Parallel Real-Time Scheduling of DAGs

Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill

Abstract—Recently, multi-core processors have become mainstream in processor design. To take full advantage of multi-core processing, computation-intensive real-time systems must exploit intra-task parallelism. In this paper, we address the problem of real-time scheduling for a general model of deterministic parallel tasks, where each task is represented as a directed acyclic graph (DAG) with nodes having arbitrary execution requirements. We prove processor-speed augmentation bounds for both preemptive and non-preemptive real-time scheduling for general DAG tasks on multi-core processors. We first decompose each DAG into sequential tasks with their own release times and deadlines. Then we prove that these decomposed tasks can be scheduled using preemptive global EDF with a resource augmentation bound of 4. This bound is as good as the best known bound for more restrictive models, and is the first for a general DAG model. We also prove that the decomposition has a resource augmentation bound of 4 plus a constant non-preemption overhead for non-preemptive global EDF scheduling. To our knowledge, this is the first resource augmentation bound for non-preemptive scheduling of parallel tasks. Finally, we evaluate our analytical results through simulations that demonstrate that the derived resource augmentation bounds are safe in practice.

Index Terms—Parallel task, multi-core processor, real-time scheduling, resource augmentation bound

1 Introduction

As the rate of increase of clock frequencies is leveling off, most processor chip manufacturers have recently moved to increasing performance by increasing the number of cores on a chip. Intel's 80-core Polaris [1], Tilera's 100-core TILE-Gx, AMD's 12-core Opteron [2], and ClearSpeed's 96-core processor [3] are some notable examples of multi-core chips. With the rapid evolution of multi-core technology, however, real-time system software and programming models have failed to keep pace. Most classic results in real-time scheduling concentrate on sequential tasks running on multiple processors [4]. While these systems allow many tasks to execute on the same multi-core host, they do not allow an individual task to run any faster on it than on a single-core machine.

To scale the capabilities of individual tasks with the number of cores, it is essential to develop new approaches for tasks with intra-task parallelism, where each real-time task itself is a parallel task that can utilize multiple cores at the same time. Here, we take autonomous vehicle [5] as a motivating example. Such a system consists of a myriad of real-time tasks such as motion planning, sensor fusion, computer vision, and decision making algorithms that exhibit intra-task parallelism. For example, the decision making subsystem processes massive amounts of data from various types of sensors, where the data processing on different types of sensors can run in parallel. Such intra-task parallelism may enable timing guarantees for many complex real-time systems requiring heavy computation, whose stringent timing constraints are difficult to meet on traditional single-core processors.

There has been some recent work on real-time scheduling for parallel tasks, but it has been mostly restricted to the synchronous task model [6], [7], [8]. In the *synchronous model*, each task consists of a sequence of segments with synchronization points at the end of each segment. In addition, each segment of a task contains threads of execution that are of *equal* length. For synchronous tasks, the result in [6], [8] proves a resource augmentation bound of four under global earliest deadline first (EDF) scheduling. A *resource augmentation bound* ν of a scheduling policy $\mathbb{A}$ indicates that if there is any way to schedule a task set on m identical unit-speed processor cores, then $\mathbb{A}$ is guaranteed to successfully schedule it on m cores with each core being ν times as fast as the original.

While the synchronous task model represents the tasks generated by the *parallel for* loop construct common to many parallel languages such as OpenMP [9] and CilkPlus [10], most parallel languages also have other constructs for generating parallel programs, notably *fork-join* constructs. A program that uses fork-join constructs will generate a *non-synchronous* task, generally represented as a directed acyclic graph (DAG), where each thread (sequence of instructions) is a node, and the edges represent dependencies between the threads. A node's execution requirement can vary arbitrarily, and different nodes in the same DAG can have different execution requirements.

Another limitation of the state-of-the-art is that all prior work on parallel real-time tasks considers *preemptive scheduling*, where threads are allowed to preempt each other in the middle of execution. Preemption can be a high-overhead since it often involves a system call and a context switch. An alternative scheduling model is to consider *node-level non-preemptive scheduling* (called non-preemptive scheduling in this paper), where once the execution of a particular node (thread) starts it cannot be preempted by any other thread. Most parallel languages and libraries have yield points at

• *The authors are with the Department of Computer Science and Engineering, Washington University in St. Louis, 1 Brookings Dr, St. Louis, MO 63130. E-mail: {saifullah, dferry, li.jing, kunal, lu, cdgill}@wustl.edu.*

Manuscript received 4 Sept. 2013; revised 24 Dec. 2013; accepted 30 Dec. 2013. Date of publication 15 Jan. 2014; date of current version 14 Nov. 2014. Recommended for acceptance by A. Mellouk.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.2297919

the end of threads (nodes of a DAG), allowing low-cost, user-space preemption at these yield points. For these, schedulers that switch context only when threads end can be implemented entirely in user-space, and therefore have low overheads. In addition, fewer switches imply lower caching overhead. In this model, since a node is never preempted, if it accesses the same memory location multiple times, those locations will be cached, and a node never has to restart on a cold cache.

This paper addresses the hard real-time scheduling of a set of generalized DAGs sharing a multi-core machine. We generalize the previous work in two important directions. First, we consider a general model of deterministic parallel tasks, where each task is represented by a general DAG in which nodes can have *arbitrary* execution requirements. Second, we address both *preemptive* and *non-preemptive* scheduling. In particular, we make the following new contributions.

- We propose a novel task decomposition to transform the nodes of a general DAG into sequential tasks. Since each node of the DAG becomes an individual sequential task, these tasks can be scheduled either preemptively or non-preemptively.
- We prove that any set of parallel tasks of a general DAG model, upon decomposition, can be scheduled using preemptive global EDF with a resource augmentation bound of 4. This bound is as good as the best known bound for more restrictive models [6] and, to our knowledge, is the *first* bound for a general DAG model.
- We prove that our decomposition requires a resource augmentation bound of $4 + 2\rho$ for non-preemptive global EDF scheduling, where ρ is the *non-preemption overhead* of the tasks. To our knowledge, this is the *first* bound for *non-preemptive* scheduling of parallel real-time tasks.
- Through simulations, we demonstrate that the derived bounds are safe, and reasonably tight in practice, especially under preemptive EDF that requires a resource augmentation of 3.2 in simulation as opposed to our analytical bound of four.

Section 2 reviews related work. Section 3 describes the task model. Section 4 presents the decomposition algorithm. Sections 5 and 6 present analyses for preemptive and non-preemptive global EDF scheduling, respectively. Section 7 presents the simulation results.

2 Related Work

There has been a substantial amount of work on traditional multiprocessor real-time scheduling focused on sequential tasks [4]. Scheduling of parallel tasks without deadlines has been addressed in [11], [12]. *Soft real-time scheduling*, where the goal is to meet a subset of deadlines based on some application-specific criterion, for parallel task has been studied for optimizing cache misses [13], makespan [14], and total work done within the deadlines [15]. In contrast, we address *hard real-time scheduling* where the goal is to meet all task deadlines. Hard real-time scheduling is a fundamental requirement in many important application domains such as video surveillance, radar tracking, and autonomous vehicle [5].

An exact (i.e., both sufficient and necessary) schedulability analysis under hard real-time system is intractable for most cases of parallel tasks [16]. Early works on hard real-time parallel scheduling make simplifying assumptions about task models. For example, the study in [17], [18] considers EDF scheduling of parallel tasks where the actual number of processors used by a particular task is determined before starting the system, and remains unchanged.

Recently, *preemptive* real-time scheduling has been studied [6], [7], [8] for *synchronous* parallel tasks with implicit deadlines. In [7], every task is an alternate sequence of parallel and sequential *segments* with each parallel segment consisting of multiple threads of *equal* length that synchronize at the end of the segment. All parallel segments in a task have an *equal* number of threads which cannot *exceed* the number of processor cores. Each thread is transformed into a subtask, and a resource augmentation bound of 3.42 is claimed under partitioned deadline monotonic (DM) scheduling. This result was later generalized for synchronous model with arbitrary numbers of threads in segments, with bounds of four and five for global EDF and partitioned DM scheduling, respectively [6], and also to minimize the required number of processors [19].

Scheduling and analysis of DAGs introduces a challenging open problem. For this general model, an augmentation bound has been analyzed recently in [20], but it considers a *single* DAG on a multi-core machine with preemption. Our earlier work [6] has proposed a simple extension to a synchronous task scheduling approach that handles *unit-node DAG* where each node has unit execution requirement. The work in [8] is an implementation of our work in [6]. However, most parallel languages that use fork-join constructs generate a *non-synchronous* task, generally represented as a DAG where each node's execution requirement can vary arbitrarily, and different nodes in the same DAG can have different execution requirements. The decomposition in [6] for restrictive model is not applicable for general DAG. If it is extended to general DAG, it may split each node of a DAG into multiple subtasks, thereby disallowing node-level non-preemptive scheduling. Also, it will make preemptive scheduling inefficient and costly due to excessive numbers of contexts switches due to node splitting and artificially increased synchronization.

3 Parallel Task Model

We consider n periodic parallel tasks to be scheduled on a multi-core platform consisting of m identical cores. The task set is represented by $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task $\tau_i, 1 \leq i \leq n$, is represented as a directed acyclic graph, where the *nodes* stand for different execution requirements, and the *edges* represent dependencies between the nodes.

A node in τ_i is denoted by $W_i^j, 1 \leq j \leq n_i$, with n_i being the total number of nodes in τ_i. The *execution requirement* of node W_i^j is denoted by E_i^j. A directed edge from node W_i^j to node W_i^k, denoted as $W_i^j \rightarrow W_i^k$, implies that the execution of W_i^k cannot start until W_i^j finishes. W_i^j, in this case, is called a *parent* of W_i^k, while W_i^k is its *child*. A node may have 0 or more parents or children, and can start execution only

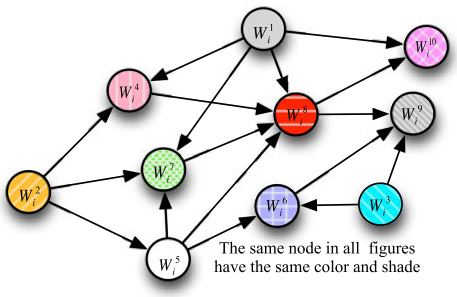


Fig. 1. A parallel task τ_i represented as a DAG.

after all of its parents have finished execution. Fig. 1 shows a task τ_i with $n_i = 10$ nodes.

The *execution requirement* (i.e., *work*) C_i of task τ_i is the sum of the execution requirements of all nodes in τ_i; that is, $C_i = \sum_{j=1}^{n_i} E_i^j$. Thus, C_i is the *maximum execution time* of τ_i if it was executing on a single processor of speed 1. For task τ_i, the *critical path length*, denoted by P_i, is the sum of execution requirements of the nodes on a critical path. A *critical path* is a directed path that has the maximum execution requirement among all other paths in DAG τ_i. Thus, P_i is the *minimum execution time* of τ_i meaning that it needs at least P_i time units on unit-speed processor cores even when the number of cores m is infinite. The *period* of task τ_i is denoted by T_i and the deadline D_i of each task τ_i is considered *implicit*, i.e., $D_i = T_i$. Since P_i is the minimum execution time of task τ_i even on a machine with an infinite number of cores, the condition $T_i \geq P_i$ must hold for τ_i to be schedulable (i.e., to meet its deadline). A task set is said to be *schedulable* when all tasks in the set meet their deadlines.

4 Task Decomposition

We schedule parallel tasks by decomposing each parallel task into smaller sequential tasks. The main intuition for decomposing a parallel task into a set of sequential tasks is that the scheduling of parallel task reduces to the scheduling of sequential tasks, allowing us to leverage existing schedulability analysis for traditional multiprocessor scheduling. In this section, we present a decomposition technique for a parallel task under a general DAG model. Upon decomposition, each node of a DAG becomes an individual sequential task, called a *subtask*, with its own deadline and with an execution requirement equal to the node's execution requirement. We use the terms 'subtask' and 'node' interchangeably. All nodes of a DAG are assigned appropriate deadlines and release offsets such that when they execute as individual subtasks all dependencies among them in the original DAG are preserved. The deadlines of the subtasks of a DAG are assigned by splitting the DAG's deadline. The decomposition ensures that if the subtasks of a DAG are schedulable, then the DAG must be schedulable. Thus, an implicit deadline DAG is decomposed into a set of constrained deadline (i.e., deadline is no greater than period) sequential subtasks with each subtask corresponding to a node of the DAG.

Our schedulability analysis for parallel tasks entails deriving a resource augmentation bound [6], [7]. In particular, our result aims at procuring the following claim: If an optimal algorithm can schedule a task set on a machine of m unit-speed processor cores, then our algorithm can schedule this task set on m processor cores, each of speed ν, where ν is the *resource augmentation* factor. Since an optimal algorithm is unknown, we pessimistically assume that an optimal scheduler can schedule a task set if each task of the set has a critical-path length no greater than its deadline, and the total utilization of the task set is no greater than m. No algorithm can schedule a task set that does not meet these conditions. Our resource augmentation analysis is based on the densities of the decomposed tasks, where the *density* of any task is the ratio of its execution requirement to its deadline.

4.1 Terminology

The *utilization* u_i of a task τ_i, and the *total utilization* $u_{\text{sum}}(\tau)$ for any task set τ of n tasks are defined as

$$u_i = \frac{C_i}{T_i}; \qquad u_{\text{sum}}(\tau) = \sum_{i=1}^{n} \frac{C_i}{T_i}.$$

If u_{sum} is greater than m, then no algorithm can schedule τ on m identical unit-speed processor cores.

The *density* δ_i of any task τ_i, and the *total density* $\delta_{\text{sum}}(\tau)$ and the *maximum density* $\delta_{\max}(\tau)$ for any set τ of n tasks are defined as follows:

$$\delta_i = \frac{C_i}{D_i}; \quad \delta_{\text{sum}}(\tau) = \sum_{i=1}^{n} \delta_i; \quad \delta_{\max}(\tau) = \max\{\delta_i | 1 \leq i \leq n\}. \tag{1}$$

The *demand bound function* (DBF) of task τ_i is the largest cumulative execution requirement of all jobs generated by τ_i that have both arrival times and deadlines within a contiguous interval of t time units. For any task τ_i, the DBF is given by

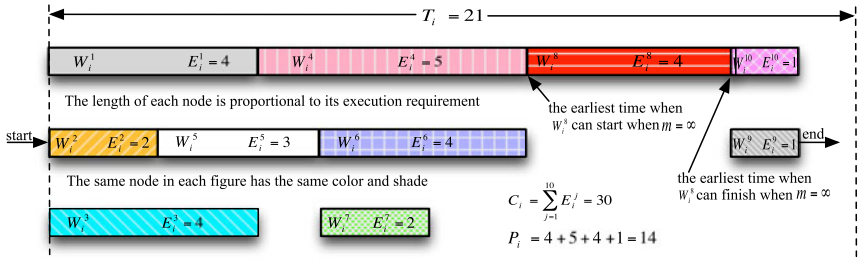
$$\text{DBF}(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right). \tag{2}$$

Based on the DBF, the *load*, denoted by $\lambda(\tau)$, of any task set τ consisting of n tasks is defined as follows:

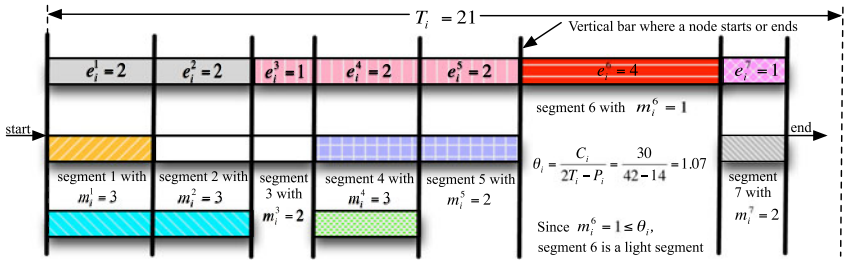
$$\lambda(\tau) = \max_{t>0}\left(\frac{\sum_{i=1}^{n} \text{DBF}(\tau_i, t)}{t}\right). \tag{3}$$

4.2 Decomposition Algorithm

In the decomposition, the intermediate subdeadline assigned to a node is called *node deadline*. Note that once task τ_i is released, it has a total of T_i time units to finish its execution. The proposed decomposition algorithm splits this deadline T_i into node deadlines by preserving the dependencies in τ_i. For task τ_i, the deadline and the offset assigned to node W_i^j are denoted by D_i^j and Φ_i^j, respectively. Once appropriate values of D_i^j and Φ_i^j are determined for each node W_i^j (respecting the dependencies in the DAG), task τ_i is decomposed into nodes. Upon decomposition, the



(a) τ_i^{∞}: a timing diagram for when τ_i executes on an infinite number of processor cores



(b) τ_i^{syn}

Fig. 2. τ_i^{∞} and τ_i^{syn} of DAG τ_i (of Fig. 1).

dependencies in the DAG need not be considered, and each node can execute as a traditional sequential multiprocessor task. Hence, the decomposition technique for τ_i boils down to determining D_i^j and Φ_i^j for each node W_i^j as presented below. The presentation is accompanied by an example using the DAG τ_i from Fig. 1. For the example, we assign execution requirement of each node W_i^j as follows: $E_i^1 = 4$, $E_i^2 = 2$, $E_i^3 = 4$, $E_i^4 = 5$, $E_i^5 = 3$, $E_i^6 = 4$, $E_i^7 = 2$, $E_i^8 = 4$, $E_i^9 = 1$, $E_i^{10} = 1$. Hence, $C_i = 30$, $P_i = 14$. Let period $T_i = 21$.

To perform the decomposition, we first represent DAG τ_i as a *timing diagram* τ_i^{∞} (Fig. 2a) that shows its execution time on an infinite number of unit-speed processor cores. Specifically, τ_i^{∞} indicates the earliest start time and the earliest finishing time (of the worst case execution requirement) of each node when $m = \infty$. For any node W_i^j that has no parents, the *earliest start time* and the *earliest finishing time* are 0 and E_i^j, respectively. For every other node W_i^j, the *earliest start time* is the latest finishing time among its parents, and the *earliest finishing time* is E_i^j time units after that. For example, in τ_i of Fig. 1, nodes W_i^1, W_i^2, and W_i^3 can start execution at time 0, and their earliest finishing times are 4, 2, and 4, respectively. Node W_i^4 can start after W_i^1 and W_i^2 complete, and finish after 5 time units at its earliest, and so on. Fig. 2a shows τ_i^{∞} for DAG τ_i. Next, based on τ_i^{∞}, the calculation of D_i^j and Φ_i^j for each node W_i^j involves the following two steps. In Step 1, for each node, we estimate the time requirement at different parts of the node. In Step 2, the total estimated time requirements at different parts of the node is assigned as the node's deadline.

As stated before, our resource augmentation analysis is based on the densities of the decomposed tasks. The efficiency of the analysis is largely dependent on the total density (δ_{sum}) and the maximum density (δ_{max}) of the decomposed tasks. Namely, we need to keep both δ_{sum} and δ_{max} bounded and as small as possible to minimize the resource augmentation requirement. Therefore, the objective of the decomposition algorithm is to split the entire task deadline into node deadlines and to keep their densities small so that each node (subtask) has enough slack. The *slack* of any task represents the extra time beyond its execution requirement and is defined as the difference between its deadline and execution requirement.

4.2.1 Estimating Time Requirements of the Nodes

In DAG τ_i, a node can execute with different numbers of nodes in parallel at different times. Such a degree of parallelism can be estimated based on τ_i^{∞}. For example, in Fig. 2a, node W_i^5 executes with W_i^1 and W_i^3 in parallel for the first 2 time units, and then executes with W_i^4 in parallel for the next time unit. In this way, we first identify the degrees of parallelism at different parts of each node. Intuitively, the parts of a node that may execute with a large number of nodes in parallel demand more time. Therefore, different parts of a node are assigned different amounts of time considering these degrees of parallelism and execution requirements. Later, the total time of all parts of a node is assigned to the node as its deadline.

To identify the degree of parallelism for different portions of a node based on τ_i^{∞}, we assign time units to a node in different (consecutive) segments. In different segments of a node, the task may have different degrees of parallelism. In τ_i^{∞}, starting from the beginning, we draw a vertical line at every time instant where a node starts or ends (as shown in Fig. 2b). This is done in linear time using a breadth-first search over the DAG. The vertical lines now split τ_i^{∞} into segments. For example, in Fig. 2b, τ_i is split into seven segments (numbered from left to right).

Once τ_i^{∞} is split into segments, each segment consists of an equal amount of execution by the nodes that lie in the segment. Parts of different nodes in the same segment can now be thought of as *threads of execution* that run in parallel,

and the threads in a segment can start only after those in the preceding segment finish. We denote this synchronous form of τ_i^{∞} by τ_i^{syn}. We first allot time to the segments, and finally add all times allotted to different segments of a node to calculate its deadline.

We split T_i time units among the nodes based on the number of threads and execution requirement of the segments where a node lies in τ_i^{syn}. We first estimate time requirement for each segment. Let τ_i^{syn} be a sequence of s_i segments numbered as $1, 2, \ldots, s_i$. For any segment j, we use m_i^j to denote the *number of threads in the segment*, and e_i^j to denote the *execution requirement of each thread* in the segment (see Fig. 2b). Since τ_i^{syn} has the same critical path and total execution requirements as those of τ_i,

$$P_i = \sum_{j=1}^{s_i} e_i^j; \qquad C_i = \sum_{j=1}^{s_i} m_i^j.e_i^j.$$

For any segment j of τ_i^{syn}, we calculate a value d_i^j, called the *segment deadline*, so that the segment is assigned a total of d_i^j time units to finish all its threads. Now we calculate the value d_i^j that minimizes both thread density and segment density that would lead to minimizing δ_{sum} and $\delta_{\max}$ upon decomposition.

Since segment j consists of m_i^j parallel threads, with each thread having an execution requirement of e_i^j, the total execution requirement of segment j is $m_i^j e_i^j$. Thus, the segments with larger numbers of threads and with longer threads are computation-intensive, and demand more time to finish execution. Therefore, a reasonable way to assign the segment deadlines is to split T_i proportionally among the segments by considering their total execution requirement. Such a policy assigns a segment deadline of $\frac{T_i}{C_i} m_i^j e_i^j$ to segment j. Since this is the deadline for each parallel thread of segment j, by Equation (1), the density of a thread becomes $\frac{C_i}{m_i^j T_i}$ which can be as large as m. Hence, such a method does not minimize $\delta_{\max}$, and is not useful. Instead, we classify the segments of τ_i^{syn} into two groups based on a threshold θ_i on the number threads per segment: each segment j with $m_i^j > \theta_i$ is called a *heavy segment*, and each segment j with $m_i^j \leq \theta_i$ is called a *light segment*. Among the heavy segments, we allocate a portion of time T_i that is no less than that allocated among the light ones. Before assigning time among the segments, we determine a value of θ_i and the fraction of time T_i to be split among the heavy and light segments.

We show below that choosing $\theta_i = \frac{C_i}{2T_i - P_i}$ helps us keep both thread density and segment density bounded. Therefore, each segment j with $m_i^j > \frac{C_i}{2T_i - P_i}$ is classified as a *heavy segment* while other segments are called *light segments*. Let H_i denote the *set of heavy segments*, and L_i denote the *set of light segments* of τ_i^{syn}. This raises three different cases: when $L_i = \emptyset$ (i.e., when τ_i^{syn} consists of only heavy segments), when $H_i = \emptyset$ (i.e., when τ_i^{syn} consists of only light segments), and when $H_i \neq \emptyset, L_i \neq \emptyset$ (i.e., when τ_i^{syn} consists of both light segments and heavy segments). We use three different approaches for these three scenarios.

Case 1: when $H_i = \emptyset$. Since each segment has a smaller number ($\leq \frac{C_i}{2T_i - P_i}$) of threads, we only consider the length of a thread in each segment to assign time for it. Hence, T_i time units is split proportionally among all segments according to the length of each thread. For each segment j, its deadline d_i^j is calculated as follows:

$$d_i^j = \frac{T_i}{P_i} e_i^j. \tag{4}$$

Since the condition $T_i \geq P_i$ must hold for every task τ_i to be schedulable,

$$d_i^j = \frac{T_i}{P_i} e_i^j \geq \frac{T_i}{T_i} e_i^j = e_i^j. \tag{5}$$

Hence, the maximum density of a thread in any segment is at most 1. Since a segment has at most $\frac{C_i}{2T_i - P_i}$ threads, and $T_i \geq P_i$, the segment's density is at most

$$\frac{C_i}{2T_i - P_i} \leq \frac{C_i}{2T_i - T_i} = \frac{C_i}{T_i}. \tag{6}$$

Case 2: when $L_i = \emptyset$. All segments are heavy, and T_i time units is split proportionally among all segments according to the work (i.e., total execution requirement) of each segment. For each segment j, its deadline d_i^j is given by

$$d_i^j = \frac{T_i}{C_i} m_i^j e_i^j. \tag{7}$$

Since for every segment j, $m_i^j > \frac{C_i}{2T_i - P_i}$, we have

$$d_i^j = \frac{T_i}{C_i} m_i^j e_i^j > \frac{T_i}{C_i} \frac{C_i}{2T_i - P_i} e_i^j = \frac{2T_i}{2(2T_i - P_i)} e_i^j \geq \frac{e_i^j}{2}. \tag{8}$$

Hence, the maximum density of any thread is at most 2. The total density of segment j is at most

$$\frac{m_i^j e_i^j}{d_i^j} = \frac{m_i^j e_i^j}{\frac{T_i}{C_i} m_i^j e_i^j} = \frac{C_i}{T_i}. \tag{9}$$

Case 3: when $H_i \neq \emptyset$ and $L_i \neq \emptyset$. The task has both heavy segments and light segments. A total of $(T_i - P_i/2)$ time units is assigned to heavy segments, and the remaining $P_i/2$ time units is assigned to light segments. $(T_i - P_i/2)$ time units is split proportionally among heavy segments according to the work of each segment. The total execution requirement of heavy segments of τ_i^{syn} is denoted by C_i^{heavy}, defined as

$$C_i^{\text{heavy}} = \sum_{j \in H_i} m_i^j.e_i^j.$$

For each heavy segment j, the deadline d_i^j is

$$d_i^j = \frac{T_i - \frac{P_i}{2}}{C_i^{\text{heavy}}} m_i^j e_i^j. \tag{10}$$

Since for each heavy segment j, $m_i^j > \frac{C_i}{2T_i - P_i}$, we have

$$d_i^j = \frac{(T_i - \frac{P_i}{2}) m_i^j e_i^j}{C_i^{\text{heavy}}} > \frac{(T_i - \frac{P_i}{2}) \frac{C_i}{2T_i - P_i} e_i^j}{C_i^{\text{heavy}}} \geq \frac{e_i^j}{2}. \tag{11}$$

Hence, maximum density of a thread in any heavy segment is at most 2. As $T_i \geq P_i$, the total density of a heavy segment becomes

$$\frac{m_i^j e_i^j}{d_i^j} = \frac{m_i^j e_i^j}{\frac{T_i - \frac{P_i}{2}}{C_i^{\text{heavy}}} m_i^j e_i^j} = \frac{C_i^{\text{heavy}}}{T_i - \frac{P_i}{2}} \leq \frac{C_i}{T_i - \frac{T_i}{2}} = \frac{2C_i}{T_i}. \quad (12)$$

Now, to distribute time among the light segments, $P_i/2$ time units is split proportionally among light segments according to the length of each thread. The critical path length of light segments is denoted by P_i^{light}, and is defined as follows:

$$P_i^{\text{light}} = \sum_{j \in L_i} e_i^j.$$

For each light segment j, the deadline d_i^j is

$$d_i^j = \frac{\frac{P_i}{2}}{P_i^{\text{light}}} e_i^j. \quad (13)$$

The density of a thread in any light segment is at most 2 since

$$d_i^j = \frac{\frac{P_i}{2}}{P_i^{\text{light}}} e_i^j \geq \frac{\frac{P_i}{2}}{P_i} e_i^j = \frac{e_i^j}{2}. \quad (14)$$

Since a light segment has at most $\frac{C_i}{2T_i - P_i}$ threads, and $T_i \geq P_i$, the total density of a light segment is at most

$$\frac{2C_i}{2T_i - P_i} \leq \frac{2C_i}{2T_i - T_i} = \frac{2C_i}{T_i}. \quad (15)$$

4.2.2 Calculating Deadline and Offset for Nodes

We have assigned segment deadlines to (the threads of) each segment of τ_i^{syn} in Step 1 (Equations (4), (7), (10), (13)). Since a node may be split into multiple (consecutive) segments in τ_i^{syn}, now we have to remove all segment deadlines of a node to reconstruct (restore) the node. Namely, we add all segment deadlines of a node, and assign the total as the node's deadline.

Now let a node W_i^j of τ_i belong to segments k to r $(1 \leq k \leq r \leq s_i)$ in τ_i^{syn}. Therefore, the deadline D_i^j of node W_i^j is calculated as follows.

$$D_i^j = d_i^k + d_i^{k+1} + \cdots + d_i^r. \quad (16)$$

Note the execution requirement E_i^j of node W_i^j is

$$E_i^j = e_i^k + e_i^{k+1} + \cdots + e_i^r. \quad (17)$$

Node W_i^j cannot start until all of its parents complete. Hence, its release offset Φ_i^j is determined as follows.

$$\Phi_i^j = \begin{cases} 0; & \text{if } W_i^j \text{ has no parent} \\ \max\{\Phi_i^l + D_i^l | W_i^l & \text{otherwise.} \\ \quad \text{is a parent of } W_i^j\};. & \end{cases}$$

Now that we have assigned an appropriate deadline D_i^j and release offset Φ_i^j to each node W_i^j of τ_i, the DAG τ_i is now decomposed into nodes. Each node W_i^j is now an individual (sequential) multiprocessor subtask with an execution requirement E_i^j, a constrained deadline D_i^j, and a release offset Φ_i^j. Note that the period of W_i^j is still the same as that of the original DAG which is T_i. The release offset Φ_i^j ensures that node W_i^j can start execution no earlier than W_i^j time units following the release time of the original DAG. Our method guarantees that for a general DAG no node is split into smaller subtasks to ensure node-level non-preemption. Thus, the (node-level) non-preemptive behavior of the original task is preserved in scheduling the nodes as individual tasks, where nodes of the DAG are never preempted. The entire decomposition method is presented as Algorithm 1 in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.2297919, which runs in linear time (in terms of the DAG size i.e., number of nodes and edges). Fig. 7 in Appendix B, available in the online supplemental material, shows the complete decomposition of τ_i. Appendix C, available in the online supplemental material, provides a sketch (Fig. 8) on how it can be implemented on a real system.

4.3 Density Analysis after Decomposition

After decomposition, let τ_i^{dec} denote all subtasks (i.e., nodes) that τ_i generates. Note that the densities of all such subtasks comprise the density of τ_i^{dec}. Now we analyze the density of τ_i^{dec} which will later be used to analyze schedulability.

Let node W_i^j of τ_i belong to segments k to r $(1 \leq k \leq r \leq s_i)$ in τ_i^{syn}. Since W_i^j has been assigned deadline D_i^j, by Equations (16) and (17), its density δ_i^j after decomposition is

$$\delta_i^j = \frac{E_i^j}{D_i^j} = \frac{e_i^k + e_i^{k+1} + \cdots + e_i^r}{d_i^k + d_i^{k+1} + \cdots + d_i^r}. \quad (18)$$

By Equations (5), (8), (11), (14), $d_i^k \geq \frac{e_i^k}{2}$, $\forall i, k$. Hence, from (18),

$$\delta_i^j = \frac{E_i^j}{D_i^j} \leq \frac{2e_i^k + 2e_i^{k+1} + \cdots + 2e_i^r}{e_i^k + e_i^{k+1} + \cdots + e_i^r} = 2. \quad (19)$$

Let τ^{dec} be the set of all generated subtasks of all original DAG tasks, and $\delta_{\max}$ be the *maximum density* among all subtasks in τ^{dec}. By Equation (19),

$$\delta_{\max} = \max\left\{\delta_i^j \middle| 1 \leq j \leq n_i, 1 \leq i \leq n\right\} \leq 2. \quad (20)$$

We use $D_{\min}$ to denote the minimum deadline among all subtasks in τ^{dec}. That is,

$$D_{\min} = \min\left\{D_i^j \middle| 1 \leq j \leq n_i, 1 \leq i \leq n\right\}. \quad (21)$$

Theorem 1. *Let a DAG τ_i, $1 \leq i \leq n$, with period T_i, critical path length P_i where $T_i \geq P_i$, and maximum execution requirement C_i be decomposed into subtasks (nodes) denoted τ_i^{dec} using the decomposition technique (Algorithm 1 in Appendix, available in the online supplemental material). The density of τ_i^{dec} is at most $\frac{2C_i}{T_i}$.*

Proof. Since we decompose τ_i into nodes, the densities of all decomposed nodes W_i^j, $1 \leq j \leq n_i$, comprise the density of τ_i^{dec}. In Step 1, every node W_i^j of τ_i is split into threads in different segments of τ_i^{syn}, and each segment is assigned a segment deadline. In Step 2, we remove all segment deadlines in the node, and their total is assigned as the node's deadline. If τ_i is scheduled in the form of τ_i^{syn}, then each segment is scheduled after its preceding segment is complete. That is, at any time at most one segment is active. By Equations (6), (9), (12), (15), a segment has density at most $\frac{2C_i}{T_i}$ (considering $T_i \geq P_i$). Hence, the overall density of τ_i^{syn} never exceeds $\frac{2C_i}{T_i}$. Therefore, it is sufficient to prove that removing segment deadlines in the nodes does not increase the task's overall density. That is, it is sufficient to prove that the density δ_i^j (Equation (18)) of any node W_i^j after removing its segment deadlines is no greater than the density $\delta_i^{j,\text{syn}}$ that it had before removing its segment deadlines.

Let node W_i^j of the original DAG task τ_i be split into threads in segments k to r ($1 \leq k \leq r \leq s_i$) in τ_i^{syn}. Since the total density of any set of tasks is an upper bound on its load (as proven in [21]), the load of the threads of W_i^j must be no greater than the total density of these threads. Since each of these threads is executed only once in the interval of D_i^j time units, based on Equation (2), the DBF of the thread, $thread_i^l$, in segment l, $k \leq l \leq r$, in the interval of D_i^j time units is expressed as

$$\text{DBF}\left(thread_i^l, D_i^j\right) = e_i^l.$$

Therefore, using Equation (3), the load, denoted by $\lambda_i^{j,\text{syn}}$, of the threads of W_i^j in τ_i^{syn} for interval D_i^j is

$$\lambda_i^{j,\text{syn}} \geq \frac{e_i^k}{D_i^j} + \frac{e_i^{k+1}}{D_i^j} + \cdots + \frac{e_i^r}{D_i^j} = \frac{E_i^j}{D_i^j} = \delta_i^j.$$

Since $\delta_i^{j,\text{syn}} \geq \lambda_i^{j,\text{syn}}$, for any W_i^j, we have $\delta_i^{j,\text{syn}} \geq \delta_i^j$. □

Let δ_{sum} be the *total density* of all subtasks τ^{dec}. Since, from Theorem 1, the density of each τ_i^{dec} is at most $\frac{2C_i}{T_i}$ where $T_i \geq P_i$,

$$\delta_{\text{sum}} \leq \sum_{i=1}^{n} \frac{2C_i}{T_i} = 2\sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{22}$$

5 Preemptive EDF Scheduling

Once all DAG tasks are decomposed into nodes (i.e., subtasks), we consider scheduling the nodes. Since every node after decomposition becomes a sequential task, we schedule them using traditional multiprocessor scheduling policies. In this section, we consider the preemptive global EDF policy.

Lemma 2. *For any set of DAGs $\tau = \{\tau_1, \ldots, \tau_n\}$, let τ^{dec} be the decomposed task set. If τ^{dec} is schedulable under some preemptive scheduling, then τ is preemptively schedulable.*

Proof. See Appendix D, available in the online supplemental material. □

To schedule the decomposed subtasks τ^{dec}, the EDF policy is the same as the traditional global EDF policy where jobs with earlier absolute deadlines have higher priorities. Due to the *preemptive* policy, a job can be suspended (preempted) at any time by arriving higher-priority jobs, and is later resumed with (in theory) no cost or penalty. Under preemptive global EDF, we now present a schedulability analysis for τ^{dec} in terms of a resource augmentation bound which, by Lemma 2, is also a sufficient analysis for the original DAG task set τ. For a task set, a *resource augmentation bound* ν of a scheduling policy $\mathbb{A}$ on an m-core machine is a processor speed-up factor. That is, if there exists any way to schedule the task set on m identical unit-speed processor cores, then $\mathbb{A}$ is guaranteed to successfully schedule it on an m-core processor with each core being ν times as fast as the original.

Our analysis hinges on a result (Theorem 3) for preemptive global EDF scheduling of constrained deadline sporadic tasks on a traditional multiprocessor platform [22]. This result is a generalization of the result for implicit deadline tasks [23].

Theorem 3. *(From [22]) Any constrained deadline sporadic sequential task set π with total density $\delta_{\text{sum}}(\pi)$ and maximum density $\delta_{\max}(\pi)$ is schedulable using preemptive global EDF policy on m unit-speed processor cores if*

$$\delta_{\text{sum}}(\pi) \leq m - (m-1)\delta_{\max}(\pi).$$

Note that τ^{dec} consists of constrained deadline (sub) tasks that are periodic with offsets. If they do not have offsets, then the above condition directly applies. Taking the offsets into account, the execution requirement, the deadline, and the period (which is equal to the period of the original DAG) of each subtask remains unchanged. The release offsets only ensure that some subtasks of the same original DAG are not executed simultaneously to preserve the precedence relations in the DAG. This implies that both δ_{sum} and $\delta_{\max}$ of the subtasks with offsets are no greater than δ_{sum} and $\delta_{\max}$, respectively, of the same set of tasks with no offsets. Hence, Theorem 3 holds for τ^{dec}. We now use the results of density analysis from Section 4.3, and prove that τ^{dec} is guaranteed to be schedulable with a resource augmentation of at most 4 in Corollary 1 that follows Theorem 4.

Theorem 4. *For any set of DAGs $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, let τ^{dec} be the decomposed task set. If every DAG τ_i satisfies the condition $T_i \geq P_i$, and the DAG set τ satisfies the condition $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m$ on m identical unit-speed processor cores, then the decomposed task set τ^{dec} is guaranteed to be schedulable under preemptive global EDF on m processor cores, each of speed 4.*

Proof. If each DAG τ_i satisfies the condition $T_i \geq P_i$, then the total density δ_{sum} of the decomposed task set τ^{dec} is at most $2\sum_{i=1}^{n} \frac{C_i}{T_i}$ (Equation (22)), and the maximum density $\delta_{\max}$ of τ^{dec} is at most 2 (Equation (20)) on unit-speed processors. To be able to schedule the decomposed tasks τ^{dec}, let each processor core be of speed ν, where $\nu > 1$. On an m-core platform where each core has speed ν, let the total density and the maximum density of task set τ^{dec} be denoted by $\delta_{\text{sum},\nu}$ and $\delta_{\max,\nu}$, respectively.

Considering that the condition $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m$ holds for τ, the total density of decomposed tasks τ^{dec} from

Equation (22) is derived as follows on ν-speed cores.

$$\delta_{\text{sum},\nu} = \frac{\delta_{\text{sum}}}{\nu} \leq 2\sum_{i=1}^{n}\frac{\frac{C_i}{\nu}}{T_i} = \frac{2}{\nu}\sum_{i=1}^{n}\frac{C_i}{T_i} \leq \frac{2m}{\nu}. \tag{23}$$

On ν-speed cores, the maximum density of τ^{dec} is derived from Equation (20) as follows.

$$\delta_{\max,\nu} = \frac{\delta_{\max}}{\nu} \leq \frac{2}{\nu}. \tag{24}$$

Using Conditions (24) and (23) in Theorem 3, τ^{dec} is schedulable under preemptive EDF policy on m processor cores each of speed ν if

$$\frac{2m}{\nu} \leq m - (m-1)\frac{2}{\nu} \Leftrightarrow \frac{4}{\nu} - \frac{2}{m\nu} \leq 1.$$

From the above condition, τ^{dec} must be schedulable if

$$\frac{4}{\nu} \leq 1 \Leftrightarrow \nu \geq 4.$$

□

Corollary 1. *For any set of DAGs $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$, let τ^{dec} be the decomposed task set. If there exists any algorithm that can schedule τ on m unit-speed processor cores, then the decomposed task set τ^{dec} is guaranteed to be schedulable under preemptive global EDF on m cores, each of speed 4.*

Proof. If there exists any algorithm that can schedule τ on m unit-speed processor cores, then the following two conditions must hold.

$$\sum_{i=1}^{n}\frac{C_i}{T_i} \leq m \tag{25}$$

$$T_i \geq P_i, \text{ for each } \tau_i. \tag{26}$$

Hence, the proof follows from Theorem 4. □

Since Theorem 4 holds, we have the following straightforward schedulability test based on the resource augmentation bound of 4 for any set of DAGs: *For any set of DAGs $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, if the total utilization $u_{\text{sum}}(\tau) \leq \frac{m}{4}$ and every DAG τ_i individually satisfies condition $P_i \leq \frac{T_i}{4}$, then the task set is schedulable under preemptive EDF policy upon decomposition.*

6 Non-Preemptive EDF Scheduling

We now address non-preemptive global EDF scheduling considering that the original task set τ is scheduled based on node-level non-preemption. In *node-level non-preemptive scheduling*, whenever the execution of a node in a DAG starts, the node's execution cannot be preempted by any task.

The decomposition converts each node of a DAG to a traditional multiprocessor (sub)task. Therefore, we consider fully non-preemptive global EDF scheduling of the decomposed tasks. Namely, once a job of a decomposed (sub)task starts execution, it cannot be preempted by any other job.

Lemma 5. *For any set of DAGs $\tau = \{\tau_1, \ldots, \tau_n\}$, let τ^{dec} be the decomposed task set. If τ^{dec} is schedulable under some fully non-preemptive scheduling, then τ is schedulable under node-level non-preemption.*

Proof. See Appendix E, available in the online supplemental material. □

Under non-preemptive global EDF, we now present a schedulability analysis for τ^{dec} in terms of a resource augmentation bound which, by Lemma 5, is also a sufficient analysis for the DAG task set τ. This analysis exploits Theorem 6 for non-preemptive global EDF scheduling of constrained deadline periodic tasks on traditional multiprocessor. The theorem is a generalization of the result for implicit deadline tasks [24].

For a task set π, let $C_{\max}(\pi)$ and $D_{\min}(\pi)$ be the maximum execution requirement and the minimum deadline among all tasks in π. In non-preemptive scheduling, $C_{\max}(\pi)$ represents the *maximum blocking time* that a task may experience, and plays a major role in schedulability. Hence, a *non-preemption overhead*, defined in [24], for the task set π is given by $\rho(\pi) = \frac{C_{\max}(\pi)}{D_{\min}(\pi)}$. The value of $\rho(\pi)$ indicates the added penalty or overhead associated with non-preemptivity. In other words, since preemption is not allowed, the capacity of each processor is reduced (at most) by a factor of $\rho(\pi)$. In non-preemptive scheduling, this capacity reduction is recompensed by reducing the cost associated with context-switch, saving state, etc.

Theorem 6. *(From [24]) Any constrained deadline periodic task set π with total density $\delta_{\text{sum}}(\pi)$, maximum density $\delta_{\max}(\pi)$, and a non-preemption overhead $\rho(\pi)$ is schedulable using non-preemptive global EDF on m unit-speed cores if*

$$\delta_{\text{sum}}(\pi) \leq m\big(1 - \rho(\pi)\big) - (m-1)\delta_{\max}(\pi).$$

Let $E_{\max}$ and $E_{\min}$ be the maximum and minimum execution requirement, respectively, among all nodes of all DAG tasks. That is,

$$E_{\max} = \max\Big\{E_i^j \big| 1 \leq j \leq n_i, 1 \leq i \leq n\Big\} \tag{27}$$

$$E_{\min} = \min\Big\{E_i^j \big| 1 \leq j \leq n_i, 1 \leq i \leq n\Big\}. \tag{28}$$

In node-level non-preemptive scheduling of the DAGs, the processor capacity reduction due to non-preemptivity is at most $\frac{E_{\max}}{E_{\min}}$. Hence, this value is the *non-preemption overhead* of the DAGs denoted by ρ:

$$\rho = \frac{E_{\max}}{E_{\min}}. \tag{29}$$

Theorem 7 derives a resource augmentation bound of $4 + 2\rho$ for non-preemptive global EDF scheduling of the decomposed tasks. A tighter bound analysis is provided in Appendix E, available in the online supplemental material.

Theorem 7. *For DAG model parallel tasks $\tau = \{\tau_1, \ldots, \tau_n\}$, let τ^{dec} be the decomposed task set with non-preemption overhead ρ. If there exists any way to schedule τ on m unit-speed processor cores, then τ^{dec} is schedulable under non-preemptive global EDF on m cores, each of speed $4 + 2\rho$.*

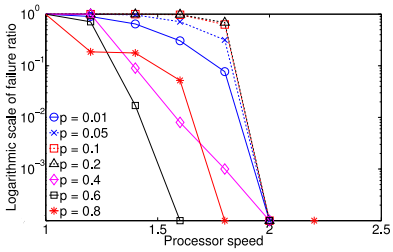


Fig. 3. Failure ratio in preemptive EDF on 32 cores under different edge probability.

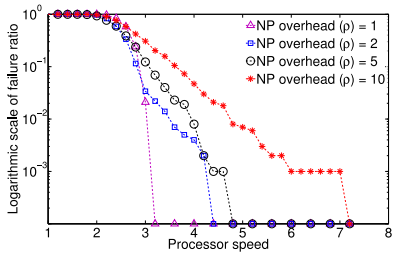


Fig. 5. Failure ratio in non-preemptive EDF on 8 cores under different non-preemption overhead.

Proof. After decomposition, $D_{\min}$ (Equation (21)) is the minimum deadline among all subtasks in τ^{dec}. Since $E_{\max}$ (Equation (27)) represents the maximum blocking time that a subtask may experience, the non-preemption overhead of the decomposed tasks is $\frac{E_{\max}}{D_{\min}}$. From Equations (19) and (29), the non-preemption overhead of the decomposed tasks

$$\frac{E_{\max}}{D_{\min}} \leq \frac{E_{\max}}{E_{\min}/2} = \frac{2E_{\max}}{E_{\min}} = 2\rho. \tag{30}$$

Similar to Theorem 4 and Corollary 1, suppose we need each core to be of speed ν to be able to schedule the decomposed tasks τ^{dec}. From Equation (30), the non-preemption overhead of τ^{dec} on ν-speed cores is

$$\frac{E_{\max}/\nu}{D_{\min}} \leq \frac{2\rho}{\nu}. \tag{31}$$

Considering a non-preemption overhead of at most $\frac{2\rho}{\nu}$ on ν-speed processor cores, and using Equations (24) and (23) in Theorem 6, τ^{dec} is schedulable under non-preemptive EDF on m cores each of speed ν if

$$\frac{2m}{\nu} \leq m\left(1 - \frac{2\rho}{\nu}\right) - (m-1)\frac{2}{\nu} \Leftrightarrow \frac{4+2\rho}{\nu} - \frac{1}{m\nu} \leq 1.$$

From the above condition, task set τ^{dec} is schedulable if

$$\frac{4+2\rho}{\nu} \leq 1 \Leftrightarrow \nu \geq 4 + 2\rho. \qquad \square$$

7 Evaluation

In this section, we evaluate our analytical results. We simulate the execution of a set of parallel tasks under scheduling algorithms to observe deadline misses. We developed a simple event-driven simulator detailed in Appendix F, available in the online supplemental material, where task executions are simulated in parallel as if they executed on m cores.

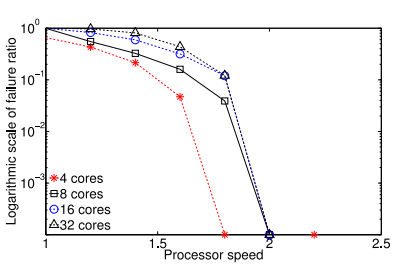


Fig. 4. Failure ratio in preemptive EDF on different numbers of cores.

We use the Erdös-Rényi method $G(n_i, p)$ [25] to generate task sets for evaluation. For each value of m (i.e., the number of cores), we generate task sets whose utilization is exactly m, fully loading a machine of 1-speed processors. The complete task generation method is explained in Appendix F, available in the online supplemental material. We experiment by varying the following 4 parameters: type of task period (harmonic versus arbitrary periods), number of cores (m), probability of an edge in DAG (p), and non-preemption overhead (ρ). The experimental methodology is detailed in Appendix F, available in the online supplemental material.

In all experiments, we simulate $1,000$ task sets. For each task set, we start by simulating its execution on 1-speed processors, and increase the speed by 0.1 intervals until all task sets are schedulable. Using these different task sets, we conduct two sets of experiments. In our first set, we evaluate the scheduler under preemptive global EDF. Hence, we vary the types of period, m and p, but keep ρ constant at 2, leading to 112 combinations. In the second set, we evaluate under non-preemptive global EDF by varying all four factors, leading to 896 combinations.

7.1 Results

Effect of harmonic versus arbitrary periods. This result is discussed in Appendix F, available in the online supplemental material.

Effect of p in preemptive scheduling. For each value of p, Fig. 3 shows the *failure ratio* defined as the ratio of the number of task sets where some task missed a deadline to the total number of task sets (which is $1,000$ in our experiment) attempted to be scheduled. To preserve resolution of the figure, we show the results for only seven (out of 14) values of p. In these experiments, $\rho = 2$, $m = 32$. Note that the failure ratio increases as p increases from 0.01 to 0.1, and then falls again. We have detailed the reasons in Appendix F, available in the online supplemental material. *Effect of m in preemptive scheduling.* Fig. 4 shows that he failure ratio increases as m increases. We have detailed the results in Appendix F, available in the online supplemental material.

Effect of ρ in non-preemptive scheduling. Fig. 5 shows that the failure ratio increases as the discrete ρ increases. The results are detailed in Appendix F, available in the online supplemental material.

Effect of m in non-preemptive scheduling. Fig. 6 shows the required speed for each combination of m and ρ, with $p = 0.2$. We have detailed the results in Appendix F, available in the online supplemental material.

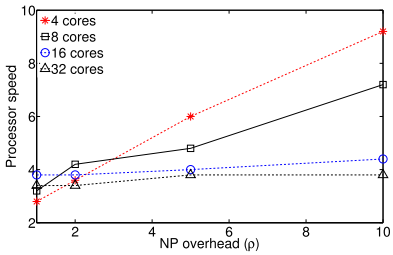


Fig. 6. Required speed in non-preemptive EDF on different numbers of cores with increasing non-preemption overhead.

The simulation results show a maximum speed requirement of 3.2 for preemptive EDF suggesting that our analytical resource augmentation bound of four is reasonably tight. The corresponding bounds for non-preemptive EDF sound relatively looser in our simulation results. This is because, as stated in Section 6, non-preemptivity can cause processor capacity reduction of up to ρ in the worst case. We have discussed this issue in more details in Appendix F, available in the online supplemental material.

8 Conclusions

As multi-core technology becomes mainstream in processor design, real-time scheduling of parallel tasks is crucial to exploit its potential. In this paper, we consider a general task model and through a novel task decomposition we prove a resource augmentation bound of 4 for preemptive EDF, and 4 plus a non-preemption overhead for non-preemptive EDF scheduling. To our knowledge, these are the first bounds for real-time scheduling of general DAGs.

Acknowledgments

This research was supported by US National Science Foundation (NSF) under XPS grant (1337218), CPS grant (1136073), NeTS grant (1017701).

References

[1] http://en.wikipedia.org/wiki/Teraflops_Research_Chip, 2014.
[2] www.amd.com/us/products/server/processors, 2014.
[3] www.clearspeed.com, 2014.
[4] R. Davis and A. Burns, "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems," *ACM Computing Surveys*, vol. 43, aricle, 35, 2011.
[5] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar, "Parallel Scheduling for Cyber-Physical Systems: Analysis and Case Study on a Self-Driving Car," *Proc. ACM/IEEE Fourth Int'l Conf. Cyber-Physical Systems (ICCPS '13)*, 2013.
[6] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-Core Real-Time Scheduling for Generalized Parallel Task Models," *Proc. IEEE 32nd Real-Time Systems Symp. (RTSS '11)*, 2011.
[7] K. Lakshmanan, S. Kato, and R.R. Rajkumar, "Scheduling Parallel Real-Time Tasks on Multi-Core Processors," *Proc. IEEE 31st Real-Time Systems Symp. (RTSS '10)*, 2010.
[8] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, "A Real-Time Scheduling Service for Parallel Tasks," *Proc. IEEE 19th Real-Time and Embedded Tech. and Applications Symp. (RTAS '13)*, 2013.
[9] "OpenMP,," http://openmp.org, 2014.
[10] http://software.intel.com/en-us/articles/intel-cilk-plus, 2014.
[11] K. Agrawal, Y. He, W.J. Hsu, and C.E. Leiserson, "Adaptive Task Scheduling with Parallelism Feedback," *Proc. ACM SIGPLAN 11th Symp. Principles and Practice of Parallel Programming (PPoPP '06)*, 2006.
[12] K. Agrawal, C.E. Leiserson, Y. He, and W.J. Hsu, "Adaptive Work-Stealing with Parallelism Feedback," *ACM Trans. Computer Systems*, vol. 26, no. 3, article 7, 2008.
[13] J. Anderson and J. Calandrino, "Parallel Real-Time Task Scheduling on Multicore Platforms," *Proc. IEEE 27th Real-Time Systems Symp. (RTSS '06)*, 2006.
[14] Q. Wang and K.H. Cheng, "A Heuristic of Scheduling Parallel Tasks and Its Analysis," *SIAM J. Computing*, vol. 21, no. 2, pp. 281-294, 1992.
[15] O. Kwon and K. Chwa, "Scheduling Parallel Tasks with Individual Deadlines," *Theoretical Computer Science*, vol. 215, pp. 209-223, 1999.
[16] C.-C. Han and K.-J. Lin, "Scheduling Parallelizable Jobs on Multiprocessors," *Proc. Real-Time Systems Symp. (RTSS '89)*, 1989.
[17] G. Manimaran, C. Murthy, and K. Ramamritham, "A New Approach for Scheduling of Parallelizable Tasks in Real-Time Multiprocessor Systems," *Real-Time Systems*, vol. 15, no. 1, pp. 39-60, 1998.
[18] S. Kato and Y. Ishikawa, "Gang EDF Scheduling of Parallel Task Systems," *Proc. IEEE 30th Real-Time Systems Symp. (RTSS '09)*, 2009.
[19] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques Optimizing the Number of Processors to Schedule Multi-Threaded Tasks," *Proc. 24th Euromicro Conf. Real-Time Systems (ECRTS '12)*, 2012.
[20] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A Generalized Parallel Task Model for Recurrent Real-Time Processes," *Proc. IEEE 30th Real-Time Systems Symp. (RTSS '12)*, 2012.
[21] N. Fisher, T.P. Baker, and S. Baruah, "Algorithms for Determining the Demand-Based Load of a Sporadic Task System," *Proc. IEEE 12th Int'l Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA '06)*, 2006.
[22] S. Baruah, "Techniques for Multiprocessor Global Schedulability Analysis," *Proc. IEEE 28th Real-Time Systems Symp. (RTSS '07)*, 2007.
[23] J. Goossens, S. Funk, and S. Baruah, "Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors," *Real-Time Systems*, vol. 25, no. 2-3, pp. 187-205, 2003.
[24] S. Baruah, "The Non-Preemptive Scheduling of Periodic Tasks Upon Multiprocessors," *Real-Time Systems*, vol. 32, pp. 9-20, 2006.
[25] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random Graph Generation for Scheduling Simulations," *Proc. Third Int'l ICST Conf. Simulation Tools and Techniques (SIMUTools '10)*, 2010.
[26] T. Abdelzaher, B. Andersson, J. Jonsson, V. Sharma, and M. Nguyen, "The Aperiodic Multiprocessor Utilization Bound for Liquid Tasks," *Proc. IEEE Eighth Real-Time and Embedded Technology and Applications Symp. (RTAS '02)*, 2002.
[27] http://en.wikipedia.org/wiki/Gamma_distribution, 2014.

Abusayeed Saifullah is working toward the PhD degree in the Department of Computer Science and Engineering, Washington University in St. Louis. Advised by Chenyang Lu, he is a member of the Cyber-Physical Systems Laboratory at Washington University. His research focuses on real-time wireless sensor-actuator networks used in emerging cyber-physical systems, and spans a broad range of topics in wireless sensor networks, embedded systems, real-time systems, and multi-core parallel computing. He has received the Best Student Paper Awards at the 32nd IEEE Real-Time Systems Symposium (RTSS 2011) and at the fifth International Symposium on Parallel and Distributed Processing and Applications (ISPA 2007), and best paper nomination at the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012).

David Ferry is working toward the PhD degree in the Department of Computer Science and Engineering, Washington University in St. Louis. His research interests include parallel computing, real-time parallel systems, and cyber-physical systems.

Jing Li received the bachelor's of science degree in computer science and engineering from the Harbin Institute of Technology, China, in 2011 and is working toward the PhD degree in Computer Science and Engineering department, Washington University in St. Louis. Her research interests include real-time parallel scheduling theory, real-time parallel system and cyber-physical systems.

Kunal Agrawal received the PhD degree in 2009 from MIT on the topic scheduling and synchronization for multicore concurrency platforms. She is an assistant professor in the Department of Computer Science and Engineering at Washington University in Saint Louis. Her research interests include scheduling of parallel programs, parallel algorithms and data structures, synchronization mechanisms, transactional memory, and cache-efficient algorithms.

Chenyang Lu received the BS degree from the University of Science and Technology of China, in 1995, the MS degree from the Chinese Academy of Sciences in 1997, and the PhD degree from the University of Virginia in 2001, all in computer science. He is a professor of computer science and engineering at Washington University in St. Louis. He is an editor-in-chief of *ACM Transactions on Sensor Networks*, area editor of *IEEE Internet of Things Journal* and an associate editor of *Real-Time Systems*. He also serves as a program chair of premier conferences such as IEEE Real-Time Systems Symposium (RTSS 2012), ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS 2012) and ACM Conference on Embedded Networked Sensor Systems (SenSys 2014). He is the author and co-author of more than 100 research papers with over 10,000 citations and an h-index of 47. His research interests include real-time systems, wireless sensor networks and cyber-physical systems.

Christopher D. Gill is a professor of computer science and engineering at Washington University in St. Louis. His research interests include formal modeling, verification, implementation, and empirical evaluation of policies and mechanisms for enforcing timing, concurrency, footprint, fault-tolerance, and security properties in distributed, mobile, embedded, real-time, and cyber-physical systems. He developed the Kokyu real-time scheduling and dispatching framework used in several AFRL and DARPA projects. He led development of the nORB small-footprint real-time object request broker at Washington University. He has also led research projects under which a number of real-time and fault-tolerant services for The ACE ORB (TAO), and the Component Integrated ACE ORB (CIAO), were developed. He has more than 50 refereed technical publications and has an extensive record of service in review panels, standards bodies, workshops, and conferences for distributed real-time and embedded computing.