scheduling of Parallel Jobs on Multiprocessors”. In: *SODA ’96*.

[21] M. Drozdowski. “Real-time scheduling of linear speedup parallel tasks”. In: *Inf. Process. Lett.* 57 (1 1996), pp. 35–40.

[22] J. Edmonds, D. D. Chinn, T. Brecht, and X. Deng. “Non-clairvoyant Multiprocessor Scheduling of Jobs with Changing Execution Characteristics”. In: *Journal of Scheduling* 6.3 (2003), pp. 231–250.

[23] N. Fisher, J. Goossens, and S. Baruah. “Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible”. In: *Real-Time Systems Journal* 45.1-2 (2010), pp. 26–71.

[24] R. L. Graham. “Bounds on Multiprocessing Anomalies”. In: *SIAM Journal on Applied Mathematics* (1969), 17(2):416–429.

[25] H.-M. Huang, T. Tidwell, C. Gill, C. Lu, X. Gao, and S. Dyke. “Cyber-physical systems for real-time hybrid structural testing: a case study”. In: *International Conference on Cyber Physical Systems*. 2010.

[26] *Intel CilkPlus*. http://software.intel.com/en-us/articles/intel-cilk-plus.

[27] S. Kato and Y. Ishikawa. “Gang EDF Scheduling of Parallel Task Systems”. In: *Proceedings of the Real Time Systems Symposium*. 2009.

[28] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. “Parallel scheduling for cyber-physical systems: analysis and case study on a self-driving car”. In: *ICCPS*. 2013, pp. 31–40.

[29] K. Lakshmanan, S. Kato, and R. R. Rajkumar. “Scheduling Parallel Real-Time Tasks on Multi-core Processors”. In: *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*. RTSS ’10. 2010, pp. 259–268. ISBN: 978-0-7695-4298-0.

[30] W. Y. Lee and H. Lee. “Optimal Scheduling for Real-Time Parallel Tasks”. In: *IEICE Transactions on Information Systems* E89-D.6 (2006), pp. 1962–1966.

[31] J. Li, K. Agrawal, C.Lu, and C. Gill. “Analysis of Global EDF for Parallel Tasks”. In: *Euromicro Conference on Real Time Systems*. 2013.

[32] C. Liu and J. Anderson. “Supporting Soft Real-Time Parallel Applications on Multicore Processors”. In: *RTCSA*. 2012.

[33] J. M. López, J. L. Díaz, and D. F. García. “Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems”. In: *Real-Time Systems Journal* 28.1 (Oct. 2004), pp. 39–68.

[34] G. Manimaran, C. S. R. Murthy, and K. Ramamritham. “A New Approach for Scheduling of Parallelizable Tasks inReal-Time Multiprocessor Systems”. In: *Real-Time Syst.* 15 (1 1998), pp. 39–60.

[35] A. K. Mok. *FUNDAMENTAL DESIGN PROBLEMS OF DISTRIBUTED SYSTEMS FOR THE HARD-REAL-TIME ENVIRONMENT*. Tech. rep. Cambridge, MA, USA, 1983.

[36] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. “Techniques optimizing the number of processors to schedule multi-threaded tasks”. In: *Euromicro Conference Real Time Systems*. 2012.

[37] *OpenMP Application Program Interface v3.1*. http://www.openmp.org/mp-documents/OpenMP3.1.pdf. 2011.

[38] C. D. Polychronopoulos and D. J. Kuck. “Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers”. In: *Computers, IEEE Transactions on* C-36.12 (1987), pp. 1425 – 1439.

[39] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O’Reilly Media, 2010.

[40] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. “Multi-core Real-Time Scheduling for Generalized Parallel Task Models”. In: *Real Time Systems Symposium*. 2011.

[41] R. Sedgewick and K. D. Wayne. *Algorithms*. 4th. Addison-Wesley Professional, 2011. ISBN: 9780321573513.