Exploiting Simultaneous Multithreading in Priority-Driven Hard Real-Time Systems

Sims Hill Osborne, Shareef Ahmed, Saujas Nandi, and James H. Anderson
Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina, U.S.A.
{shosborn, shareef, saujas, anderson} @cs.unc.edu

Abstract—Simultaneous Multithreading (SMT) has the ability to dramatically improve real-time scheduling, but existing methods are cumbersome, frequently need specialized hardware, or are limited to producing table-based schedules. Here, an easily portable method for quickly applying SMT to priority-driven hard real-time systems is given. Using a combination of integer linear programming and heuristic bin-packing, a partitioned Earliest-Deadline-First (EDF) scheduler that takes advantage of SMT is produced. The integer linear programming and partitioning are done offline, but generally require only a few seconds, even given over a hundred tasks. A large-scale schedulability study is conducted, showing that compared to partitioned scheduling without SMT, the schedulable utilization for the considered hardware platform is nearly doubled in the best cases.

Index Terms—real-time systems, simultaneous multithreading, hard real-time, scheduling algorithms

I. INTRODUCTION

Simultaneous Multithreading (SMT), a technology that allows multiple programs to execute in parallel on a single computing core, is capable of dramatically increasing the ability of a given hardware platform to schedule real-time systems [5, 8, 9, 10, 14, 15, 16, 17]. This benefit can be achieved by taking advantage of SMT’s ability to increase throughput while avoiding situations where increased execution times for individual programs—an inevitable consequence of SMT—cause deadline misses.

Unfortunately, previous work on applying SMT to hard real-time systems requires either purpose-built hardware [16, 17], modifications to basic interactions between the operating system (OS) and hardware [5, 8, 9], or a table-driven schedule [14], which may be undesirable for many applications. In the last case, our own prior work, the methods used to create scheduling tables are time-consuming, thus limiting their applicability to larger systems. These drawbacks limit the industrial applicability of SMT. Even so, industrial users are eager to make use of SMT; in particular, multiple developers have expressed interest to the U.S. Federal Aviation Administration (FAA) in using SMT in safety-critical systems [12].

In this work, we attempt to bridge the gap between the theoretical potential of SMT and applying that potential to an existing system that assumes off-the-shelf hardware and priority-driven scheduling. With that in mind, we show how to use SMT to transform an otherwise unschedulable task system into a task system that can be scheduled using Earliest-Deadline-First (EDF) or another priority-driven scheduler. We do so without sacrificing safety and without relying on a customized hardware platform or OS. As an added bonus to industrial users, we show that our transformation step can be performed quickly—under three seconds, in the majority of scenarios we considered, and in less than a minute for all considered scenarios—so that this stage of testing potential systems will not become a bottleneck in the development process. We judge our methods’ success via a large-scale schedulability study in which we track both the proportion of task systems that can be scheduled on a given platform and how much time is needed to conduct each test.

Contribution and organization. We show within the context of our schedulability study that when we apply our transformation process, the transformed system can be successfully scheduled with partitioned EDF, even, in some cases, when the original system has total utilization approaching double what could be scheduled on the given hardware platform without SMT. Furthermore, the transformation step can be completed in under a minute, even given a system that includes hundreds of tasks. While this may seem like a long time, it is reasonable for a one-time, offline step, particularly considering the possible benefits. The resulting schedule will be no less safe than scheduling the same task system without using SMT.

In Sec. II, we cover necessary background information, including an overview of SMT technology, a review of partitioned EDF scheduling, and an explanation of how we quantify safety. In Sec. III, we give a solution to our problem. The steps of our solution are depicted graphically in Fig. 1. Beginning with an initial task system \( \tau \), we use an integer linear program (ILP) to transform it into a system
A task and platform model

We consider the problem of scheduling a periodic hard real-time task system \( \tau \) that consists of \( n \) independent tasks. Each task \( \tau_i \) releases a single job every \( T_i \) time units—\( T_i \) gives the task’s period—starting at time 0, and each job is assumed to have a maximum cost of \( C_i \leq T_i \) time units. We briefly discuss the determination of safe \( C_i \) values in Sec. II-C below. The \( a^{th} \) job released by \( \tau_i \) is denoted \( \tau_i,a \), and tasks are denoted \( \tau_i = (C_i, T_i) \). Each task \( \tau_i \) has a utilization given by \( u_i = \frac{C_i}{T_i} \). The total utilization of all tasks is given by \( U \). We assume implicit deadlines: every job must complete within \( T_i \) time units of its release. Our methods work best when the number of distinct periods within \( \tau \) is small relative to our hardware platform’s core count, but we do not enforce a strict cutoff for the number of different periods.

When SMT is not used for a particular task, every job of that task is fully preemptable; accounting for the costs of preemption is a well-studied topic. Here, we assume that all task costs are inflated to account for the costs of preemptions without SMT. We assume that interference between jobs executing on separate cores, due to causes including cache conflicts, DRAM conflicts, memory bus conflicts, general OS support, and I/O conflicts, is negligible. The system is scheduled correctly if it can be shown that no job will ever miss a deadline. An individual task is said to be scheduled correctly if no job of that task will ever miss a deadline. We introduce additional task notation, and discuss how we handle preemptions when SMT is used, as part of our overview of SMT below.

B. Overview of SMT Technology

On modern computers, each core uses instruction-level parallelism within jobs to execute multiple instructions per cycle. When SMT is enabled, this behavior is expanded to allow multiple jobs to execute instructions within a single cycle. An overview is given in Ex. 1 and Fig. 2 below, both of which closely follow explanations found in our previous work [14]. Further information on the fundamentals of SMT can be found in the works of Eggers et al. [6]. For a detailed discussion of factors that can affect SMT execution in practice, see [2, 3].

Ex. 1. At the top of Fig. 2, jobs of tasks \( \tau_1 \) (darker colored) and \( \tau_2 \) (lighter colored) execute sequentially without SMT on a core that can accept two instructions per cycle. When fewer than two instructions are ready, as in cycles 3 and 4, execution resources are wasted. \( \tau_1 \) finishes at the end of 6 cycles and \( \tau_2 \) at the end of 12. In the second part of the figure, the same jobs employ SMT to execute in parallel, thereby reducing the number of lost cycles. \( \tau_1 \) finishes after 8 cycles and \( \tau_2 \) after 10. SMT thus delays the completion of \( \tau_1 \), but speeds up the completion of \( \tau_2 \) since it does not have to wait for \( \tau_1 \) to complete before beginning its own execution.

In addition to increasing the execution time of individual jobs, SMT can make it more difficult to predict job execution times due to interactions between jobs that share a core. To mitigate this problem, we require that jobs employing SMT be simultaneously co-scheduled.

Def. 1. [14] Two jobs are simultaneously co-scheduled if both begin execution simultaneously on separate hardware.
threads of the same core, and when one job completes, the remaining job continues on the same core until complete. 

Simultaneously co-scheduled jobs require their own definitions for execution costs.

**Def. 2.** [14] The joint cost to simultaneously execute jobs of \( \tau_i \) and \( \tau_j \), denoted by \( C_{i:j} \), is defined as the execution time for both jobs assuming they begin simultaneously. In Fig. 2, the joint cost of \( \tau_1 \) and \( \tau_2 \) is given by \( C_{1:2} = 10 \). If \( i = j \), then \( C_{i:i} = C_i \), indicating solo execution for \( \tau_i \). Jobs with nothing co-scheduled are solo jobs.

We require simultaneous co-scheduling to limit the possibilities we need to consider when determining the execution costs of paired tasks. Without this restriction, we would need to consider in addition to the case of Ex. 1 the time required to execute \( \tau_1 \) if it began while \( \tau_2 \) was already executing on the same core, the time required to execute \( \tau_1 \) if \( \tau_2 \) began executing later on the same core, and many other possibilities, creating an insurmountable timing-analysis burden.

C. Safety

With or without SMT, safely running a task system requires that stated values for \( C_i \) and \( C_{i:j} \) accurately reflect the true costs for these items. Determining these values is non-trivial; indeed, finding the true worst-case execution times (WCETs) for tasks on modern, complex processors may be essentially impossible [4]. If a platform includes multiple cores or supports SMT, timing analysis becomes harder still. For this reason, we use measurement-based probabilistic timing analysis (MBPTA) for both solo and paired tasks. The key to producing a safe measurement-based analysis is that execution times need to be measured in circumstances that match their anticipated runtime circumstances. For this reason, we do not allow unrestricted preemptions on tasks that use SMT. If we did allow unrestricted preemptions, our measurements would need to account for all possible ways in which tasks could be preempted, but we are not aware of any existing work that considers how to account for preemption overheads with SMT (we plan to address this topic in future work).

In our model, the stated costs \( C_i \) and \( C_{i:j} \) are estimates of the true WCETs based on the maximum observed execution time for a given task over many jobs. A task system is, roughly speaking, safe enough if all stated costs \( C_i \) and \( C_{i:j} \) are such that the probability of the actual cost of an arbitrary job of \( \tau_i \) or \( \tau_{i:j} \) being no more than its stated cost is at least a given value \( q \), where \( q \) approaches one. Determining an appropriate value of \( q \) is an application-specific decision; further details of this model, including how to determine \( C_i \) and \( C_{i:j} \) values given a particular \( q \), are covered in [14].

D. Scheduling Tasks using SMT with Partitioned EDF

In this work, we make co-scheduling decisions—our transformation step—at the task level rather than the job level. Doing so creates a much simpler decision process and allows for the use of priority-based scheduling algorithms, such as EDF. To do so, we combine individual tasks into paired tasks.

**Def. 3.** If \( \tau_i \) and \( \tau_j \) are paired tasks, then the scheduler views \( \tau_{i:a} \) and \( \tau_{j:a} \) as a single schedulable entity with cost \( C_{i:j} \) and relative deadline \( T_i \) for all \( a \), i.e., \( \tau_{i:a,j:a} \) is simultaneously co-scheduled for all \( a \). We require that two paired tasks share a common period.

To schedule \( \tau \) across multiple cores, we first determine which tasks should be paired together—we discuss this topic further in Sec. III—and then assign tasks and paired tasks to individual cores. Treating each paired task as a single unit, we then test the tasks assigned to each core for schedulability. We refer to the subset of \( \tau \) assigned to core \( \pi_t \) as \( \tau^t \) and say that it has total utilization \( U^t \).

**Dealing with preemptions.** Safely preempting paired tasks requires careful consideration. The most conservative approach is to make all paired tasks non-preemptable; this is essentially what we did in [14]. However, we can allow more flexibility without introducing undue variation in execution costs by permitting preemption only when SMT is not actually in use. Notice that in Fig. 2, \( \tau_1 \) finishes before \( \tau_2 \). We suspect this scenario to be the typical case; it is unlikely that two jobs will finish at the exact same time. Once the first job has been completed, there is no reason that the remaining job cannot be preempted.

To test for schedulability under the rule that paired tasks are preemptable only at certain times, we need a term for the time during which a task is not preemptable.

**Def. 4.** Let the inner cost of paired task \( \tau_{i:j} \), denoted \( C'_{i:j} \), give the maximum amount of time during which a job of the paired task \( \tau_{i:j} \) is non-preemptable. Typically, this is equivalent to the time required for the first of the paired jobs \( \tau_i \) and \( \tau_j \) to complete, although we will discuss other possibilities. For example, in Fig. 2, \( C'_{1:2} \) is 8, assuming the pair can be preempted only if one of the two jobs has completed. If we assume that each job of \( \tau_{i:j} \) is completely non-preemptable, then \( C_{i:j} = C_{i:j} \).
In terms of schedulability testing, a paired task’s inner cost is equivalent to a non-preemptable section. A uniprocessor EDF schedulability test that accounts for non-preemptable sections within otherwise preemptable tasks is given by Liu in [11].

**Def. 5.** Let \( \tau_i \)'s blocking term \( b_i \) be the maximum total time for which a job of task \( \tau_i \) may be prevented from executing by lower-priority jobs. ▶

**Theorem 1.** [11] Scheduling \( \tau \) via EDF on a uniprocessor will result in all deadlines being met if

\[
\sum_{k=1}^{n} u_k + \frac{b_i}{T_i} \leq 1 
\]

holds for all \( \tau_i \in \tau \).

If tasks have been partitioned, Exp. (1) can be applied to partitioned EDF by considering only the tasks in \( \tau^f \) for each core \( \tau_i \).

When we partition \( \tau^R \) onto individual cores, we will make use of the following corollary:

**Corollary 1.** [11] Given task \( \tau_k \), \( b_k \) is equal to the maximum value of \( C'_{i,j} \) for any paired task \( \tau_{i,j} \) on the same core for which \( T_k < T_i \) holds (recall that for \( \tau_i \) and \( \tau_j \) to be paired, \( T_i = T_j \) must hold).

**Preemption points.** If banning preemptions while SMT is in use prevents a task system from being scheduled correctly, we can consider using preemption points. Preemption points are statically inserted into a task’s source code prior to runtime. At runtime, a job that is blocking a higher-priority job will be preempted once a preemption point is reached. The programmer’s challenge in this case is to place preemption points to limit the maximum amount of time for which a job can be non-preemptable, thereby capping \( b_i \) in Theorem 1. This topic has been recently addressed by Baruah and Fisher [1]. A similar principle can be applied to paired tasks. With preemption points in place, it is possible to measure execution times between them, allowing for tasks to be preempted at the selected points without compromising safety.

In our schedulability tests (Sec. IV), we consider the effect of placing preemption points so that maximum \( C'_{i,j} \) values can be guaranteed. We find that in some cases, particularly when a task system contains many periods, their use can improve schedulability dramatically, but in other cases they make little to no difference.

### III. Scheduling Heuristics

In this section, we describe the full process of scheduling a system with SMT. We do so in three steps; each step is detailed in its own subsection. First, we **transform** our starting task system \( \tau \) into a new system, \( \tau^R \), that employs SMT for some tasks. Second, we **partition** the tasks and paired tasks of \( \tau^R \) onto individual computing cores. Third, we **test** each core individually to see if employing EDF on that core will produce a correct schedule. The first two steps are to be done offline, but the scheduling of individual cores is to be done online.

#### A. Transforming the System

In this subsection we show how to transform \( \tau \) into an equivalent system \( \tau^R \) in which some tasks are replaced by paired tasks. “Equivalent” here means that if \( \tau^R \) is scheduled correctly, then \( \tau \) is also scheduled correctly. The idea behind using paired tasks is to decrease the total amount of time needed to correctly schedule both of the two tasks within a pair. Since we require that paired tasks share a period, a paired task \( \tau_{i,j} \) has the same relative deadline as its component tasks \( \tau_i \) and \( \tau_j \). Consequently, if \( \tau_{i,j} \) is scheduled correctly, then \( \tau_i \) and \( \tau_j \) are also scheduled correctly. It follows that \( \tau \) can be correctly scheduled by combining some tasks into pairs and then correctly scheduling all solo tasks and all task pairs, treating each task pair as if it were a single task.

To aid in our explanations, we define a system’s total utilization when task pairs are treated as if they were individual tasks.

**Def. 6.** The transformed utilization \( U^R \) of system \( \tau^R \) is given by

\[
\sum_{\forall i: \tau_i \text{ is a solo task}} u_i + \sum_{\forall i,j: i > j, \tau_i \text{ and } \tau_j \text{ are paired}} \frac{C_{i,j}}{T_i} 
\]

We use \( U^{Rf} \) for the equivalent term when considering only the tasks and task-pairs assigned to a single core \( \pi_f \). ▶

When considering if a portion of \( \tau^R \) is schedulable on a single core, we can safely replace the summation in Exp. (1) with \( U^{Rf} \).

**Which tasks should be paired?** Given the role that total utilization plays in determining schedulability, it is reasonable to define task pairs so as to minimize total paired utilization. We can do so using an ILP with decision variables \( x_{i,j} \) for all tasks \( \tau_i \) and \( \tau_j \) within \( \tau \).

**Def. 7.** For all \( i \) and all \( j \) such that \( T_i = T_j \), let \( x_{i,j} \) equal 1 if \( \tau_i \) and \( \tau_j \) are paired with each other and 0 otherwise. For \( i = j \), let \( x_{i,j} \) equal 1 if \( \tau_i \) is a solo task in \( \tau^R \) and 0 otherwise. Since we do not consider pairing tasks where \( T_i \neq T_j \), we define \( x_{i,j} = 0 \) for those cases. ▶

With Def. 7 in place we can write \( U^R \) as follows:

\[
\sum_{i=1}^{n} \sum_{j=1}^{n} x_{i,j} \frac{C_{i,j}}{T_i} 
\]

Recall from Def. 2 that for solo tasks, we define \( C_{i,j} = C_i \); hence for \( i = j \), \( \frac{C_{i,j}}{T_i} = u_i \).
ILP constraints. In order for \( \tau^R \) to be equivalent to \( \tau \), all tasks within \( \tau \) must be accounted for in \( \tau^R \). To enforce this rule, we require that

\[
\forall i \leq n : \sum_{j=1}^{n} x_{i,j} = 1
\]

holds; essentially, all tasks within \( \tau \) must appear in \( \tau^R \) either as a solo task or as part of a paired task.

Additionally, just as \( \tau \) will be unschedulable if \( C_i > T_i \) holds for any task, \( \tau^R \) will be unschedulable if \( C_{i,j} > T_i \) holds for any paired task \( \tau_{i,j} \). We therefore require that the following holds:

\[
 x_{i,j} \cdot C_{i,j} \leq T_i,
\]

i.e., \( \tau_i \) and \( \tau_j \) may be paired only if \( C_{i,j} \leq T_i \) holds. Since we require that only tasks sharing a period may be paired, we do not need a separate restriction governing the relative values of \( C_{i,j} \) and \( T_j \).

Finally, note that Def. 7 actually defines both \( x_{i,j} \) and \( x_{j,i} \) for each possible task pair. To avoid any inconsistencies, we add the restriction that

\[
\forall i, j : x_{i,j} = x_{j,i}.
\]

We define our ILP as minimizing Exp. (3) subject to Exps. (4) through (6). Despite using integer variables, our ILP executed reasonably quickly in the experiments presented in Sec. IV. We discuss execution times in more detail in Sec. IV.

B. Partitioning the Transformed System

After defining \( \tau^R \), our next step is to partition it onto the individual cores of \( \pi \). Even without considering non-preemptive sections, assigning tasks and task pairs to cores so that all cores are schedulable is a bin-packing problem. While bin-packing is NP-complete in the strong sense, multiple well-studied approximation algorithms for it exist.

We use two of these algorithms—worst-fit decreasing and best-fit decreasing bin-packing—and two algorithm variations of our own, giving us a total of four partitioning algorithms. After assigning tasks to cores, schedulability is tested per Theorem 1 and Corollary 1. In all cases, we assign tasks to cores in non-increasing order of \( C_{i,j} \) and view each core as a single bin with capacity 1.0.

Worst-fit decreasing and best-fit decreasing. In worst-fit bin-packing, each task or paired task is placed on the core that will maximize remaining capacity on the selected core. In best-fit bin packing, each task or paired task is placed on the core that will minimize remaining capacity on the selected core.

Period-aware bin-packing. In our second two algorithms, we modify the worst-fit and best-fit algorithms in an attempt to limit the number of different periods on any one core; note that one consequence of Corollary 1 is that if all tasks on a given core share the same period, then no task is subject to priority-inversion blocking. In this case, the core is schedulable if and only if \( U^R \leq 1 \) holds.

In period-aware worst-fit partitioning, we again attempt to place tasks and task-pairs on cores in non-increasing order of \( C_i \). In this method, we potentially make two attempts to assign each task to a core. In the first attempt, we use worst-fit bin-packing to assign a task or paired task to a core, but we consider only cores on which all previously assigned tasks have the same period as the current task. If a task or paired task is assigned to a core at this point, we move on to the next task or paired task. If the task or pair cannot be placed onto a core using this method, we consider all cores of the platform and assign the task using the standard worst-fit decision process.

Period-aware best-fit partitioning is similar—we first attempt to schedule each task considering only cores without any different periods—but using best-fit rather than worst-fit bin-packing to determine the assignments of tasks to cores.

C. Testing Individual Cores

Our final step is to test each core for schedulability using Theorem 1. To do so, we treat each paired task as if it were a single task. After tasks have been partitioned, the process is no different from uniprocessor scheduling without SMT. While we use EDF in this paper, there is no reason why another uniprocessor scheduling algorithm, such as rate-monotonic (RM) scheduling, cannot be used; the only change needed to use a different per-core schedulability test would be to use a different schedulability test than that of Theorem 1.

IV. EXPERIMENTS

In this section, we present our experimental results. To evaluate our scheduling methods, we conducted a schedulability study in which we created tens of thousands of synthetic tasks across nearly 2,000 scheduling scenarios. For each scenario, we compared the effectiveness of scheduling task systems using our ILP combined with our four bin-packing algorithms—worst-fit, best-fit, worst-fit, and period-aware best-fit—against scheduling the same systems without SMT. In the last case, our baseline, scheduling is attempted using partitioned EDF, with worst-fit decreasing bin-packing as the partitioning algorithm.

A. Experimental Setup

We examined 1,728 scenarios, with each scenario defined by a core count, per-task utilization range, period set, SMT interaction model, and an inner cost model.

The first three factors of our scenario definition require only a brief explanation. The last two are covered in more detail below. We considered core counts of four, eight, and sixteen. Solo per-task utilizations were drawn from
four uniform ranges: (0, 0.4) (low), (0.3, 0.7) (medium), (0.6, 1) (high), or (0, 1) (wide). Periods were drawn from either the set {20, 40, 60, 80, 100} (five periods) or the set {10, 20, 30, 40, 50, 60, 70, 80, 90, 100} (ten periods).

Each task was created by selecting a utilization from the appropriate distribution and a period from the appropriate set, with all periods within a set having equal probability. Periods and utilizations were selected independently. A solo task execution cost was then assigned as a function of utilization and period. For each scenario, we determined schedulability ratios (i.e., the percentage of schedulable task sets) for task systems ranging in total utilization from $\frac{m}{2}$ to $2m$—recall that $m$ denotes the core count—using each of our four bin-packing algorithms to partition $\tau^R$ onto separate cores after having transformed $\tau$ into $\tau^R$ using our ILP. We compared these results to a baseline of table-based scheduling and may require prohibitively large amounts of time to compute a scheduling table.

**Double cost.** In this model, we defined $C'_{i,j}$ as the time during which both jobs are executing and then assumed that $C'_{i,j} = \min(C_i, 2 \cdot C_j)$, with $\tau_j$ being the task with the shorter solo cost. This model is still quite conservative; we found our data from [14] shows only one paired task in which $C'_{i,j} > 2 \cdot C_j$ held after excluding tasks for which $C_i$ and $C_j$ differ by more than a factor of 10. Other works [2, 3, 10, 15] have also found that it is rare for execution times to double in the presence of SMT.

**Data driven.** In this model, we used the same definition of $C'_{i,j}$ as in the double cost model, but rather than assuming that $C'_{i,j} = \min(C_i, 2 \cdot C_j)$, we based our $C'_{i,j}$ values on how much time was actually required in [14] for the faster-executing job of each pair to finish, with minimal added conservatism. Again excluding cases where $C_i$ and $C_j$ differed by more than a factor of 10, we found that in the majority of cases, the execution time of $\tau_j$ alone within the pair $\tau_{i,j}$—i.e., $C'_{i,j}$ under this model—ranged from slightly greater than $C_j$ to $1.8 \cdot C_j$. There was one outlier for which we had $C'_{i,j} \approx 10 \cdot C_j$. This data is shown graphically in Fig. 4.

\[3\] This value occurred with a benchmark, petrinet, that is extremely short and was often difficult to measure.
This step allows for the possibility of C\textsuperscript{\textprime}\textsubscript{i,j} being as great as the maximum relative to C\textsubscript{j}, 10 \cdot C\textsubscript{j}, that we observed in practice. 2% overstates our observed frequency of this occurrence; our outlier was a single sample out of 84 possibilities. Apart from that possibility, we set each C\textsuperscript{\textprime}\textsubscript{i,j} value as a uniform random variable in the range [1.1 \cdot C\textsubscript{j}, \min(1.8 \cdot C\textsubscript{j}, C\textsubscript{i,j})]. Some pessimism persists in this model, as our observed C\textsuperscript{\textprime}\textsubscript{i,j} values can be seen in Fig. 4 to skew towards the lower end of that range and we do not allow the possibility in our model of C\textsuperscript{\textprime}\textsubscript{i,j} < 1.1 \cdot C\textsubscript{j} holding despite having observed that possibility in practice.

It is noteworthy that in some of our collected data, C\textsuperscript{\textprime}\textsubscript{i,j} < C\textsubscript{j} holds, meaning that in some cases, a job requires less time to complete with SMT than without. At present, we do not have a good explanation for this behavior, and so we pessimistically exclude it from our modeling. We intend to investigate this phenomenon in future work.

Preemption points. In this model, we attempted to capture the effects of allowing preemption points within a task system. In our preemption points model, we assumed that any code we execute has preemption points inserted so that no job will be non-preemptable for more than 10 time units. In this case, we defined C\textsuperscript{\textprime}\textsubscript{i,j} as the minimum of 10 and what it would have been under the data driven model. We chose 10 with the idea that if each time unit corresponds to one millisecond, placing preemption points to limit non-preemptable sections to 10 ms should be achievable without causing exceptionally high overheads.

Full preemption. Here we assumed that all tasks are fully preemptable, even when there are paired jobs running at the same time. In practice, allowing unrestricted preemptions along with SMT would tend to make the already difficult timing-analysis problem discussed in [14] even harder, possibly making it impossible to guarantee a safe timing analysis for hard-real tasks. However, testing this approach allowed us to see the cost of limiting preemptions. In addition, this approach may be viable for soft real-time and non-safety-critical systems, where some additional uncertainty in timing analysis may be tolerable.

B. Schedulability Results

To determine whether a task system was schedulable, we attempted to transform\textsuperscript{4} and partition each system created so that the resulting sub-systems were all schedulable on their assigned cores per Theorem 1.

For each scenario considered, we summarize our results in a graph that shows the schedulability ratio of systems ranging in total utilization from \( \frac{3m}{T} \) to \( 2m \), with each point on the graph corresponding to approximately 100 systems.\textsuperscript{5} For each scenario, we show only the partitioning algorithm that produced the best results. Since the partitioning algorithms all execute quickly, it is entirely practical to run all four for each task system and then choose the best result.

We use two metrics to summarize the proportion of systems that are schedulable under each scenario. We define relative schedulable area (RSA) as the area under the schedulability curve divided by the core count \( m \) for each scenario. In calculating RSAs, we assumed that the schedulability ratio is constant between total utilization 0.0 and \( \frac{m}{T} \), which is the smallest utilization we tested in each scenario. This assumption results in RSAs being somewhat understated in the lowest-performing scenarios. An ideal (e.g., fluid) scheduler, not using SMT, that can preempt and migrate jobs arbitrarily would have an RSA of 1.0; it could schedule all task systems with total utilization at most \( m \) and no task systems with greater utilization.

In addition to RSA, we define a scenario’s partitioned improvement (PI) as the RSA for a given scenario and partitioning algorithm divided by the RSA for that same scenario using our baseline scheduling algorithm. We use PI to show the benefit of our methods in cases where partitioned scheduling without SMT falls well short of an ideal scheduler to begin with; for example, the scenario shown in Fig. 13 has an RSA of 0.93 Based on that statistic alone, one might include that SMT is not effective in this case. However, the scenario has a PI of 1.11 showing an improvement over partitioned scheduling without SMT.

Our full set of graphs is included in an online appendix [13]. Here, we show the graphs that give the best, worst, and median results for both RSA and PI when using the full-preemption and data-driven models. These graphs demonstrate several trends we saw in our results.

\textsuperscript{4}We used Gurobi Optimizer, a commercial optimization programming solver with free academic licensing, to execute the required ILP.

\textsuperscript{5}While we calculated schedulability for utilizations in the range \([\frac{m}{T}, 2m]\), our graphs only show results in the range \([\frac{3m}{T}, 2m]\); in the majority of cases, we found that all systems with utilization less than \( \frac{3m}{T} \) could be scheduled.
Fig. 5: The best RSA and PI in the full preemptions model and the best RSA in the data driven model.

Fig. 6: The best PI in the data driven model.

In some cases, we found that two or more different inner cost models produced near-identical results. Graphically, this result produced graphs that were difficult to read. To avoid this problem, we do not print lines for inner cost models that had the same RSA within two significant digits as the data-driven model.

**Obs. 1.** With the full preemption model, applying SMT always gave an improvement compared to partitioned EDF, and, in the best cases, nearly doubled schedulable utilization. RSAs ranged from a high of 1.91 (Fig. 5) to a low of 0.83 (Fig. 7) and PIs from a high of 1.93 (Fig. 5) to a low of 1.01 (Fig. 8).

**Obs. 2.** With the data driven model, applying SMT improved schedulability in more than half of all scenarios, as shown by the median PI of 1.11 (Fig. 13). In the best data driven case, RSA equaled 1.66 (Fig. 5) and PI 1.84 (Fig. 6).

**Obs. 3.** Applying SMT did not always improve schedulability when using the data-driven model, as shown in Fig. 9. The worst results for models other than full preemption typically occurred when the period count was greater than the core count, as seen in Fig. 9. In these cases, the problem is that the ILP creates a task system that can be scheduled only by allowing more task preemption than we permit; recall that the ILP does not consider inner costs, and that its only restriction on per-task costs is that $C_{i1} \leq T_i$ must hold.

**Obs. 4.** The impact of which preemption model is used on overall schedulability varies greatly. In some scenarios, such as that of Fig. 8, it makes essentially no difference, whereas in others, such as Fig. 9, the difference is dramatic. This result suggests that in some cases, further work on means to allow more preemptions of SMT-enabled tasks
Utilization
0.0
0.2
0.4
0.6
0.8
1.0
Schedulability Ratio

8 Cores, Task Util~U(0, 1), 10 Periods

Mult. Thread Score~N(0.45, 0.06), split=0

Baseline (RSA=0.92)
No Pre. (RSA=0.93, PI=1.16)
Data Driven (RSA=1.05, PI=1.16)
Pre. Points (RSA=1.09, PI=1.19)
Full Pre. (RSA=1.30, PI=1.41)

TABLE I: Period-aware advantages (Def. 9)

<table>
<thead>
<tr>
<th>Inner cost model</th>
<th>Period-aware advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Preemption</td>
<td>0.75</td>
</tr>
<tr>
<td>Double Cost</td>
<td>0.75</td>
</tr>
<tr>
<td>Data Driven</td>
<td>0.74</td>
</tr>
<tr>
<td>Preemption Points</td>
<td>0.21</td>
</tr>
<tr>
<td>Full Preemption</td>
<td>0.01</td>
</tr>
</tbody>
</table>

would be time well spent.

Obs. 5. The period-aware algorithms provided a large advantage under the no-preemption, double-cost, and data-driven preemption models, but were less advantageous using the preemption points and full preemption models. We summarize our findings on the period-aware advantage (defined below) of each model in Table 1.

Def. 9. For each inner cost model, we define its period-aware advantage as the proportion of scenarios in which at least one of the period-aware partitioning algorithms gave a strictly greater RSA than both the best-fit and worst-fit algorithms.

Obs. 6. In no case did our ILP require more than 37 seconds to execute, and only one required more than 30 seconds. The median time required was 2.54 seconds. No four-core system required more than 9 seconds, and no eight-core system required more than 24 seconds. In contrast, ILP execution times of 60 seconds were frequently insufficient in our previous work [14], even on systems of only four cores.

Execution times are summarized in Fig. 14. For each scenario, we recorded only the maximum time required by any ILP, meaning that our discussion here overstates the typical execution time needed. Note that since we considered preemption only after partitioning tasks, each ILP provided data for all five of our preemption models. In total we recorded 432 execution times. Our schedulability tests were performed on a research cluster consisting of 2.5 and 2.3 GHz cores, with tests for many scenarios running in parallel. We suspect that individual ILPs ran significantly slower than they would have had our experiments not run in parallel.

As to whether our ILP execution-time requirements are practical, less than 1 minute is certainly reasonable for an offline step, since that will only be done once per system. Our shortest times—the fastest 5% of our ILPs required less than 100 ms to run—could even be practical to run online as part of a task system allowing dynamic task entry and exit.

V. CONCLUSION

Within the context of our schedulability study, we found that when allowing tasks to be preemptable, schedulability was increased by a factor of 1.5 or more in 31% of tested scenarios. The same improvement was seen in 13% of
scenarios using either data driven inner costs or allowing no preemption at all. Furthermore, we saw schedulability improvements of 1.8 or more in 11% of scenarios that allowed full preemption and in 2% of scenarios using either the data driven or no preemption models.

In future work, we plan to investigate the effects of allowing preemption while SMT is active; by doing so, we hope to enable results that are close to those of our somewhat idealized full-preemption model. If we find that preemptions have a significant detrimental effect on SMT-enabled execution, we will need to rely more on the ability of our period-aware partitioning algorithms to obviate the need for preemptions; in practice, though we do not model it here, increasing preemptions in a system may come at a cost of reduced schedulability. If that cost can be avoided, so much the better.

The process we have given here may also be suitable for dynamic systems, in which tasks can enter and leave a system during run-time. Given that our transformation step and partitioning algorithms both execute quickly, it may be possible to execute them periodically as scheduled jobs in a live system, with the goal of rebalancing a system whose task mix has changed. For this to be practical, we would need to be able to guarantee run-times for the currently offline portions of our algorithm. In addition, we intend to integrate our work on SMT into a mixed-criticality context.

REFERENCES


