Report Number: WUCSE-2013-25

2013

Parallel Real-Time Scheduling of DAGs

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

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 open 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...

Read complete abstract on page 2.

Parallel Real-Time Scheduling of DAGs

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

Complete 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 open 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 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 bounds are safe, and reasonably tight in practice, especially under preemptive EDF scheduling.

Parallel Real-Time Scheduling of DAGs

Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill
Department of Computer Science and Engineering
Washington University in St. Louis

***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 open 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 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 bounds are safe, and reasonably tight in practice, especially under preemptive EDF scheduling.**

I. 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.

If we want 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. Such intra-task parallelism may enable timing guarantees for complex real-time systems that require heavy computation, such as video surveillance, computer vision, radar tracking, and hybrid real-time structural testing [5] 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]. 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] proves a resource augmentation bound of 4 under global earliest deadline first (EDF) scheduling. A *resource augmentation* under a scheduling policy quantifies processor speed-up factor (how much we have to increase the speed) with respect to an optimal algorithm to guarantee the schedulability of a task set.

While the synchronous task model represents the kind of tasks generated by the *parallel for* loop construct that is common to many parallel languages such as OpenMP [8] and CilkPlus [9], 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. While this is a reasonable model, preemption can be a high-overhead operation since it often involves a system call and a context switch. An alternative scheduling model is to consider *node-level non-preemptive scheduling* (simply 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 end of threads (nodes of the DAG), allowing low-cost, user-space preemption at these yield points. For these languages and libraries, schedulers that switch context only when threads end (in other words, where threads do not preempt each other) can be implemented entirely in user-space (without interaction with the kernel), and therefore have low overheads. In addition, fewer switches usually imply lower caching overhead. In this model, since a node is never preempted, if it accesses the same memory location multiple times, those memory locations will be cached, and a node never has to restart on a cold cache.

This paper addresses the hard real-time scheduling problem 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 is transformed into a single sequential subtask, these subtasks 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.
- We implement the proposed decomposition algorithm, and evaluate our analytical results for both preemptive and non-preemptive scheduling through simulations. The results indicate 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 4.

Section II reviews related work. Section III describes the task model. Section IV presents the decomposition algorithm. Sections V and VI present analyses for preemptive and non-preemptive global EDF scheduling, respectively. Section VII presents the simulation results. Section VIII offers conclusions.

II. 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 [10]–[15]. *Soft real-time scheduling* (where the goal is to meet a subset of deadlines based on application-specific criteria) has been studied for various parallel task models and optimization criteria such as cache misses [16], [17], makespan [18] and total work done within deadlines [19].

The schedulability analysis under *hard real-time system* (where the goal is to meet all task deadlines) is intractable for most cases of parallel tasks without resource augmentation [20]. Some early work makes simplifying assumptions about task models [21]–[24]. For example, some approaches [21], [22] address the scheduling of *malleable tasks*, where tasks can execute on varying numbers of processors without loss in efficiency. The study in [23] considers non-preemptive EDF scheduling of *moldable tasks*, where the actual number of processors used by a particular task is determined before starting the system, and remains unchanged. Gang EDF scheduling [24] of moldable parallel tasks requires the users to select (at submission time) a fixed number of processors upon which their task will run, and the task must then always use that number of threads.

Recently, *preemptive* real-time scheduling has been studied [6], [7] 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 4 and 5 for global EDF and partitioned DM scheduling, respectively [6], and also to minimize the required number of processors [25].

Our earlier work [6] has proposed a simple extension to a synchronous task scheduling approach that handles *unit-node DAG* where each node has a unit execution requirement by converting each task to a synchronous task allowing direct application of the same approach. This model is quite *restrictive* and *over-simplified* since each node or thread of execution has unit-execution requirement that simplifies the analysis for resource augmentation. However, these assumptions do not hold in general since this model does not represent a parallel task that most parallel languages generate. 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. Notably, the decomposition in [6] for restrictive model is not applicable for a general DAG. If one does so, a single node will split into multiple smaller subtasks, each with its own release time and deadline. As a result, when the decomposed tasks are scheduled, there is no easy way of preserving the node-level non-preemptive behavior of original tasks.

Scheduling and analysis of general DAGs introduces a challenging open problem. For this general model, an augmentation bound has been analyzed recently in [26], but it considers the restricted case of a *single* DAG on a multi-core machine with preemption. In this paper, we investigate the open problem of scheduling and analysis for a set of any number of general DAGs on a multi-core machine. We consider both preemptive and non-preemptive real-time scheduling of general DAG tasks on multi-core processors, and provide resource augmentation bound under both policies.

III. 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 $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$. Each task $\tau_i, 1 \leq i \leq n$, is represented as a Directed Acyclic Graph (DAG), 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

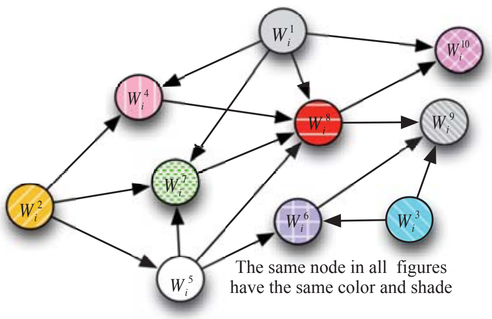


Fig. 1. A parallel task τ_i represented as a DAG

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 unless W_i^j has finished execution. 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. A node can start execution only after all of its parents have finished execution. Figure 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.

IV. Task Decomposition

We schedule parallel tasks by decomposing them into smaller sequential tasks. This strategy allows us to leverage existing schedulability analysis for traditional multiprocessor scheduling (both preemptive and non-preemptive) of sequential tasks. 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. (Henceforth, we will 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 task are preserved. 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. Note that 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. We first present terminology used in decomposition. Then, we present the proposed technique for decomposition, followed by a density analysis of the decomposed tasks.

A. Terminology

Our proposed decomposition technique converts each implicit deadline DAG task into a set of constrained deadline sequential tasks, and is based on the following definitions that are applicable for any task, not limited to just parallel tasks.

The *utilization* u_i of any task τ_i, and the *total utilization* $u_{\text{sum}}(\tau)$ for any task set τ consisting 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 the total utilization 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

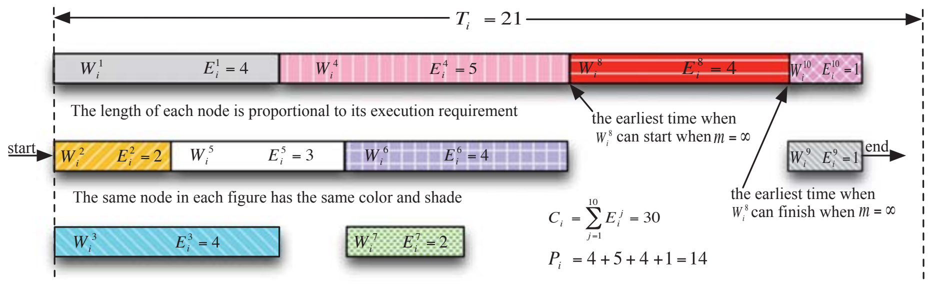
$$\text{DBF}(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right) \tag{2}$$

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

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

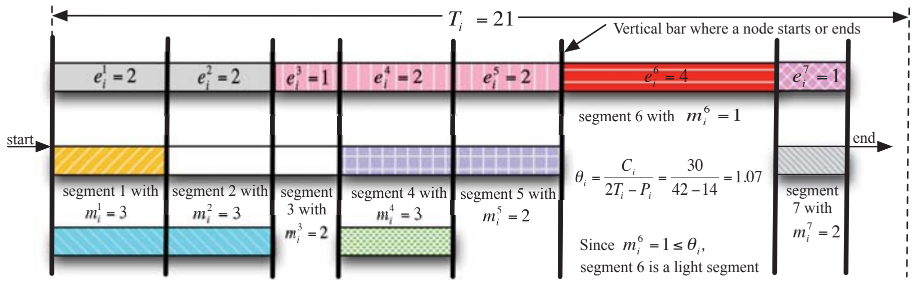
B. Decomposition Algorithm

The decomposition algorithm converts each node of a DAG into an individual sequential subtask with its own execution requirement, release offset, and a constrained deadline. The release offsets are assigned so as to preserve the dependencies



$T_i = 21$
W_i^1 $E_i^1 = 4$
W_i^4 $E_i^4 = 5$
W_i^8 $E_i^8 = 4$
W_i^{10} $E_i^{10} = 1$
The length of each node is proportional to its execution requirement
the earliest time when W_i^8 can start when $m = \infty$
start
W_i^2 $E_i^2 = 2$
W_i^5 $E_i^5 = 3$
W_i^6 $E_i^6 = 4$
W_i^9 $E_i^9 = 1$
end
the earliest time when W_i^8 can finish when $m = \infty$
The same node in each figure has the same color and shade
$C_i = \sum_{j=1}^{10} E_i^j = 30$
W_i^3 $E_i^3 = 4$
W_i^7 $E_i^7 = 2$
$P_i = 4 + 5 + 4 + 1 = 14$

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



$T_i = 21$
Vertical bar where a node starts or ends
$e_i^1 = 2$
$e_i^2 = 2$
$e_i^3 = 1$
$e_i^4 = 2$
$e_i^5 = 2$
$e_i^6 = 4$
$e_i^7 = 1$
segment 6 with $m_i^6 = 1$
start
end
$\theta_i = \frac{C_i}{2T_i - P_i} = \frac{30}{42 - 14} = 1.07$
segment 1 with $m_i^1 = 3$
segment 2 with $m_i^2 = 3$
segment 3 with $m_i^3 = 2$
segment 4 with $m_i^4 = 3$
segment 5 with $m_i^5 = 2$
segment 7 with $m_i^7 = 2$
Since $m_i^6 = 1 \le \theta_i$, segment 6 is a light segment

(b) τ_i^{syn}

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

of the original DAG, namely, to ensure that a node (subtask) can start after the deadlines of all the parent nodes (subtasks). That is, a node starts after its *latest* parent finishes. The (relative) deadlines of the nodes are assigned by splitting the task deadline into intermediate subdeadlines. 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 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 Figure 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^∞ (Figure 2(a)) 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 Figure 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. Figure 2(a) 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, we analyze the schedulability of the decomposed tasks based on their densities. The efficiency of the analysis is largely dependent on the total density (δ_{sum}) and the maximum density ($\delta_{\max}$) of the decomposed tasks. Namely, we need to keep both δ_{sum} and $\delta_{\max}$ bounded and as small as possible (since a higher value of density implies a higher value of execution requirement to deadline ratio) to minimize the resource augmentation requirement. Therefore, the objective of the decomposition algorithm is to split the entire task deadline into node deadlines 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.

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 Figure 2(a), 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 Figure 2(b)). 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 Figure 2(b), τ_i is split into 7 segments (numbered in increasing order 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. Note that τ_i *is never converted to a synchronous model; the procedure only identifies segments to estimate time requirements of nodes, and does not decompose* τ_i *in this step.*

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 $1, 2, \cdots, s_i$. 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 Figure 2(b)). 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 (i.e. total number of processor cores). 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 of the number threads per segment: each segment j with $m_i^j > \theta_i$ is classified as a *heavy segment*, and each segment j with $m_i^j \leq \theta_i$ is classified as 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 segments. Before assigning time among the segments, an important issue is to 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,

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

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

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

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

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

Algorithm 1: Decomposition Algorithm

```
Input: a DAG task $\tau_i$ with period and deadline $T_i$, total execution requirement $C_i$, critical path length $P_i$;
Output: node deadline $D_i^j$, release offset $\Phi_i^j$ for each node $W_i^j$ of $\tau_i$;
for each node $W_i^j$ of $\tau_i$ do $\Phi_i^j \leftarrow 0$; $D_i^j \leftarrow 0$; end
Represent $\tau_i$ as $\tau_i^{\text{syn}}$;
$\theta_i \leftarrow C_i/(2T_i - P_i)$;      /* heavy or light threshold */
$total\_heavy \leftarrow 0$;                 /* total heavy segments */
$total\_light \leftarrow 0$;                 /* total heavy segments */
$C_i^{\text{heavy}} \leftarrow 0$;          /* total work of heavy segments */
$P_i^{\text{light}} \leftarrow 0$;  /* light segments' critical path len. */
for each j-th segment in $\tau_i^{syn}$ do
    if $m_i^j > \theta_i$ then          /* it is a heavy segment */
        $total\_heavy \leftarrow total\_heavy + 1$;
        $C_i^{\text{heavy}} \leftarrow C_i^{\text{heavy}} + m_i^j e_i^j$;
    else                                /* it is a light segment */
        $total\_light \leftarrow total\_light + 1$;
        $P_i^{\text{light}} \leftarrow P_i^{\text{light}} + e_i^j$;
    end
end
if $total\_heavy = 0$ then        /* all segments are light */
    for each j-th segment in $\tau_i^{syn}$ do $d_i^j = \frac{T_i}{P_i} e_i^j$;
else if $total\_light = 0$ then  /* all segments are heavy */
    for each j-th segment in $\tau_i^{syn}$ do $d_i^j = \frac{T_i}{C_i} m_i^j e_i^j$;
else   /* $\tau_i^{\text{syn}}$ has both heavy and light segments */
    for each j-th segment in $\tau_i^{syn}$ do
        if $m_i^j > \theta_i$ then          /* for heavy segment */
            $d_i^j = \frac{T_i - P_i/2}{C_i^{\text{heavy}}} m_i^j e_i^j$;
        else                                /* for light segment */
            $d_i^j = \frac{P_i/2}{P_i^{\text{light}}} e_i^j$;
        end
    end
end
/* Remove seg. deadlines. Assign node deadline */
for each node $W_i^j$ of $\tau_i$ in breadth-first search order do
    if $W_i^j$ belongs to segments k to r in $\tau_i^{syn}$ then
        $D_i^j = d_i^k + d_i^{k+1} + \cdots + d_i^r$;    /* node deadline */
    $\Phi_i^j \leftarrow \max\{\Phi_i^l + D_i^l | W_i^l$ is a parent of $W_i^j\}$;  /* offset */
end
```

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 work (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 calculated as

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

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. The total density of a heavy segment becomes

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

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, its deadline d_i^j is calculated as

$$d_i^j = \frac{\frac{P_i}{2}}{P_i^{\text{light}}} e_i^j \tag{13}$$

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

$$d_i^j = \frac{\frac{P_i}{2}}{P_i^{\text{light}}} e_i^j \geq \frac{\frac{P_i}{2}}{P_i} e_i^j = \frac{e_i^j}{2} \tag{14}$$

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

$$\frac{2C_i}{2T_i - P_i} \leq \frac{2C_i}{2T_i - T_i} = \frac{2C_i}{T_i} \tag{15}$$

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.

$$D_i^j = d_i^k + d_i^{k+1} + \cdots + d_i^r \tag{16}$$

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

$$E_i^j = e_i^k + e_i^{k+1} + \cdots + e_i^r \tag{17}$$

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

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

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 which runs in linear time (in terms of the DAG size i.e., number of nodes and edges). Figure 3 shows the complete decomposition of τ_i.

C. 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 ($1 \le k \le r \le s_i$) 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

$$\delta_i^j = \frac{E_i^j}{D_i^j} = \frac{e_i^k + e_i^{k+1} + \cdots + e_i^r}{d_i^k + d_i^{k+1} + \cdots + d_i^r} \tag{18}$$

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

$$\delta_i^j = \frac{E_i^j}{D_i^j} \le \frac{2e_i^k + 2e_i^{k+1} + \cdots + 2e_i^r}{e_i^k + e_i^{k+1} + \cdots + e_i^r} = 2 \tag{19}$$

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,

$$\delta_{\max} = \max\left\{\delta_i^j \middle| 1 \le j \le n_i,\ 1 \le i \le n\right\} \le 2 \tag{20}$$

Theorem 1. *Let a DAG τ_i, $1 \le i \le n$, with period T_i, critical path length P_i, and maximum execution requirement C_i be decomposed into subtasks (nodes) denoted τ_i^{dec} using Algorithm 1. 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, $1 \le j \le n_i$, 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. Since a segment has density at most $\frac{2C_i}{T_i}$ (Equations 6, 9, 12, 15), the overall density of τ_i^{syn} never exceeds $\frac{2C_i}{T_i}$.

Hence, 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 τ_i be split into threads in segments k to r ($1 \le k \le r \le s_i$) in τ_i^{syn}. Since the total density of any set of tasks is an upper bound on its load (proven in [27]), 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, by Equation 2, the DBF of the thread, $thread_i^l$, in segment l, $k \le l \le r$, in the interval D_i^j

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

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

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

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

Let δ_{sum} be the *total density* of all subtasks τ^{dec}. Then, from Theorem 1,

$$\delta_{\text{sum}} \le \sum_{i=1}^{n} \frac{2C_i}{T_i} = 2\sum_{i=1}^{n} \frac{C_i}{T_i} \tag{21}$$

V. 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.

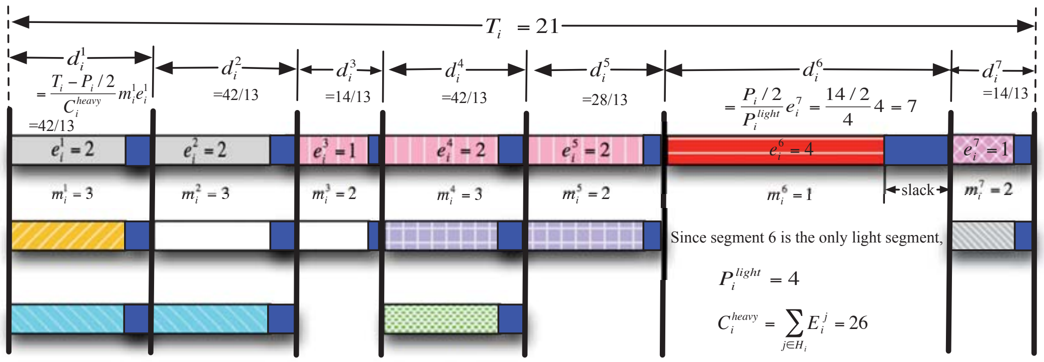
Lemma 2. *For any set of parallel DAG tasks $\tau = \{\tau_1, \cdots, \tau_n\}$, let τ^{dec} be the decomposed task set. If τ^{dec} is schedulable under some preemptive scheduling, then τ is preemptively schedulable.*

Proof: In each τ_i^{dec}, a node is released only after all of its parents finish execution. Hence, the precedence relations in original task τ_i are retained in τ_i^{dec}. Besides, for each τ_i^{dec}, the deadline and the execution requirement are the same as those of original task τ_i. Hence, if τ^{dec} is preemptively schedulable, a preemptive schedule must exist for τ where each task in τ meets its deadline. ■

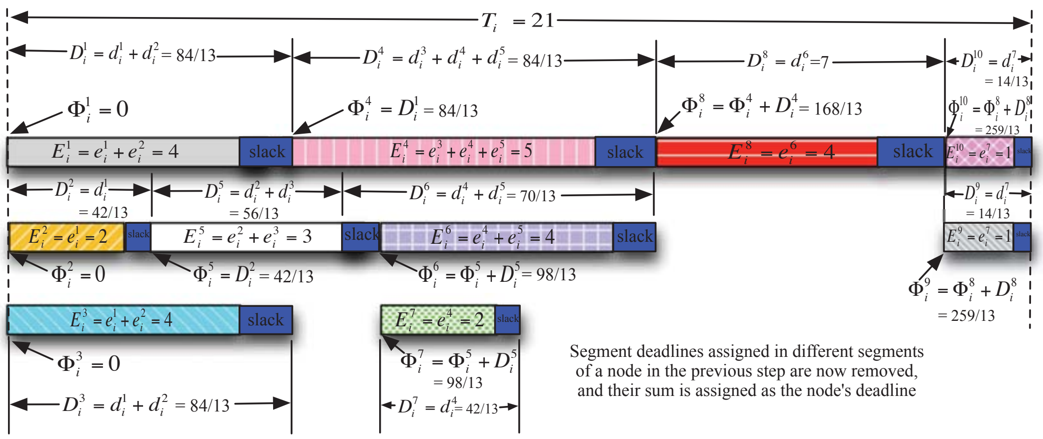
To schedule the decomposed subtasks τ^{dec}, the EDF policy is the same as the traditional global EDF policy where jobs with earlier absolute deadlines have higher priorities. Due to the *preemptive* policy, a job can be suspended (preempted) at any time by arriving higher-priority jobs, and is later 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 a multi-core processor with m cores 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 [28]. This result is a generalization of the result for implicit deadline tasks [29].

Theorem 3. *(From [28]) Any constrained deadline sporadic sequential task set π with total density $\delta_{\text{sum}}(\pi)$ and maximum*



(a) Calculating segment deadlines of τ_i^{syn}



$T_i = 21$
$D_i^1 = d_i^1 + d_i^2 = 84/13$
$D_i^4 = d_i^3 + d_i^4 + d_i^5 = 84/13$
$D_i^8 = d_i^6 = 7$
$D_i^{10} = d_i^7 = 14/13$
$\Phi_i^1 = 0$
$\Phi_i^4 = D_i^1 = 84/13$
$\Phi_i^8 = \Phi_i^4 + D_i^4 = 168/13$
$\Phi_i^{10} = \Phi_i^8 + D_i^8 = 259/13$
$E_i^1 = e_i^1 + e_i^2 = 4$
slack
$E_i^4 = e_i^3 + e_i^4 + e_i^5 = 5$
slack
$E_i^8 = e_i^6 = 4$
slack
$E_i^{10} = e_i^7 = 1$
$D_i^2 = d_i^1 = 42/13$
$D_i^5 = d_i^2 + d_i^3 = 56/13$
$D_i^6 = d_i^4 + d_i^5 = 70/13$
$D_i^9 = d_i^7 = 14/13$
$E_i^2 = e_i^1 = 2$
slack
$E_i^5 = e_i^2 + e_i^3 = 3$
slack
$E_i^6 = e_i^4 + e_i^5 = 4$
slack
$E_i^9 = e_i^7 = 1$
$\Phi_i^2 = 0$
$\Phi_i^5 = D_i^2 = 42/13$
$\Phi_i^6 = \Phi_i^5 + D_i^5 = 98/13$
$\Phi_i^9 = \Phi_i^8 + D_i^8 = 259/13$
$E_i^3 = e_i^1 + e_i^2 = 4$
slack
$E_i^7 = e_i^4 = 2$
slack
$\Phi_i^3 = 0$
$\Phi_i^7 = \Phi_i^5 + D_i^5 = 98/13$
Segment deadlines assigned in different segments of a node in the previous step are now removed, and their sum is assigned as the node's deadline
$D_i^3 = d_i^1 + d_i^2 = 84/13$
$D_i^7 = d_i^4 = 42/13$

(b) Removing segment deadlines, and calculating node deadlines and offsets

Fig. 3. Decomposition of τ_i (shown in Figure 1) when $T_i = 21$

density $\delta_{\max}(\pi)$ is schedulable using preemptive global EDF policy on m unit-speed processor cores if

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

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 Subsection IV-C, and prove that τ^{dec} is guaranteed to be schedulable with a resource augmentation of at most 4 (Theorem 4).

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

Proof: If τ is schedulable on m identical unit-speed processor cores, the following condition must hold.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \le m \tag{22}$$

To be able to schedule the decomposed tasks τ^{dec}, let each processor core be of speed ν. 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. From Equation 20, we have

$$\delta_{\max,\nu} = \frac{\delta_{\max}}{\nu} \le \frac{2}{\nu} \tag{23}$$

Based on Equation 22, when each processor core is of speed ν, the total density of τ^{dec} given in Equation 21 becomes

$$\delta_{\text{sum},\nu} = \frac{\delta_{\text{sum}}}{\nu} \le 2\sum_{i=1}^{n} \frac{\frac{C_i}{\nu}}{T_i} = \frac{2}{\nu}\sum_{i=1}^{n} \frac{C_i}{T_i} \le \frac{2m}{\nu} \tag{24}$$

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

$$\frac{2m}{\nu} \le m - (m-1)\frac{2}{\nu} \iff \frac{4}{\nu} - \frac{2}{m\nu} \le 1$$

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

$$\frac{4}{\nu} \le 1 \iff \nu \ge 4.$$ ■

VI. 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. Most parallel languages and libraries have yield points at the ends of threads (nodes of the DAG), where they allow low cost, user-space preemption. For these languages and libraries, schedulers that switch context only when threads end (in other words, where threads do not preempt each other) can be implemented entirely in user-space (without interaction with the kernel), and therefore have low overheads.

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 a set of DAG parallel tasks $\tau = \{\tau_1, \cdots, \tau_n\}$, 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: Since the decomposition converts each node of a DAG to an individual task, a fully non-preemptive scheduling of τ^{dec} preserves the node-level non-preemptive behavior of task set τ. The rest of the proof follows from Lemma 2. ■

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 [30].

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 [30], 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 [30]) 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. In node-level non-preemptive scheduling of the DAG tasks, 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 DAG tasks, and is denoted by ρ:

$$\rho = \frac{E_{\max}}{E_{\min}} \tag{25}$$

Theorem 7 derives a resource augmentation bound of $4+2\rho$ for non-preemptive global EDF scheduling after decomposition.

Theorem 7. *For DAG model parallel tasks $\tau = \{\tau_1, \cdots, \tau_n\}$, 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 $4 + 2\rho$.*

Proof: After decomposition, let $D_{\min}$ be the minimum deadline among all subtasks in τ^{dec}. Since $E_{\max}$ (i.e. the maximum execution requirement among all subtasks in τ^{dec}) 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 25, the non-preemption overhead of the decomposed tasks

$$\frac{E_{\max}}{D_{\min}} \leq \frac{E_{\max}}{E_{\min}/2} = \frac{2E_{\max}}{E_{\min}} = 2\rho \tag{26}$$

Similar to Theorem 4, suppose we need each processor core to be of speed ν to be able to schedule the decomposed tasks τ^{dec}. From Equation 26, the non-preemption overhead of τ^{dec} on ν-speed processor cores is

$$\frac{E_{\max}/\nu}{D_{\min}} \leq \frac{2\rho}{\nu} \tag{27}$$

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

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

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

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

VII. Evaluation

The derived resource augmentation bounds provide a sufficient condition for schedulability. Namely, if a set of DAG tasks is schedulable on a unit-speed m-core machine by a (potentially unrealizable) ideal scheduler, then the tasks upon our proposed decomposition are guaranteed to be schedulable under global EDF on an m-core machine where each core has a speed of 4 (with preemption) or $4+2\rho$ (without preemption).

In this section, we evaluate our scheduler using simulations. We want to accomplish two things. First, we want to validate that our theoretical bounds are correct, that is, an augmentation of 4 for preemptive EDF (or $4+2\rho$ for non-preemptive EDF) is sufficient to schedule any task set that an ideal scheduler can schedule. Second, we want to see how effective our scheduling strategy is in practice and, if the bounds are an accurate representation of how much augmentation is needed in practice. We do not compare with any baseline since no other strategies for real-time scheduling of general DAGs exist.

A. Task and Task Set Generation

We want to evaluate our scheduler using task sets that an optimal scheduler could schedule on 1-speed processors. However, as we cannot determine this ideal scheduler, we assume that an ideal scheduler can schedule any task set whose total utilization is no greater than m, and that each individual task is schedulable in isolation (that is, its critical path length is no greater than its deadline). Therefore, in our experiments, for each value of m (i.e. the number of processor cores), we generate task sets whose utilization is exactly m, fully loading a machine of 1-speed processors.

We use the Erdös-Rényi method $G(n_i, p)$ [31] to generate task sets for evaluation. The precise methodology is as follows.

Number of nodes. To generate a DAG τ_i, we pick the number of nodes n_i uniformly at random in range $[50, 350]$. We found that these values would allow us to generate varied task sets within a reasonable amount of time.

Adding edges. We add edges to the graph using the Erdös-Rényi method $G(n_i, p)$ [31]. We scan all the possible edges directing from lower node id to higher node id to avoid introducing a cycle into the graph. For each possible edge, we generate a random value in range $[0, 1]$ and add the edge only if the generated value is less than a predefined probability p. (We will vary p in our experiments to explore the effect of changing p.) Finally, we add an additional minimum number of edges so that each node (except the first and the last node) has at least one incoming and one outgoing edge in order to make the DAG weakly connected. Note that the critical path length of a DAG generated using the pure Erdös-Rényi method increases as p increases. However, since our method is slightly modified, the critical path is also large when p is small. Therefore, as p increases, the critical path first decreases up to a certain value of p and then increases again.

Execution time of nodes. We assign every node an execution time chosen randomly from a specified range. The range is based on the value and type (continuous or discrete) of the non-preemption overhead ρ (explained in the next subsection).

At this point, we have the DAG structure and the execution times for its nodes. For each DAG τ_i, we now assign a period T_i (which is also its deadline) that is no less than the critical path length P_i. We consider two types of task sets:

Task sets with harmonic periods. These deadlines are carefully picked so that they are multiples of each other, so as to ensure that we can run our experiments up to the hyper-period of the task sets. In particular, we pick deadlines that are powers of two. We find the smallest value a such that $P_i \leq 2^a$, and randomly set T_i to be one of 2^a, 2^{a+1}, or 2^{a+2}. These choices for period are due to the fact that we want some high utilization tasks and some low utilization tasks. The ratio P_i/T_i of the task is in the range $[1, 1/2]$, (1/2,1/4], or (1/4, 1/8], when its period T_i is 2^a, 2^{a+1}, or 2^{a+2}, respectively.

Task sets with arbitrary periods. We first generate a random number $\text{Gamma}(2, 1)$ using the *gamma distribution* [32]. Then we set period T_i to be $(P_i + \frac{C_i}{0.5m}) * (1 + 0.25 * \text{Gamma}(2, 1))$. We choose this formula for three reasons. First, we want to ensure that the assigned value is a *valid* period, i.e., $P_i \leq T_i$. Second, we want to ensure that each task set contains a reasonable number of tasks even when the number of cores is small. At the same time, with more cores, we do not want to limit average DAG utilization to a certain small value. Hence the minimum period is a function of m. Third, while we want the average period to be close to the minimum valid period (to have high utilization tasks), we also want some tasks with large periods. Table I shows the average number of DAGs per task set achieved by the random period generation process.

TABLE I
NUMBER OF TASKS PER TASK SET

m \ p	0.01	0.05	0.1	0.2	0.4	0.6	0.8
4	4	4	4	5	6	7	8
8	4	4	5	7	9	11	13
16	4	6	7	10	15	19	22
32	5	8	11	17	26	34	41

To create a task set we combine individual DAGs as follows. We add DAGs to the task set until the total utilization of the set exceeds m. We then remove the last generated DAG. Thus, at this point, the total utilization is smaller than m. To make the total utilization exactly m, we add small DAGs with long periods (and therefore small utilization). We stop adding small DAGs when the total utilization is larger than 99% of m.

B. Experimental Methodology

We run experiments by varying the following 4 parameters.

Harmonic vs. arbitrary periods. We want to evaluate whether arbitrary periods are better or worse than harmonic ones. For harmonic period task sets, we run simulation up to their hyper-period. For arbitrary period task sets, the hyper-period can be too long to simulate, and hence we run simulation up to 20 times the maximum period.

Number of cores (m). We want to evaluate if parallel scheduling is easier or harder as the number of cores increases. We run experiments on m: $4, 8, 16$, and 32.

Probability of an edge (p). As stated before, p affects the critical path length, the density, and the structure of the DAG. We test using 14 values of p: $0.01, 0.02, 0.03, 0.05, 0.07, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8$, and 0.9.

Non-preemption overhead (ρ). This is the ratio of the maximum node execution length to the minimum node execution length. For non-preemptive EDF scheduling, the resource augmentation bound increases as ρ increases. We want to evaluate whether the effect of increased ρ is really that severe in practice. For all of our experiments, we set the minimum node execution requirement to be 50, and vary the maximum execution requirement. To get $\rho = 1, 2, 5$, and 10, the maximum execution requirements are chosen to be $50, 100, 250$, and 500, respectively. In addition, when we evaluate the performance of non-preemptive EDF, we want to maximize the influence of ρ. Therefore, besides using uniformly generated node execution time between maximum and minimum (called *continuous* ρ), we also generate by choosing from discrete values 50, $2*50$, $\cdots$, $\rho*50$ (called *discrete* ρ).

In all experiments, we simulate 1000 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 the scheduler under non-preemptive global EDF by varying all four factors, leading to 896 combinations.

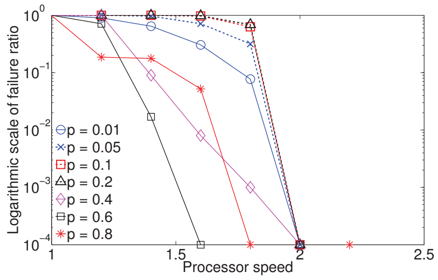


Fig. 4. Failure ratio in preemptive EDF on 32 cores under different edge probability

C. Results

Of the 896 combinations of parameters (each having 1000 task sets) we have tested, preemptive EDF has the maximum required speed of 3.2 to meet all deadlines (this data point is not shown in figures for better resolution), which is close to our analytical resource augmentation bound of 4. In contrast, among the combinations of parameters with $\rho = 1, 2, 5, 10$, the maximum required speed for non-preemptive EDF are 4.0, 5.8, 8.6, and 12.6, respectively, which look much smaller than the analytical bound of 6, 8, 14, and 24, respectively. These issues are discussed upon presenting the results. For brevity, we present only a subset of the experimental results.

Effect of harmonic vs. arbitrary periods. We find that it is slightly harder to schedule harmonic period tasks using preemptive EDF, and vice-versa for non-preemptive EDF. However, the difference is not significant, and the trends are very similar under both. Here we will only show the experiments for arbitrary periods.

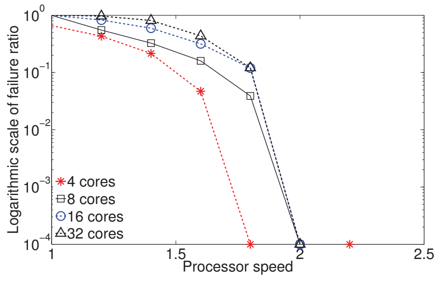


Fig. 5. Failure ratio in preemptive EDF on different numbers of cores

Effect of p in preemptive scheduling. For each value of p, Figure 4 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 1000 in our experiment) attempted to be scheduled. To preserve resolution of the figure, we show the results for only 7 (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. As we explained in Section VII-A, as p increases, the critical-path length first decreases (making the tasks more "parallel" or "DAG-like") and then increases again (making the tasks more sequential). Therefore, for both small and large p, the tasks are largely sequential. These results seem to conform to our intuition that, in general, parallel tasks are more difficult to schedule than sequential ones. The results for 4, 8, and 16 cores also follow this trend, and hence are omitted.

Effect of m in preemptive scheduling. Figure 5 shows the failure ratio in logarithmic scale for each value of m with fixed $p = 0.2$, and $\rho = 2$. We can see that the failure ratio increases as m increases, suggesting that it is harder to schedule on larger number of cores. The trend is similar for different values of p, and hence is not shown.

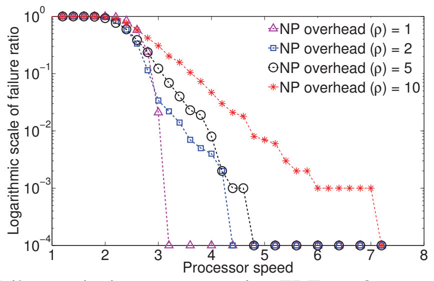


Fig. 6. Failure ratio in non-preemptive EDF on 8 cores under different non-preemption overhead

Effect of ρ in non-preemptive scheduling. The most important factor to evaluate is the effect of ρ. Figure 6 shows the failure ratio for discrete ρ for each value of ρ, with fixed $p = 0.2$, $m = 8$. With the increase in ρ, the failure ratio becomes much higher, which is expected. However, this trend is not quite strong for continuous ρ, and we omit plotting those results. Following may be the reason for this anomaly. The maximum value of ρ only affects the schedule if a node having the maximum execution interferes with a node having the minimum execution. Since ρ is continuous, a node's execution requirement is assigned from many different values. This causes only a small number of nodes to be at these extremes, thereby reducing the chances of such interference.

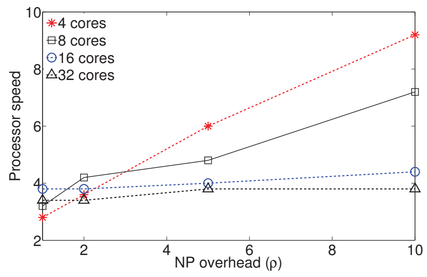


Fig. 7. Required speed in non-preemptive EDF on different numbers of cores with increasing non-preemption overhead

Effect of m in non-preemptive scheduling. Figure 6 shows the required speed for each combination of m and ρ, with

$p = 0.2$. Note that this figure is different from the previous ones in that it only shows the speed at which all task sets become schedulable. We can see that for each value of m, when ρ increases, the required speed increases, which is expected. However, when m increases, this trend becomes less obvious. One possible reason is that when there are more cores, the overhead from interference between executing low priority subtask and a newly released higher priority subtask will, on average, be smaller. This happens because the overhead is the minimum remaining work of all m running lower priority subtasks, instead of the average or worst case subtask execution time. When m is higher, the minimum will be much smaller than average, making the system much less influenced when ρ increases.

The simulation results show a maximum speed requirement of 3.2 for preemptive EDF suggesting that our analytical resource augmentation bound of 4 is reasonably tight. The corresponding bounds for non-preemptive EDF sound relatively looser in our simulation results. However, considering that the bound of 4 for preemptive EDF is tight, it is unlikely that a bound better than $4+\rho$ can be derived for non-preemptive EDF, since non-preemptivity can cause processor capacity reduction of up to ρ in the worst case. Due to decomposition, this value increases to 2ρ (see Equation 26). Therefore, for the sake of non-preemptivity in scheduling the decomposed tasks, the processor capacity reduction can be up to 2ρ in extreme cases, requiring a speed increase of 2ρ in addition to that for preemptive scheduling. Hence, there may be task sets that require a resource augmentation of $4 + 2\rho$, but our simulation does not encounter those tasks. In other words, our results may be an artifact of our experimental set up and random task generation strategies. Randomly generated tasks may be very unlikely to exhibit the pathological behavior required for the worst case to manifest itself. More work, both theoretical and experimental, is needed to decide whether the bounds are pessimistic, or if these simulations are optimistic.

VIII. 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 DAG model tasks. Through simulations, we have observed that the required augmentation is close to 4 in practice for preemptive tasks. However, for non-preemptive task sets, the worst augmentation requirement we found in practice was much smaller than the theoretical bounds. Our results suggest several possible directions of future work. One direction is to provide better bounds and/or provide lower bound arguments to argue that the bounds are in fact tight. Another possible direction is to study the effect of caches on scheduling overhead. While non-preemption mitigates this problem to some extent, more can be done to optimize cache-locality. Finally, we can generalize our results to models that take into account the effects of non-deterministic synchronization such as locks.

References

[1] http://en.wikipedia.org/wiki/Teraflops_Research_Chip.
[2] www.amd.com/us/products/server/processors.
[3] www.clearspeed.com/newsevents/news/ClearSpeed_Ace_011708.php.
[4] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Comp. Surv.*, vol. 43, pp. 35:1–44, 2011.
[5] 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 *ICCPS ’10*.
[6] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, “Multi-core real-time scheduling for generalized parallel task models,” in *RTSS ’11*.
[7] K. Lakshmanan, S. Kato, and R. R. Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” in *RTSS ’10*.
[8] “OpenMP,” http://openmp.org.
[9] “Intel CilkPlus,” http://software.intel.com/en-us/articles/intel-cilk-plus.
[10] C. D. Polychronopoulos and D. J. Kuck, “Guided self-scheduling: A practical scheduling scheme for parallel supercomputers,” *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1425–1439, 1987.
[11] M. Drozdowski, “Real-time scheduling of linear speedup parallel tasks,” *Inf. Process. Lett.*, vol. 57, no. 1, pp. 35–40, 1996.
[12] X. Deng, N. Gu, T. Brecht, and K. Lu, “Preemptive scheduling of parallel jobs on multiprocessors,” in *SODA ’96*.
[13] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, “Thread scheduling for multiprogrammed multiprocessors,” in *SPAA ’98*.
[14] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson, “Adaptive task scheduling with parallelism feedback,” in *PPoPP ’06*.
[15] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu, “Adaptive work-stealing with parallelism feedback,” *ACM Trans. Comput. Syst.*, vol. 26, September 2008.
[16] J. M. Calandrino and J. H. Anderson, “On the design and implementation of a cache-aware multicore real-time scheduler,” in *ECRTS ’09*.
[17] J. H. Anderson and J. M. Calandrino, “Parallel real-time task scheduling on multicore platforms,” in *RTSS ’06*.
[18] Q. Wang and K. H. Cheng, “A heuristic of scheduling parallel tasks and its analysis,” *SIAM J. Comput.*, vol. 21, no. 2, 1992.
[19] O.-H. Kwon and K.-Y. Chwa, “Scheduling parallel tasks with individual deadlines,” *Theor. Comput. Sci.*, vol. 215, no. 1-2, pp. 209–223, 1999.
[20] C.-C. Han and K.-J. Lin, “Scheduling parallelizable jobs on multiprocessors,” in *RTSS ’89*.
[21] W. Y. Lee and H. Lee, “Optimal scheduling for real-time parallel tasks,” *IEICE Trans. Inf. Syst.*, vol. E89-D, no. 6, pp. 1962–1966, 2006.
[22] S. Collette, L. Cucu, and J. Goossens, “Integrating job parallelism in real-time scheduling theory,” *Inf. Process. Lett.*, vol. 106, no. 5, pp. 180–187, 2008.
[23] G. Manimaran, C. S. R. Murthy, and K. Ramamritham, “A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems,” *Real-Time Syst.*, vol. 15, no. 1, pp. 39–60, 1998.
[24] S. Kato and Y. Ishikawa, “Gang EDF scheduling of parallel task systems,” in *RTSS ’09*.
[25] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, “Techniques optimizing the number of processors to schedule multi-threaded tasks,” in *ECRTS ’12*.
[26] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, “A generalized parallel task model for recurrent real-time processes,” in *RTSS ’12*.
[27] N. Fisher, T. P. Baker, and S. Baruah, “Algorithms for determining the demand-based load of a sporadic task system,” in *RTCSA ’06*.
[28] S. Baruah, “Techniques for multiprocessor global schedulability analysis,” in *RTSS ’07*.
[29] J. Goossens, S. Funk, and S. Baruah, “Priority-driven scheduling of periodic task systems on multiprocessors,” *Real-Time Syst.*, vol. 25, no. 2-3, pp. 187–205, 2003.
[30] S. Baruah, “The non-preemptive scheduling of periodic tasks upon multiprocessors,” *Real-Time Syst.*, vol. 32, pp. 9–20, 2006.
[31] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, “Random graph generation for scheduling simulations,” in *SIMUTools ’10*.
[32] “Gamma distribution,” http://en.wikipedia.org/wiki/Gamma_distribution.