Parallel Real-Time Scheduling of DAGs

Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, Christopher 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]–[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 4 under global earliest deadline first (EDF) scheduling. A resource augmentation bound $\nu$ of a scheduling policy $\kappa$ indicates that if there is any way to schedule a task set on $m$ identical unit-speed processor cores, then $\kappa$ is guaranteed to successfully schedule it on $m$ cores with each core being $\nu$ 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 alter-
native 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 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 $\rho$ 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 4.

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]–[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 4 and 5 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 (DAG), where the nodes stand for different execution requirements, and the edges represent dependencies between the nodes.

A node in $\tau_i$ is denoted by $W^j_i$, $1 \leq j \leq n_i$, with $n_i$ being the total number of nodes in $\tau_i$. The execution requirement of node $W^j_i$ is denoted by $E^j_i$. A directed edge from node $W^j_i$ to node $W^k_i$, denoted as $W^j_i \rightarrow W^k_i$, implies that the execution of $W^k_i$ cannot start until $W^j_i$ finishes. $W^1_i$, in this case, is called a parent of $W^k_i$, while $W^k_i$ is its child. A node may have 0 or more parents or children, and can start execution only after all of its parents have finished execution. Figure 1 shows a task $\tau_i$ with $n_i = 10$ nodes.

The execution requirement (i.e., work) $C_i$ of task $\tau_i$ is the sum of the execution requirements of all nodes in $\tau_i$; that is, $C_i = \sum_{j=1}^{n_i} E^j_i$. Thus, $C_i$ is the maximum execution time of $\tau_i$ if it was executing on a single processor of speed 1. For task $\tau_i$, 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 $\tau_i$. Thus, $P_i$ is the minimum execution time of $\tau_i$ 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 $\tau_i$ is denoted by $T_i$ and the deadline $D_i$ of each task $\tau_i$ is considered implicit, i.e., $D_i = T_i$. Since $P_i$ is the minimum execution time of task $\tau_i$ even on a machine with an infinite number of cores, the condition $T_i \geq P_i$ must hold for $\tau_i$ 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 $\nu$, where $\nu$ 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 $\tau_i$, and the total utilization $u_{\text{sum}}(\tau)$ for any task set $\tau$ of $n$ tasks are defined as

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

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

The density $\delta_i$ of any task $\tau_i$, and the total density $\delta_{\text{sum}}(\tau)$ and the maximum density $\delta_{\text{max}}(\tau)$ for any set $\tau$ of $n$ tasks are defined as follows.

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

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

$$\text{DBF}(\tau_i, t) = \max \left( 0, \left( \frac{t - D_i}{T_i} \right) + 1 \right) C_i$$
Based on the DBF, the load, denoted by $\lambda(\tau)$, of any task set $\tau$ consisting of $n$ tasks is defined as follows.

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

### 4.2 Decomposition Algorithm

In the decomposition, the intermediate subdeadline assigned to a node is called node deadline. Note that once task $\tau_i$ 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 $\tau_i$. For task $\tau_i$, the deadline and the offset assigned to node $W_i^j$ are denoted by $D_i^j$ and $\Phi_i^j$, respectively. Once appropriate values of $D_i^j$ and $\Phi_i^j$ are determined for each node $W_i^j$ (respecting the dependencies in the DAG), task $\tau_i$ 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 $\tau_i$ boils down to determining $D_i^j$ and $\Phi_i^j$ for each node $W_i^j$ as presented below. The presentation is accompanied by an example using the DAG $\tau_i$ 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 $\tau_i$ as a timing diagram $\tau_i^\infty$ (Figure 2(a)) that shows its execution time on an infinite number of unit-speed processor cores. Specifically, $\tau_i^\infty$ 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 $\tau_i$ 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 $\tau_i^\infty$ for DAG $\tau_i$. Next, based on $\tau_i^\infty$, the calculation of $D_i^j$ and $\Phi_i^j$ 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 ($\delta_{\text{sum}}$) and the maximum density ($\delta_{\text{max}}$) of the decomposed tasks. Namely, we need to keep both $\delta_{\text{sum}}$ and $\delta_{\text{max}}$ 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 $\tau_i$, a node can execute with different numbers of nodes in parallel at different times. Such a degree of parallelism can be estimated based on $\tau_i^\infty$. 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 $\tau_i^\infty$, 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 $\tau_i^\infty$, 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 $\tau_i^\infty$ into segments. For example, in Figure 2(b), $\tau_i$ is split into 7 segments (numbered from left to right).

Once $\tau_i^\infty$ 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 $\tau_i^\infty$ by $\tau_i^{\text{syn}}$. 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 $\tau_i^{\text{syn}}$. We first estimate time requirement for each segment. Let $\tau_i^{\text{syn}}$ be a sequence of $s_i$ segments numbered as 1, 2, · · · , $s_i$. For any segment $j$, we use $m_i^j$ to denote the number of threads in the segment, and $c_i^j$ to denote the execution requirement of each thread in the segment (see Figure 2(b)). Since $\tau_i^{\text{syn}}$ has the same critical path and
total execution requirements as those of \( \tau_i \),
\[
P_i = \sum_{j=1}^{s_i} e_i^j; \quad C_i = \sum_{j=1}^{s_i} m_i^j e_i^j
\]
For any segment \( j \) of \( \tau_i^{\text{syn}} \), 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 \( \delta_{\text{sum}} \) and \( \delta_{\text{max}} \) 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 requirements. Such a policy assigns a segment deadline of \( \frac{T_i}{m_i^j} m_i^j e_i^j \) 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 \( \delta_{\text{max}} \), and is not useful. Instead, we classify the segments of \( \tau_i^{\text{syn}} \) into two groups based on a threshold \( \theta_i \), 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 \( \theta_i \) 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 \( \tau_i^{\text{syn}} \).

This raises three different cases: when \( L_i = \emptyset \) (i.e., when \( \tau_i^{\text{syn}} \) consists of only heavy segments), when \( H_i = \emptyset \) (i.e., when \( \tau_i^{\text{syn}} \) consists of only light segments), and when \( H_i \neq \emptyset \, L_i \neq \emptyset \) (i.e., when \( \tau_i^{\text{syn}} \) consists of both light segments and heavy segments). We use three different approaches for these three scenarios.

Case 1: when \( L_i = \emptyset \). 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.
\[
d_i^j = \frac{T_i}{P_i} e_i^j
\]
Since the condition \( T_i \geq P_i \) must hold for every task \( \tau_i \) to be schedulable,
\[
d_i^j = \frac{T_i}{P_i} e_i^j \geq \frac{T_i}{T_i} e_i^j = e_i^j
\]
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
\[
\frac{C_i}{2T_i-P_i} \leq \frac{C_i}{2T_i-P_i} = \frac{C_i}{T_i}
\]

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 $d_i$ is given by
\[ d_i = \frac{T_i}{C_i} m_i e_i \] (7)
Since for every segment $j$, $m_j > \frac{C_j}{2T_j - P_j}$, we have
\[ d_i = \frac{T_i}{C_i} m_i e_i > \frac{T_i}{2T_i - P_i} C_i = \frac{2T_i}{2(2T_i - P_i)} e_i \geq \frac{e_i}{2} \] (8)
Hence, the maximum density of any thread is at most 2. The total density of segment $j$ is at most
\[ \frac{m_j e_j}{d_j} = \frac{m_j e_j}{T_j} \leq \frac{C_i}{T_i} \] (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 $\tau_i^{syn}$ is denoted by $C_{i^{\text{heavy}}}$, defined as
\[ C_i^{\text{heavy}} = \sum_{j \in H_i} m_i e_i \]
For each heavy segment $j$, the deadline $d_i$ is
\[ d_i = \frac{T_i - P_i}{C_i^{\text{heavy}}} m_i e_i \] (10)
Since for each heavy segment $j$, $m_j > \frac{C_j}{2T_j - P_j}$, we have
\[ d_i = \frac{(T_i - P_i)}{C_i^{\text{heavy}}} m_i e_i > \frac{(T_i - P_i)}{2T_i - P_i} C_i^{\text{heavy}} e_i \geq \frac{e_i}{2} \] (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
\[ \frac{m_i e_i}{d_i} = \frac{m_i e_i}{T_i - P_i} \leq \frac{C_i}{T_i - P_i/2} = \frac{2C_i}{T_i} \] (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 $P_i^{\text{light}}$, and is defined as follows.
\[ P_i^{\text{light}} = \sum_{j \in L_i} e_i \]
For each light segment $j$, the deadline $d_i$ is
\[ d_i = \frac{P_i}{P_i^{\text{light}}} e_i \] (13)
The density of a thread in any light segment is at most 2 since
\[ \frac{m_i e_i}{d_i} = \frac{m_i e_i}{P_i^{\text{light}}} \geq \frac{P_i}{P_i^{\text{light}}} e_i \geq \frac{e_i}{2} \] (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
\[ \frac{2C_i}{2T_i - P_i} \leq \frac{2C_i}{2T_i - P_i} = \frac{2C_i}{T_i} \] (15)

4.2.2 Calculating Deadline and Offset for Nodes
We have assigned segment deadlines to (the threads of) each segment of $\tau_i^{syn}$ in Step 1 (Equations (4), (7), (10), (13)). Since a node may be split into multiple (consecutive) segments in $\tau_i^{syn}$, 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 $W_i$ of $\tau_i$ belong to segments $k$ to $r$ $(1 \leq k \leq r \leq s_i)$ in $\tau_i^{syn}$. Therefore, the deadline $D_i$ of node $W_i$ is calculated as follows.
\[ D_i = d_k + d_k + 1 + \cdots + d_r \] (16)
Note the execution requirement $E_i$ of node $W_i$ is
\[ E_i = e_i + e_i + 1 + \cdots + e_i \] (17)
Node $W_i$ cannot start until all of its parents complete. Hence, its release offset $\Phi_i$ is determined as follows.
\[ \Phi_i = \begin{cases} 0; & \text{if } W_i \text{ has no parent} \\ \max\{\Phi_i + D_i | W_i \text{ is a parent of } W_i\}; & \text{otherwise} \end{cases} \]
Now that we have assigned an appropriate deadline $D_i$ and release offset $\Phi_i$ to each node $W_i$ of $\tau_i$, the DAG $\tau_i$ is now decomposed into nodes. Each node $W_i$ is now an individual (sequential) multiprocessor subtask with an execution requirement $E_i$, a constrained deadline $D_i$, and a release offset $\Phi_i$. Note that the period of $W_i^{\text{dec}}$ is still the same as that of the original DAG which is $T_i$. The release offset $\Phi_i$ ensures that node $W_i$ can start execution no earlier than $W_i$ 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 runs in linear time (in terms of the DAG size i.e., number of nodes and edges). Figure 7 in Appendix B shows the complete decomposition of $\tau_i$. Appendix C provides a sketch (Figure 8) on how it can be implemented on a real system.

4.3 Density Analysis after Decomposition
After decomposition, let $\tau_i^{\text{dec}}$ denote all subtasks (i.e., nodes) that $\tau_i$ generates. Note that the densities of all such subtasks comprise the density of $\tau_i^{\text{dec}}$. Now we analyze the density of $\tau_i^{\text{dec}}$ which will later be used to analyze schedulability.
Let node $W_i^j$ of $\tau_i$ belong to segments $k$ to $r$ ($1 \leq k \leq r \leq s_i$) in $\tau_i^{syn}$. Since $W_i^j$ has been assigned deadline $D_i^j$, by Equations (16) and (17), its density $\delta_i^j$ after decomposition is

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

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

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

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

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

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

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

Theorem 1: Let a DAG $\tau_i$, $1 \leq i \leq n$, 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 $\tau_{dec}$ using the decomposition technique (Algorithm 1 in Appendix). The density of $\tau_{dec}$ is at most \(\frac{2C_i}{T_i}\).

Proof: Since we decompose $\tau_i$ into nodes, the densities of all decomposed nodes $W_i^j$, $1 \leq j \leq n_i$, comprise the density of $\tau_{dec}$. In Step 1, every node $W_i^j$ of $\tau_i$ is split into threads in different segments of $\tau_i^{syn}$, 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 $\tau_i$ is scheduled in the form of $\tau_i^{syn}$, 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 $\tau_i^{syn}$ 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 $\delta_i^j$ (Equation (18)) of any node $W_i^j$ after removing its segment deadlines is no greater than the density $\delta_i^{j,syn}$ that it had before removing its segment deadlines.

Let node $W_i^j$ of the original DAG task $\tau_i$ be split into threads in segments $k$ to $r$ ($1 \leq k \leq r \leq s_i$) in $\tau_i^{syn}$. 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, $thread_i^j$, in segment $l, k \leq l \leq r$, in the interval of $D_i^j$ time units is expressed as

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

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

$$\lambda_i^{j,syn} = \frac{e_i^k + e_i^{k+1} + \ldots + e_i^r}{D_i^j} = \delta_i^j$$

Since $\delta_i^{j,syn} \geq \delta_i^j$, for any $W_i^j$, we have $\delta_i^{j,syn} \geq \delta_i^j$. \(\square\)

Let $\delta_{sum}$ be the total density of all subtasks $\tau_{dec}$. From Theorem 1, the density of each $\tau_{dec}$ is at most $\frac{2C_i}{T_i}$ where $T_i \geq P_i$.

$$\delta_{sum} \leq \sum_{i=1}^{n} \frac{2C_i}{T_i} = \sum_{i=1}^{n} \frac{C_i}{T_i} \quad (22)$$

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 $\tau_{dec}$ be the decomposed task set. If $\tau_{dec}$ is schedulable under some preemptive scheduling, then $\tau$ is preemptively schedulable.

Proof: See Appendix D. \(\square\)

To schedule the decomposed subtasks $\tau_{dec}$, 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 resource or penalty. Under preemptive global EDF, we now present a schedulability analysis for $\tau_{dec}$ in terms of a resource augmentation bound which, by Lemma 2, is also a sufficient analysis for the original DAG task set $\tau$. For a task set, a resource augmentation bound $\nu$ of a scheduling policy $\lambda$ 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 $\lambda$ is guaranteed to successfully schedule it on an $m$-core processor with each core being $\nu$ 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 $\pi$ with total density $\delta_{sum}(\pi)$ and maximum density $\delta_{max}(\pi)$ is schedulable
using preemptive global EDF policy on \( m \) unit-speed processor cores if

\[
\delta_{\text{sum}}(\pi) \leq m - (m - 1)\delta_{\text{max}}(\pi)
\]

Note that \( \tau_{\text{dec}} \) 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 \( \delta_{\text{sum}} \) and \( \delta_{\text{max}} \) of the subtasks with offsets are no greater than \( \delta_{\text{sum}} \) and \( \delta_{\text{max}} \), respectively, of the same set of tasks with no offsets. Hence, Theorem 3 holds for \( \tau_{\text{dec}} \). We now use the results of density analysis from Subsection 4.3, and prove that \( \tau_{\text{dec}} \) 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 \( \tau_{\text{dec}} \) be the decomposed task set. If every DAG \( \tau_i \) satisfies the condition \( T_i \geq P_i \), and the DAG set \( \tau \) 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 \( \tau_{\text{dec}} \) is guaranteed to be schedulable under preemptive global EDF on \( m \) processor cores, each of speed 4.

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

Considering that the condition \( \sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \) holds for \( \tau \), the total density of decomposed tasks \( \tau_{\text{dec}} \) from Equation (22) is derived as follows on \( \nu \)-speed cores.

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

On \( \nu \)-speed cores, the maximum density of \( \tau_{\text{dec}} \) is derived from Equation (20) as follows.

\[
\delta_{\text{max}}, \nu = \frac{\delta_{\text{max}}}{\nu} \leq \frac{2}{\nu} \tag{24}
\]

Using Conditions (24) and (23) in Theorem 3, \( \tau_{\text{dec}} \) is schedulable under preemptive EDF policy on \( m \) processor cores each of speed \( \nu \) 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, \( \tau_{\text{dec}} \) must be schedulable if

\[
\frac{4}{\nu} \leq 1 \Leftrightarrow \nu \geq 4.
\]

**Corollary 1:** For any set of DAGs \( \tau = \{ \tau_1, \tau_2, \ldots, \tau_n \} \), let \( \tau_{\text{dec}} \) be the decomposed task set. If there exists any algorithm that can schedule \( \tau \) on \( m \) unit-speed processor cores, then the decomposed task set \( \tau_{\text{dec}} \) 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 \( \tau \) 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 \( u_{\text{sum}}(\tau) \leq \frac{m}{4} \), then \( \tau \) is guaranteed to be 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 \( \tau \) 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 \( \tau_{\text{dec}} \) be the decomposed task set. If \( \tau_{\text{dec}} \) is schedulable under some fully non-preemptive scheduling, then \( \tau \) is schedulable under node-level non-preemption.

**Proof:** See Appendix E.

Under non-preemptive global EDF, we now present a schedulability analysis for \( \tau_{\text{dec}} \) in terms of a resource augmentation bound which, by Lemma 5, is also a sufficient analysis for the DAG task set \( \tau \). 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 \( \pi \), let \( C_{\text{max}}(\pi) \) and \( D_{\text{min}}(\pi) \) be the maximum execution requirement and the minimum deadline among all tasks in \( \pi \). In non-preemptive scheduling, \( C_{\text{max}}(\pi) \) 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 \( \pi \) is given by

\[
\rho(\pi) = \frac{C_{\text{max}}(\pi) - D_{\text{min}}(\pi)}{D_{\text{min}}(\pi)}.
\]
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 $\pi$ with total density $\delta_{\text{sum}}(\pi)$, maximum density $\delta_{\text{max}}(\pi)$, and a non-preemption overhead $\rho(\pi)$ is schedulable using non-preemptive global EDF on $m$ unit-speed cores if

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

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

$$E_{\text{max}} = \max \left\{ E_i^j | 1 \leq j \leq n_i, 1 \leq i \leq n \right\}$$

$$E_{\text{min}} = \min \left\{ E_i^j | 1 \leq j \leq n_i, 1 \leq i \leq n \right\}$$

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

$$\rho = \frac{E_{\text{max}}}{E_{\text{min}}}$$

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.

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

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

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

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

$$\frac{E_{\text{max}}}{D_{\text{min}}} \leq \frac{2\rho}{\nu}$$

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

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

From the above condition, task set $\tau_{\text{dec}}$ is schedulable if

$$\frac{4 + 2\rho}{\nu} \leq 1 \iff \nu \geq 4 + 2\rho.$$ 

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 where task executions are simulated in parallel as if they executed on $m$ cores.

We use the Erdős-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. We experiment by varying the following 4 parameters: type of task period (harmonic vs. arbitrary periods), number of cores ($m$), probability of an edge in DAG ($p$), and non-preemption overhead ($\rho$). The experimental methodology is detailed in Appendix F.

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 $\rho$ 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.

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

7.1 Results

Effect of harmonic vs. arbitrary periods. This result is discussed in Appendix F.

Effect of $p$ in preemptive scheduling. For each value of $p$, Figure 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 1000 in our experiment) attempted to be scheduled. To
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.

ACKNOWLEDGEMENTS

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

REFERENCES


8 Conclusions

As multi-core technology becomes mainstream in processor design, real-time scheduling of parallel tasks
Abusayeed Saifullah is a Ph.D. candidate in the Department of Computer Science and Engineering at Washington University in St Louis. Advised by Chenyang Lu, he is a member of the Cyber-Physical Systems Laboratory at Washington University. Abu’s 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 5th 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 a PhD student in the department of Computer Science and Engineering at Washington University in St. Louis. His research interests include parallel computing, real-time parallel systems, and cyber-physical systems.

Jing Li is a third year Ph.D. student in Computer Science and Engineering department at Washington University in St. Louis. She received the Bachelor of Science degree in Computer Science and Engineering from Harbin Institute of Technology in China in 2011. Her research interests include real-time parallel scheduling theory, real-time parallel system and cyber-physical systems.

Kunal Agrawal is an assistant professor in the Department of Computer Science and Engineering at Washington University in Saint Louis. She completed her Ph.D. in 2009 at MIT on the topic Scheduling and Synchronization for Multicore Concurrency Platforms. Her research interests include scheduling of parallel programs, parallel algorithms and data structures, synchronization mechanisms, transactional memory, and cache-efficient algorithms.

Chenyang Lu is a Professor of Computer Science and Engineering at Washington University in St. Louis. Professor Lu is Editor-in-Chief of ACM Transactions on Sensor Networks, Area Editor of IEEE Internet of Things Journal and Associate Editor of Real-Time Systems. He also serves as 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). Professor Lu is the author and co-author of over 100 research papers with over 10000 citations and an h-index of 47. He received the Ph.D. degree from University of Virginia in 2001, the M.S. degree from Chinese Academy of Sciences in 1997, and the B.S. degree from University of Science and Technology of China in 1995, all in computer science. His research interests include real-time systems, wireless sensor networks and cyber-physical systems.
APPENDIX

SUPPLEMENTAL MATERIALS

Parallel Real-Time Scheduling of DAGs

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

APPENDIX A

Pseudo Code of the Decomposition Algorithm

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: deadline $D_i^j$ 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 $\tau_i^{W_i}$;
$\theta_i \leftarrow C_i/(2T_i - P_i)$; /* heavy or light threshold */
total heavy $\leftarrow 0$; /* total heavy segments */
total light $\leftarrow 0$; /* total light segments */
$C_{\text{heavy}} \leftarrow 0$; /* total work of heavy segments */
$P_{\text{light}} \leftarrow 0$; /* light segments' critical path length */
for each $j$-th segment in $\tau_i^{W_i}$ do
  if $m_j^i > \theta_i$ then /* it is a heavy segment */
    total heavy $\leftarrow$ total heavy $+ 1$;
    $C_{\text{heavy}} \leftarrow C_{\text{heavy}} + m_j^i e_j^i$;
  else /* it is a light segment */
    total light $\leftarrow$ total light $+ 1$;
    $P_{\text{light}} \leftarrow P_{\text{light}} + e_j^i$;
  end
if total heavy $= 0$ then /* all segments are light */
  for each $j$-th segment in $\tau_i^{W_i}$ do $d_j^i = \frac{T_i}{m_j^i} e_j^i$;
else if total light $= 0$ then /* all segments are heavy */
  for each $j$-th segment in $\tau_i^{W_i}$ do $d_j^i = \frac{T_i}{m_j^i} e_j^i$;
else /* $\tau_i^{W_i}$ has both heavy and light segment */
  for each $j$-th segment in $\tau_i^{W_i}$ do
    if $m_j^i > \theta_i$ then /* heavy segment */
      $d_j^i = \frac{T_i - \theta_i}{m_j^i} e_j^i$;
    else /* light segment */
      $d_j^i = \frac{P_i}{m_j^i} e_j^i$;
  end
/* Remove segments. Assign node deadlines */
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 $\tau_i^{W_i}$ then
    $d_j^i = d_j^i + d_{i+1}^i + \ldots + d_r^i$; /* node deadline */
  $\Phi_i^j \leftarrow \max\{\Phi_i^j + D_i^j|W_i^j$ is a parent of $W_i^j\}$;
end

APPENDIX B

An Example Decomposition

Figure 7 shows the complete decomposition of $\tau_i$.

APPENDIX C

Implementation Considerations

This paper provides the algorithmic foundation for building a real-time parallel scheduler for parallel tasks. We now provide a sketch (Figure 8) on how it be implemented on a real system. In principle, one can use any parallel language such as OpenMP [9] and CilkPlus [10] that provides parallel programming support through library routines and directives. For example, OpenMP directives are compiler pragma statements that indicate where and how parallelization can occur within a program. One such directive converts a regular for loop to a parallel-for loop, by prefacing the loop with #pragma omp parallel for.

The programmer can specify their task set as a set of parallel programs written in some such parallel language. To make these tasks real-time tasks, the programmer must also specify task deadlines and periods. We assume that these things are specified using a separate task specification file that is also an input to the scheduler. In addition, the decomposition algorithm also needs the execution requirements of each node in task, which can also be either specified in the task specification file, or measured using a profiler.

Using the task specification file and the task set, the scheduler computes the intermediate deadlines and release times for each node. In addition, the compiler decomposes the task into individual nodes/subtasks. Once the intermediate deadlines are known, we can use a global priority queue to keep the subtasks sorted by priorities according to EDF. Now at runtime, the scheduler schedules these subtasks on $m$ processors using OS support for scheduling priorities, runtime preemption, and synchronization. When a subtask becomes available, if a worker is free, it is simply scheduled on this worker. If no worker is free, it is added to the priority queue. In the preemptive scheduler, we can use Linux support for preemption to preempt tasks with lower priorities when a high-priority task becomes available. For non-preemptive scheduling, we disable preemption and add yield points after each node.

When a subtask yields at its yield point, the scheduler can schedule the highest priority task that is available in the priority queue.

APPENDIX D

Proof of Lemma 2

Proof: In each $\tau_{i}^{\text{dec}}$, a node is released only after all of its parents finish execution. Hence, the precedence relations in original task $\tau_i$ are retained in $\tau_{i}^{\text{dec}}$ (that represent all subtasks of $\tau_i$). Besides, the time by which the last subtask of $\tau_{i}^{\text{dec}}$ has to finish is equal to the deadline of the original task $\tau_i$, and the sum

1. Most parallel languages already have this support, since this is when control returns to the scheduler. For others, these can be added by the compiler.
of the execution requirements of these subtasks is equal to the execution requirement of the original task \(\tau_i\). Hence, if \(\tau_{\text{dec}}\) is preemptively schedulable, a preemptive schedule must exist for \(\tau\) where each task in \(\tau\) meets its deadline.

**APPENDIX E**

**Proof of Lemma 5**

Proof: Since the decomposition converts each node of a DAG to an individual task, a fully non-preemptive scheduling of \(\tau_{\text{dec}}\) preserves the node-level non-preemptive behavior of task set \(\tau\). The rest of the proof follows from Lemma 2.

**A Tighter Bound for Non-Preemptive EDF**

A resource augmentation of \(4 + 2\rho\) for non-preemptive EDF is relatively looser than the corresponding bound of 4 for preemptive EDF. This is mainly because non-preemptivity can cause processor capacity reduction of up to \(\rho\). Due to decomposition, this value increases to \(2\rho\) (see Equation (30)). However, we can express the augmentation bound in a tighter form by using a tighter bound on non-preemption overhead. As shown in Equation (30) the non-preemption overhead of the decomposed task is indeed at most \(\frac{E_{\text{max}}}{D_{\text{min}}} \rho\). But we used a pessimistic upper bound of this value by replacing \(D_{\text{min}}\) with \(E_{\text{min}}\) as the value of \(D_{\text{min}}\) is unknown before decomposition. \(E_{\text{min}}\) is a lower bound of \(D_{\text{min}}\) and is known (from input) before decomposition. Therefore, if we define the bound upon decomposition, we can use \(\frac{E_{\text{max}}}{D_{\text{min}}} \rho\) as the maximum non-preemption overhead. Using this value of non-preemption overhead in Theorem 7, our bound will be \(4 + \frac{E_{\text{max}}}{D_{\text{min}}} \rho\) which is a lot smaller than \(4 + 2\rho\).

Notably, the work in [26] has identified a large class of applications such as high-performance web
and data-servers that consist of many real-time tasks, called liquid tasks, in which the smallest deadline of any job in the system is orders of magnitude greater than the largest execution requirement of any job. Upon decomposing the liquid tasks, the value of $D_{\text{min}}$ can be very close to $E_{\text{max}}$. Thus the value of $\frac{E_{\text{max}}}{D_{\text{min}}}$ approaches 1, and a resource augmentation of $4 + 2\rho$ is tight and quite useful in scheduling liquid parallel tasks. Our result provides the first such bound for non-preemptive real-time scheduling of parallel tasks, and provides the basis for future directions to derive tighter bounds for all classes of real-time tasks.

### APPENDIX F

#### Event-Driven Simulation

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 non-existing) 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 simulations, we first randomly create tasks and then calculate subtask deadlines using our proposed decomposition method. We then simulate the execution of these subtasks. The environment consists of $m$ cores and a global priority queue which keeps subtasks in the order of priorities based on EDF. An event occurs when a subtask is released or completed. When a subtask $t$ is released, a preemptive and non-preemptive schedulers behave differently. In a non-preemptive scheduler, two things can occur: (i) If a core is free, then $t$ is scheduled on that core; (ii) If all cores are busy, then the task is added to the priority queue. On a preemptive scheduler, if all cores are busy, but another subtask $s$ with a deadline later than $t$’s deadline is executing, then $s$ is preempted and placed in the priority queue, and $t$ is scheduled instead of $s$. When a subtask completes, the highest priority subtask from the queue is executed on the core that has just become free. This is a simple simulator which only simulates the task executions and ignores overheads due to migration, cache misses, preemption, and synchronization.

#### 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 (i.e. its critical path length is no greater than its deadline). Therefore, in our experiments, 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.

We use the Erdős-Rényi method $G(n, p)$ [25] as presented below to generate task sets for evaluation.

**Number of nodes.** To generate a DAG $\tau_i$, we pick the number of nodes $n_i$ uniformly at random in range $[50, 350]$. 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 Erdős-Rényi method $G(n, p)$ [25]. 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 Erdős-Rényi method increases as \( p \) increases. Since our method is slightly modified, the critical path is also large when \( p \) is small. Hence, 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 \( \rho \) (explained in the next subsection).

At this point, we have the DAG structure and the execution times for its nodes. For each DAG \( \tau_i \), we now assign a period \( T_i \) 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 \( 2^a+2 \). We choose such periods because 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 Gamma(2, 1) using the gamma distribution [27]. Then we set period \( T_i \) to be \((P_i + C_{\text{end}})/(1 + 0.25 * \text{Gamma}(2, 1))\). 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 \( m \) 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 1 shows the average number of DAGs per task set achieved by the random period generation process.

**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 \( p = 0.01, 0.02, 0.03, 0.05, 0.07, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 \).

**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 \( ρ = 1, 2, 5, \) 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, \( \cdots \), \( ρ * 50 \) (called discrete \( ρ \)).

**Detailed 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 \( p = 1, 2, 5, 10 \), 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.

**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 minor, and the trends are very similar under both. Here we will only show the experiments for arbitrary periods.

**Effect of p in preemptive scheduling.** For each value of \( p \), Figure 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 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, \( p = 2, m = 32 \). Note that the failure ratio increases as \( p \) increases from 0.01 to 0.1, and then falls again. As we explained in Section A, as \( p \) increases, the critical-path length first decreases (making the tasks more...
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. While the corresponding bounds for non-preemptive EDF sound relatively looser in our simulation results, we clarify the tightness and the usefulness of this bound in practice from two points of view. First, 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 $\rho$ in the worst case. For the sake of non-preemptivity in scheduling the decomposed tasks, the processor capacity reduction can be up to $2\rho$ in extreme cases, requiring a speed increase of $2\rho$ 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 that is unlikely to generate the worst-case task set. Second, as explained in Appendix E, there are practical task sets for which such a bound is tight. Specifically, the work in [26] has identified a large class of applications such as high-performance web and data-servers that consist of many real-time tasks, called liquid tasks, in which the smallest deadline of any job in the system is orders of magnitude greater than the largest execution requirement of any job. As the value of non-preemptive overhead approaches very small value (close to 1) for these tasks, a resource augmentation of $4 + 2\rho$ is tight and quite useful in scheduling liquid parallel tasks. Deriving tighter bounds for all classes of real-time tasks is an important future work.

**TABLE 1**

<table>
<thead>
<tr>
<th>$m$</th>
<th>$p$</th>
<th>0.01</th>
<th>0.02</th>
<th>0.03</th>
<th>0.05</th>
<th>0.07</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>20</td>
<td>22</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>17</td>
<td>22</td>
<td>26</td>
<td>30</td>
<td>34</td>
<td>37</td>
<td>41</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

“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 4 shows the failure ratio in logarithmic scale for each value of $m$, when $p = 0.2$ and $\rho = 2$. The failure ratio increases as $m$ increases, indicating that it is harder to schedule on larger numbers of cores. The trend is similar for different values of $p$, and hence is not shown.

**Effect of $\rho$ in non-preemptive scheduling.** The most important factor to evaluate is the effect of $\rho$. Figure 5 shows the failure ratio for discrete $\rho$ for each value of $\rho$, with fixed $p = 0.2$, $m = 8$. With the increase in $\rho$, the failure ratio becomes much higher, which is expected. However, this trend is not quite strong for continuous $\rho$, and we omit plotting those results. Following may be the reason for this anomaly. The maximum value of $\rho$ only affects the schedule if a node having the maximum execution interferes with a node having the minimum execution. Since $\rho$ 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.

**Effect of $m$ in non-preemptive scheduling.** Figure 6 shows the required speed for each combination of $m$ and $\rho$, with $p = 0.2$. 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 $\rho$ increases, the required speed increases, which is expected. This trend is less visible when $m$ increases. 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 $\rho$ increases.

The simulation results show a maximum speed