# Simultaneous Multithreading Applied to Real Time

**Sims Hill Osborne**
University of North Carolina, Chapel Hill, North Carolina, U.S.A.
http://www.cs.unc.edu/~shosborn/
shosborn@cs.unc.edu

**Joshua J. Bakita**
University of North Carolina, Chapel Hill, North Carolina, U.S.A.
https://jbakita.me/
jbakita@cs.unc.edu

**James H. Anderson**
University of North Carolina, Chapel Hill, North Carolina, U.S.A.
http://jamesanderson.web.unc.edu/
anderson@cs.unc.edu

## Abstract

*Existing models used in real-time scheduling are inadequate to take advantage of simultaneous multithreading (SMT), which has been shown to improve performance in many areas of computing, but has seen little application to real-time systems. The SMART task model, which allows for combining SMT and real time by accounting for the variable task execution costs caused by SMT, is introduced, along with methods and conditions for scheduling SMT tasks under global earliest-deadline-first scheduling. The benefits of using SMT are demonstrated through a large-scale schedulability study in which we show that task systems with utilizations up to 30% larger than what would be schedulable without SMT can be correctly scheduled.*

**2012 ACM Subject Classification** Computer systems organization → Real-time systems; Computer systems organization → Real-time system specification; Software and its engineering → Multithreading; Software and its engineering → Scheduling

**Keywords and phrases** real-time systems, simultaneous multithreading, soft real-time, scheduling algorithms

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

**Funding** Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors.

## 1 Introduction

Simultaneous multithreading (SMT) is a technology developed in the 1980s and 90s that allows multiple processes to issue instructions to different processor contexts, or threads, on a single physical computing core, creating the illusion of multiple cores for every one core that is actually present. It was designed to increase system utilization, particularly in the presence of memory latency [6, 25]. SMT became widely available in 2002, when it was made available on Intel processors [17]. Early experiments on the Pentium 4 showed that SMT could increase throughput by a factor of more than 1.5 in the best case [1, 2, 24]. The first attempt to utilize SMT in a real-time context was made in 2002 by Jain et al. [14], who showed that, by enabling SMT and making every thread available for real-time work, it is possible to schedule workloads with total utilizations up to 50 percent greater than what would be possible on the same platform without SMT. While Jain et al. gave ample experimental evidence that SMT can enable systems with higher utilization to be supported, neither they nor anyone else, to our knowledge, has provided a schedulability test that takes SMT into account.

© Sims Hill Osborne, Joshua J. Bakita, and James H. Anderson;
licensed under Creative Commons License CC-BY

Unfortunately, SMT's increase in throughput comes at the cost of longer and less predictable execution times, caused by contention for limited hardware resources. Apparently, the real-time systems community decided that this uncertainty makes SMT inappropriate for real-time work. We question the validity of this assessment for soft real-time (SRT) systems that may tolerate some degree of tardiness. Evidence suggest that others are even beginning to question this assessment in the context of safety-critical domains. In particular, the U.S. Federal Aviation Administration has received requests to certify safety-critical applications that use SMT, though they currently lack adequate techniques for doing so [20]. As evidence of the potential benefits of SMT, we present a sample of our results in Fig. 1; a platform with 16 cores is capable of scheduling task systems with utilizations of more than 20. We discuss this graph and others in Sec. 5.

**Considered problem.** We consider the problem of defining a scheduler for SRT systems that reaps the benefits of SMT without sacrificing execution-cost predictability. (We defer consideration of safety-critical applications to future work.) Existing models for analyzing real-time workloads do not allow us to specify how enabling SMT affects a task, so to quantify the per-task effects of SMT, we introduce a new task model, SMART (**S**imultaneous **M**ultithreading **A**pplied to **R**eal **T**ime). Using the SMART model, we attack our problem by dividing it into three sub-problems:

- **Sub-Problem 1:** Determine execution costs for tasks with SMT enabled. "Costs" is plural for each task; with SMT, one worst-case execution cost is not enough to define a task.
- **Sub-Problem 2:** Decide which tasks should use SMT. How using SMT will affect any given task is a function of what other tasks are using SMT.
- **Sub-Problem 3:** Schedule tasks so tasks using SMT do not interfere with tasks not using SMT.

The second sub-problem is particularly interesting. In general, allowing a task to execute with SMT will decrease the demand the task places on the hardware platform but increase the time needed for the task to execute. To address our problem, we need to balance the advantages of decreasing platform demand with the disadvantages of increasing task execution time. It is not enough to evaluate a task in isolation; every task that uses SMT may influence every other task that uses SMT.

**Motivation.** Processors are expensive. For any workload, real time or not, it is desirable to minimize the hardware cost needed to obtain a given level of performance. SMT is a means to get the most work out of a given processor. Presently, SMT is widely implemented, meaning there is a high chance that users are paying for SMT even if they are not using it. A better understanding of SMT would allow for better use of existing hardware resources.

**Related works.** Snavely and Tullsen demonstrated that SMT performance is dependent on which tasks share a core and introduced the term "symbiosis" to describe this concept [23]. We have already mentioned Jain et al.'s work on SMT and real-time scheduling from 2002 [14]. Since then, Cazorla et al. [3], Gomes et al. [11, 12], and Zimmer et al. [27] have proposed ways to eliminate the timing

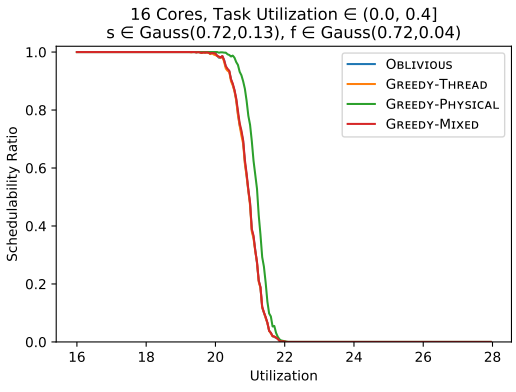


**Figure 1** Schedulability on 16 cores with SMT. Note that the horizontal axis begins at utilization 16 and that schedulability does not begin to drop until utilization 20. Effectively, more than 20 cores worth of capacity can be had on a 16-core platform. We discuss this graph and others like it in detail later.

uncertainties associated with SMT by means of detailed control over program execution and, in the case of Zimmer et al., a purpose-built processor, FlexPRET. Cazorla et al. [3] and Lo et al. [16] gave methods to limit real-time work to a small number of threads, leaving the remaining threads to execute only when doing so will not interfere with real-time work. Mische et al. [19] proposed to use SMT to hide context-switch times by using threads to switch task state in and out in the background. Early work on the performance of tasks executed by hardware threads was done by Bulpin [1] , Bulpin and Pratt [2], Huang et al. [13], and Tuck and Tulsen [24]. A preliminary version of our paper was presented as a work in progress at RTSS 2018 [21].

**Contribution and organization.** We introduce the SMART task model, a method for scheduling SMART tasks, and a related schedulability test. While other works focus on modifying hardware to make SMT more predictable, our work allows for SMT-supported real-time work to run on existing hardware and operating systems. We give results of benchmark tests measuring the performance impacts of SMT with regard to execution times. We show, using a schedulability study based on our benchmark results, that it is possible to correctly schedule task systems with utilizations more than 25% greater than what would be schedulable on the same platform without SMT enabled.[1]

The rest of this paper is organized as follows. In Sec. 2, we give a brief overview of SMT technology, discuss the shortcomings of the sporadic task model with regard to SMT, and introduce the SMART model. In Sec. 3, we address Sub-Problems 2 and 3, showing how SMT can be used to schedule otherwise unschedulable task systems. In Sec. 4, we address Sub-Problem 1, how to determine appropriate costs. (Note that we address our sub-problems in reverse order.) In Sec. 5, we present our schedulability experiments and results. In Sec. 6, we conclude and suggest future directions for our research.

## 2 What is a SMART Task?

In this section, we first give a brief overview of SMT technology. We then give a review of the sporadic task model and consider aspects of SMT that cannot be expressed with the sporadic model. We introduce SMART as an alternative model, under which task execution characteristics in the presence of SMT can be better described.

### 2.1 SMT Basics

Cores with SMT enabled accept multiple instructions per cycle from multiple tasks, reducing wasted instructions per cycle. A detailed explanation is available in Eggers et al.[6], but we illustrate the essentials in Example 1.

▶ **Example 1.** Fig. 2 shows the effect of enabling SMT. At the top of the figure, tasks $\tau_1$ and $\tau_2$ execute sequentially without SMT on a processor that can accept two instructions per cycle. When less than two instructions are available for execution, as at times 2, 3, and elsewhere, processor cycles are lost. $\tau_1$ finishes at time 6 and $\tau_2$ at time 12. At the bottom of the figure, the same tasks execute in parallel with SMT enabled, reducing the number of lost processor cycles. Both tasks finish at time 9. In this case, SMT has the effect of delaying the completion of $\tau_1$, but speeding up the completion of $\tau_2$, since it does not have to wait for $\tau_1$ to complete before beginning its own execution. ◀

Fig. 3 gives a more task-centric view of the two tasks seen in Fig. 2. For the remainder of this paper, we will conceptualize tasks as seen in Fig. 3; we are interested in how long a task takes

[1] While Jain et al. [14] were able to schedule systems with up to 50% greater utilization, they define a "correctly scheduled system" as one having a low number of observed deadline misses, whereas we define correctness as all tasks having analytically guaranteed bounded tardiness.

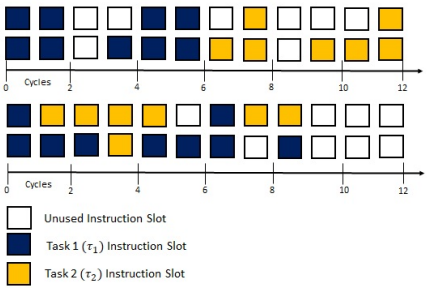


**Figure 2** Top: task execution without SMT. Bottom: task execution with SMT.

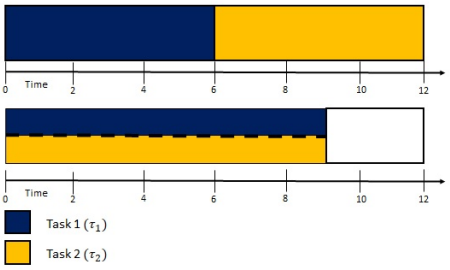


**Figure 3** Two tasks executing without SMT (top) and with SMT (bottom). With SMT, each task requires more time to complete individually, but time for both tasks to complete is reduced.

to execute and how much of a core it uses, not an exact cycle-by-cycle accounting. As shown in Fig. 3, SMT can cause individual tasks to take longer to complete, but total throughput is potentially increased, since the number of wasted instruction slots can be decreased. The challenge for real-time scheduling is to take advantage of this increased throughput without allowing increased execution costs to render the system unschedulable. The effect of SMT on task execution times is not constant across tasks; how much a task's execution time is increased by SMT depends on both the task itself and on other tasks that might be executing on the same core.

To discuss SMT more easily, we make a distinction between a core and a processor. A *core* is the hardware unit responsible for executing instructions. A *processor* is a single instruction context on a core. Every computer core, by definition, supports at least one processor, but computer cores capable of SMT may support multiple processors. We define a *physical processor* as a processor that occupies an entire core, while a *threaded processor* corresponds to a single hardware thread. Different threaded processors on the same core are *sibling processors*. Tasks scheduled on sibling processors are said to be *co-scheduled*.

We focus on a platform $\pi$ that has $m$ cores where every core supports one physical processor or two threaded processors at a time. For example, Fig. 4 shows a system of six cores. Cores 1-3 have SMT enabled and support two threaded processors each. Cores 5 and 6 have SMT disabled and support one physical processor each. Core 4 initially has SMT disabled and supports one physical processor, but at time 1, SMT is enabled on core 4, causing the single physical processor to be replaced by two threaded processors. We only consider two threads per core because this is what Intel currently supports.

## 2.2 Task Model

In the traditional implicit-deadline sporadic task model, a task $\tau_i = (T_i, C_i)$ is defined by its *period,* $T_i$, and its worst-case execution *cost,* $C_i$. The *utilization* of $\tau_i$ is given by $u_i = \frac{C_i}{T_i}$. Every task releases an unlimited number of *jobs*, with the $k^{th}$ job released by $\tau_i$ denoted by $\tau_{i,k}$. Jobs of $\tau_i$ are released at least $T_i$ units of time apart and have an implicit deadline of $T_i$. If the jobs of each task $\tau_i$ are released exactly $T_i$ units apart, then the task system is *periodic.* We consider only SRT systems here, in which some deadline misses are acceptable. In our model, a job's *tardiness* is the difference between its completion time and deadline, if the job completes after its deadline, and zero otherwise. A task's tardiness is the maximum tardiness of any of its jobs. We define an SRT system as being *correctly scheduled* if all tasks have guaranteed bounded tardiness. A task system is *SRT-schedulable* under

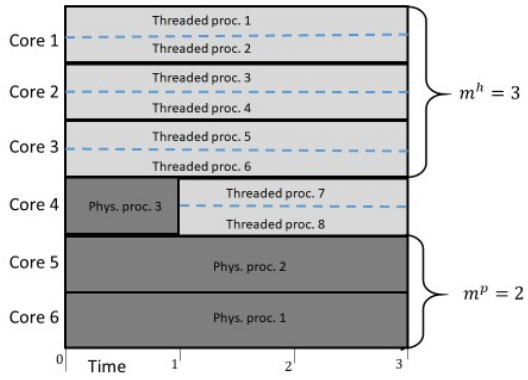


**Figure 4** Example of cores supporting threaded processors only, physical processors only, or both.

scheduling algorithm $A$ if it can be correctly scheduled by the specific algorithm $A$, and *SRT-feasible* if it is SRT-schedulable by some algorithm $A$. An algorithm is *SRT-optimal* if it can schedule all SRT-feasible task systems.

Given a platform $\pi$ consisting of $m$ identical cores and no SMT, a task system $\tau$ is SRT-feasible if and only if

$$\forall \tau_i \in \tau \; u_i \leq 1 \quad \text{and} \quad \sum_{i=1}^{n} u_i \leq m \tag{1}$$

both hold [4].

**The SMART model.** The shortcoming of the sporadic model in regard to SMT is that it only allows one worst-case execution cost per task, and therefore cannot adequately characterize a task system's behavior in the presence of SMT. For example, it is not possible to specify the task behavior seen in Fig. 3 using the sporadic model. To address this shortcoming, we introduce the SMART model. In this model, every task is modeled as $\tau_i = (T_i, (C_{i:j}))$. All parameters must be rational. As in the sporadic model, $T_i$ is the period of $\tau_i$. The parameter $(C_{i:j})$ is a list of costs that indicate the worst-case execution cost of a job of $\tau_i$ given that the entire job is co-scheduled with one or more jobs of $\tau_j$. We define $C_{i:i}$ to be $\tau_i$'s cost when it executes on a normal physical processor. For all $i \neq j$, $C_{i:j} \geq C_{i:i}$.[2] We define $u_{i:j} = \frac{C_{i:j}}{T_i}$.

Notice that we are implicitly making two simplifying assumptions here: **(i)** $\tau_i$'s worst-cast execution time can be determined by examining how it is interfered with when co-scheduled with each other task *individually*; and **(ii)** when $\tau_i$ is co-scheduled with $\tau_j$, every portion of $\tau_i$ receives the same amount of interference from every portion of $\tau_j$. In practice, (i) and (ii) will not necessarily hold, but we maintain that our model is sufficient for non-safety critical SRT workloads. We discuss this issue further in Sec. 4 when we delve into the issue of how to actually determine execution costs.

▶ **Definition 1.** The execution *rate* of $\tau_i$ given that it is co-scheduled with $\tau_j$ is given by $r_{i:j} = \frac{C_i}{C_{i:j}}$, where both $C_i$ and $C_{i:j}$ are maximum observed execution times.

We assume no relationship between $r_{i:j}$ and $r_{j:i}$; in fact, as we show in our benchmark experiments, the two can differ significantly. Our definition assumes two hardware threads per core, but could be expanded to allow for additional threads. In general, $r_{i:j} > 0.5$ indicates that $\tau_i$ could benefit from being co-scheduled with $\tau_j$ assuming that $C_{i:j} \leq T_i$ and $C_{j:i} \leq T_j$ hold.

[2] In the rare event that $C_{i:j} < C_{i:i}$ holds, the two are likely close in value, and we can simply redefine $C_{i:j}$ to equal $C_{i:i}$.

▶ **Example 2.** Suppose Fig. 3 depicts one job each of SMART tasks $\tau_1$ and $\tau_2$. $C_1 = 6$ and $C_2 = 6$, but $C_{1:2} = 9$ and $C_{2:1} = 9$, giving $r_{1:2} = r_{2:1} = \frac{2}{3}$. Task $\tau_2$ benefits from SMT; the job completes at time 9 with SMT as opposed to time 12 without. If both jobs are released at time 0 and have a deadline at time 10, then SMT allows for both jobs to complete on time, whereas without SMT, $\tau_2$'s job misses its deadline. ◀

**Scheduling SMART tasks.** We need to schedule $n$ tasks that have $n$ costs each; this problem poses obvious difficulties. In the next section, we show how we can schedule SMART tasks similarly to traditional sporadic tasks without sacrificing the advantages of SMT.

## 3 Scheduling Physical and Threaded Tasks

Not all tasks will benefit from SMT. We label tasks that should and should not use SMT as *threaded tasks* and *physical tasks*, respectively. Physical tasks can execute only on physical processors and threaded tasks only on threaded processors. To keep the task types separate, we divide them into two task subsystems, $\tau^p$ and $\tau^h$, that we schedule separately.

▶ **Definition 2.** Subsystem $\tau^p$ is the set of all physical tasks in $\tau$. $n^p = |\tau^p|$. Subsystem $\tau^h$ is the set of all threaded[3] tasks in $\tau$. $n^h = |\tau^h|$.

After partitioning $\tau$ into $\tau^p$ and $\tau^h$, physical tasks have cost $C_i^p$ and utilization $u_i^p = \frac{C_i^p}{T_i}$; threaded tasks have cost $C_i^h$ and utilization $u_i^h = \frac{C_i^h}{T_i}$. Costs for physical tasks are no different than costs in a sporadic task system, but costs for threaded tasks are a function of how the task system is divided. These cost parameters are a simplification of the full SMART parameters; we will show how to obtain them in Sec. 3.2.

▶ **Definition 3.** The total utilizations of $\tau^p$ and $\tau^h$ are given by $U^p = \sum_{i=1}^{n^p} u_i^p$ and $U^h = \sum_{i=1}^{n^h} u_i^h$ respectively. To measure the total demand placed on the platform, we define *effective utilization*, $U^E = U^p + \frac{U^h}{2}$. $U^h$ is halved in the sum to reflect the fact that each threaded task requires only half a core at a time to execute. ◀

### 3.1 Sub-Problem 3: Scheduling Task Subsystems

In this section, we give conditions for scheduling $\tau^p$ and $\tau^h$ on $\pi$. We assume the decision of which tasks should be physical and which should be threaded has already been made. Our current problem is how to schedule those tasks, but the best way to do so is not immediately obvious.

▶ **Example 3.** Suppose we attempt to schedule a task system $\tau$ using global earliest-deadline-first scheduling (GEDF). Let $\tau_1$ be a threaded task and $\tau_2$ a physical task such that at time $t$, a job of $\tau_1$ with a deadline of $t + 1$ is contending for a single core with a job of $\tau_2$ with a deadline of $t + 2$. Following GEDF, the job of $\tau_1$ should be given priority over that of $\tau_2$. However, if no other threaded task has an active job at time $t$, then doing so will cause the second threaded processor of a core in $\pi$ to be unused, negating any advantage gained by having $\tau_1$ be threaded. If we avoid this problem by giving priority to $\tau_2$, then we are not wasting processor capacity, but we are violating EDF priority rules. If we co-schedule the tasks on threaded processors despite $\tau_2$ being a physical task, then unanticipated task interference may ensue, potentially invalidating assigned per-task worst-case execution costs. None of these approaches is particularly satisfactory. ◀

[3] We use *h* rather than *t* for threaded so as to avoid confusion with *t* for time.

To address the problems raised in Example 3, we divide $\pi$ into *sub-platforms* $\pi^p$ and $\pi^h$.

▶ **Definition 4.** $\pi^p$ is the sub-platform of $\pi$ that schedules only tasks in $\tau^p$. It includes $m^p = \lfloor U^p \rfloor$ fully available cores and one partially available core. Given a length-$W$ interval, denoted a *window*, the partially available core belongs to $\pi^p$ for $a^p W$ time units per window, where $a^p = U^p - \lfloor U^p \rfloor$. $\pi^p$ can exist only if $U^p \leq m$.

▶ **Definition 5.** $\pi^h$ is the sub-platform of $\pi$ that schedules only tasks in $\tau^h$. It has $m^h = m - \lceil U^p \rceil$ fully available cores and one core available for $a^h W$ time units per window, where $a^h = \lceil U^p \rceil - U^p$. Consequently, $m^h + a^h = m - U^p$. If $a^p > 0$, then $a^h = 1 - a^p$.

We refer to the core shared by both platforms as the *shared core.* If there is no shared core, then $a^p = a^h = 0$. Note that $m^p + a^p + m^h + a^h = m$ must hold.

▶ **Example 4.** In Fig. 4, $\pi^p$ is shown in dark gray and $\pi^h$ in light gray. The sub-platforms are defined by $m^p = 2$, $m^h = 3$, $W = 3$, $a^p = \frac{1}{3}$, and $a^h = \frac{2}{3}$. ◀.

We now give schedulability results for $\tau^p$ and $\tau^h$ individually and then combine those conditions to get an overall schedulability result. For the most part, we will focus on the case where a shared core exists. Our results are based on Devi and Anderson's EDF-high-low (EDF-hl) algorithm [5]. EDF-hl gives schedulability conditions and tardiness bounds for "low" SRT tasks that are scheduled according to GEDF but are subject to interruption from periodic "high" hard real-time tasks, with at most one such task fixed on each processor. For our purposes, we can view $\tau^p$ as a set of low tasks scheduled on $m^p + \lceil a^p \rceil$ processors and subject to preemption by a single high task with period $W$ and cost $a^h W$. This reflects the fact that, from the perspective of $\tau^p$, work on the shared core is periodically preempted. Likewise, we can view $\tau^h$ as a set of low tasks scheduled on $2(m^h + 1)$ processors that are periodically preempted by two high tasks, both with period $W$ and cost $a^p W$. The following definitions apply to the EDF-hl results.

▶ **Definition 6.** Devi and Anderson define $\tau_H$ as the set of all high tasks, $\tau_L$ as the set of all low tasks, $u_{max}(\tau_L)$ as the highest-utilization task within $\tau_L$, $U_{sum}$ as the total utilization of both $\tau_H$ and $\tau_L$, $U_H$ is the sum of all the utilizations of all tasks in $\tau_H$, and $U_L$ is the sum of the $min(\lceil U_{sum} \rceil - 2, n)$ largest utilization of tasks in $\tau_L$.

We state an abridged version of Theorem 1 in [5] here. The full version defines the tardiness bound $B$ as a function of the task system and platform. We omit that portion of the theorem due to space constraints.

▶ **Theorem 1.** *EDF-hl ensures a tardiness bound of at most $B$ to every task $\tau_i$ of $\tau_L$ if $|\tau_H| \leq m$ and $U_{sum} \leq m$ and at least one of (2) or (3) holds.*

$$m - |\tau_H| - U_L > 0 \tag{2}$$

$$m - \max(|\tau_H| - 1, 0) u_{max}(\tau_L) - U_L - U_H > 0 \tag{3}$$

Returning to our problem, our schedulabilty conditions rely on the following assumptions. These assumptions allow us to schedule $\tau^p$ and $\tau^h$ as if they both consisted of standard sporadic tasks. We will show how to support Assumptions 1 and 2 in Sec. 3.2.

▶ **Assumption 1.** Tasks have been divided into threaded and physical tasks such that $\forall \tau_i^p \in \tau^p, u_i^p \leq 1$ and $\forall \tau_i^h \in \tau^h, u_i^h \leq 1$ both hold. Without loss of generality, we assume that the tasks in each of the sets $\tau^p$ and $\tau^h$ are indexed in decreasing-utilization order, e.g., $u_1^p$ (resp., $u_1^h$) is the largest utilization in $\tau^p$ (resp., $\tau^h$).

▶ **Assumption 2.** Worst-case costs for physical and threaded tasks have been determined.

▶ **Assumption 3.** Physical tasks are not permitted to execute on threaded processors.[4]

▶ **Lemma 2.** *$\tau^p$ is schedulable on $\pi^p$ under GEDF such that all tasks have guaranteed bounded tardiness if (4) holds.*

$$U^p \leq m^p + a^p. \tag{4}$$

**Proof.** If $a^p = 0$, then the result restates the SRT feasibility condition for $m$ identical, fully available processors given by (1). GEDF has been shown to be SRT-optimal [4], so the result follows.

If $m^p = 0$, then it can easily be shown that the system is schedulable only if $U^p \leq a^p$.

In the rest of the proof, we consider the remaining possibility, i.e., that $a^p > 0$ and $m^p > 0$ both hold. For this case, we show that Theorem 1 can be applied.

From the perspective of $\tau^p$, there exists a set of low tasks $\tau^p$ with total utilization $U^p$, one high task with utilization $a^h$, and $m^p + 1$ processors. Thus, we want to apply Theorem 1 with the substitutions $m \leftarrow m^p + 1$, $\tau_L \leftarrow \tau^p$, $U_{sum} \leftarrow U^p + a^h$, and $|\tau_H| = 1$. With these substitutions, (4), and Def. 5, it is straightforward to see that both $|\tau_H| \leq m$ and $U_{sum} \leq m$ hold, as required by Theorem 1. We now show that (2) holds, from which bounded tardiness for the tasks in $\tau_L$, i.e., those in $\tau^p$, follows. To see this, note that from Def. 5 and $U_{sum} = U^p + a^h$, we have

$$\begin{aligned}
&U_L = \sum_{i=1}^{min(\lceil U^p + a^h \rceil - 2, n^p)} u_i^p \\
&= \{\text{by Defs. 4 and 5, } U^p + a^h = m^p + 1\} \\
&U_L = \sum_{i=1}^{min(m^p - 1, n^p)} u_i^p \\
&\Rightarrow \{\text{because } u_i^p \leq 1 \text{ holds, by Assumption 1}\} \\
&U_L < m^p.
\end{aligned}$$

From this last inequality, we have $m - |\tau_H| - U_L = m^p + 1 - 1 - U_L > 0$, as required by (2). ◀

The schedulability condition for $\tau^h$ is slightly more complicated, due to it potentially having two partially available processors.

▶ **Lemma 3.** *$\tau^h$ is schedulable on $\pi^h$ under GEDF such that all tasks have guaranteed bounded tardiness if (5) and at least one of (6) or (7) hold, where $u_{max}(\tau^h)$ denotes the maximum task utilization in $\tau^h$.*

$$U^h \leq 2(m^h + a^h) \tag{5}$$

$$2m^h > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \tag{6}$$

$$2(m^h + a^h) - u_{max}(\tau^h) > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \tag{7}$$

[4] When the shared core belongs to $\pi^p$, it supports a physical processor, not a threaded processor.

**Proof.** As in the prior proof, the proof is straightforward if either $a^h = 0$ holds or $m^h = 0$ holds, so we focus on the remaining possibility, i.e, $m^h > 0$ and $a^h > 0$ both hold; note that the latter implies that $a^p > 0$ holds as well. As before, we will use Theorem 1. In this case, we are attempting to schedule a set of low tasks $\tau^h$ with total utilization $U^h$ on $2(m^h + 1)$ processors given two high tasks, each with utilization $a^p$. Thus, we want to apply Theorem 1 with the substitutions $m \leftarrow 2(m^h + 1)$, $\tau_L \leftarrow \tau^h$, $U_{sum} \leftarrow U^h + 2a^p$, and $|\tau_H| = 2$. With these substitutions, (5), and Def. 5, it is straightforward to see that both $|\tau_H| \leq m$ and $U_{sum} \leq m$ hold, as required by Theorem 1. In the rest of the proof, we show that, with these substitutions, (6) implies (2) and (7) implies (3), from which bounded tardiness for the tasks in $\tau_L$, i.e., those in $\tau^h$, follows.

To see that (6) implies (2), first note that, because $m^h$ is an integer, we have $\lceil U_{sum} \rceil - 2 \leq m - 2 = \lceil 2(m^h + 1) \rceil - 2 = \lceil 2m^h \rceil = 2m^h$. Therefore,

$$
\begin{aligned}
& 2m^h > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \\
\Rightarrow\ & \{\text{because } \lceil U_{sum} \rceil - 2 \leq 2m^h\} \\
& 2m^h > \sum_{i=1}^{\min(\lceil U_{sum} \rceil - 2, n^h)} u_i^h \\
=\ & \{\text{by the definition of } U_L \text{ in Def. 6}\} \\
& 2m^h > U_L,
\end{aligned}
$$

i.e., $2m^h - U_L > 0$ holds, which is equivalent to (2), since $m = 2(m^h + 1)$ and $|\tau_H| = 2$.

To see that (7) implies (3), observe that

$$
\begin{aligned}
& 2(m^h + a^h) - u_{max}(\tau^h) > \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \\
\Rightarrow\ & \{\text{reasoning as above}\} \\
& 2(m^h + a^h) - u_{max}(\tau^h) > U_L \\
=\ & \{\text{because } a^h = 1 - a^p\} \\
& 2(m^h + 1 - a^p) - u_{max}(\tau^h) > U_L, \\
=\ & \{\text{in our context, } u_{max}(\tau^h) = u_{max}(\tau_L),\ |\tau_H| - 1 = 2,\ U_H = 2a^p, \text{ and } m = 2(m^h + 1)\} \\
& m - \max(|\tau_H| - 1, 0) u_{max}(\tau_L) - U_H > U_L,
\end{aligned}
$$

which is equivalent to (3).

◀

A special case applies when there is no shared core.

▶ **Lemma 4.** *If $a^h = 0$, then $\tau^h$ is schedulable on $\pi^h$ under GEDF if and only if $U^h \leq 2m^h$ holds.*

**Proof.** With no shared core, the platform consists of $2m^h$ identical cores. The standard SRT feasibility test given by (1) applies. ◀

Our next step is to give a schedulability condition for $\tau^p$ and $\tau^h$ combined on $\pi$. This condition is a straightforward extension of the preceding lemmas, but it has the benefit of letting us focus on $\tau$ rather than on how $\pi$ is partitioned.

▶ **Theorem 5.** *Platform $\pi$ can be partitioned such that $\tau^p$ is schedulable on $\pi^p$ and $\tau^h$ is schedulable on $\pi^h$, both under GEDF, if (8) and at least one of (9) or (10) hold.*

$$U^E \leq m \tag{8}$$

$$2(m - \lceil U^p \rceil) > \sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^p)} u_i^h \tag{9}$$

$$2(m - U^p) - u_1^h > \sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^p)} u_i^h \tag{10}$$

**Proof.** In order to define $m^p$ and $a^p$ so that $m^p + a^p = U^p$ holds, as in Def. 4, we merely require $U^p \leq m$ to hold, and by Def. 3, this is implied by (8). Note that $m^p + a^p = U^p$ satisfies Condition (4) in Lemma 2.

Schedulability of $\tau^p$ on $\pi^p$ is implied by (8):

$$\begin{aligned} U^E &\leq m \\ &= \{\text{by Def. 3, } U^E = U^p + \frac{U^h}{2}\} \\ U^p &\leq m \\ &= \{\text{by Def. 4, } m^p + a^p = U^p\} \\ U^p &= m^p + a^p, \end{aligned}$$

which is the condition for $\tau^p$ per Lemma 2.

We next show that (8) implies Condition (5) of Lemma 3. To see this, observe that, by Def. 3, $U^E \leq m \Rightarrow \frac{U^h}{2} \leq m - U^p$. Also, by Def. 5, $m^h + a^h = m - U^p$. Putting these facts together, we have $U^h \leq 2(m^h + a^h)$, which is (5).

We conclude the proof by showing that (9) is equivalent to Condition (6) of Lemma 3, and that (9) is equivalent to Condition (7) of Lemma 3. To see the former, note the following.

$$\begin{aligned} 2(m - \lceil U^p \rceil) &> \sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^h)} u_i^h \\ &= \{\text{by Def. 5, } m - \lceil U^p \rceil = m^h\} \\ 2m^h &> \sum_{i=1}^{\min(2m^h, n^h)} u_i^h \end{aligned}$$

Similarly, to see that (10) holds, note the following.

$$\begin{aligned} 2(m - U^p) - u_1^h &> \sum_{i=1}^{\min(2(m-\lceil U^p \rceil), n^h)} u_i^h \\ &= \{\text{by Def. 5, } m - \lceil U^p \rceil = m^h.\} \\ 2(m^h + a^h) - u_1^h &> \sum_{i=1}^{\min(2m^h, n^h)} u_i^h. \end{aligned}$$

Having verified all conditions of Lemmas 2 and 3, we conclude that $\tau^p$ is schedulable on $\pi^p$ and $\tau^h$ is schedulable on $\pi^h$. ◀

Again, a special case applies if $U^p$ is integral.

▶ **Corollary 6.** *If $U^p$ is integral, then both $\tau^p$ and $\tau^h$ are schedulable on their respective sub-platforms under GEDF so long as $U^E \leq m$ holds.*

**Proof.** Similar to the proof of Lemma 4. ◀

It is not strictly necessary that $\pi^p$ be defined as we do here. If we allow other design considerations, such as maximizing cache affinity or minimizing tardiness, different platform definitions may be preferable, but we defer those possibilities to future work.

By themselves, the results of this section are not very useful, since there are an exponential number of possible ways to partition $\pi$. In the next section, we show how to efficiently find $\tau^p$ and $\tau^h$ that will be schedulable under Theorem 5.

## 3.2 Sub-Problem 2: Dividing the Tasks

So far, we have addressed how to schedule a task system $\tau$ for a given pair of subsystems $\tau^p$ and $\tau^h$. Here, we show how we arrive at Assumption 1—$\tau$ has already been divided—and weaken Assumption 2, which states that all execution costs have been determined, to the following:

▶ **Assumption 4.** If $\tau_i$ is a threaded task, then $C_i^h = \max_{\forall \tau_j \in \tau} C_{i:j}$.

**Oblivious scheduling.** We first work through a simple example of dividing a task system and then formalize that approach into what we term *symbiosis-oblivious partitioning*.[5] We then show how our approach can be improved by modifying Assumption 4.

▶ **Example 5.** Let $\tau$ consist of four SMART tasks,

$$\tau_1 = \left(8, \left(7, 10, 10, 9.\overline{3}\right)\right), \qquad \tau_2 = \left(4, \left(4, 1, 2, 1.\overline{3}\right)\right),$$
$$\tau_3 = \left(4, \left(3, 2.\overline{6}, 2, 2.5\right)\right), \qquad \tau_4 = \left(8, \left(6, 6, 5.\overline{3}, 4\right)\right).$$

Under traditional sporadic scheduling, where we consider only physical costs, $\tau$ has total utilization $\frac{7}{8} + \frac{1}{4} + \frac{2}{4} + \frac{4}{8} = 2.125$ and will require three cores to be feasibly scheduled (recall that $C_{i:i}$ gives $\tau_i$'s cost with nothing co-scheduled, i.e., without SMT). Based on Assumption 4, we see that $C_1^h = 10$ if $\tau_1$ is threaded. Because $T_1 = 8$, making $\tau_1$ threaded would give $u_1^h = \frac{10}{8}$, making the system unschedulable. For $\tau_2$, $C_2^h$ would be at most $\tau^2$'s period, but $C_2^h = 4$ would be more than twice $C_2^p = 1$. Part of the schedulability condition given in Theorem 5 is that $U^E \leq m$. Because $U^E$ is defined as $U^E = U^p + \frac{U^h}{2}$ (Def. 3), placing $\tau_2$ in $\tau^h$ would increase $U^E$ more than placing $\tau_2$ in $\tau^p$, so we do not wish for $\tau_2$ to be threaded. For both $\tau_3$ and $\tau_4$, $\max C_{i:j} \leq T_i$ and $\min(\frac{C_i}{C_{i:j}}) \geq .5$ both hold, so letting those tasks be threaded would decrease $U^E$ compared to placing them in $\tau^p$ without violating $u_i^h \leq 1$, so we allow those tasks to be threaded, giving $u_3^h = \frac{3}{4}$ and $u_4^h = \frac{6}{8}$. The resulting partition has $U^p = \frac{7}{8} + \frac{1}{4}$, $U^h = \frac{3}{4} + \frac{6}{8}$, and $U^E = 1.875$. It can, per Theorem 5, be scheduled on only two cores. ◀

We formally state the steps we just took in Algorithm 1, which partitions $\tau$ into $\tau^p$ and $\tau^h$ so as to minimizes $U^E$ subject to $u_i^h \leq 1$ for all threaded tasks and $|\tau^h| \geq 2$. The resulting partition is then schedulable if Theorem 5 holds. We require that $|\tau^h| \geq 2$ holds since allowing a single threaded task will give no schedulability advantage compared to letting all tasks by physical. We refer to partitions of $\tau$ that obey both these constraints as *legal*. We will examine the effectiveness of Algorithm 1 in our schedulability study.

▶ **Definition 7.** A partition of $\tau$ into $\tau^p$ and $\tau^h$ is *legal* if and only if $\forall\ \tau_i^h \in \tau^h$, $u_i^h \leq 1$ and $|\tau^h| \neq 1$ hold.

[5] The terms symbiosis-oblivious and symbiosis-aware scheduling were previously used by Jain et al.[14].

```
1: for all τ_i ∈ τ do
2:    C_i^h ← max_{∀j≤n} C_{i:j}
3:    if C_i^h ≤ T_i and C_i / C_i^h ≥ 2 then
4:       τ^h ← τ^h ∪ τ_i
5:    else
6:       C_i^p ← C_{i:i}
7:       τ^p ← τ^p ∪ τ_i
8:    end if
9: end for
10: if |τ^h| < 2 then
11:    τ^p ← τ^p ∪ τ^h
12:    τ^h ← ∅
13: end if
14: return τ^p, τ^h
```

**Algorithm 1:** Oblivious Partitioning

**A more complex cost model.** Under Assumption 4, the only variable that influences the cost of $\tau_i$ is whether $\tau_i$ is physical or threaded. However, Assumption 4, and consequently Algorithm 1, is highly pessimistic with regard to assigning $C_i^h$ values. Returning to Example 5, we declared $C_3^h = 3$ on the grounds that $\forall j, \max C_{3:j} = 3$ holds. However, there is a limitation to that logic; $C_3^h = 3$ is based on the assumption that $\tau_1$ can interfere with $\tau_3$, but in our example, we decided that $\tau_1$ should not be threaded. We can remove this limitation, thereby improving our model, by replacing Assumption 4 with Assumption 5. The difference is that under Assumption 5, $C_i^h$ is only based on other threaded tasks, not on all tasks in $\tau$.

▶ **Assumption 5.** If $\tau_i$ is threaded, then $C_i^h = \max_{\forall \tau_j \in \tau^h} C_{i:j}$.

The difference is that while Assumption 4 considers interference from all tasks in $\tau$, Assumption 5 considers interference only from other tasks in $\tau^h$. While this approach removes some of the pessimism present in symbiosis-oblivious scheduling, it has the disadvantage that every time a task is added to or removed from $\tau^h$, $C_i^h$ may change for all tasks in $\tau^h$. We refer to task-partitioning algorithms that incorporate Assumption 5 as *symbiosis-aware partitioning.* We give a brief demonstration of symbiosis-aware partitioning in Example 6, using the same task set as in Example 5.

▶ **Example 6.** We first decide that $\tau_1$ must be physical, since $\forall\, j \neq 1, C_{1:j} > T_1$. Knowing that no task will be co-scheduled with $\tau_1$, we have $C_2^h = 2$ and $C_3^h = 2.\overline{6}$, giving $u_2^h = \frac{2}{4}$ and $u_3^h = \frac{2.\overline{6}}{4}$, but leaving $u_4^h$ unchanged. (In Example 5, we made $\tau_2$ a physical task and $\tau_3$ a threaded task with $u_3^h = \frac{3}{4}$.) Now we make all of $\tau_2$, $\tau_3$, and $\tau_4$ threaded, with $\tau_3$ having a lower utilization than before. We now get $U^p = \frac{7}{8}$ and $U^h = \frac{2}{4} + \frac{2.\overline{6}}{4} + \frac{6}{8}$, so that $U^E = 1.8\overline{3}$. a reduction from $U^E = 1.875$ in Example 5. Again, $\tau^p$ and $\tau^h$ are schedulable on two cores per Theorem 5.

**A greedy approach to schedulability.** We propose to use Algorithm 2 to partition $\tau$. The algorithm seeks to minimize $U^E$ by repeatedly moving the task whose movement from $\tau^p$ to $\tau^h$, or vice versa, will give the greatest decrease to $U^E$. It does so until either a specified maximum number of attempts has been made or it reaches a partition that cannot be improved by the movement of any single task. The algorithm is not optimal, even given an unlimited number of attempts, as there may exist partitions of $\tau$ that cannot be improved by moving any one task but can be improved by moving two or more tasks.

The **for** loop of lines 3 through 16 determines, for every $\tau_i$ in $\tau^p$, the benefit of moving that task to $\tau^h$. Line 4 tests what $C_i^h$ would be if $\tau_i$ were in $\tau^h$. Lines 10 through 13 calculate the change to tasks already in $\tau^h$ caused by moving $\tau_i$, and line 15 gives the total change to $U^E$ caused by moving $\tau_i$ to $\tau^h$.

**Require:** $\tau$ partitioned such that $\forall \tau_i \in \tau^h, u_i^h \leq 1$ and $|\tau^h| \geq 2$
1: **for** $\ell \leftarrow 1...maxLoops$ **do**
2: ▷ Identify best move from $\tau^p$ to $\tau^h$
3: **for all** $\tau_i \in \tau^p$ **do**
4: $C_i^h = \max_{\tau_i \in \tau^h} C_{i:j}$
5: $u_i^h = \frac{C_i^h}{T_i}$
6: **if** $u_i^h > 1$ **then**
7: **continue**
8: **end if**
9: ▷ Calculate how adding $\tau_i$ to $\tau^h$ will affect tasks already in $\tau_h$
10: **if** moving $\tau_i$ to $\tau^h$ will cause $u_j^h \geq 1$ for any $\tau_j \in \tau^h$ **then**
11: **continue**
12: **end if**
13: $I(\tau_i^h) \leftarrow$ total increase in util. of tasks already in $\tau^h$ caused by moving $\tau_i$
14: ▷ $\Delta(i)$ gives decrease to $U^E$ caused by moving $\tau_i$.
15: $\Delta(i) \leftarrow u_i^p - \frac{u_i^h + I(\tau_i^h)}{2}$
16: **end for**
17: ▷ Identify best move from $\tau^h$ to $\tau^p$
18: **if** $|\tau^h| > 2$ **then**
19: **for all** $\tau_j \in \tau_h$ **do**
20: $D(\tau_j^h) \leftarrow$ total decrease in util. of tasks already in $\tau^h$ caused by moving $\tau_j$
21: ▷ $\Delta(j)$ gives decrease to $U^E$ caused by moving $\tau_j$.
22: $\Delta(j) \leftarrow \frac{u_j^h + D(\tau_j^h)}{2} - u_j^p$
23: **end for**
24: **end if**
25: **if** no task has a positive $\Delta$ value **then**
26: **break**
27: **end if**
28: Move task with maximum $\Delta$ to other subsystem and update threaded costs
29: **end for**
30: **return**$(\tau^p, \tau^h)$

**Algorithm 2:** Greedy Partitioning

Similarly, the **for** loop of lines 19 through 23 determines the benefit of moving $\tau_j$ to $\tau^p$, for every $\tau_j$ currently in $\tau^h$. Line 20 gives the change to tasks remaining in $\tau^h$ caused by moving $\tau_j$, and line 22 gives the total change to $U^E$ caused by moving $\tau_j$ to $\tau^p$. The **if** of line 25 guarantees that no task will be moved unless moving that task will decrease $U^E$, preventing the algorithm from placing $\tau$ into any one partition more than once.

The algorithm returns a partition that can be tested for schedulability by Theorem 5.

Algorithm 2 assumes, and maintains as an invariant, that the partition is legal, as defined in Def. 7. To begin Algorithm 2, $\tau$ must already be in a legal partition. We propose three ways to achieve this. First, in the *threaded start* approach, we begin with all tasks in $\tau^h$ and then place into $\tau^p$ all

tasks for which any possible $C_i^h$ value will give $u_i^h > 1$. Intuitively, putting tasks in $\tau^h$ whenever possible should be beneficial, so we should start with as many tasks in $\tau^h$ as possible.

Second, in the *physical start* approach, we start with all tasks in $\tau^p$ apart from the single pair of tasks that will give the greatest decrease to $U^E$. This can be done by defining the decrease to $U^E$ associated with a single pair of tasks $(\tau_i, \tau_j)$ as

$$\forall\ (i,j), \Delta(i,j) = u_i^p + u_j^p - \frac{1}{2}\left(\frac{C_{i:j}}{T_i} + \frac{C_{j:i}}{T_j}\right)$$

and adding to $\tau^h$ the pair of tasks that maximize $\Delta(i,j)$ subject to $\frac{C_{i:j}}{T_i} \leq 1$ and $\frac{C_{j:i}}{T_j} \leq 1$. When $\tau_i$ and $\tau_j$ are placed into $\tau^h$, $u_i^p$ and $u_j^p$ are no longer part of $U^p$ and can thus be subtracted from $U^E$. However, we must add half of the new $U^h$ value, $\left(\frac{C_{i:j}}{T_i} + \frac{C_{j:i}}{T_j}\right)$, to $U^E$. We expect this approach will be more efficient than the first one in task systems where $u_i^p$ is typically large or $\frac{C_i}{C_{i:j}}$ is typically small, since there will be relatively few tasks that can be placed in $\tau^h$, making it more efficient to begin with the majority of tasks in $\tau^p$. If no satisfactory pair of tasks exists, then we conclude that SMT should not be used with this task system.

Third, in the *mixed start* approach, we first run Algorithm 1 and use the partition given by doing so as our starting point. Intuitively, Algorithm 1 by itself should give a partition with a lower $U^E$ value than either of the other two approaches, so using it is a starting point should yield better results. As with the physical approach, if Algorithm 1 places no tasks in $\tau^h$, then we conclude that SMT should not be used. We compare these three approaches in our schedulability experiments, presented in Sec. 5. We expected that the no default approach would dominate, but we found that for all three versions of Algorithm 2, there existed task systems that were schedulable according to that version alone. In fact, the default physical approach seemed to find more schedulable task systems than the other two.

# 4 Sub-Problem 1: SMT and Execution Times

Current literature does not address how SMT affects worst-case execution costs. While the early 2000s saw multiple detailed analyses of the performance effects of SMT [1, 2, 24], little work of this type has been done since then. While ongoing research into scheduling with SMT exists outside of real time [7, 8, 10, 22], this current research does not suit our needs for two reasons. First, it tends to be oriented towards total throughput and average execution costs, whereas we need information on worst-case execution costs. Second, the current works we are aware of compare different methods of implementing SMT, but do not compare systems that use SMT to those that do not use it.

## 4.1 Benchmark Experiments

To analyze the effects of SMT on worst-case execution costs, we ran a series of experiments using the TACLeBench sequential benchmarks [9], which consist of 23 C implementations of functions commonly found in embedded and real-time systems. All of our experiments were conducted in Linux on an Intel Xeon Silver 4110 2.1 GHz CPU with eight cores, each capable of supporting two threaded processors, running Linux.[6]

To get baseline results for execution times without SMT enabled, we looped each benchmark 1,000 to 100,000 times—lower cost benchmarks got more loops—and timed the execution of each loop using a nanosecond resolution timer. Between loops, an array the size of the L3 cache was

[6] The code used for these experiments is available at https://github.com/JoshuaJB/SMART-ECRTS19

**Table 1** Baseline Execution Times (ns)

| Benchmark | max | mean | $CV\left(\frac{\text{std. dev.}}{\text{mean}}\right)$ |
|---|---|---|---|
| adpcm_dec | 167,380 | 151,914 | 0.006659 |
| adpcm_enc | 158,053 | 147,394 | 0.006463 |
| ammunition | 47,979,870 | 47,899,553 | 0.001589 |
| cjpeg_transupp | 844,791 | 827,661 | 0.002087 |
| cjpeg_wrbmp | 32,420 | 26,712 | 0.010552 |
| dijkstra | 15,740,782 | 15,719,309 | 0.000445 |
| epic | 665,837 | 649,170 | 0.002284 |
| fmref | 154,776 | 99,280 | 0.068863 |
| gsm_dec | 470,193 | 463,592 | 0.002546 |
| gsm_enc | 1,337,465 | 1,320,787 | 0.001934 |
| h264_dec | 93,361 | 82,045 | 0.006340 |
| huff_enc | 247,232 | 234,213 | 0.005431 |
| mpeg2 | 135,009,849 | 134,898,300 | 0.000248 |
| ndes | 21,600 | 15,426 | 0.015071 |
| petrinet | 3,682 | 62 | 0.215268 |
| rijndael_dec | 965,022 | 940,081 | 0.007688 |
| rijndael_enc | 872,400 | 858,645 | 0.002224 |
| statemate | 11,928 | 6,495 | 0.026602 |
| susan | 10,958,260 | 10,932,188 | 0.000379 |

allocated and set, guaranteeing that every execution started with a cold cache. Benchmarks were assigned a Linux real-time priority, prioritizing them above all normal tasks, pinned to a single processor, and executed sequentially. We were forced to exclude four benchmarks from the set—anagram, audiobeam, g723_enc, and huff_dec—as they would not correctly execute in a loop. Results of our baseline experiments are summarized in Table 1. The last column gives the coefficient of variation, defined as the standard deviation divided by the mean.

For threaded execution times, every task was executed alongside every other task. For each pair, the measured task was executed the same number of times as in the baseline experiments while an interfering task executed continuously at equal priority on the second thread of the same core. Our results are summarized in Fig. 5, which shows $r_{i:j}$ for every pair of tasks, with the measured task as $\tau_i$ and the interfering task as $\tau_j$. Observed rates ranged from 0.51 (mpeg2 interfering with epic) to 1.00, with the exception of values involving petrinet. Petrinet has an extremly short execution time, as indicated in Table 1; we suspect its strange behavior is merely random noise.

While we have defined $C_{i:i}$ as the cost of $\tau_i$ with no co-schedule, the main diagonal of Fig. 5 shows how much slower a task runs when executed with a second copy of itself. This is irrelevant for real-time systems in which task parallelism is forbidden, but is relevant to systems in which different jobs of the same task may execute in parallel, as discussed by Voronov, Anderson, and Yang [26]. Prior to performing our experiments, we had expected that tasks executed alongside copies of themselves would have very low $r_{i:j}$, values, due to competing for the same resources, but our experiments show this is not necessarily the case.

## 4.2 Benchmark Characterization

In our results, we observe that tasks are relatively consistent both in how vulnerable they are to interference from other tasks and in how much interference they cause to other tasks. This is similar

| measured benchmark \ interfering benchmark | adpcm_dec | adpcm_enc | ammunition | cjpeg_transupp | cjpeg_wrbmp | dijkstra | epic | fmref | gsm_dec | gsm_enc | h264_dec | huff_enc | mpeg2 | ndes | petrinet | rijndael_dec | rijndael_enc | statemate | susan | MINIMUM | max CV |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| adpcm_dec | 0.97 | 0.96 | 0.98 | 0.98 | 0.99 | 0.99 | 0.94 | 0.96 | 0.96 | 0.98 | 0.97 | 0.97 | 0.97 | 0.96 | 1.00 | 0.92 | 0.97 | 1.00 | 0.94 | **0.92** | 0.012617 |
| adpcm_enc | 0.91 | 0.85 | 0.91 | 0.94 | 0.97 | 0.95 | 0.94 | 0.93 | 0.95 | 0.95 | 0.94 | 0.94 | 0.93 | 0.95 | 0.96 | 0.94 | 0.96 | 0.92 | 0.92 | **0.85** | 0.011473 |
| ammunition | 0.67 | 0.66 | 0.67 | 0.65 | 0.69 | 0.68 | 0.69 | 0.64 | 0.64 | 0.66 | 0.68 | 0.68 | 0.66 | 0.68 | 0.69 | 0.68 | 0.70 | 0.71 | 0.68 | **0.64** | 0.001080 |
| cjpeg_transupp | 0.68 | 0.67 | 0.70 | 0.63 | 0.65 | 0.64 | 0.72 | 0.63 | 0.62 | 0.66 | 0.63 | 0.65 | 0.64 | 0.67 | 0.77 | 0.68 | 0.68 | 0.68 | 0.62 | **0.62** | 0.010452 |
| cjpeg_wrbmp | 0.69 | 0.63 | 0.63 | 0.62 | 0.59 | 0.65 | 0.68 | 0.69 | 0.60 | 0.65 | 0.54 | 0.55 | 0.60 | 0.63 | 0.74 | 0.66 | 0.52 | 0.66 | 0.62 | **0.52** | 0.061954 |
| dijkstra | 0.70 | 0.69 | 0.74 | 0.68 | 0.71 | 0.70 | 0.74 | 0.67 | 0.66 | 0.69 | 0.70 | 0.71 | 0.69 | 0.71 | 0.79 | 0.72 | 0.72 | 0.72 | 0.70 | **0.66** | 0.002617 |
| epic | 0.54 | 0.53 | 0.57 | 0.54 | 0.54 | 0.59 | 0.57 | 0.57 | 0.56 | 0.54 | 0.57 | 0.55 | 0.51 | 0.55 | 0.59 | 0.55 | 0.55 | 0.54 | 0.54 | **0.51** | 0.013115 |
| fmref | 0.75 | 0.75 | 0.76 | 0.73 | 0.75 | 0.66 | 0.77 | 0.74 | 0.73 | 0.73 | 0.71 | 0.71 | 0.73 | 0.75 | 0.79 | 0.76 | 0.76 | 0.76 | 0.67 | **0.66** | 0.060101 |
| gsm_dec | 0.64 | 0.63 | 0.64 | 0.61 | 0.65 | 0.63 | 0.68 | 0.60 | 0.60 | 0.61 | 0.62 | 0.63 | 0.61 | 0.63 | 0.71 | 0.64 | 0.64 | 0.65 | 0.61 | **0.60** | 0.011700 |
| gsm_enc | 0.59 | 0.58 | 0.60 | 0.56 | 0.61 | 0.62 | 0.64 | 0.57 | 0.57 | 0.59 | 0.60 | 0.61 | 0.58 | 0.61 | 0.65 | 0.61 | 0.62 | 0.63 | 0.59 | **0.56** | 0.012556 |
| h264_dec | 0.91 | 0.92 | 0.90 | 0.86 | 0.87 | 0.87 | 0.96 | 0.85 | 0.84 | 0.87 | 0.85 | 0.88 | 0.88 | 0.91 | 1.00 | 0.88 | 0.79 | 0.75 | 0.87 | **0.75** | 0.030283 |
| huff_enc | 0.73 | 0.70 | 0.74 | 0.67 | 0.69 | 0.66 | 0.79 | 0.71 | 0.67 | 0.69 | 0.69 | 0.71 | 0.66 | 0.72 | 0.79 | 0.69 | 0.71 | 0.67 | 0.69 | **0.66** | 0.023466 |
| mpeg2 | 0.72 | 0.71 | 0.73 | 0.66 | 0.72 | 0.70 | 0.75 | 0.69 | 0.68 | 0.70 | 0.69 | 0.70 | 0.67 | 0.71 | 0.64 | 0.72 | 0.72 | 0.72 | 0.70 | **0.64** | 0.144416 |
| ndes | 0.66 | 0.68 | 0.69 | 0.67 | 0.71 | 0.69 | 0.72 | 0.70 | 0.67 | 0.57 | 0.66 | 0.59 | 0.61 | 0.55 | 0.74 | 0.70 | 0.56 | 0.72 | 0.67 | **0.55** | 0.029245 |
| petrinet | 6.11 | 1.24 | 0.91 | 5.93 | 1.05 | 5.68 | 0.88 | 0.94 | 1.20 | 0.78 | 1.18 | 0.69 | 0.78 | 0.82 | 0.71 | 0.60 | 1.22 | 0.82 | 0.98 | **0.60** | 0.875009 |
| rijndael_dec | 0.58 | 0.59 | 0.58 | 0.64 | 0.65 | 0.65 | 0.67 | 0.64 | 0.61 | 0.63 | 0.65 | 0.65 | 0.62 | 0.63 | 0.64 | 0.61 | 0.61 | 0.64 | 0.63 | **0.58** | 0.016778 |
| rijndael_enc | 0.56 | 0.56 | 0.57 | 0.61 | 0.63 | 0.64 | 0.65 | 0.62 | 0.59 | 0.60 | 0.63 | 0.63 | 0.59 | 0.60 | 0.60 | 0.59 | 0.58 | 0.61 | 0.61 | **0.56** | 0.013859 |
| statemate | 0.66 | 0.99 | 0.88 | 0.61 | 0.84 | 0.86 | 1.00 | 0.89 | 0.61 | 0.68 | 0.77 | 0.97 | 0.97 | 0.96 | 0.95 | 0.55 | 0.97 | 0.73 | 0.93 | **0.55** | 0.027924 |
| susan | 0.62 | 0.62 | 0.61 | 0.56 | 0.58 | 0.55 | 0.67 | 0.59 | 0.56 | 0.60 | 0.56 | 0.58 | 0.58 | 0.61 | 0.69 | 0.61 | 0.62 | 0.64 | 0.57 | **0.55** | 0.004294 |

**Figure 5** Effect of SMT on execution times. Measured benchmarks execute with the listed $r_{i:j}$ values when sharing a thread with a given interfering benchmark. Shading is darkest on smallest values. Right column shows the maximum coefficient of variation experienced by each measured benchmark over all interfering benchmarks.

to other results in the literature [1, 2, 13, 24]. Visually, this produces a banding effect in Fig. 5. We say that tasks that experience little interference from other tasks—i.e. tasks $\tau_i$ for which $r_{i:j}$ tends to be high—are *strong*, and that tasks which cause little interference to other tasks—i.e. $\tau_i$ for which $r_{j:i}$ tends to be high—are *friendly*. When we define a *strength score* $s_i = mean_j(r_{i:j})$ and *friendliness score* $f_i = mean_j(r_{j:i})$, no task has a Pearson correlation[7] with absolute value greater than $0.14$ between $s_i$ and $f_i$ values. Bulpin's work on the behavior of threaded tasks discusses this lack of correlation in detail [1, 2].

For both values, we centered and standardized each row and column before fitting them to several common statistical distributions via a log-likelihood maximization. We found the Gaussian distribution to best approximate the results from our experiments. Applying a maximum likelihood (MLE) estimation, we found that mean 0.72 and standard deviation 0.13 were the best for $s_i$ while mean 0.72 and standard deviation 0.04 were best for $f_i$.

## 4.3 Reliability of Measured Worst-Case Costs

We stated in Assumption 4 that $C_i^h$ is no more than $\max_{\tau_j \in \tau^h} C_{i:j}$. While we are confident that violations will be rare, we cannot guarantee there will not be any. In particular, our assumption that all portions of $\tau_i$ receive the same amount of interference from all portions of $\tau_j$ is a potential source of timing violations. For example, let $\tau^h = \{\tau_1^h, \tau_2^h, \tau_3^h\}$ be such that $C_{1:2} = C_{1:3} = 6$. Under Assumption 5, the worst-case execution time for $\tau_1$ is 6. Suppose $\tau_1$ can be broken into two segments, $\tau_{1a}$ and $\tau_{1b}$, such that $C_{1a:2} = 4$, $C_{1b:2} = 2$, $C_{1a:3} = 2$, and $C_{1b:3} = 4$. If $\tau_{1a}$ is co-scheduled with $\tau_2$ and $\tau_{1b}$ is co-scheduled with $\tau_3$, $\tau_1$'s total execution time would be 8, violating our stated worst-case execution costs. At present, our benchmark tests would not discover this scenario and our model does not account for it. In the future, we would like to resolve this issue with finer-grained timing analysis and a model that does not assume task interference is independent from location within the task. In

[7] A Pearson correlation of $\pm 1$ indicates total positive or negative linear correlation; 0 indicates no correlation.

particular, breaking tasks into segments, determining execution costs per segment, as in our example, and conducting an analysis similar to this paper, but at a finer granularity, seems like a promising way forward. For now, we reiterate that we are only considering applications that are not safety-critical and where some tardiness is acceptable.

In general, precise timing analysis on multicore is a hard problem. While SMT may make it harder, uncertainty in timing analysis is present even without SMT. Fortunately, Mills and Anderson have shown that in SRT systems, expected tardiness can be bounded based on average rather than worst-case execution times [18]. Their approach relies on designating per-task execution budgets so that if any one job overruns its budget, it will not receive further execution time until a subsequent job of the same task could have been executed had the first job completed. Mills and Anderson base their budgets on average execution times. Therefore, so long as our stated costs are greater than the true average costs, any system $\tau$ that can be scheduled as we have described will remain schedulable, though possibly with increased tardiness, even if not all of our stated costs are true worst-case costs.

# 5 Schedulability Experiments

Having shown how to schedule SMT-enabled systems and having analyzed the behavior of our benchmark tasks when using SMT, it remains to be seen whether we can generally schedule otherwise unschedulable systems. To answer this question, we ran a series of schedulabilty experiments.

## 5.1 Experimental Procedure

To run our experiments, we created synthetic task systems to be scheduled on platforms with $m$ cores, $m \in \{4, 8, 16\}$ such that the total system utilization ranged from $m$ to $2m$. For each task system, we partitioned the system into $\tau^p$ and $\tau^h$ using Algorithm 1 and all three versions of Algorithm 2. We then tested for schedulability per Theorem 5. We created enough task systems that each data point in our graphs represents the composite schedulability of approximately 1,000 task systems. We created approximately 1,000 graphs, with a few thousand to hundreds of thousands of task systems per graph. The process of creating task sets, partitioning task sets, and testing for schedulability consumed over 30 days of CPU time.

We plotted these results on a series of schedulability graphs with total utilizations on the horizontal axis and the proportion of systems that were schedulable on the vertical axis. Since we started at utilization $m$, and the standard SRT feasibility condition given by (1) requires that $\sum_{i=1}^{n} u_i \leq m$ hold, every single system we created was infeasible without using SMT. Every system that we could schedule is an argument for adapting SMT in real-time.

Each graph shows results for tasks created using a common set of parameters for utilization and $r_{i:j}$ values. Task utilizations were assigned from one of four ranges: the uniform distributions (0, .4], [.3, .7], [.6, 1], and (0, 1].

We used two different approaches for determining $r_{i:j}$ values. In the *Gaussian-average* approach, we drew $s_i$ and $f_i$ from the Gaussian distributions with mean 0.72 for both values and standard deviations ranging from 0.13 to 0.39 for $s_i$ and from 0.04 to 0.12 for $f_i$. These parameters are based on the distributions we fitted to our models, as discussed in the previous section. We allowed larger standard deviations than we obtained from our benchmarks to make our results more widely applicable.

In the *uniform-normal* approach, both $s_i$ and $f_i$ were chosen from one of four uniform distributions: [.65, 1], [.7, 1], [.75, 1], or [.8, 1]. The two ranges were not necessarily the same for a given graph. Each $r_{i:j}$ value was then chosen from the normal distribution with mean $s_i f_i$ and standard deviation $\sigma$, where $\sigma$ had the value of .01, .05, or .1. This method may result in both negative values and values greater than 1; in our schedulability tests, we interpret such values as 0 and 1, respectively. The

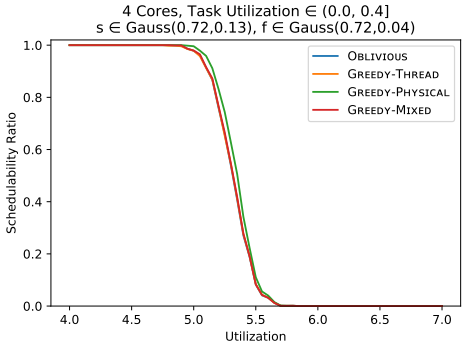


**Figure 6** Graph shape is similar to Fig. 1, which has more cores.

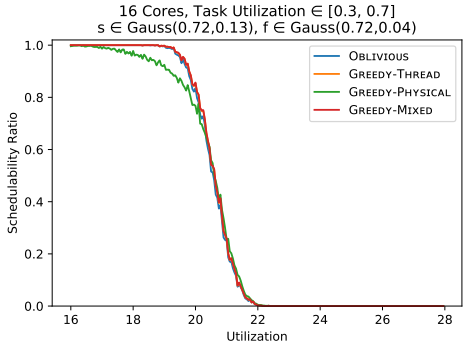


**Figure 7** Schedulability is similar to that of Figs. 1 and 6, despite the higher per-task utils.

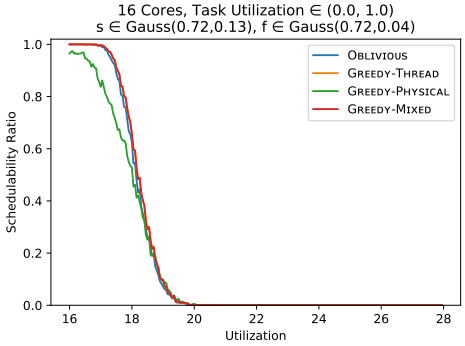


**Figure 8** Despite the same expected per-task util. as Fig. 7, schedulability is reduced.

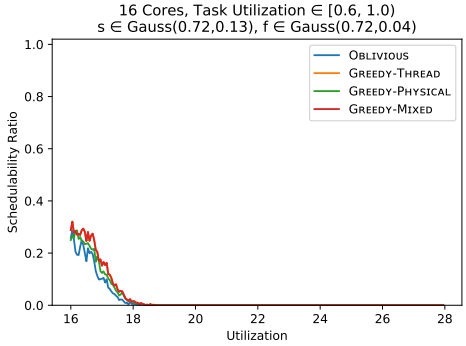


**Figure 9** Given high per-task utilizations, only small schedulability gains can be achieved.

intuition behind the uniform-normal approach was to create $r_{i:j}$ values that were broadly similar to the benchmark values we obtained, but using a radically different method from the Gaussian-average approach so as to avoid having our results be overly dependent on that model. While high $s_i$ values in this context still indicate tasks that receive little interference from other tasks, and high $f_i$ values indicate tasks that cause little interference to others, they are used differently here than in the Gaussian average approach and should not be directly compared.

## 5.2 Results

Due to space constraints, we select and present only a small portion of our graphs here to highlight general trends. A full set of graphs will be made available in our online appendix.[8]

▶ **Observation 1.** Task systems with low per-task utilization received the greatest improvement in schedulability, and task systems with high utilization saw the least.

When we partition our task systems, we require that $C_i^h \leq T_i$ holds for all threaded tasks (as a threaded task with $C_i^h > T_i$ would not be schedulable). Since threading tasks increases individual execution costs, it will typically not be possible to thread tasks that already have high utilizations. Fig. 6 shows schedulability for task systems with individual utilizations drawn from the

[8] The appendix is available at http://jamesanderson.web.unc.edu/papers/ The code used for these experiments is available at https://github.com/JoshuaJB/SMART-ECRTS19

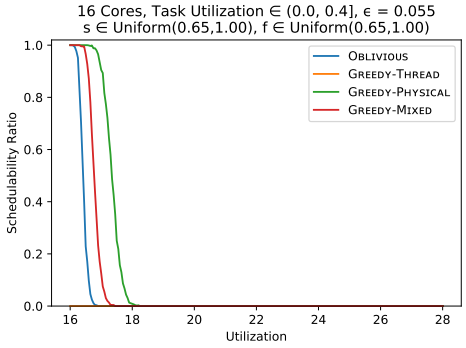


**Figure 10** System with uniform-normal $r_{i:j}$ values. Note the variation in algorithm performance. Greedy-thread achieves no improvement.

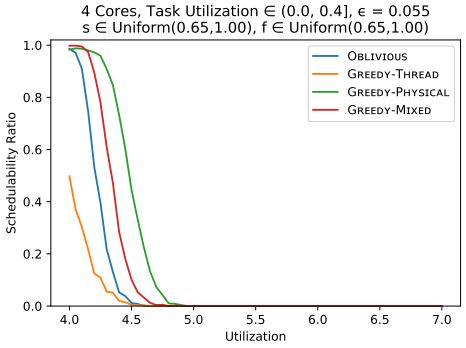


**Figure 11** Uniform-normal $r_{i:j}$ schedulability on 4 cores. Unlike the Gaussian model, core count does influence improvement from SMT here.

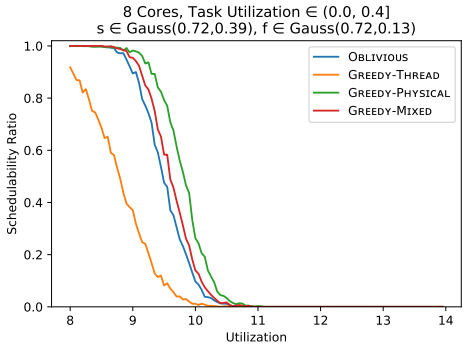


**Figure 12** A higher variance version of the Gaussian $r_{i:j}$ approach. Improvement from SMT is reduced compared to Figs. 1, 6, and 7.

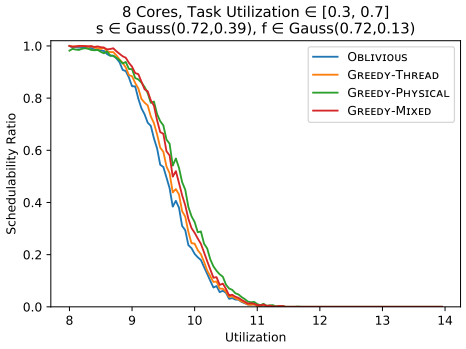


**Figure 13** Underperformance of GREEDY-THREAD as in Fig. 12 disappears as task utilizations increase, with all else constant.

uniform distribution $(0, 0.4]$, and shows that the majority of systems considered are schedulable with utilizations as high as $5.34$. Fig. 9 has the same parameters as Figs. 1 and 6, but draws utilizations instead from the range $[.6, 1]$. This graph shows virtually no improvement when run with SMT.

The presence or absence of high-utilization tasks seems to be more important to schedulability than average per-task utilization. Figs. 7 and 8 have the same per-task expected utilizations, but while Fig. 7, which has task utilizations from the range $[.3, 7]$ gives nearly the same results as Figs. 1 and 6, Fig. 8, which has utilizations in the range $(0, 1]$, does noticeably worse than Fig. 7.

▶ **Observation 2.** Returns do not diminish as core counts increase.

Systems scheduled on higher core counts see similar increases to schedulable utilization as do systems scheduled on low core counts. This can be observed by comparing figures Fig. 6 and Fig. 1. They both exhibit similar schedulability improvements proportional to their core count even though one is a 4-core system and the other one is a 16-core system.

▶ **Observation 3.** Algorithm 1, oblivious partitioning, competes with the more complex greedy algorithms.

Looking at our best results, such as Figs. 1, 6, 7, and 13, Algorithm 1 is indistinguishable from the greedy algorithms. Some difference occurs in Figs. 10 and 11 but even when Algorithm 1 does not perform as well as the variants of Algorithm 2, the difference is small enough that in some

circumstances, such as dynamic task entry and exit, the lower algorithm complexity might make it a better choice.

▶ **Observation 4.** Lower $r_{i:j}$ variability yields improved schedulability, even when controlling for average $r_{i:j}$ value.

Consider Fig. 12. The task systems sample from the same utilization range as those of Figs. 1 and 6, but here the standard deviation of the distribution from which $s_i$ and $f_i$ are sampled is larger by a factor of three. This increased variance causes the algorithms charted in Fig. 12 to be unable to schedule as many task sets as in Figs. 1 and 6.

▶ **Observation 5.** Schedulability benefits of our methods are not limited to task systems generated using a single model.

While the Gaussian-average approach generally created systems that saw more improvement from SMT, the benefits of SMT are not limited to task systems created under that model, suggesting that SMT can benefit a wide variety of task systems, as opposed to only those that are particularly similar to the TACLeBench tasks. While Figs. 10 and 11—the only uniform-normal graphs we show—give less improvement than many of our other examples, they still allow for task systems with utilizations significantly greater than $m$ to be scheduled. Note that Figs. 10 and 11 use pessimistic ranges for both $s_i$ and $f_i$ values; other uniform-normal based task systems saw more improvement from SMT.

## 6 Conclusion

In this paper, we have given a task model, SMART, that allows for reasoning about SMT-enabled task systems by defining multiple cost parameters per task. We also have shown how to decide which tasks should and should not use SMT and how to take advantage of SMT to schedule otherwise unschedulable task systems, without sacrificing bounded tardiness guarantees. We measured the execution times of benchmark tasks with and without SMT enabled, with the SMT-enabled case covering interference from all other tasks in the set. We conducted an extensive schedulability study using synthetic tasks modeled on our benchmark tasks and showed that for task systems consisting of low utilization tasks, it is possible to schedule virtually all systems with utilization as large as $1.25m$ and to schedule many task systems with utilizations approaching $1.33m$.

In the future, we plan to consider task systems with dynamic task entry and exit. The challenge of handling this case is that when threaded tasks enter or exit the system, the costs of other threaded tasks could change, possibly necessitating run-time versions of our partitioning algorithms. In addition, we want to consider the problem of partitioning both task systems and hardware platforms to minimize tardiness, as opposed to simply maximizing schedulability. The problem is intriguing; making tasks threaded tends to decrease demand on the platform, potentially reducing tardiness, but at the same time will increase execution costs, which tends to increase tardiness bounds [4, 5, 15]. While the potential gains shown in this paper are substantial, we have only just begun to expose the potentials of hardware multithreading.

## References

1 J. Bulpin. *Operating system support for simultaneous multithreaded processors*. PhD thesis, University of Cambridge, King's College, 2005.
2 J. Bulpin and I. Pratt. Multiprogramming performance of the Pentium 4 with hyperthreading. In *Third Annual Workshop on Duplicating, Deconstruction and Debunking*, June 2004.
3 F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors: synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799, 2006.

**4** U. M. C. Devi and J. H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *(RTSS'05)*, 2005.
**5** U. M. C. Devi and J. H. Anderson. Flexible tardiness bounds for sporadic real-time task systems on multiprocessors. In *20th IEEE International Parallel Distributed Processing Symposium*, 2006.
**6** S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.
**7** S. Eyerman and L. Eeckhout. The benefit of SMT in the multi-core era: Flexibility towards degrees of thread-level parallelism. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 591–606, 2014.
**8** S. Eyerman, P. Michaud, and W. Rogiest. Revisiting symbiotic job scheduling. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 124–134, 2015.
**9** H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. S, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*.
**10** J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Perf fair: A progress-aware scheduler to enhance performance and fairness in smt multicores. *IEEE Transactions on Computers*, 66(5):905–911, 2017.
**11** T. Gomes, P. Garcia, S. Pinto, J. Monteiro, and A. Tavares. Bringing hardware multithreading to the real-time domain. *IEEE Embedded Systems Letters*, 8(1):2–5, 2016.
**12** T. Gomes, S. Pinto, P. Garcia, and A. Tavares. RT-SHADOWS: Real-time system hardware for agnostic and deterministic OSes within softcore. In *ETFA '15*.
**13** W. Huang, J. Lin, Z. Zhang, and J.M. Chang. Performance characterization of Java applications on SMT processors. In *ISPASS '05*.
**14** R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors.
**15** H. Leontyev and J. H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1):26–71, Mar 2010.
**16** S. Lo, K. Lam, and T. Kuo. Real-time task scheduling for SMT systems. In *RTCSA'05*.
**17** D. Marr, F. Binns, D. Hill, G. Hinton, K. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. In *Intel Technology Journal*, volume 6, pages 4–15, Feb 2002.
**18** A. F. Mills and J. H. Anderson. A stochastic framework for multiprocessor soft real-time scheduling. In *RTAS '10*.
**19** J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Using SMT to hide context switch times of large real-time tasksets. In *RTAS '10*.
**20** B. Ocker. Faa special topics. In *Collaborative Workshop: Solutions for Certification of Multicore Processors*, November 2018.
**21** S. Osborne and J. H. Anderson. Work in progress: Combining real time and multithreading. In *RTSS '18*.
**22** P. Radojković, P. M. Carpenter, M. Moretó, V. Čakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Thread assignment in multicore/multithreaded processors: A statistical approach. *IEEE Transactions on Computers*, 65(1):256–269, 2016.
**23** A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS '2000*.
**24** N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *PACT '03*.
**25** D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA '95*.
**26** S. Voronov, J. H. Anderson, and K. Yang. Tardiness bounds for fixed-priority global scheduling without intra-task precedence constraints. In *RTNS '18*.
**27** M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *RTAS '14*.