Work in Progress: Combining Real Time and Multithreading *

Sims Osborne and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

The existing sporadic task model is inadequate for real-time systems to take advantage of Simultaneous Multithreading (SMT), which has been shown to improve performance in many areas of computing, but has seen little application to real-time systems. A new family of task models, collectively referred to as SMART, is introduced. SMART models allow for combining SMT and real time by accounting for the variable task execution costs caused by SMT.

1 Introduction

Simultaneous multithreading (SMT) is a technology developed in the 1980s and 90s that allows multiple processes to issue instructions to different threads on a single computing core, creating the illusion of multiple cores for every one core that is actually present. It was designed to increase system utilization, particularly in the presence of memory latency [4, 17]. SMT became widely available in 2002, when it was made available on Intel processors [12]. Experiments on the Pentium 4 showed that SMT can potentially increase throughput by a factor of more than 1.5 [1, 2, 16].

The first attempt to utilize SMT in a real-time context was made in 2002 by Jain et al. [9], who showed that, by making every thread available for real-time work, it is possible to schedule workloads with total utilizations up to 50 percent greater than would be possible on the same platform without SMT. However, neither Jain et al. nor anyone else, to our knowledge, has provided a schedulability test that takes SMT into account.

Unfortunately, SMT's increase in throughput comes at the cost of longer and less predictable execution times due to interference caused by contention for limited hardware resources. Apparently, the real-time systems community decided that this uncertainty makes SMT inappropriate for real-time work. However, we question the validity of this assessment for soft real-time (SRT) systems that may tolerate some job tardiness. In this paper, we present two models for analyzing the effects of using SMT in a real-time context that make it possible to take advantage of SMT without sacrificing analytically guaranteed tardiness bounds.

Our approach. If a task may execute on a single thread, its execution cost is a function of the additional workload on the same core. Variable costs bring about a new scheduling problem: how to test for schedulability when neither individual tasks nor the task system has a fixed utilization. We consider both a *symbiosis-oblivious* model, in which we assume every task has a single worst-case threaded execution time that is independent of other work on the same core, and a *symbiosis-aware* model in which we explicitly address variations in task execution costs.

Related works. Snavely and Tullsen demonstrated that SMT performance is dependent on which tasks share a core and introduced the term symbiosis to describe this concept [15]. We have mentioned Jain et al.'s work on SMT and real-time scheduling from 2002 [9], which introduced the terms symbiosis-oblivious and symbiois-aware. Since then, Cazorla et al. [3], Gomes et al. [6, 7], and Zimmer et al. [20] have proposed ways to eliminate the timing uncertainties associated with SMT by means of detailed control over program execution. Cazorla et al. [3] and Lo et al. [11] gave methods to limit real-time work to a small number of threads, leaving the remaining threads to execute only when doing so will not interfere with real-time work. Mische et al. [14] proposed to use SMT to hide context-switch times by using threads to switch task state in and out in the background. Early work on the performance of tasks executed by threads was done by Bulpin [1], Bulpin and Pratt [2], Huang et al. [8], and Tuck and Tulsen [16].

Contribution and organization. We introduce the SMART (Simultaneous Multithreading Applied to Real Time) family of task models and scheduling methods that allow multiple threads to be used for real-time work. The rest of this paper is organized as follows. In Sec. 2, we give information common to all models. In Sec. 3, we give our symbiosis-oblivious model, SMART-SIMPLE, and an associated scheduling method. In Sec. 4, we give our symbiois-aware model, SMART-ADVANCED, and three associated scheduling methods. In Sec. 5, we conclude and discuss future directions for our research.

2 Background

We consider a sporadic, SRT task system τ composed of n tasks. In our model, an SRT system is correctly scheduled if all tasks can be guaranteed to have bounded tardiness. A job's *tardiness* is the difference between its completion time and deadline, if the job completes after its deadline, and zero otherwise. A task's tardiness is the maximum tardiness of any job released by the task. A task system is *SRT-feasible* if there exists some algorithm that can correctly schedule it. We consider the scheduling of τ on a platform π composed of m identical cores. Tasks scheduled on the same core are said to be *co-scheduled.* We assume overhead costs for preemption, task migration, and context-switching are negligible. No task can execute in parallel to itself.

Task execution costs and utilizations are not constant in our models; a task executing on a single thread will take

*Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors.

2576-3172/18/$31.00 ©2018 IEEE
DOI 10.1109/RTSS.2018.00024

longer to execute than the same task would on a whole core. For a given task τ_i, we define E_i^p as the time spent per job executing on a full physical core and E_i^h as the time spent per job on a single thread. If a task never executes on a physical core, $E_i^p = 0$. If a task only executes on a physical core, $E_i^h = 0$. If a task executes on both a physical core and a single thread at different times, neither is 0. Our models require that for every task, both values are constant across all jobs.

On a platform where each core supports h hardware threads, the *effective utilization* of τ_i is given by

$$u_i^E = \frac{E_i^p + \frac{E_i^h}{h}}{T_i}.$$

Effective utilization measures the share of a core used by τ_i. For example, suppose that τ_i has a period of 8. If $E_i^p = 4$ and $E_i^h = 0$, then τ_i uses half a core per period and $u_i^E = \frac{1}{2}$. If $E_i^p = 0$ and $E_i^h = 6$, then τ_i uses a $\frac{3}{4h}$ share of a core per period since it uses $\frac{3}{4}$ of a single thread and each thread is a $\frac{1}{h}$ share of a core. Determining E^p and E^h values that allow each job to complete are discussed in the next two sections.

The *total effective utilization* of τ is given by

$$U^E = \sum_{i=1}^{n} u_i^E.$$

Roughly speaking, τ is feasible if and only if

$$u_i^E \leq 1 \forall i \text{ and} \quad (1)$$

$$U^E \leq m \quad (2)$$

hold for some combination of E^p and E^h values that allows every job to complete.[1] Most of our scheduling decisions involve adjusting E_i^p and E_i^h to minimize U^E subject to (1) while making sure every job will be able to complete.

3 SMART-SIMPLE

In this model, we simplify our problem by considering only whether a task is to execute on a full core or a single thread.

Task model. Every task $\tau_i = (T_i, C_i^p, C_i^h)$ has a period, T_i, a worst-case cost to execute on a physical core, C_i^p, and a worst-case cost[2] to execute on a thread, C_i^h. Every task is either a *physical task* that only executes on a physical core or a *threaded task* that only executes on a thread. For every job to complete, $E_i^p = C_i^p$ must hold for physical tasks and $E_i^h = C_i^h$ must hold for threaded tasks. Therefore, $u_i^E = \frac{C_i^p}{T_i}$ for physical tasks and $u_i^E = \frac{C_i^h}{hT_i}$ for threaded tasks. U^E can then be minimized by having tasks be threaded if

$$C_i^h < hC_i^p \text{ and}$$
$$C_i^h \leq T_i,$$

both hold, or physical otherwise. Making tasks physical or threaded is a preliminary choice that must precede other scheduling decisions.

Example. Let τ consist of four tasks, $\tau_1 = (8, 7, 10)$, $\tau_2 = (4, 1, 4)$, $\tau_3 = (4, 2, 3)$, and $\tau_4 = (8, 4, 6)$ If we consider only physical costs, τ has $U^E = 2.125$ and will require 3 cores. However, on a platform where each core can support two threads, τ_3 and τ_4 can execute as threads with $\frac{C_i^h}{T_i} = .75$ for each as opposed to their physical utilizations of .5 each. Although τ_3 and τ_4 will take longer individually as threads, their combined effective utilization is only .75 when they are threaded, since $h = 2$, as opposed to 1 when they are physical. By making τ_3 and τ_4 threaded, we reduce U^E to 1.875. τ_1 and τ_2 should not execute as threads; τ_1 would be infeasible, since $C_1^h > T_1$, and having τ_2 be a thread would actually increase U^E, since $C_2^h < 2C_2^p$.

While our example considers only two threads per core, the SMART-SIMPLE model can be applied to hardware platforms that support any number of threads per core. We will return to this sample task set throughout the paper.

Implementation. Let τ^p be the set of all physical tasks—τ_1 and τ_2 in our example—and τ^h be the set of all threaded tasks—τ_3 and τ_4 in our example. Scheduling τ^p and τ^h together raises questions about how the two task types should interact. Should a single threaded task preempt a physical task, leaving an $\frac{h-1}{h}$ share of a core unused? If low- and high-priority threaded tasks are running in parallel, should a physical task with intermediate priority preempt both?

We resolve these questions by dividing π into sub-platforms π^p and π^h so that τ^p and τ^h can be scheduled independently. We allow one core to be shared between the sub-platforms. In our example, τ^p has effective utilization of 1.125 and τ^h of .75. We give one core to π^p and allow a shared core to be used by π^p for at least a .125 share of some amount of time X, and by π^h for at least a .75 share of X so that each sub-platform has a core count, including its fraction of the shared core, equal to the effective utilization of the task sub-system it needs to support. Both τ^p and τ^h can then be scheduled independently using GEDF or another method on π^p and π^h, which are effectively *limited-supply* platforms. Information on limited-supply platforms can be found in the work of Leontyev and Anderson [10].

Let m^p and m^h be the number of cores fully owned by π^p and π^h, respectively, and let a^p and a^h be the share each sub-platform gets of the shared core. Let n^p and n^h be the number of tasks in τ^p and τ^h. It can be shown that τ^p is feasible on π^p if and only if

$$\sum_{i=1}^{n^p} \frac{C_i^p}{T_i} \leq m^p + a^p$$

[1] Exceptions to this claim come from scheduling fewer than h threads at a time; if only one threaded task is scheduled, $h - 1$ threads are wasted.

[2] C_i^h is calculated as the worst observed cost when τ_i is executed on a thread while interfering tasks execute on other threads of the same core. While this method may not give a true worst-case cost, Mills and Anderson [13] have shown that average-case execution costs are sufficient for bounded tardiness in SRT.

holds, and that with tasks in τ^h indexed by non-increasing utilization, τ^h is feasible on π^h if and only if

$$\sum_{i=1}^{n^h} \frac{C_i^h}{T_i} \leq h(m^h + a^h) \text{ and}$$
$$\sum_{i=1}^{\min(hm^h+k, n^h)} \frac{C_i^h}{T_i} \leq hm^h + ka^h \; \forall \; k, \; 1 \leq k \leq h,$$

both hold. These conditions are based on feasibility for platforms with different-speed cores (uniform platforms), as described by Funk [5]; intuitively, a limited-supply platform is a very coarse-grained uniform platform.

τ is feasible if and only if it can be partitioned into τ^p and τ^h such that sufficiently sized platforms π^p and π^h can co-exist on π. In many cases, feasibility can be determined without examining every partition of τ. In future work, we will describe these cases precisely so that feasibility can be determined quickly. It can be shown that dividing the platform with one shared core is optimal for this model.

4 SMART-ADVANCED

Task model. If we limit ourselves to two threads per core, we can improve SMART-SIMPLE by modeling every task as $\tau_i = (T_i, C_i, (r_{i:j}))$ with a period T_i, a worst-case cost to execute as a physical task, C_i, and a list of *rates* that indicate how quickly τ_i executes when co-scheduled with τ_j. This model is SMART-ADVANCED. Rates are defined relative to a task's execution with nothing co-scheduled. We define $r_{i:i} = 1$ and $r_{i:j} \leq 1$ for all tasks.[3] $r_{i:j}$ states how much work is done by τ_i in the worst case when co-scheduled with τ_j in one time unit. A task not co-scheduled with another task receives one unit of work per time unit. A job is complete when the amount of work received by the job is equal to the task's cost C_i. For example, if $r_{i:j} = .5$, then τ_i executes at half its normal speed when co-scheduled with τ_j and would take twice as long to execute when co-scheduled with τ_j as it would by itself. We do not assume any relationship between $r_{i:j}$ and $r_{j:i}$.

Example. Returning to our earlier example, we consider the same four tasks with additional information. For example, we might have $\tau_1 = \left(8, 7, \left(1, \frac{7}{10}, \frac{7}{10}, \frac{3}{4}\right)\right)$, $\tau_2 = \left(4, 1, \left(\frac{1}{4}, 1, \frac{1}{2}, \frac{3}{4}\right)\right)$, $\tau_3 = \left(4, 2, \left(\frac{2}{3}, \frac{3}{4}, 1, \frac{4}{5}\right)\right)$, and $\tau_4 = \left(8, 4, \left(\frac{2}{3}, \frac{2}{3}, \frac{3}{4}, 1\right)\right)$.

Semi-aware scheduling. The easiest way to use this additional information is to improve threaded costs by not allowing tasks that dramatically slow down others to be threaded. Looking at the tasks in our example, τ_1 is responsible for the lowest rate, and therefore the worst-case threaded cost, of τ_2 and τ_3. If we declare that τ_1 will only ever be given a full processor, then we can improve the threaded costs from SMART-SIMPLE so that $\tau_2 = (4, 1, 2)$ and $\tau_3 = (4, 2, \frac{8}{3})$ by not considering those tasks' costs when co-scheduled with τ_1. Now, making τ_2 threaded will decrease effective utilization and τ_3 has a lower threaded cost than it did in the SMART-SIMPLE model. Using the new values, we redefine $\tau^p = \{\tau_1\}$and $\tau^h = \{\tau_2, \tau_3, \tau_4\}$, giving $U^E \approx 1.833$, an improvement over $U^E = 1.875$ that we got for SMART-SIMPLE. The task system can be scheduled with the same divided platform scheme that was described for SMART-SIMPLE. We call this method semi-aware as we are aware of all thread interactions during a preliminary partitioning phase, but then ignore them when constructing the actual schedule. Considering individual thread interactions allows us to use less pessimistic worst-cast costs by limiting which threads can interact, thereby preventing some of the worst-case situations from ever happening.

Fully aware scheduling. With our semi-aware scheduling method, we used our knowledge of execution rates to determine which tasks should and should not be threaded, but did not dictate exactly how tasks should be co-scheduled. Here, we create a co-schedule for every task that states how long it should be co-scheduled with every other task. To do so, we define a set of *containers* $\tau_{i:j}$ such that scheduling container $\tau_{i:j}$, $i \neq j$, is equivalent to co-scheduling τ_i and τ_j. Scheduling container $\tau_{i:i}$ is equivalent to scheduling τ_i to execute by itself, with no co-scheduled task. To schedule τ, each container executes once per *frame*, where a frame is an interval of length f. Further information on frames can be found in [18, 19].

All containers have a utilization $0 \leq u_{i:j} \leq 1$, a period of f, and a cost $C_{i:j} = fu_{i:j}$. Defining containers for all $i \leq j$ defines a co-schedule for the entire task system consisting of $\frac{n^2+n}{2}$ distinct containers. The *co-schedule* for τ is defined as the set of containers $\tau_{i:j}$ for all i and all j. We write $\tau_{i:j}$ and $\tau_{j:i}$ as two separate terms to simplify notation, but they are equivalent. For all i and j, $C_{i:j} = C_{j:i}$. If $C_{i:j} = 0$, τ_i and τ_j are never co-scheduled. Tasks τ_i and τ_j are the *parents* of container $\tau_{i:j}$ and $\tau_{i:j}$ is the *child* of τ_i and τ_j. Containers that share a parent are *siblings*. If $\tau_{i:j}$ is scheduled when τ_i but not τ_j is active, then we allow τ_i to execute with no co-scheduled job until either τ_j becomes active or the budget of $\tau_{i:j}$ is exhausted. Since $r_{i:j} \leq 1$ and $r_{i:i} = 1$, this will not cause any job of τ_i to complete later than it would have had τ_j been active.

Correctness. There are two conditions for correctness in fully aware scheduling: 1) every task τ_i must have child containers that will allow τ_i to receive adequate work given that all containers execute once per frame (containers are sufficient) and 2) all containers execute once per frame.[4] Formally, the first condition can be stated as

$$\sum_{j=1}^{n} (u_{i:j} \times r_{i:j}) \geq \frac{C_i}{T_i} \; \forall i. \tag{3}$$

[3] In the rare case that $r_{i:j} > 1$, we can force $r_{i:j} \leq 1$ to hold by assigning a dummy copy of τ_j to execute alongside τ_i as a helper whenever τ_i would otherwise execute alone.

[4] Arbitrarily large containers will be sufficient for all τ_i but impossible to schedule; zero utilization containers can be scheduled correctly, but the parent tasks will make no progress.

Unrestricted containers. It seems like the best overall approach would be to pack tasks into containers such that (3) holds and all containers have utilization of at most 1, and then schedule the resulting containers using standard multicore methods. However, it is with this approach that the difficulties of scheduling on threads become most apparent: if τ_i is split among multiple sibling containers, the containers are not independent, since allowing sibling containers to execute in parallel would mean τ_i executes in parallel, which is forbidden. Therefore, scheduling $\tau_{i:j}$ requires knowing when all siblings of $\tau_{i:j}$ will execute, which requires knowing when the siblings of the siblings will execute, and so forth. Scheduling quickly becomes intractable.

Returning to our example task system, a possible co-schedule for τ is given by $u_{1:1} = .875$, $u_{2:2} = 0.033$, $u_{2:3} = 0.134$, $u_{2:4} = 0.2$, $u_{3:4} = 0.5$, and $u_{i:j} = 0$ for all other i, j. This co-schedule gives $U^E = 1.742$, which is the smallest value of our methods, but difficulties are present even in this small example. None of either $\{\tau_{2:2}, \tau_{2:3}, \tau_{2:4}\}$ or $\{\tau_{2:3}, \tau_{2:4}, \tau_{3:4}\}$ can execute in parallel. As n grows so will the number of containers that cannot be executed in parallel, making scheduling more difficult. Note that since each container is given a full core, we calculate U^E as the sum of the containers' utilizations.

Restricted containers. The difficulties in scheduling containers come from allowing one task to be in multiple containers. If we limit each task to one container, all containers are independent and can be scheduled using existing algorithms. With that in mind, we propose a greedy strategy of placing in one container the pair τ_i, τ_j that will bring the greatest reduction in U^E without giving either task an effective utilization greater than 1. We choose task pairings in this manner until either all tasks have been put into containers or no suitable pairings remain. In the latter case, the remaining tasks are placed into containers of one task each.

The resulting container has two tasks, with each task belonging entirely to one container. Since no two containers include the same parent tasks, containers can be scheduled as independent tasks using existing multi-core scheduling methods. Typically, executing τ_i and τ_j will cause one of them to get work equivalent to its utilization done first. In this case, we allow the remaining task to execute on a full core until it has accomplished work equal to its utilization.

For our sample task system, the best pairing is τ_3 and τ_4. We create container $\tau_{3:4}$. Setting $u_{3:4} = .625$ makes $\tau_{3:4}$ sufficient for τ_3 but not τ_4. To make $\tau_{3:4}$ sufficient for τ_4,, we increase its utilization by .03125 time units to .65625 and allow τ_4 to execute alone in the additional container space provided. To make containers sufficient, we can view τ_4 as being a parent to two separate containers, $\tau_{3:4}$ and $\tau_{4:4}$, but to prevent the task parallelism problems we encountered with unrestricted containers we schedule $\tau_{3:4}$ and $\tau_{4:4}$ as one unit. Pairing τ_1 and τ_2 will not reduce U^E further. Therefore, each gets its own container and is allowed to execute on a full core, giving $U^E = 1.78125$. This value is slightly higher than that for unrestricted containers, but the resulting set of containers can be scheduled using existing theory.

5 Conclusion

In experiments with two threads per core, we have found that tasks running as threads have worst-case costs consistently less than twice their normal worst-case physical costs. Based on our data, even our simplest model, SMART-SIMPLE, is an improvement over ignoring multithreading. Moving forward, we expect to focus on developing algorithms to classify tasks as threaded or physical in the semi-aware method. We plan to conduct a schedulability study to demonstrate the impact of combing real time and SMT.

References

[1] J. Bulpin. *Operating system support for simultaneous multithreaded processors*. PhD thesis, University of Cambridge, King's College, 2005.

[2] J. Bulpin and I. Pratt. Multiprogramming performance of the pentium 4 with hyperthreading. In *Third Annual Workshop on Duplicating, Deconstruction and Debunking*, June 2004.

[3] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors: synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799, July 2006.

[4] 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, Sept 1997.

[5] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. In *RTSS '01*.

[6] 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.

[7] T. Gomes, S. Pinto, P. Garcia, and A. Tavares. RT-SHADOWS: Real-time system hardware for agnostic and deterministic OSes within softcore. In *ETFA '15*.

[8] W. Huang, J. Lin, Z. Zhang, and J.M. Chang. Performance characterization of Java applications on SMT processors. In *ISPASS '05*.

[9] R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *RTSS '02*.

[10] H. Leontyev and J. H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1):26–71, Mar 2010.

[11] S. Lo, K. Lam, and T. Kuo. Real-time task scheduling for SMT systems. In *RTCSA'05*.

[12] D. Marr, F. Binns, D. Hill, G. Hinton, K. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. In *Intel Technology Journal*, volume 6, pages 4–15, Feb 2002.

[13] A. F. Mills and J. H. Anderson. A stochastic framework for multiprocessor soft real-time scheduling. In *RTAS '10*.

[14] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Using SMT to hide context switch times of large real-time tasksets. In *RTAS '10*.

[15] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS '2000*.

[16] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *PACT '03*.

[17] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA '95*.

[18] S. Voronov and J. H. Anderson. An optimal semi-partitioned scheduler assuming arbitrary affinity masks. In *RTSS '18*.

[19] K. Yang and J. H. Anderson. An optimal semi-partitioned scheduler for uniform heterogeneous multiprocessors. In *ECRTS '15*.

[20] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *RTAS '14*.