Simultaneous Multithreading and Hard Real Time: Can It Be Safe?

Sims Hill Osborne
University of North Carolina, Chapel Hill, NC, USA
http://www.cs.unc.edu/~shosborn/
shosborn@cs.unc.edu

James H. Anderson
University of North Carolina, Chapel Hill, NC, USA
http://jamesanderson.web.unc.edu/
anderson@cs.unc.edu

Abstract

The applicability of Simultaneous Multithreading (SMT) to real-time systems has been hampered by the difficulty of obtaining reliable execution costs in an SMT-enabled system. This problem is addressed by introducing a scheduling framework, called CERT-MT, that combines scheduling-aware timing analysis with a cyclic-executive scheduler in a way that minimizes SMT-related timing variations. The proposed scheduling-aware timing analysis is based on maximum observed execution times and accounts for the uncertainty inherent in measurement-based timing analysis. The timing analysis is found to work for tasks with and without SMT, though some adjustments are required in the former case. A large-scale schedulability study is presented that shows CERT-MT can schedule systems with total utilizations approaching 1.4 times the core count, without sacrificing safety.

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

Keywords and phrases real-time systems, simultaneous multithreading, hard real-time, scheduling algorithms, probability, statistics, timing analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.14

Related Version Longer version with all graphs, code, and data available at http://jamesanderson.web.unc.edu/papers/

Supplementary Material ECRTS 2020 Artifact Evaluation approved artifact available at https://doi.org/10.4230/DARTS.6.1.1.

Funding Work was supported by NSF grants CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, ONR grant N00014-20-1-2698, and funding from General Motors.

Acknowledgements We thank Prof. Alex Mills (Zicklin School of Business, Baruch College, New York, NY) for reviewing an early draft and offering suggestions, particularly with regards to Sec. 4. We thank Joshua Bakita (UNC-Chapel Hill) for contributions to the code used to execute our benchmark tests. Finally, we thank our anonymous reviewers and Shepherd for the improvements they suggested to the final version of this paper.

1 Introduction

Simultaneous Multithreading (SMT) is a technology that allows a single physical computing core to act as two or more logical cores, or hardware *threads*. Ideally, enabling SMT allows multiple jobs to execute in parallel on a single core in significantly less time than would be required for the same jobs to execute sequentially. If this ideal can be achieved, the potential benefits are clear. Users in industry are eager to make more 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 [55].

Unfortunately, with SMT enabled, a task's execution time is dependent on code that executes on other thread(s) of the same core. This fact makes it difficult to obtain worst-case execution times (WCETs) with SMT enabled, to the point that the real-time community has largely given up on SMT for hard real-time systems. However, multicore systems face problems with interdependence between scheduling and execution times even without SMT. Such problems make static timing analysis on multicore platforms extremely difficult, which has led to interest in Measurement Based Probabilistic Timing Analysis (MBPTA), a family of methods that determine probabilistic worst-case execution times (pWCETs) by sampling observed execution times [17, 20].

Considered problem. We claim the following: *if a hard real-time task system τ can be guaranteed a given level of safety on a modern multicore architecture via MBPTA, enabling SMT can, in many cases, allow τ to run on fewer cores without sacrificing safety.* We provide a formal definition of safety in Sec. 2. We justify this claim by providing and examining CERT-MT (**C**ontrolled **E**xecution of **R**eal **T**ime with **M**ulti-**T**hreading), a scheduling framework based on a cyclic-executive scheduler and MBPTA timing analysis.

The CERT-MT framework. CERT-MT is a framework that includes both scheduling analysis and timing analysis designed to minimize uncertainties resulting from SMT. The overall structure of CERT-MT is illustrated in Fig. 1. The timing analysis of CERT-MT is based on MBPTA. We add to the MBPTA family of techniques by analyzing the probabilistic and statistical implications of basing timing analysis on worst observed execution times. As part of this analysis, we point out a potential source of error: the measurements used in MBPTA to build estimates may themselves be subject to stochastic variation.

Historically, there has been a separation of concerns in the real-time community between timing analysis and schedulability analysis. There are good reasons for this separation: both topics are complex on their own, making it difficult to address both within a single work. However, maintaining this separation comes at the cost of imprecise timing analysis that can be made safe only by resorting to more pessimism than needed. To avoid both unsafe conditions and excessive pessimism, CERT-MT is built around two key rules: first, conditions when testing tasks with MBPTA should match runtime conditions as closely as possible; second, timing analysis and scheduling decisions should be considered at the same time, since scheduling can affect task execution times. These are well-known, standard principles of measurement-based analysis; we consider scheduling and timing analysis simultaneously in deference to these principles.

Contribution and organization. We give an MBPTA method that is straightforward to use and, given basic assumptions about the underlying data, provides mathematically guaranteed results. Using this method, we analyze the execution times of tasks employing SMT and thereby show that using SMT can bring substantial benefits to hard real-time systems without sacrificing safety.

More specifically, in Sec. 3 we define the CERT-MT scheduler, which is designed to minimize variations of job execution times caused by SMT, and in Sec. 4, we give our new MBPTA method, which upper-bounds the likelihood of an observed maximum execution time being exceeded. This method, which is also applicable to systems without SMT, breaks new ground by accounting for the fact that if measurements contain any random elements, then any measurement-based estimate is itself a random variable. The theoretical work of Secs. 3 and 4 is backed by empirical results.

In Sec. 5, we present our experiments. First, in Sec. 5.1, we show that our timing analysis works very well in practice, even when certain assumptions underlying our analysis do not hold. Second, in Sec. 5.2, we discuss how enabling SMT affects the timing requirements of individual tasks. Finally, in Sec. 5.3, we give the results of a large-scale schedulability study built around the understanding of SMT behavior we have developed. In this study, CERT-MT enabled systems with total utilizations approaching 140% of the core count to be scheduled with no loss of safety. In Sec. 6, we conclude and discuss future research directions.

We add to existing work on MBPTA, but there are many elements within MBPTA we do not address. In order to focus on the role of SMT, we do not consider the role of interference from tasks on other cores, nor do we address the complex question of how to determine appropriate task inputs for timing analysis. For our benchmark tasks, we used the built-in inputs and proceeded to work under the assumption that they are appropriate. Finally, we do not consider how the probability of individual jobs exceeding stated costs affects the probability of larger system failures. We plan to address these topics in future work.

2 Background

In this section, we provide some basic background information about SMT, define our task and platform models, and give a brief overview of concepts related to MBPTA and pWCETs.

SMT basics. Modern superscalar cores execute multiple instructions per cycle, using instruction-level parallelism within jobs to decrease execution time. SMT extends this behavior by allowing multiple jobs to execute instructions within a single cycle. An overview is given in Ex. 1 below. More details can be found in Eggers et al. [22].

▶ **Example 1.** Fig. 2 shows the effects of enabling SMT. At the top of Fig. 2, jobs of tasks τ_1 and τ_2 execute sequentially without SMT on a core that can accept two instructions per cycle. When fewer than two instructions are ready, as in cycles 2 and 3, execution resources are wasted. τ_1 finishes in cycle 6 and τ_2 in cycle 12.

In the middle of the figure, the same jobs execute in parallel with SMT enabled, reducing the number of lost cycles. Both tasks finish in cycle 8. In this case, SMT has the effect of delaying the completion of the darker-colored τ_1, but speeding up the completion of the lighter τ_2, since it does not have to wait for τ_1 to complete before beginning its own execution.

The bottom shows SMT execution with the start of τ_2 delayed; this delay actually causes τ_1 to require more time to execute due to different interactions between the two tasks. For a detailed discussion of factors that can affect SMT execution, see Bulpin [10, 11].

Hardware platform. Our hardware platform π consists of m identical cores, each supporting two hardware threads. Different threads on the same core are referred to as *sibling threads*. Jobs scheduled on sibling threads are *sibling jobs*. Sibling jobs are said to be *co-scheduled*. It is not possible to adjust the priority of sibling jobs relative to one another; in particular, switching which job executes on which sibling thread has no effect. These conditions match those of Hyperthreading, Intel's SMT implementation [25]. Jobs that do not use SMT are referred to as *solo jobs*.

Task model. We consider a hard real-time task system τ that consists of n independent synchronous periodic, implicit-deadline tasks. More specifically, we assume there are no inter-task precedence constraints, critical sections, or other synchronization requirements. Each

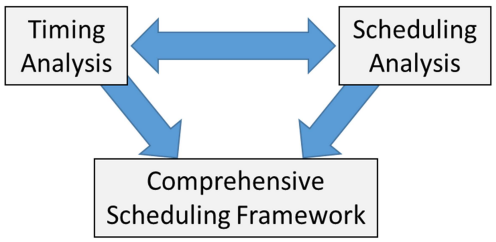

Figure 1 The CERT-MT framework.

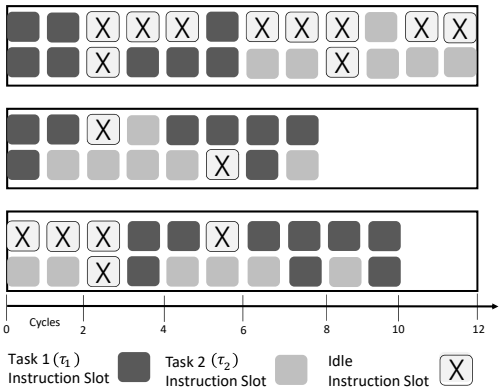

Figure 2 Top: task execution without SMT. Middle: execution with SMT. Bottom: execution with SMT, different starting times.

task $\tau_i = (C_i, T_i)$ is defined by its execution *cost,* C_i, and its *period,* T_i. Time is continuous. The *utilization* of τ_i is given by $u_i = \frac{C_i}{T_i}$. Every task releases an unlimited number of *jobs*, with the a^{th} job released by τ_i denoted by $\tau_{i.a}$. Jobs of τ_i are released every T_i units of time and have a relative deadline of T_i; the a^{th} job is released at time $r_{i.a} = T_i(a-1)$ and has an absolute deadline at time $d_{i.a} = T_i \cdot a$. The system is scheduled correctly if it can be shown that no job will ever miss a deadline if all task execution costs are upper-bounded by C_i.

Typically, C_i is said to give a task τ_i's WCET. Our C_i values are determined based on pWCETs, or probabilistic WCETs. We give a more precise definition of C_i below, after establishing some necessary statistical concepts.

We require task periods to be harmonic, i.e., every period must be an integer multiple of every smaller period. The least common multiple of all periods is referred to as the *hyperperiod* and denoted H. We add to this task model random variables that correspond to the execution times of individual jobs:

▶ **Definition 2.** *E_i is a random variable corresponding to the execution time of a randomly selected job of τ_i.*

pWCETs and system safety. There are two problems with modeling a task based on its WCET. First, determining the true WCET may be prohibitively difficult, especially on a multicore machine. Second, if the WCET can be determined, it may be a very rare event that a job actually requires that many units of execution time, leading to very pessimistic scheduling if it is assumed that every job of τ_i will require its WCET; depending on the application, this pessimism may be unnecessary. As an alternative to WCETs, we could consider pWCETs instead.

▶ **Definition 3.** *τ_i has the* pWCET C_i^p *if and only if execution times for τ_i follow a probability distribution*[1] *such that*

$$\Pr(E_i \leq C_i^p) = p \tag{1}$$

holds. We refer to p as the provisioning level *of τ_i.*

[1] In some sources, the distribution itself is referred to as the pWCET

However, finding a task's pWCET has its own problems. Validating that C_i^p is a pWCET for a task τ_i requires a detailed understanding of the task's behavior in many possible execution states and of the likelihoods of those states occurring. It can be argued that determining a pWCET is no more practical than determining a WCET. Furthermore, it is potentially dangerous to determine a pWCET based only on observed run-times. To understand why, consider a simpler problem: rolling a die. Finding exact probabilities of a given die roll result, which is analogous to a pWCET, requires detailed knowledge of the die's physical properties, which is analogous to perfect knowledge of a task's behavior. Lacking detailed knowledge of a die or task, we can make predictions based on observed past rolls or execution times. This method is practical, but does not produce a pWCET for the task.

▶ **Example 4.** We wish to know Pr(an arbitrary future roll ≤ 5) for a given die, but we cannot examine the die directly. We only know the results of rolling the die in the past. Suppose that out of ten roles, the maximum was 5. Any of the following are possible: the die has no faces greater than 5, the die has faces greater than 5 but these are unlikely to be rolled, or the die has faces greater than 5, which are reasonably likely to be rolled, but our set of ten rolls was not representative of the die's true long term behavior. To illustrate the last case, rolling a fair six-sided die ten times gives

$$\Pr(\text{ten rolls} \leq 5 \mid \text{a fair six-sided die}) = \left(\frac{5}{6}\right)^{10} \approx 0.162.$$

Consequently, a naive interpretation of observed results, such as the rule that getting zero 6s out of ten rolls means a 6 is less likely than other roles, will frequently be wrong.

Any estimate we can collect from measured random data – such as the observed maximum given ten rolls – is a random variable, but to make probabilistic statements of the form Pr(arbitrary future roll or execution time $\leq Y$) where Y is a constant rather than a random variable, we need more information than can be obtained by observing results. For a die, this would mean physically examining it. For a task, we would need static analysis, which is impractical for tasks utilizing SMT or in multicore systems. Since static timing analysis is impractical, we are left with measurement-based methods. With MBPTA, each trial is used to gather data, similarly to each roll of a die. A set of execution times is called a *trace*.

▶ **Definition 5.** *A trace is an ordered set of execution times for a task and constitutes a finite-sized sample from the population of all possible execution times for that task. For trace R with $|R|$ elements, let $\{R_1, R_2, ...R_{|R|}\}$ be the ordered sample times within the trace. Let the maximum of the trace be denoted by R_{max}. Note that the R_k values refer to times within a finite-sized sample, while E_i is from the entire population of execution times.*

The trace is analyzed to produce an estimate of future behavior. How to do so safely lies at the heart of MBPTA analysis. EVT methods attempt to use traces that are representative of possible task behaviors to produce an upper bound on C_i^p, typically much larger than the observed maximum execution time, for a given value of p. By doing so, practitioners can avoid many hazards associated with estimates based on purely random data. Unfortunately, how to obtain data that is sufficiently representative is very much an open question; there is no universal agreement on how to do so [17, 20, 36]. Consequently, it is not always possible to correctly apply EVT.

We use a different approach. We analyze a trace to produce a *safe WCET*, which accounts for both randomness in future behavior and randomness involved in creating the estimate.

▶ **Definition 6.** *Given a task τ_i, a* safe WCET S_i^q *is a random variable such that*

$$\Pr(E_i \leq S_i^q) = q \tag{2}$$

holds given trace R of τ_i, where R constitutes a random sampling of τ_i's possible execution times. We refer to q as the safety level[2] *of τ_i.*

The difference between Defs. 3 and 6 is that, while C_i^p in Def. 3 is a constant, S_i^q is a random variable. In practice, the execution times forming R may not *actually* be random, but we analyze R *as if* they were. We formally state our assumptions regarding R in Sec. 4. Later, in Sec. 5, we show empirically that our analysis holds even when applied to execution traces for which our assumptions of randomness may not. This approach of analyzing non-random data as if it were random may seem strange, but it is not original to us. In particular, pseudo-random number generators (pRNGs) actually produce values that are purely deterministic, but are then used to produce "randomess" that powers a vast variety of applications across many domains. Our traces do not exhibit the same degree of random-like behavior that is expected of widely-used pRNGs, but our experiments in Sec. 5 show the power of assuming randomness to gain insight into task behaviors.

Costs and correctness revisited. We define the execution-cost parameter C_i in our model so that $C_i = S_i^q$ holds for a specified q. The traditional notion of correctness – all jobs are guaranteed to complete on time – cannot be adhered to without known upper bounds on all job execution times. For this reason, we supplement the binary idea of correctness with a quantifiable level of *safety*.

▶ **Definition 7.** *Task system τ is q-safe if all tasks have sWCETs with safety level at least q, and the system would be correctly scheduled if all tasks had true WCETs no greater than the stated sWCETs.*

Some may question the wisdom of using sWCETs within a safety-critical context. However, probabilistic reasoning is already present within hard real-time contexts. In particular, FAA standards for commercial aircraft state acceptable failure rates, essentially giving a probabilistic bound. *We are not weakening hard real-time correctness; we are making explicit a dependence on timing analysis that is often left implicit.*

Determining an appropriate safety level is an application-specific decision. Our concept of safety is applicable to any system designed around an acceptable failure rate, which could include applications ranging from streaming media to aviation. We highlight aviation rules to show that probabilistic timing analysis does not preclude systems that have both hard real-time and safety-critical requirements.

Related work. The foundations of SMT were given by Tulsen et al. [67] and Eggers et al. [22] in the mid 1990s. SMT first became widely available in 2002, when it was made available on Intel processors [50], prompting works that gave detailed descriptions of SMT on Intel hardware [10, 11, 34, 64, 66], albeit outside of a real-time context.

Jain et al. [35] did early work on applying SMT to real time; they showed experimentally that significant performance gains in soft real-time systems are possible. Kato et al. [37, 38] gave algorithms for determining which tasks should share a core to maximize efficiency, provided that it is possible to statically determine execution costs of SMT-enabled tasks.

[2] Note that q is unrelated to safety integrity levels used in risk analysis, despite the similar name.

Cazorla et al. [16] and Gomes et al. [26, 27] have proposed ways to exert greater control over SMT timings by modifying the interaction between the operating system and the hardware, while Zimmer et al. [73] and Suito et al. [65] have proposed purpose-built hardware aimed towards real-time work with multithreading. Lo et al. [48] gave methods to limit real-time work to a small number of threads, executing non-real-time work only when safe to do so. Mische et al. [53] suggested using SMT to hide context-switch costs by using threads to switch task state in and out in the background. More recently, Osborne et al. [56, 58] have benchmarked SMT performance and given schedulability results within a soft real-time context based on worst-case observed execution times. Detailed analysis of Intel's microarchitecture, including the resource constraints that are relevant with SMT, have been performed by Fog [25]. Scheduling with SMT is related to the more general case of scheduling in parallel any set of tasks that may influence one another's execution times, as discussed by Andersson et al. [4].

Cyclic-executive (CE) scheduling was developed to combine high predictability with low run-time overheads [5, 47]. The essence of the method is to pre-compute a table stating when every job should execute. Recent work on CE scheduling for multicore platforms has focused on mixed-criticalty scheduling [14, 12] and more efficient schedule computation [21].

The difficulties with guaranteeing execution costs in a multicore context arise from the possibility of tasks on different cores interfering with one another. An overview of these problems, and possible solutions, is given by Kim in his dissertation [40]. More specific concerns include cache conflicts [2, 9, 19, 41, 49, 69, 70], DRAM conflicts [29, 30, 68, 71, 72], memory bus conflicts [54, 63], general OS support [3, 18, 31], and I/O conflicts [39, 59].

MBPTA was first proposed by Burns and Edgar in 2000 [13], in part as a means to address the difficulty of timing analysis on complex processors; concerns over the practicality of static timing analysis predate multicore computing. Traditionally, EVT has required that execution times be identically and independently distributed [6, 24, 60]. This requirement has been an obstacle to the use of EVT, since observed execution times will often include statistical dependencies. One way real-time practitioners have worked around this limitation by has been by requiring that executions be performed with randomized caches [1, 7, 8, 32, 42, 43, 51, 52]. Others have explored ways to introduce randomness to execution traces after measurement is complete [45, 46]. Alternatively, the works of Leadbetter et al. [44] and Hsing [33] imply that variables need not be identically distributed so long as they are *stationary* – a data set is stationary if it maintains the same distribution over time – and either all values are independently distributed over time or extreme values are independent with respect to time (extremal independence). This fact has been used by Guet, Santinelli, and various co-authors in [28, 61, 62] and has been noted by Carzola et al. and Davis and Cucu-Grosjean in recent survey papers [17, 20].

3 The CERT-MT Scheduler

In this section, we describe the CERT-MT scheduler, which is a CE scheduler that allows for SMT. We do so in two parts. First, we describe the restrictions we place on jobs using SMT that will make our timing analysis possible. Second, we discuss the correctness conditions of CE schedulers and outline the construction of a mathematical optimization program that will produce a correct schedule for SMT-enabled tasks that adheres to our rules if one exists. We will return to the topic of timing analysis in Sec. 4.

3.1 Rules for Using SMT

Again, the fundamental rule of CERT-MT is that *jobs employing SMT must execute as they are tested in determining execution times.*[3] To enforce this rule, we implement the following two sub-rules:

1. All jobs employing SMT must be *simultaneously co-scheduled,* as defined in Def. 8 below.
2. All jobs employing SMT must be executed non-preemptively. We include scheduler interruptions as preemptions.

▶ **Definition 8.** *We say that two jobs are* simultaneously co-scheduled *if both begin execution simultaneously on sibling threads of the same core, and when one job completes, the remaining job continues on the same core with no sibling job until complete. We use $\tau_{i.a:j.b}$ to denote the simultaneously co-scheduled jobs $\tau_{i.a}$ and $\tau_{j.b}$ and $\tau_{i:j}$ to denote non-specific simultaneously co-scheduled jobs of τ_i and τ_j.*

With these sub-rules in place, we can enforce our primary rule – execution as tested – without creating an excessive timing-analysis burden. The first sub-rule allows us to limit our testing to simultaneously co-scheduled jobs, as opposed to considering, for example, what happens if τ_i begins executing as a solo job and τ_j begins later on the same core, similar to what we saw in the final case of Ex. 1 and Fig. 2. We thereby ensure that our timing analysis accounts for hardware resource sharing between jobs on the same core. From here onwards, we use the term "co-scheduled" to mean "simultaneously co-scheduled" since that is the only scheme we consider. Banning preemptions for SMT-enabled jobs is a precautionary step; while there exists substantial work on accounting for preemptions without SMT, no such work exists for SMT. In its absence, we choose to use a conservative approach. We allow for preemptions when not using SMT.

Disadvantages of non-preemptive scheduling. While requiring non-preemptive execution simplifies our timing analysis, it can make otherwise schedulable systems unschedulable, as shown in Ex. 9 below. The example does not explicitly reference SMT or co-scheduled jobs; we explain later how co-scheduled jobs can be treated as a single schedulable unit.

In a CE schedule, jobs are assigned to execute within specific time intervals called *frames* [5]. A correct schedule is built by assigning jobs to frames. We give a precise definition of frames – our definition elaborates on the standard one – in Def. 10 below, following Ex. 9.

▶ **Example 9.** Let τ be a task system with periods in the set $\{10, 20\}$, with some jobs having $T_i = 20$ and $C_i > 10$. Consider the frame size f needed to execute τ non-preemptively. If $f > 10$ holds, then no job with $T_i = 10$ can be scheduled, since the first frame will not complete until after the deadline of the first job, so such a job could miss its deadline. If $f \leq 10$ holds, then no job with $C_i > 10$ can be scheduled, since the frame boundary causes a preemption by the scheduler.

To facilitate the scheduling of non-preemptive jobs, we allow the frame size to be defined per-core, rather than requiring one frame size for the entire platform. We replace the notation f for a platform-wide frame size – the standard in CE scheduling – with $f(\ell)$.

[3] Recall that we consider only independent tasks; we defer more complex systems to future work.

▶ **Definition 10.** *A* frame *is a time interval of length $f(\ell)$ on core ℓ. Frames are indexed using g, starting from $g = 1$, such that the g^{th} frame on core ℓ starts at time $(g-1) \cdot f(\ell)$ and ends at time $g \cdot f(\ell)$. We require that no frame extends beyond the hyperperiod boundary,*[4] *allowing the schedule to repeat every hyperperiod. The last frame per hyperperiod on core ℓ has index $g = \left\lfloor \frac{H}{f(\ell)} \right\rfloor$.*

As a consequence of allowing multiple frame sizes, frames with the same indices may cover different lengths of time on different cores. For example, suppose that on a two-core system, $f(1) = 10$ and $f(2) = 20$. Frames 1, 2, and 3 of core 1 would start at times 0, 10, and 20, but frames 1, 2, and 3 of core 2 would start at times 0, 20, and 40. In addition, the maximum valid frame index varies based on $f(\ell)$; since we only need to define a schedule for the first hyperperiod, we do not consider frames for which $g \cdot f(\ell) > H$ holds.

Scheduling co-scheduled jobs. We define three scheduling parameters for co-scheduled job pairs: *joint cost*, *joint release*, and *joint deadline*. These parameters allow us to treat pairs of jobs as schedulable entities.

▶ **Definition 11.** *The* joint cost *to simultaneously execute jobs of τ_i and τ_j is given by $C_{i:j}$, defined as the execution time for both jobs assuming they begin simultaneously. If $i = j$, then $C_{i:j} = C_i$, indicating solo execution for τ_i. Jobs with nothing co-scheduled are* solo jobs*.*

As with single-task costs, determining $C_{i:j}$ such that both jobs are absolutely guaranteed to complete is impractical. For this reason, we define $C_{i:j} = S^q_{i:j}$ for a specified value of q, with $S^q_{i:j}$ analogous to S^q_i (Def. 6).

▶ **Definition 12.** *Let $E_{i:j}$ be a random variable corresponding to the time required to simultaneously co-schedule a pair of randomly selected jobs of τ_i and τ_j. We define the* joint safe WCET *(jsWCET) $S^q_{i:j}$ of $\tau_{i:j}$ to be a function of a trace R of $\tau_{i:j}$ such that*

$$\Pr(E_{i:j} \leq S^q_{i:j}) = q \tag{3}$$

holds. As with solo tasks, we assume R is a random sample, making $S^q_{i:j}$ a random variable.

Our definition of safety (Def. 7) can be easily expanded to accommodate jsWCET values. Along with defining a joint cost for each pair, we define a *joint release* and *joint deadline*.

▶ **Definition 13.** *Given $\tau_{i.a:j.b}$, the* joint release *and* joint deadline *are given, respectively, by*

$r(i.a, j.b) = \max(T_i \cdot (a-1), T_j \cdot (b-1))$ *and*
$d(i.a, j.b) = \min(T_i \cdot a, T_j \cdot b)$.

For the case where $i = j$ and $a = b$ – i.e., a solo job – the r and d terms are simply the job's release time and absolute deadline.

In order for both jobs of a pair to finish on time, the pair must begin no sooner than $r(i.a, j.b)$ – both jobs must have been released – and must finish no later than $d(i.a, j.b)$ – prior to either jobs' deadline.

[4] This requirement weakens the typical CE requirement that $f(\ell)$ divide H while still allowing the schedule over each hyperperiod to repeat.

3.2 Creating a Schedule

In this subsection, we review what it means for a CE scheduler to be correct and expand upon standard CE rules to apply to SMT and when frame size can vary. For now, we assume we have already determined safe execution costs for all tasks and pairs of co-scheduled tasks. We will discuss how to determine such costs in Secs. 4 and 5. A CE schedule is correct if the rules stated in Def. 14 below, adapted from Baker and Shaw [5], hold.

▶ **Definition 14.** *A CE schedule is* correct *if over the course of each hyperperiod: (i) all jobs are scheduled; (ii) any non-preemptable job is scheduled in exactly one frame; (iii) every job completes in a frame that ends no later than its deadline; (iv) no job executes in a frame that begins before its release; (v) the total execution time scheduled in each frame is no greater than the frame size; and (vi) no job executes in parallel with itself.*

If we allow a particular job to execute on multiple cores, rule (vi) of Def. 14 becomes quite challenging. For this reason, we require that each job, with or without SMT, executes on only one core. However, a task may execute different jobs on different cores.

In a conventional CE scheduler, not permitting SMT and requiring one frame size across all cores, the assignment of jobs or job portions to frames so that the system is schedulable is the primary decision to make. With CERT-MT, additional decisions are needed: what frame size should be used for each core, on what core should each job be scheduled – with different frame sizes, cores are not interchangeable – and how, if at all, should jobs be co-scheduled? Even so, the requirements given for correctness in Def. 14 are unchanged.

However, these correctness requirements are more complicated, particularly since the scheduling decisions needed cannot be made independently. To make them, we employ a mathematical optimization program. To convert our correctness conditions into mathematical form, we define a variable for every possible job pair, core, and frame.

▶ **Definition 15.** *For $i \neq j$, let the variable $x(i.a, j.b, \ell, g)$ be defined as*

$$x(i.a, j.b, \ell, g) = \frac{\textit{time budgeted for } \tau_{i.a:j.b} \textit{ on core } \ell \textit{ in frame } g}{C_{i:j}}.$$

This actually defines two variables for every pair of distinct jobs: one for which $i < j$ holds, and a second for which the opposite is true. For $i = j$, we define the variable only for $a = b$ and keep the same definition with the provision that a solo job is said to be "simultaneously co-scheduled" with itself.[5] *We refer to these variables as the x variables.*

▶ **Example 16.** Let the task system τ consist of $\tau_1 = (7.5, 10)$, $\tau_2 = (5, 20)$, $\tau_3 = (5, 20)$, $\tau_4 = (10, 20)$, and $\tau_5 = (20, 40)$. Furthermore, let $C_{1:2}$ and $C_{1:3} = 10$. Fig. 3 shows a possible schedule for τ on two cores (note that τ has total utilization of 2.25; it cannot be scheduled on two cores without SMT), along with the non-zero corresponding x variables. Frame borders are indicated by dashed lines; we have $f(1) = 10$ and $f(2) = 20$. The first two variables, $x(1.1, 2.1, 1, 1)$ and $x(2.1, 1.1, 1, 1)$, show that the first jobs of τ_1 and τ_2 are co-scheduled in frame 1 of core 1 for 10 time units, their joint cost. The last variable listed, $x(5.1, 5.1, 2, 2)$, shows that job 1 of τ_5 is scheduled in frame 2 of core 2 for half of its total cost. Note that the preemption of $\tau_{5.1}$ by the scheduler at time 20 is permitted, as our non-preemption requirement only applies to jobs using SMT.

[5] We do not propose scheduling a job with a second copy of itself; we overload the term simultaneously co-scheduled to avoid constantly addressing solo jobs as a special case.

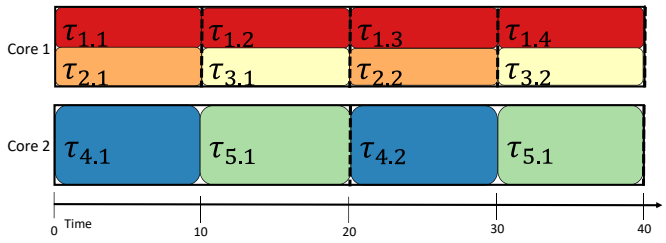

Figure 3 A possible schedule and corresponding x variables for the task system of Ex. 16.
x(1.1, 2.1, 1, 1)=x(2.1, 1.1, 1, 1)=1
x(1.2, 3.1, 1, 2)=x(3.1, 1.2, 1, 2)=1
x(1.3, 2.2, 1, 3)=x(2.2, 1.3, 1, 3)=1
x(1.4, 3.2, 1, 4)=x(3.2, 1.4, 1, 4)=1
x(4.1, 4.1, 2, 1)=1
x(5.1, 5.1, 2, 1)=0.5
x(4.2, 4.2, 2, 2)=1
x(5.1, 5.1, 2, 2)=0.5 .

We outline the constraints needed to build an optimization program that will produce a correct schedule for τ on platform π if such a schedule exists. Rather than giving the full program here, we outline the constraints needed to fulfill the conditions of Def. 14. The full program is available online [57].

Our first constraint is needed to address the existence of two variables for every pair of jobs. We require that

$$x(i.a, j.b, \ell, g) = x(j.b, i.a, \ell, g) \tag{4}$$

holds. In Fig. 3, we see the first eight variables listed as pairs for this reason. Our remaining constraints fulfill the requirements of Def. 14.

(i) All jobs are scheduled. To guarantee that all jobs released over the hyperperiod are scheduled, we require that for all jobs $\tau_{i.a}$ released within the first hyperperiod, the x variables corresponding to each job must sum to 1. Mathematically,[6]

$$\forall i, a \sum_{j=i}^{n} \sum_{b=1}^{\frac{H}{T_j}} \sum_{\ell=1}^{m} \sum_{g=1}^{\left\lfloor \frac{H}{f(\ell)} \right\rfloor} x(i.a, j.b, \ell, g) = 1.$$

In our example, τ_1 through τ_4 fulfill this requirement by having one variable valued at 1 for each job within the hyperperiod; τ_5 has two non-zero variables for its one job, which is scheduled in two different frames of core 2; each variable is valued at 0.5

(ii) Any non-preemptable job must be scheduled in exactly one frame. For $i \neq j$, we require that x be an integer variable, which can be seen with the variables corresponding to τ_1 through τ_3 in Ex. 16. While each job of τ_4 is also scheduled in a single frame, doing so is not necessary for a correct schedule, since τ_4 does not use SMT.

[6] Note that variables for which $i = j$ and $a \neq b$ do not exist per Def. 15. Here and elsewhere when $i = j$, $\sum_{b=1}^{\frac{H}{T_j}}$ would more properly written as $\sum_{b=a}^{a}$. We leave it as is for the sake of brevity.

(iii) **Every job completes in a frame that ends no later than its deadline.** Frame g of core ℓ ends at time $g \cdot f(\ell)$. We make the per-core frame size into a variable – $f(\ell)$ for all $\ell \leq m$ – and require that if a job is scheduled in a given frame, then the job's deadline may be no sooner than the end of the frame. In our optimization program, we express this rule using the following constraint:

$$\forall i.a, j.b, \ell, g, \lceil x(i.a, j.b, \ell, g) \rceil \cdot g \cdot f(\ell) \leq d(i.a, j.b).$$

If $\lceil x(i.a, j.b, \ell, g) \rceil = 1$ holds, then the restriction is true if and only if the frame's end time, $g \cdot f(\ell)$, is no more than the deadline. If $\lceil x(i.a, j.b, \ell, g) \rceil = 0$ holds, i.e., the job(s) is (are) not scheduled in frame g of core ℓ, making the frame size irrelevant, then the constraint is always true. While the ceiling operator is meaningless for non-preemptable co-scheduled jobs, which already have $x(i.a, j.b, \ell, g) \in \{0, 1\}$, it is necessary for the solo jobs to test whether *any* portion of the jobs is within frame g.
This constraint holds in Fig. 3; using the first jobs of τ_1 and τ_2 as an example, we have

$$\begin{aligned} \lceil x(1.1, 2.1, 1, 1) \rceil \cdot 1 \cdot f(1) &\leq d(1.1, 2.1) \\ 1 \cdot 1 \cdot 10 &\leq 10. \end{aligned}$$

Observe also that scheduling $\tau_{1.1}$ anywhere other than frame 1 of core 1, given that $f(2) = 20$, would violate this constraint. In addition, constraint (v) below guarantees that every job can execute for its scheduled time within each frame.

(iv) **No job executes in a frame that begins before its release.** Frame g of core ℓ begins at time $(g-1) \cdot f(\ell)$. Consequently, we require

$$\lceil x(i.a, j.b, \ell, g) \rceil \cdot r(i.a, j.b) \leq (g-1) f(\ell).$$

The logic here is similar to that in (iii) above; if $x(i.a, j.b, \ell, g) = 0$ holds, then the constraint will always be true. Otherwise, the job's release must fall no later than the beginning of the frame for the constraint to hold. In Fig. 3, using the third job of τ_1 and the second job of τ_2 as examples, we have

$$\begin{aligned} \lceil x(1.3, 2.2, 1, 2) \rceil \cdot r(1.3, 2.2) &\leq (3-1) \cdot f(1) \\ 1 \cdot 20 &\leq 2 \cdot 10. \end{aligned}$$

(v) **The total execution time scheduled in each frame is no greater than the frame size.** Each x variable requires that $x(i.a, j.b, \ell, g) \cdot C_{i:j}$ units of time be allocated to the corresponding job in the selected frame in order for the corresponding job or job pair to complete. Clearly, we must avoid overscheduling frames. The following constraint accomplishes that goal:

$$\forall\ l, g: \sum_{i=1}^{n} \sum_{a=1}^{\frac{H}{T_i}} \sum_{j=i}^{n} \sum_{b=1}^{\frac{H}{T_j}} x(i.a, j.b, \ell, g) \cdot C_{i:j} \leq f(\ell).$$

For example, $C_{1:2}$ and $f(1)$ are equal to 10 in Ex. 16, giving us

$$\begin{aligned} x(1.1, 2.1, 1, 1) \cdot C_{1:2} &\leq f(1) \\ 1 \cdot 10 &\leq 10 \end{aligned}$$

for frame 1 of core 1; note that all other variables connected to this frame, apart from $x(2.1, 1.1, 1, 1)$, are equal to 0.

(vi) No job executes in parallel with itself. For non-preemptable jobs, this rule holds automatically, since every job is scheduled in exactly one frame. For preemptable solo jobs, we add the constraint

$$\forall i, a, \ell \sum_{g=1}^{\left\lfloor \frac{H}{f(\ell)} \right\rfloor} x(i.a, i.a, \ell, g) \in \{0, 1\},$$

which states that for every solo job, either all or none of it must be on a given core. In Ex. 16, this constrains each job of τ_4 and τ_5 to a single core.

If a correct schedule exists, then an optimization program that follows the above constraints can find one. Despite the name, there is no objective function that needs to be optimized; if a set of decision variables that fulfills all restrictions exists, a correct schedule can be formed by making the scheduling choices corresponding to those variables. In our experiments, we found that when executing this program on a platform with 24 cores, most systems containing in the range of 20 tasks on 4 or 8 cores could be scheduled within a few seconds; we counted as a failure any system for which we had not found a solution within 60 seconds. Since solving the optimization program is done offline, this time is acceptable for many applications.

4 Timing Analysis

In this section, we present the timing-analysis portion of CERT-MT. While using the observed maximum for task costs, as we do here, is somewhat common practice, we are not aware of any prior work that considers exactly what the observed maximum is telling us, nor that addresses the fact that the observed maximum may itself be a random variable.

We use our results here to analyze our benchmark tests, presented in Sec. 5. All discussions of job execution costs in this section also apply to costs for simultaneously co-scheduled pairs of jobs. As mentioned in Sec. 2, we analyze execution times *as if* they were random. More specifically, we assume for now that for any given task, the execution times of individual jobs follow Premise 1 below. In Sec. 5 we show empirically that we can safely predict task behavior even without Premise 1 in most cases.

▶ **Premise 1.** The following properties hold for E_i and all $R_k \in R$: First, E_i and all values within R are drawn from the same probability distribution; second, individual R_k values are not dependent on k; and third, individual R_k values are not dependent on other values within R.

If we schedule our system using R_{max} as the task cost, how safe is that? More formally, consider Premise 2 below:

▶ **Premise 2.** Given task τ_i and trace R, assume that S_i^q is defined by $S_i^q = R_{max}$.

Combining Premise 2 and Def. 6 gives

$$\Pr(E_i \leq R_{max}) = q. \tag{5}$$

We want to determine the value of q. To do so, we need to consider both the relationship between E_i and R_{max} and that between $R_{max} = S_i^q$ and C_i^p for an arbitrary value of p. We make use of three basic rules of probability to determine q, given below without proof. Using these, rules, we give a lower bound for q in terms of p and $|R|$ in Lemma 20 below.

▶ **Proposition 17.** *Let events B_1 through B_v partition a probability space, and let A be an event belonging to the same probability space. The law of total probability states that the following holds:* $\Pr(A) = \sum_{i=1}^{v} \Pr(A|B_i) \cdot \Pr(B_i)$.

▶ **Proposition 18.** *Let A be a possible outcome of a repeated random trial. It follows for a series of trials, where each trial's result is independent from all previous results, that* $\Pr(A \text{ holds for some trial}) = 1 - \Pr(A \text{ holds for no trials})$ *holds.*

▶ **Proposition 19.** *Let A be a possible outcome of v repeated, independent trials. Then,* $\Pr(A \text{ holds for all trials}) = \Pr(A)^v$.

▶ **Lemma 20.** *Assume Premises 1 and 2 hold. Given a trace R and an arbitrary p associated with some value for C_i^p, the following holds:*

$$q \geq p \cdot \left(1 - p^{|R|}\right). \tag{6}$$

Note that since (6) holds for an arbitrary p, q is not a function of p.

Proof. The lemma is established by the following derivation:

$$
\begin{aligned}
& q \\
= \; & \{\text{by Exp. (5)}\} \\
& \Pr\left(E_i \leq R_{max}\right) \\
= \; & \{\text{by Prop. 17}\} \\
& \Pr(E_i \leq R_{max} | R_{max} \geq C_i^p) \cdot \Pr(R_{max} \geq C_i^p) + \Pr(E_i \leq R_{max} | R_{max} < C_i^p) \cdot \Pr(R_{max} < C_i^p) \\
\geq \; & \{\text{since } \Pr\left(E_i \leq R_{max} \mid R_{max} < C_i^p\right) \cdot \Pr\left(R_{max} < C_i^p\right) \geq 0\} \\
& \Pr\left(E_i \leq R_{max} \mid R_{max} \geq C_i^p\right) \cdot \Pr\left(R_{max} \geq C_i^p\right) \\
\geq \; & \{\text{since } \Pr\left(E_i \leq R_{max} \mid R_{max} \geq C_i^p\right) \geq \Pr(E_i \leq C_i^p)\} \\
& \Pr(E_i \leq C_i^p) \cdot \Pr\left(R_{max} \geq C_i^p\right) \\
= \; & \{\text{by Exp. (1)}\} \\
& p \cdot \Pr\left(R_{max} \geq C_i^p\right) \\
= \; & \{\text{by the definition of } R_{max} \text{ (Def. 5)}\} \\
& p \cdot \Pr(R_k \geq C_i^p \text{ holds for some } R_k \in R) \\
= \; & \{\text{by Prop. 18}\} \\
& p \cdot (1 - \Pr(R_k < C_i^p \text{ holds for all } R_k \in R)) \\
= \; & \{\text{by Prop. 19}\} \\
& p \cdot (1 - \Pr(R_k < C_i^p)^{|R|}) \\
= \; & \{\text{since } R_k \text{ terms are independent and from the same distribution as } E_i, \text{ per Pre. 1}\} \\
& p \cdot (1 - \Pr(E_i < C_i^p)^{|R|}) \\
\geq \; & \{\text{since } \Pr(E_i < C_i^p) \leq \Pr(E_i \leq C_i^p) \; \} \\
& p \cdot (1 - \Pr(E_i \leq C_i^p)^{|R|}) \\
= \; & \{\text{by Exp. (1)}\} \\
& p \cdot (1 - p^{|R|}).
\end{aligned}
$$

◀

We can now lower-bound q in terms of $|R|$ alone.

▶ **Theorem 21.** *Assume that Premises 1 and 2 hold. Then it follows that*

$$q \geq \left(\frac{1}{|R|+1}\right)^{\frac{1}{|R|}} \left(1 - \frac{1}{|R|+1}\right). \tag{7}$$

Proof. We first define the lower bound of q given in Exp. (6) as q^*, i.e. $q \geq q^*$ holds, where

$$q^* = p(1 - p^{|R|}).$$

Maximizing q^* will give a lower bound for q in terms of $|R|$ alone. To maximize q^*, we find the value of p for which the first derivative of q^* with respect to p equals 0 and the second derivative is negative. The first derivative is given by

$$\frac{dq^*}{dp} = 1 - (|R|+1) \cdot p^{|R|},$$

and the second by

$$\frac{d^2q^*}{dp^2} = -|R| \cdot (|R|+1) \cdot p^{|R|-1}.$$

Observe that $\frac{d^2q^*}{dp^2} < 0$ holds for all $p > 0$. Consequently, q^* is maximized when $\frac{dq^*}{dp} = 0$.

Solving $\frac{dq^*}{dp} = 0$ for p gives the value of p that maximizes q^*. Inserting this value into Exp. (6) in the place of q gives the result. ◀

▶ **Definition 22.** *We refer to the lower bound of Exp. (7) given trace size $|R|$ as $q_{b(|R|)}$.*

To attach some concrete values to (7), $|R| = 1000$ gives $q_{b(|R|)} = 0.992$ and $|R| = 10^5$ gives $q_{b(|R|)} = 0.9999$. Using these results, we can make comparisons between tasks with and without SMT based on maximum observed execution times. While we could have compared maximum observed times without the preceding proofs, determining $q_{b(|R|)}$ as we have allows us to meaningfully say that tasks with SMT are as safe as those without.

Violating Premise 1. The greatest potential shortfall of this approach is the reliance on Premise 1, which may not hold in practice. However, this obstacle is not unique to us; as mentioned in Sec. 2 under "Related works," EVT methods also must contend with data that may not be as independent as desired. In Sec. 5.1, we empirically test what happens when Premise 1 does not hold.

5 Experimental Results

In this section, we present our experimental results. The benchmark tests were conducted on an Intel Xeon Silver 4110 2.1 GHz (Skylake) CPU running Ubuntu 16.04.6. All code related to our experiments is available online [57]. We begin, in Sec. 5.1, by evaluating how well defining costs to be R_{max} works when Premise 1 may not hold. We also determine how many execution-time samples are needed to safely determine R_{max} within the context of our experiments. Using these sample counts, we then evaluate the benefits of allowing SMT in Sec. 5.2 by examining benchmark data involving individual tasks and task pairs. Finally, in Sec. 5.3, we evaluate these benefits more holistically on a system-wide basis via a schedulability study. The benchmark data discussed in Sec. 5.2 was used to inform parameter choices underlying this study.

Experimental studies are typically done within a framework that defines an "artificial world," and definitive conclusions can only really be drawn with respect to that "world" – further conclusions concerning the "real" world, though often interesting and relevant, are necessarily speculative. In our context, we need to be able to compare R_{max} values determined from relatively small traces to entire populations. Thus, in Secs. 5.1 and 5.2, we assume an artificial world in which 18 programs are of interest from the TACLeBench sequential benchmarks [23], which consist of functions commonly found in embedded and real-time systems. Furthermore, we assume that the entire population of execution times for the task in this world is given by a sequence of 100,000 job executions, which we denote as R^+. Note that, in the "real" world, we typically would not have the entire population of possible execution times – if we did, we would have no need for MBPTA. With this setup in place, our world thus consists of 18 solo tasks and 171 task pairs, each with 100,000 jobs or job pairs. In determining SMT execution costs in this world, we simultaneously co-scheduled task pairs per Def. 8. In all cases, we invalidated the cache between jobs or job pairs in determining such costs.

In the "real" world, appropriately dealing with task inputs in timing analysis is a complex issue. In our setting here, the initial "input" for each benchmark is hard-coded, but the program structure causes the inputs processed by each loop to vary, primarily due to the presence of non-resetting global variables. This aspect of the benchmarks allows for patterns similar to real-world applications where a task's input, and therefore its execution time, exhibits dependencies between one job and the next. For example, in many image-processing applications, the complexity of one job, and therefore the processing time needed, is predictive of the next. As our purpose here is to assess the viability of SMT, rather than fully addressing the issue of timing analysis in multicore systems, we believe these assumptions regarding input data are reasonable.

5.1 Timing-Analysis Assessment

So far, our timing-analysis has been theoretical; we have shown that if Premises 1 and 2 hold, then we can safely place a lower bound on the value of q, which we denote as $q_{b(|R|)}$ per Def. 22 ("b" denotes our theoretical bound). In practice, however, Premise 1 may not hold. What can we say about q in a more realistic setting? To avoid overloading q, we define an additional term for the value we actually compute given a trace and population.

▶ **Definition 23.** *Given a trace size $|R|$ and a population R^+, and assuming that every possible block of $|R|$ consecutive execution times taken from R^+ forms a trace and is equally likely to occur, let the result of computing q per Def. 6 be denoted $q_{c(|R|)}$ ("c" denotes a computed value of q).*

The way we have created R^+ and defined our possible traces means that Premise 1 will not typically hold; individual R_k values within each trace will tend to be dependent on both k and on other values within the same trace. However, if we find that $q_{c(|R|)} \geq q_{b(|R|)}$ holds, we then have evidence that building a system on the assumption that a proportion $q_{b(|R|)}$ of all tasks will have execution times no more than R_{max} is safe even without Premise 1. In our experiments, we use $|R| = 1000$ (i.e., we consider using 1,000 samples from the population of size 100,000), giving $q_{b(1000)} = 0.992$ using Exp. (7).

Results. When we calculated $q_{c(1000)}$ for each of our 18 solo tasks, we found a minimum $q_{c(1000)}$ of 0.997 and a mean and maximum of 0.999. We conclude that for these tasks in our experimental context, calculating costs via R_{max} is safe even without Premise 1. Our

$q_{c(1000)}$ values are closer to 0.999 than to $q_{b(1000)}$; the former is the value we would expect to see given that $R_{max} \approx C_i^{0.999}$. Our $q_{c(1000)}$ values for all solo jobs, along with additional summary statistics, are available in an online appendix [57]. For our 171 task pairs using SMT, the mean $q_{c(1000)}$ across all pairs was found to be 0.998. Further highlights are given in Table 1. Full results are in [57].

Cases where $\mathbf{q_{c(1000)} < q_{b(1000)}}$. Of the 171 task pairs, five had $q_{c(1000)} < q_{b(1000)}$, but if we exclude pairs for which C_i and C_j (defined as the R_{max} for the first 1,000 jobs of each solo task) differ by a factor of more than 10, we can eliminate these five problem pairs. In Sec. 5.2, we will see that there are additional reasons to not consider pairs with dramatically different solo execution times, making this exclusion an appealing option.

An alternative solution is to increase our trace size until we find a new value for $|R|$ such that $q_{c(|R|)} \geq q_{b(1000)}$. When we did this, our new $|R|$ values ranged from 1,250 to 1,500, in all cases giving $q_{c(|R|)} > 0.992$. An example can be seen in Table 1; the pair of tasks "cjpeg_wrbmp" and "susan" have $q_{c(1000)} = 0.987$, but if we inflate the trace size by a factor of 1.5 (column "inf."), we then have $q_{c(1500)} = 0.992$ (column "$q_{(inf.|R|)}$"). Note that inflation factors are not applicable for task pairs where $q_{c(1000)} \geq q_{b(1000)}$ holds. Within the context of our experiments, this result suggests that even when SMT is challenging from a timing-analysis perspective, reliable timings can be obtained by using a trace size that is a slightly larger than the trace size that would be needed for a given level of safety if Premise 1 held. We plan to investigate this aspect of our results further in future work, with a particular emphasis on what inflation factors can be expected given a wider range of tasks and R_{max} values.

Timing-analysis conclusions. Before moving on, we recap our conclusions from this subsection. In the experiments discussed here, we found that $q_{c(1000)} \geq q_{b(1000)}$ held perfectly, even without Premise 1, when SMT was not in play. When SMT was used, $q_{c(1000)} \geq q_{b(1000)}$ held most of the time, or all of the time if do not allow SMT to be used when C_i and C_j differ by more than a factor of 10.

The seemingly obvious solution is to use SMT only for cases where $q_{c(1000)} \geq q_{b(1000)}$ holds. The problem is that to know whether the inequality holds, we need either an assurance that Premise 1 holds or the ability to compare multiple traces to the population. In real applications, we would not have the population available. For this reason, the implementation of our timing-analysis method should be done in consultation with domain experts who could give assurances that a trace is large enough to capture relevant execution-time variations.

This approach may seem unsatisfactory, but it is similar to formal system verification under a fault hypothesis. In a safety-critical context, such as avionics, software properties are demonstrated to hold subject to a given fault hypothesis – informally, a list of possibilities that should not happen. For example, an aviation software fault hypothesis might include "foreign objects will not enter the engines." If this condition fails – perhaps a flock of geese flies into the engines – then correctness is no longer guaranteed. A fault hypothesis will be based on expert input, but it cannot be mathematically guaranteed. Selecting a trace size that will allow at least the first point of Premise 1 to hold is essentially another element of a fault hypothesis. In this case we would be claiming that $|R|$ is large enough to capture the task's overall behavior. This concern is not unique to us, but applies to any MBPTA method. Even for static analysis, the results hinge on the assumption that all relevant factors were adequately accounted for.

Table 1 Task pairs with min., median, and max. $q_{c(1000)}$ values.

benchmark 1	benchmark 2	$q_{c(1000)}$	inf.	$q_{c(inf.\|R\|)}$
cjpeg_wrbmp	susan	0.987	1.5	0.992
adpcm_dec	rijndael_enc	0.999	NA	NA
dijksta	gsm_enc	0.999	NA	NA

5.2 Execution Times with and without SMT

We now consider how execution times with SMT compare to those without. We need this information to make a realistic assessment of the potential benefits of using SMT. In Sec. 5.1, we saw that given our populations, trace sizes of 1,000 are safe for tasks without SMT and trace sizes of 1,500 are safe for tasks with SMT. With that in mind, our results in this subsection are based on the first 1,000 execution times for solo tasks and the first 1,500 execution times for pairs. This is a conservative approach; while we found only five task pairs for which inflating $|R|$ was necessary, we do so here for all pairs.

For each pair of benchmark tasks, we record their *multithreading score.*

▶ **Definition 24.** *If $\tau_{i:j}$ is a task pair for which $C_i \geq C_j$ holds, then the* multithreading score *$M_{i:j}$ satisfies the following:*

$$C_{i:j} = C_i + M_{i:j} \cdot C_j.$$

When τ_i and τ_j are co-scheduled, we are essentially hiding the execution requirement for τ_j at the cost of increasing the execution time needed for τ_i. $M_{i:j}$ thus gives how much greater $C_{i:j}$ is than C_i for every unit of C_j, i.e., the cost to τ_i, per unit of τ_j, of "hiding" τ_j.

If $M_{i:j} \geq 1$ holds, then there is no benefit to pairing τ_i and τ_j together, since SMT would not succeed in "hiding" τ_j at all. If $M_{i:j} < 1$ holds, then pairing jobs of the two tasks is potentially beneficial, with lower values indicating greater benefit. If $M_{i:j} < 0$ holds, then $\tau_{i:j}$ actually requires less measured time to execute than τ_i alone.

We found that $M_{i:j} > 1$ was frequently the case for pairs where $\frac{C_i}{C_j} > 10$ holds. For this reason, along with the difficulty in using R_{max} for timing analysis in those cases described in Sec. 5.1, we decided that using SMT in those cases is not advisable. For the remaining pairs, where $\frac{C_i}{C_j} \leq 10$ holds, we summarize our results via a histogram in Fig. 4. The remainder of our discussion considers only these remaining pairs. Again, more detailed results are available in [57]. We did see five negative values, ranging from -0.003 to -0.532. For all of these values, comparing the traces for the component solo tasks to the larger population showed unexpectedly high R_{max} values. Essentially, we overestimated both τ_i and τ_j individually, thereby distorting the value of $M_{i:j}$; our values were not unsafe.

Even if we ignore the negative values, our findings indicate that SMT can have some benefit in almost all pairs that have similar solo costs for τ_i and τ_j. We make further use of these findings, and explore their implications for schedulability, in Sec. 5.3.

5.3 Schedulability Study

In this subsection, we turn our attention to assessing the benefits of allowing SMT from a system-wide perspective. To provide such an assessment, we conducted a schedulability study involving CERT-MT.

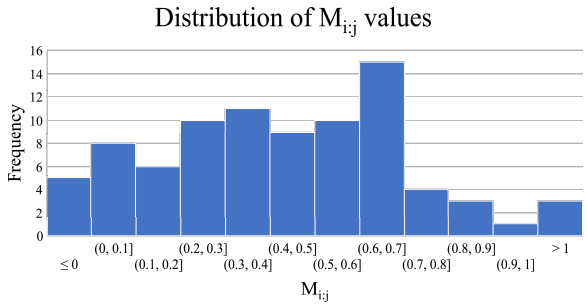

Figure 4 Distribution of $M_{i:j}$ values. The distribution has a median of 0.42 and values less than one have a mean of 0.40. Three values are greater than 1.0.

Generating task sets. We examined 104 scheduling scenarios, with each scenario defined by a core count, per-task utilization range, and a model for SMT interaction. The last factor is described in detail below; the others require only a brief explanation. We considered core counts of four and eight and per-task utilizations drawn from four uniform ranges: $(0, 0.4)$ (*low*), $(0.3, 0.7)$ (*medium*), $(0.6, 1)$ (*high*), and $(0, 1)$ (*wide*). Each task was created by selecting a utilization from the appropriate distribution and a period from the set $\{10, 20, 40, 80\}$, with all periods having equal probability. 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 $\frac{3m}{4}$ to $2m$. Recall that m gives the core count, and consequently the maximum schedulable utilization when SMT is not used.

For each scenario, the corresponding schedulability ratios are summarized in one graph. Each data point in one of these graphs represents the fraction of approximately 50 task systems that could be deemed as schedulable. The total set of graphs took over 2 CPU years to compute on a 6,500-CPU research cluster.

Modeling pair costs. A key aspect of our study is that of modeling the cost of job pairs executing with SMT. Our goal in modeling such costs was not to reproduce any system's behavior exactly, but to be able to draw broad conclusions about the value of allowing SMT without relying on one specific model for task interactions.

In Fig. 4, we see that most $M_{i:j}$ values fall between 0.1 and 0.8. To capture the possibility of $M_{i:j} \geq 1$, we first gave each job pair either a 0.0, 0.1, or 0.2 probability of having $M_{i:j} \geq 1$. We refer to this value as the *split*, i.e., a split of 0.1 indicates that each task pair has a 10% chance of being declared unsuitable for SMT. This particular split value closely reflects our observed results discussed previously. If a task pair was not selected to have $M_{i:j} \geq 1$, then we determined its $M_{i:j}$ value based on one of three normal distributions – (0.45, 0.12), (0.6, 0.07), or (0.45, .06) – or one of three uniform distributions – (0.1, 0.8), (0.4, 0.8), or (0.27, 0.63). All of the normal distributions were truncated, with any negative value produced replaced by 0.01. Of these distributions, the first uniform and first normal distributions most closely match our observed results, which had a median of 0.43 and a mean of 0.40 excluding cases where $M_{i:j} > 1$. We exclude those cases from our mean because any pair where $M_{i:j} > 1$ holds is unsuitable for threading; the exact value of $M_{i:j}$ will not impact

the decision. The second distribution of each type gives a higher (i.e., more pessimistic) mean, and the third of each type has the same mean as the first, but a smaller standard deviation (resp., range) in the normal (resp., uniform) case. All of these distributions add some pessimism to Fig. 4 by eliminating or reducing the probability of $M_{i:j} < 0.1$.

In total, we utilized 18 combinations of distributions in determining task-pair costs: three probabilities for $M_{i:j} \geq 1$, each combined with each of three uniform and three normal distributions. After determining $M_{i:j}$, we calculated $C_{i:j}$ per Def. 24. For task pairs where solo costs differ by a factor exceeding 10, we did not allow SMT. For our four-core scenarios, we tested every possible combination of per-task utilizations and $M_{i:j}$ distributions, yielding 72 scenarios. For our eight-core scenarios, we were more selective, considering 32 scenarios.

Creating and evaluating schedules. To determine whether a task system was schedulable, we attempted to create a schedule that complied with the conditions given in Sec. 3 for each system.[7] The graphs show the proportion of systems that are schedulable at each total utilization. We summarize CERT-MT's overall performance for each graph by recording its *relative schedulable area (RSA)*, defined as the area under the schedulability curve divided by the core count m. In calculating RSAs, we assumed that the schedulability ratio is constant between total utilization 0.0 and $\frac{3m}{4}$, which is the smallest utilization we tested in each scenario. This assumption results in RSAs being understated in some 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. RSAs for practical hard real-time schedulers would typically be less than 1.0, sometimes dramatically so [15].

Schedulability graphs. Our full set of graphs is included in an online appendix [57]. Here we present a small selection of our results, along with some observations on general trends we observed. In Figs. 5 – 8 we show, respectively, our best, median, and worst four-core results, along with our best eight-core results. Generally, CERT-MT's performance in the eight-core scenarios was not as good as in the four-core scenarios, for reasons discussed below.

▶ **Observation 1.** *At its best, CERT-MT proved capable of increasing RSAs by a factor of 1.4 or more. This ability is demonstrated by Fig. 5. Of our 72 four-core graphs, 26 have RSAs of at least 1.3, and 45 have RSAs are greater than 1.0.*

▶ **Observation 2.** *The performance of CERT-MT tended to decrease as the job count increased. For example, Fig. 7 uses light per-task utilization, meaning there are more tasks for a given total utilization. The poorer performance of CERT-MT for high job counts was not due to any ill effects from SMT but rather to the time required to produce a schedule. Given the magnitude of our study, it was necessary to enforce a time limit on our optimization program. If this limit was reached for a particular task set, then it was deemed as unschedulable. We used a 60-second limit. This observation also explains the relatively poor performance we saw in the eight-core scenarios. We counter this observation, however, by noting that when testing a specific task system, a much higher time limit could be used. Moreover, it is possible that a more efficient schedule-construction method could be devised.*

[7] We used Gurobi Optimizer, a commercial optimization programming solver with free academic licensing.

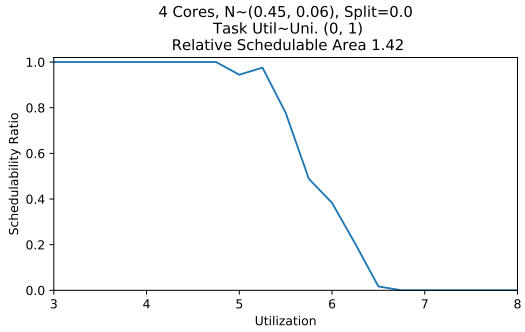

Figure 5 The best four-core scenario.

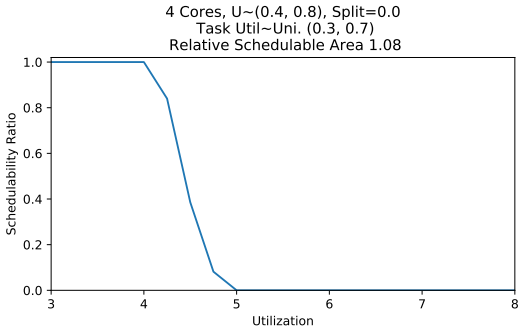

Figure 6 The median four-core scenario.

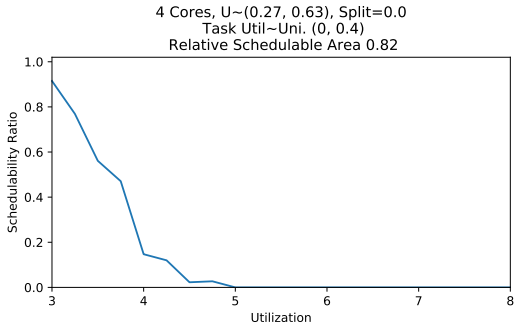

Figure 7 The worst four-core scenario.

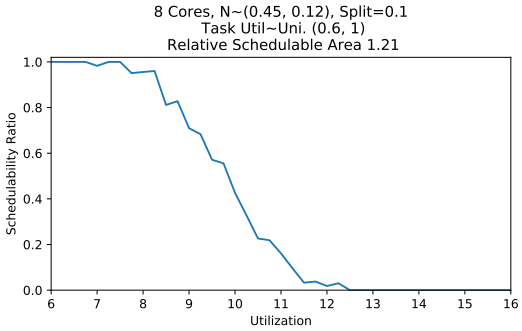

Figure 8 The best eight-core scenario.

▶ **Observation 3.** *CERT-MT performed best when the majority of $M_{i:j}$ values fell within a narrow range. For example, the standard deviation of such values is the smallest in the most successful scenario (depicted in Fig. 5). This observation is also connected to the time limits we imposed on our optimization program; a wider range of $M_{i:j}$ values gives this program more "choices" to consider, which takes more time.*

▶ **Observation 4.** *The mean of $M_{i:j}$ had surprisingly little effect on overall performance. The graphs in Figs. 5 and 7 have distributions that produce identical means for $M_{i:j}$.*

Improving schedulability. The schedulability study we conducted suggests that employing SMT can potentially enable substantial schedulability gains in hard real-time systems. Additionally, the observations above hint at several ways of further increasing schedulability beyond what we have seen. First, a larger schedule-construction timeout value could be used, as noted already. Second, for larger systems, it may be more effective to divide tasks into clusters scheduled separately. Third, given that the standard deviation of $M_{i:j}$ seems to be more important for schedulability than its mean, it might be possible, counter-intuitively, to improve schedulability by artificially increasing the lower $M_{i:j}$ values to decrease their range.

6 Conclusion

In industry today there is interest in using SMT to increase the capacity of safety-critical systems, as evidenced by the FAA inquiries noted earlier [55]. In this paper, we have shown that when SMT-enabled jobs are scheduled with care, producing a safe, measurement-based

timing analysis is comparable in difficulty to doing the same for jobs not using SMT. Within the context of our schedulability study, we demonstrated that allowing SMT can increase system capacity by up to 40%. We have also highlighted a subtlety of measurement-based timing analysis – any estimate based on random measured data is itself a random variable – that is not always emphasized, and that could compromise safety, if not accounted for.

Due to space constraints, there are many topics we have not been able to address that we plan to address in the future. These include: the impacts of (well-studied) sources of cross-core interference in multicore systems when SMT is allowed; the selection of appropriate task inputs in measurement-based timing analysis; the impacts of occasional deadline misses on overall system safety; and refined methods for constructing schedules under CERT-MT. As to the question posed in this paper's title, the evidence provided in this paper suggests that SMT can indeed be both safe and beneficial for hard real-time systems.

References

1 J. Abella, E. Quiñones, F. Wartel, T. Vardanega, and F. J. Cazorla. Heart of gold: Making the improbable happen to increase confidence in MBPTA. In *ECRTS 2014*, pages 255–265, 2014.

2 S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis. Outstanding paper: Evaluation of cache partitioning for hard real-time systems. In *ECRTS 2014*, pages 15–26. IEEE, 2014.

3 J. H. Anderson, S. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, volume 4, page 7. Euromicro, 2009.

4 B. Andersson, H. Kim, D. De Niz, M. Klein, R. Rajkumar, and J. Lehoczky. Schedulability analysis of tasks with corunner-dependent execution times. *ACM Trans. Embed. Comput. Syst.*, 17(3):71:1–71:29, May 2018.

5 T. P. Baker and A. Shaw. The cyclic executive model and ada. *Real-Time Systems*, 1(1):7–25, June 1989.

6 A. A. Balkema and L. De Haan. Residual life time at great age. *The Annals of Probability*, pages 792–804, 1974.

7 P. Benedicte, L. Kosmidis, E. Quinones, J. Abella, and F. J. Cazorla. A confidence assessment of wcet estimates for software time randomized caches. In *INDIN*, pages 90–97, 2016.

8 P. Benedicte, L. Kosmidis, E. Quinones, J. Abella, and F. J. Cazorla. Modelling the confidence of timing analysis for time randomised caches. In *SIES*, 2016.

9 B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA 2008*, pages 101–110. IEEE, 2008.

10 J. Bulpin. *Operating system support for simultaneous multithreaded processors*. PhD thesis, University of Cambridge, King's College, 2005. URL: `http://www.cl.com.ac.uk/TechReports/`.

11 J. Bulpin and I. Pratt. Multiprogramming performance of the Pentium 4 with hyperthreading. In *Third Annual Workshop on Duplicating, Deconstruction and Debunking*, pages 53–62, June 2004.

12 A. Burns and S. Baruah. Migrating mixed criticality tasks within a cyclic executive framework. In J. Blieberger and M. Bader, editors, *Reliable Software Technologies – Ada-Europe 2017*, pages 203–216, Cham, 2017. Springer International Publishing.

13 A. Burns and S. Edgar. Predicting computation time for advanced processor architectures. In *ECRTS 2000*, pages 89–96, February 2000.

14 A. Burns, T. Fleming, and S. Baruah. Cyclic executives, multi-core platforms and mixed criticality applications. In *ECRTS 2015*, pages 3–12, July 2015.

15 J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. H. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

16 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, July 2006.

17 F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52(1):14:1–14:35, February 2019.

18 M. Chisholm, N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter. Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. In *RTNS 2017*, pages 58–67. ACM, 2017.

19 M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS 2015*, pages 305–316. IEEE, 2015.

20 R. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):03–1–03:60, 2019.

21 C. Deutschbein, T. Fleming, A. Burns, and S. Baruah. Multi-core cyclic executives for safety-critical systems. *Science of Computer Programming*, 172:102–116, 2019.

22 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, September 1997.

23 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *WCET 2016*, volume 55, pages 2:1–2:10, 2016.

24 R. A. Fisher and L. H. C. Tippett. Limiting forms of the frequency distribution of the largest or smallest member of a sample. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 24, pages 180–190. Cambridge University Press, 1928.

25 A. Fog. The microarchitecture of Intel, AMD, and VIA CPUs: an optimization guide for assembly programmers and compiler makers, 2018. Available at `https://www.agner.org/optimize/microarchitecture.pdf`.

26 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, March 2016.

27 T. Gomes, S. Pinto, P. Garcia, and A. Tavares. RT-SHADOWS: Real-time system hardware for agnostic and deterministic OSes within softcore. In *ETFA 2015*, pages 1–4, September 2015.

28 F. Guet, L. Santinelli, and J. Morio. On the Reliability of the Probabilistic Worst-Case Execution Time Estimates. In *ERTS 2016*, Toulouse, France, January 2016.

29 D. Guo and R. Pellizzoni. A requests bundling DRAM controller for mixed-criticality systems. In *RTAS 2017*, pages 247–258. IEEE, 2017.

30 M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *RTAS 2015*, pages 307–316. IEEE, 2015.

31 J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. RTOS support for multicore mixed-criticality systems. In *RTAS 2012*, pages 197–208, April 2012.

32 C. Hernandez, J. Abella, A. Gianarro, J. Andersson, and F. J. Cazorla. Random modulo: a new processor cache design for real-time critical systems. In *Proceedings of the 53rd Annual Design Automation Conference*, page 29. ACM, 2016.

33 T. Hsing. On tail index estimation using dependent data. *The Annals of Statistics*, pages 1547–1569, 1991.

34 W. Huang, J. Lin, Z. Zhang, and J.M. Chang. Performance characterization of Java applications on SMT processors. In *ISPASS 2005.*, pages 102–111, March 2005.

35 R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *RTSS 2002*, pages 134–145, 2002.

36 S. Jiménez Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean. Open challenges for probabilistic measurement-based worst-case execution time. *IEEE Embedded Systems Letters*, 9(3):69–72, September 2017.

37 S. Kato, H. Kobayashi, and N. Yamasaki. U-link scheduling: bounding execution time of real-time tasks with multi-case execution time on SMT processors. In *RTCSA 2005*, pages 193–197, August 2005.

38 S. Kato and N. Yamasaki. Extended u-link scheduling to increase the execution efficiency for SMT real-time systems. In *RTCSA 2006*, pages 373–377, August 2006.

39 J. Kim, M. Yoon, R. Bradford, and L. Sha. Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 321–331, July 2014.

40 N. Kim. *Combining Hardware Management with Mixed-Criticality Provisioning in Multicore Real-Time Systems*. PhD thesis, UNC Chapel Hill, 2019. URL: `https://www.cs.unc.edu/~anderson/diss/namhoondiss.pdf`.

41 D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *RTSS 1989*, pages 229–237. IEEE, 1989.

42 L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 513–518. EDA Consortium, 2013.

43 L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla. Efficient cache designs for probabilistically analysable real-time systems. *IEEE Transactions on Computers*, 63(12):2998–3011, 2013.

44 M. R. Leadbetter, G. Lindgren, and H. Rootzén. Conditions for the convergence in distribution of maxima of stationary normal processes. *Stochastic Processes and their Applications*, 8(2):131–139, 1978.

45 G. Lima and I. Bate. Valid application of EVT in timing analysis by randomising execution time measurements. In *RTAS 2017*, pages 187–198. IEEE, 2017.

46 G. Lima, D. Dias, and E. Barros. Extreme value theory for estimating task execution time bounds: A careful look. In *ECRTS 2016*, pages 200–211. IEEE, 2016.

47 J. W. S. Liu. *Real-Time Systems*. Prentice Hall, New York, NY, USA, 2000.

48 S. Lo, K. Lam, and T. Kuo. Real-time task scheduling for SMT systems. In *RTCSA 2005*, pages 5–10, August 2005.

49 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *RTAS 2013*, pages 45–54, April 2013.

50 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, February 2002.

51 S. Milutinovic, I. Abella, J.and Agirre, M. Azkarate-Askasua, E. Mezzetti, T. Vardanega, and F. J. Cazorla. Software time reliability in the presence of cache memories. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 233–249. Springer, 2017.

52 S. Milutinovic, J. Abella, and F. J. Cazorla. Modelling probabilistic cache representativeness in the presence of arbitrary access patterns. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 142–149, May 2016.

53 J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Using SMT to hide context switch times of large real-time tasksets. In *RTAS 2010*, pages 255–264, August 2010.

54 D. Muench, M. Paulitsch, and A. Herkersdorf. Temporal separation for hardware-based I/O virtualization for mixed-criticality embedded real-time systems using PCIe SR-IOV. In *ARCS 2014; 2014 Workshop Proceedings on Architecture of Computing Systems*, pages 1–7, February 2014.

55 B. Ocker. FAA special topics. In *Collaborative Workshop: Solutions for Certification of Multicore Processors*, November 2018.

56 S. Osborne and J. H. Anderson. Work in progress: Combining real time and multithreading. In *RTSS 2019*, pages 139–142, December 2018.

57 S. Osborne and J. H. Anderson. Simultaneous multithreading and hard real time: Can it be safe? (longer version with additonal material), 2020. Available at `http://jamesanderson.web.unc.edu/papers/`.

58 S. Osborne, J. Bakita, and J. H. Anderson. Simultaneous multithreading applied to real time. In *ECRTS 2019*, July 2019.

59 R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *RTSS*, 2008.

60 J. Pickands III. Statistical inference using extreme order statistics. *The Annals of Statistics*, 3(1):119–131, 1975.

61 L. Santinelli, F. Guet, and J. Morio. Revising measurement-based probabilistic timing analysis. In *RTAS 2017*, pages 199–208, 2017.

62 L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart. On the sustainability of the extreme value theory for WCET estimation. In *14th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

63 G. N. Seetanadi, J. Camara, L. Almeida, K. Arzen, and M. Maggio. Event-driven bandwidth allocation with formal guarantees for camera networks. In *RTSS 2017*, pages 243–254, December 2017.

64 A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In *ASPLOS 2000*, ASPLOS IX, pages 234–244, New York, NY, USA, 2000. ACM.

65 K. Suito, K. Fujii, H. Matsutani, and N. Yamasaki. Dependable responsive multithreaded processor for distributed real-time systems. In *2012 IEEE COOL Chips XV*, pages 1–3, April 2012.

66 N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *PACT*, pages 26–35, Washington, DC, USA, 2003. IEEE Computer Society.

67 D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, 1995.

68 P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS 2016*, pages 1–12, April 2016.

69 B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *ECRTS 2013*, pages 157–167, July 2013.

70 M. Xu, L. T. X. Phan, H. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS 2016*, pages 1–12, April 2016.

71 H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS 2014*, pages 155–166, April 2014.

72 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, pages 55–64, April 2013.

73 M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *RTAS 2014*, pages 101–110, April 2014.