On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler[*]

John M. Calandrino and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

Multicore architectures, which have multiple processing units on a single chip, have been adopted by most chip manufacturers. Most such chips contain on-chip caches that are shared by some or all of the cores on the chip. Prior work has presented methods for improving the performance of such caches when scheduling soft real-time workloads. Given these methods, two additional research issues arise: (1) *how to automatically* profile *the cache behavior of real-time tasks within the scheduler; and* (2) *how to* implement *scheduling methods efficiently, so that scheduling overheads do not offset any cache-related performance gains. This paper addresses these two issues in an implementation of a cache-aware soft real-time scheduler within Linux, and shows that the use of this scheduler can result in performance improvements that directly result from a decrease in shared cache miss rates.*

1 Introduction

Multicore architectures, which contain multiple processing cores on a single chip, have been adopted by most chip manufacturers. Dual-core chips are commonplace, and numerous four- and eight-core options exist. In the coming years, per-chip core counts will continue to increase—for example, Sun plans to ship its 16-core "Rock" processor by the end of 2009 [24], and Intel has claimed that it will release 80-core chips as early as 2013 [1]. The shift to multicore technologies is a watershed event, as it fundamentally changes the "standard" computing platform in many settings to be a multiprocessor.

In most multicore platforms, different cores share on-chip caches. Without effective management by the scheduler, such caches can cause thrashing that severely degrades system performance. In fact, the issue of efficient cache usage on multicore platforms is one of the most important problems with which chip makers are currently grappling. In this paper, we address this issue in the context of soft real-time systems implemented on a multicore platform where all cores are symmetric and share the lowest-level cache, as shown in Fig. 1, where all cores share an L2 cache. This architecture is fairly common—the Sun UltraSPARC T1 and T2 processors contain L2 caches shared by eight cores, and the recently-released Intel Core i7 chip contains an L3 cache shared by four cores (all higher-level caches in both chips are per-core and therefore not shared).

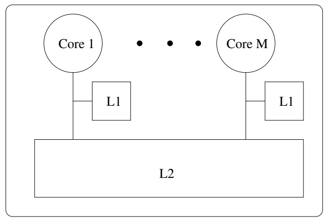


Figure 1: Multicore architecture.

In this paper, we assume that systems are organized into *multithreaded tasks* (MTTs), where each MTT consists of periodic (sequential) tasks, which may have different execution costs but a common period. MTTs are useful for specifying groups of *cooperating* tasks that reference a common set of data. (Note that an ordinary periodic task is just a "single-threaded" MTT.) MTTs arise naturally in many settings. For example, multiple threads could perform different functions on the same video frame with a common period implied by the desired frame rate. Abstractions such as MTTs allow concurrency within task models that typically handle only the sequential execution of tasks. This is important because as per-chip core counts increase, the processing power of individual cores is likely to remain the same (or decrease if cores become simpler); thus, MTTs should be useful for achieving performance gains.

In prior work [7], we explored methods for improving the performance of shared caches when scheduling soft real-time workloads by influencing *co-scheduling* choices. Namely, co-scheduling is *encouraged* for tasks within the same MTT, and *discouraged* for tasks (within different MTTs) when it would cause shared cache thrashing. Co-scheduling is influenced through *job promotions*, wherein a job is given a temporary increase in priority by moving its deadline to the current time. As determining how and when to promote jobs to improve cache performance is not straightforward, a large number of heuristics, representing different sets of scheduling policies, were evaluated in [7] within an architecture simulator. One of these heuristics was found to be particularly effective at improving system performance for a wide variety of task sets. This heuristic is of primary concern to us in this paper, and is discussed in detail in Sec. 3.

[*]Work supported by IBM, Intel, and Sun Corps., NSF grants CNS 0834270, CNS 0834132, and CNS 0615197, and ARO grant W911NF-06-1-0425. The first author was also supported by a University of North Carolina Dissertation Completion Fellowship.

Contributions. This paper complements the work in [7] by addressing two issues: (1) how to automatically *profile* the cache behavior of real-time tasks within the scheduler, so that offline profiling tools are not required; and (2) how to *implement* the aforementioned heuristic efficiently, so that scheduling overheads do not offset any cache-related performance gains. We address these issues in an implementation of a cache-aware scheduler within Linux, which allows the heuristic to work well in practice. Our scheduler often achieves substantially better cache performance than global earliest-deadline-first (G-EDF) scheduling, and this translates into better overall system performance. Note that *partitioned* scheduling approaches, wherein tasks are statically assigned to processors, and scheduling is performed independently at each processor, are not considered in this paper—achieving system-wide control over a global resource (such as a shared cache) is quite difficult when each processor is scheduled independently.

Our results, presented in Sec. 4, suggest that cache performance should be treated as a first-class concern when designing both real-time scheduling algorithms for multicore platforms with shared caches, *and the multicore hardware on which such algorithms run*. Cache performance improvements may reduce task execution costs or allow more useful computation to be performed. This could be beneficial in many settings. For example, within a multimedia server that encodes live media streams, higher-quality video, or a greater selection of videos, could be supported without upgrading hardware. Alternately, the cost of specially-designed hardware (*e.g.*, within video game consoles) could be reduced, as a smaller cache or a less powerful processor may be acceptable when they are better utilized. As interest in providing real-time support within general-purpose operating systems (*e.g.*, Linux) increases, and multicore platforms become increasingly ubiquitous within many of the hardware domains on which such operating systems run, a state-of-the-art real-time scheduler will have to address the needs of multicore platforms to remain relevant.

Related work. Prior work has explored issues related to cache-aware real-time scheduling and WCET analysis (*e.g.*, in [10, 11, 3, 15, 20, 21, 22, 19]). None of this work addresses the question of how to efficiently profile the cache behavior of real-time tasks during execution. Also closely related is work on *symbiotic scheduling* [14, 18, 23], which concerns scheduling when multiple hardware threads contend for shared resources within the same core; however, such work lacks analysis for validating real-time constraints.

Organization. The rest of this paper is organized as follows. Sec. 2 presents an overview of our task model and other background information. Sec. 3 introduces our cache profiler and the details of our implementation, and the heuristic from [7] considered herein. Sec. 4 presents an evaluation of our scheduler in terms of cache performance, profiler accuracy, and overheads for a variety of workloads, and presents a study that explores multimedia application performance. Sec. 5 concludes.

2 Background

In this section, we briefly introduce our task model as related to MTTs and other background information. For simplicity, we consider only periodic task systems, though our results apply to sporadic task systems as well.[*]

In a periodic task system τ, each task T releases successive *jobs* $T_1, T_2, \ldots$, and is characterized by a worst-case per-job *execution cost* $e(T)$ and a *period* $p(T)$. Every $p(T)$ time units, starting at time 0, T releases a new job with an execution cost of $e(T)$ time units. In this paper, all task periods are an integral multiple of the quantum length, as job releases are handled by our scheduler at quantum boundaries. The quantity $e(T)/p(T)$ is called the *utilization* of T, denoted $u(T)$.

The *deadline* $d(T_k)$ of a job T_k coincides with the release time of job T_{k+1}. If job T_k completes its execution after time $d(T_k)$, then it is *tardy*. For some scheduling algorithms, tardiness may be bounded by some amount B, meaning that any job T_k will complete execution no later than time $d(T_k)+B$.

Our goal is to schedule on M processors (or cores) a set of periodic tasks of total utilization at most M, where some tasks correspond to threads within an MTT. We assume that each MTT has at most M threads, the maximum parallelism achievable on M cores.

G-EDF scheduling. In this paper, G-EDF scheduling is used as a baseline for evaluating the performance of our cache-aware scheduler in Sec. 4. In G-EDF scheduling, jobs are scheduled in order of increasing deadlines, with ties broken arbitrarily. We use G-EDF since prior work has shown that it results in better schedulability for soft real-time systems than other approaches [8]. G-EDF is *not* optimal, so tasks may miss their deadlines; however, deadline tardiness under G-EDF is bounded [9].

Tardiness bounds under global scheduling. In recent work [17], the G-EDF tardiness-bound proof in [9] was extended to apply to a wide variety of global scheduling algorithms. In this work, *priority points* are assigned to every eligible job, with earlier priority points denoting higher priority. For example, under G-EDF, the priority point of each job is its deadline. If the priority point of every job is within a window bounded by its release time and deadline, then job priorities are *window-constrained*. It is shown in [17] that under any global scheduling algorithm with window-constrained priorities, deadline tardiness is bounded provided the system is not over-utilized, *even if the priority point of a job moves*

[*]In sporadic task systems, task periods *within* an MTT must coincide. We consider this to be reasonable since we intend an MTT to represent a single recurrent real-time computation.

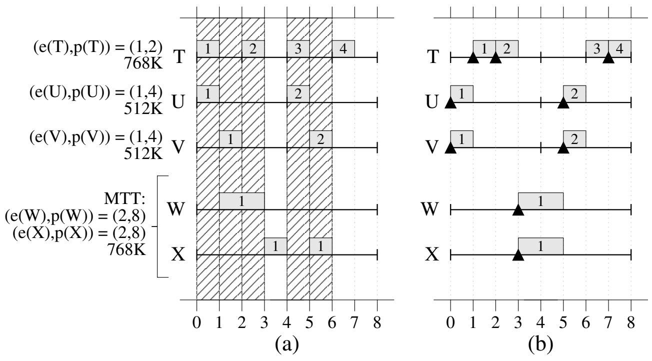


Figure 2: Two-core schedules for a set of five tasks. Tasks are scheduled using **(a)** G-EDF and **(b)** the heuristic studied herein. WSSs are shown along with task execution costs and periods. Thrashing occurs during "hatched" quanta, assuming a shared cache size of 1 MB. The number within each scheduled piece of work denotes the corresponding job index.

arbitrarily within its window. (Such a guarantee is not possible under partitioned scheduling.) This result guarantees that the heuristic considered herein, which promotes jobs to influence co-scheduling, ensures bounded deadline tardiness.

3 Cache-Aware Scheduler

In this section, we describe each component of our cache-aware real-time scheduler. Our target operating system is Linux, as it has become increasingly popular for supporting soft real-time applications. Our implementation efforts were conducted within LITMUS^{RT} [8], a Linux-based testbed produced by our research group at UNC that supports multiprocessor real-time scheduling policies within the Linux operating system. The most recent publicly-available version is a patch against Linux 2.6.24 [25].

The rest of this section is organized as follows. Sec. 3.1 presents an overview of the scheduling heuristic that we implement in this paper, including a discussion of its benefits over other heuristics from [7]. Sec. 3.2 presents our cache profiler, which collects data about tasks during execution.

3.1 Scheduling Heuristic

As stated earlier, one of the heuristics in [7] was found to improve system performance over G-EDF for a wide variety of task sets, and it is this heuristic that we implement in our scheduler. We introduce this heuristic with an example, shown in Fig. 2—the schedule generated by the heuristic is shown in inset (b). In the heuristic, cache impact is determined by examining *per-job working set sizes* (WSSs), which are specified for each MTT. The per-job WSS of an MTT indicates the amount of memory referenced by all tasks of that MTT while executing one "job" of the MTT, where the i^{th} job of an MTT consists of the i^{th} jobs of all tasks in the MTT. Shared cache thrashing is assumed to occur during a quantum if the sum of the WSSs of all MTTs with jobs scheduled in that quantum exceeds the shared cache size. For the time being, we simply assume that the heuristic has accurate WSS information—the profiler described in Sec. 3.2 is responsible for providing this information. In the example considered here, tasks W and X belong to the same MTT, so for every i, jobs W_i and X_i reference the same working set and would benefit from being co-scheduled. Also, a shared cache size of 1 MB is assumed.

Cache performance is improved by influencing co-scheduling choices through job promotions that move the deadline of a job to the current time—these promotions are indicated by black triangles in Fig. 2(b). The heuristic makes scheduling decisions iteratively over all cores at every quantum boundary—even when jobs are promoted, jobs that have already been scheduled are unaffected. Decisions are made by promoting the eligible job with the smallest WSS, as indicated by the WSS of its respective MTT, and then scheduling jobs in EDF order. This is first shown at time 0 in Fig. 2(b), where jobs U_1 and V_1, each with a 512K WSS, are promoted and scheduled.

At time 1, job T_1 is scheduled. If we chose to also schedule either job W_1 or X_1, then thrashing would occur. We avoid thrashing in this case by idling the second core, which reduces parallelism. The number of times that we can safely idle cores (so that bounded deadline tardiness can be ensured by applying the results in [17]) is a function of the total utilization of the task set. At time 2, job T_2 is scheduled, and the second core is idled again. At time 3, jobs W_1 and X_1, both belonging to the same MTT, are scheduled. Since they share the same working set, thrashing does not occur.

Cache affinity. At time 4 in Fig. 2(b), jobs W_1 and X_1 are again scheduled, even though jobs U_2 and V_2, which have smaller WSSs, were released. This demonstrates a change that we made to the heuristic from [7]—a job remains promoted until it has finished execution, rather than being immediately demoted when it is scheduled. This policy maintains cache affinity by strongly encouraging both non-preemptive execution and MTT co-scheduling. Only tardy jobs can cause preemptions or interfere with co-scheduling. This is necessary to ensure bounded deadline tardiness. This change also makes it easier to implement the heuristic efficiently.

Tardy jobs. At time 5, jobs U_2 and V_2 are promoted and scheduled, and job T_3 misses its deadline as a result. Jobs are only scheduled using their deadlines when they become tardy. Tardy jobs are not promoted, but always have priority over all non-tardy jobs, including promoted jobs. This ensures that tardy jobs are scheduled first, in EDF order, and non-tardy jobs are then scheduled in the order that they are promoted (*i.e.*, smallest WSS first). *Our scheduler ensures bounded tardiness for any task set with utilization at most* *the number of cores*—the same guarantee is provided under G-EDF. At time 6, the tardy job T_3 is scheduled, and would have had priority over any promoted job at that time, had such a job existed.

Finally, at time 7, the remaining job T_4 is scheduled. We can see from comparing insets (a) and (b) in Fig. 2 that the heuristic results in improved cache performance—thrashing occurs 62.5% of the time under G-EDF, and is eliminated entirely by the heuristic.

Implementation efficiency. Interestingly, this heuristic is one of the easiest to implement efficiently in practice, by maintaining two separate run queues for eligible jobs: one that is EDF-ordered, and a *promotion queue* in which eligible jobs are arranged in the order that they would be promoted. Jobs in the promotion queue are ordered from smallest to largest WSS, with promoted jobs remaining "pinned" at the front of the queue regardless of their WSS and future job releases. The heuristic schedules the job at the head of the EDF-ordered queue if it is tardy; otherwise, it peeks at the job at the head of the promotion queue and determines whether scheduling it will cause thrashing. This requires maintaining a running total of the WSSs of all MTTs with jobs scheduled thus far in the quantum—this total is referred to as the amount of *utilized* cache. If the amount of utilized cache plus the WSS of the job to schedule exceeds the shared cache size, then it is assumed that thrashing would occur. (Note that the job WSS is zero if a job from its MTT is already scheduled.) If thrashing would occur, and we can safely idle a core,[†] then the core is idled; otherwise, the job is scheduled. As both the "real" deadline and WSS of each job is fixed over its entire execution, the overhead incurred to maintain these run queues is relatively small.

Other heuristics in [7] are considerably less feasible to implement, since the ordering of the promotion queue depends on factors that change as scheduling decisions are made. As a result, jobs in the queue may need to be frequently reordered, resulting in high queue-maintenance overheads. For example, most of the remaining heuristics in [7] choose a job to promote based on the amount of utilized cache, defined earlier. Scheduling decisions typically cause the amount of utilized cache to change, requiring the promotion queue to be reordered. The higher overheads associated with these heuristics make them less practical from an implementation standpoint, and they are not considered further in this paper.

3.2 Cache Profiler

Our profiler provides a per-job WSS estimate for each MTT, which is required when making scheduling decisions using the heuristic described above. We profile MTTs rather than tasks since MTTs share a common working set. The profiling occurs during job execution, eliminating the need for an offline profiling tool; however, as the profiler is essentially part of the scheduler, it must be efficient.

[†]We can idle a core if an eligible *phantom task* exists. Phantom tasks are an abstraction from [7] that allows cores to be idled in a controlled way, so that bounded deadline tardiness can be ensured by applying the results in [17].

WSS as a cache behavior metric. WSS may be seen as an oversimplification of cache behavior; however, we have found it to work well for small intervals (*e.g.*, the execution time of a job). Further, it is usually the easiest metric to approximate efficiently given current hardware. Assuming a fully-associative cache (or high set-associativity, *i.e.*, eight ways or more—see [12]) so that conflict misses are avoided, the profiler, described next, can result in significant improvements in cache performance over G-EDF when used within our cache-aware scheduler, as we will see in Sec. 4.

Performance counters. Shared cache misses for each MTT are recorded by the profiler using performance counters. Performance counters are available in many processors today, and can be programmed to monitor a wide variety of events. For example, Intel has specified a set of seven *architectural performance events* that can be monitored using performance counters in most current and future Intel processors [13]. These events allow core clock cycles, retired instructions, lower-level cache misses and references, and events related to branching to be counted. (Many more events also exist that are specific to a particular type of processor.)

Typically, a separate set of performance counters is available for each core, and can be programmed to track events originating from that core. We programmed a counter at each core to track lower-level (shared) cache misses. Since jobs execute sequentially, we can measure the number of cache misses incurred for a job by resetting the counter to zero at the start of execution, and recording the total misses observed by the counter upon completion. The observed misses can then be used to calculate a per-job WSS estimate. Since accessing program counters and recording data are low-overhead operations, and computed WSS estimates are cached to minimize computation, the overhead of the profiler is relatively low.

Assumptions. The profiler as stated requires several assumptions.

(1) Each job of the same task is assumed to perform roughly the same operations on different (but similarly-sized) sets of data. This has two implications: (a) the i^{th} job of task T and the j^{th} job of task U, where T and U belong to the same MTT, do not share significant data unless $i = j$ (even if $T = U$); and (b) the per-job WSS of an MTT remains approximately the same over all jobs.

(2) Profiled jobs are not preempted and do not cause shared cache thrashing.

We consider assumption (1) to be in line with other work (*e.g.*, [22]), and natural for certain types of (multimedia) applications. To ensure assumption (2), we discard measurements obtained for jobs that are preempted, or for which cache thrashing occurred at some time during their execution. Thrashing is assumed to have occurred if for some quantum in which a job is scheduled, the sum of the WSSs of all MTTs with jobs scheduled in that quantum exceeds the shared cache size (the same approach taken in the heuristic). For MTTs, measurements for the i^{th} job of *all* tasks in the MTT must be discarded if the i^{th} job of *any* task in the MTT was preempted or caused thrashing. Assumption (2) also implies that we are not interested in profiling MTTs with per-job WSSs greater than the size of the shared cache, which would thrash the shared cache even if scheduled in isolation. Further, assumption (2) ensures that the number of capacity misses observed over non-discarded jobs is negligible.

Estimating MTT WSS. The above assumptions allow us to compute over all (non-discarded) jobs an average per-job MTT WSS, which we use as our per-job WSS estimate for an MTT. This can be computed by dividing the total cache misses observed over all profiled jobs by the total number of profiled jobs for an MTT, and multiplying the result by the cache line size. (Profiling the i^{th} job of an MTT requires profiling the i^{th} job of all tasks in the MTT, and recording the total misses observed for all jobs.) We can conclude from our assumptions that the first reference to a particular line of data by a job will almost always be the only reference that results in a cache miss, and that miss will be compulsory, since data brought into the cache should not be evicted during job execution. Thus, the aforementioned computation results in an estimate of the cache "footprint" of an MTT, which we use as a first approximation of WSS.

As an example, consider an MTT with two tasks that process video frames (loaded into main memory), each of which is 128K in size. We would expect a per-job WSS for this MTT that is slightly greater than 128K (to account for instructions, loop variables, *etc.*). When job 1 of the MTT is profiled, cache miss counts of 997 and 1,242 are recorded for each task, for a total miss count of 2,239. Next, job 2 of the MTT is discarded due to preemptions or thrashing. Finally, when job 3 of the MTT is profiled, counts of 1,072 and 1,203 are recorded for each task, bringing the total miss count to 4,514 over both profiled jobs. As a result, assuming a 64-byte cache line size, the WSS estimate that our profiler would produce before job 4 begins execution is $(4,514/2)\cdot 64 = 144,448$ bytes, or a WSS of roughly 141K, which seems reasonable.

A more accurate WSS estimate would require information about cache reuse, which could be provided online with additional hardware support. If cache reuse is high, then our profiler will overestimate the MTT WSS. This is undesirable, but such an estimate can still be used to prevent thrashing. In fact, the cache footprint of an MTT may be *more* useful than WSS in preventing thrashing, as cache interference may occur whenever a line is brought into the cache, regardless of whether that line is truly part of the working set of an MTT.

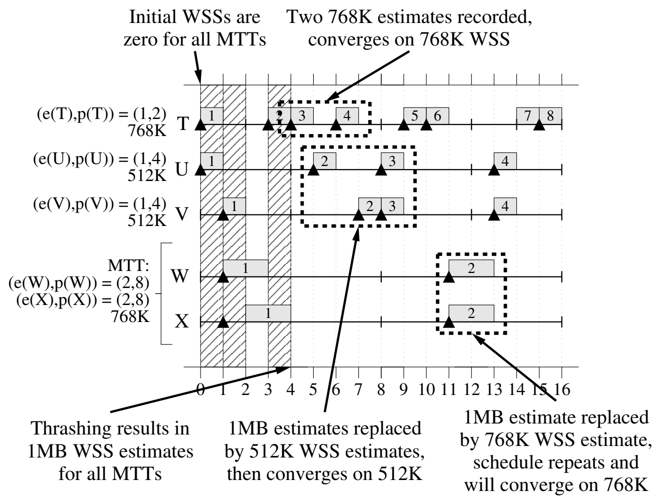

Initial WSSs are zero for all MTTs
Two 768K estimates recorded, converges on 768K WSS
(e(T),p(T)) = (1,2) 768K
T
(e(U),p(U)) = (1,4) 512K
U
(e(V),p(V)) = (1,4) 512K
V
MTT: (e(W),p(W)) = (2,8) (e(X),p(X)) = (2,8) 768K
W
X
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Thrashing results in 1MB WSS estimates for all MTTs
1MB estimates replaced by 512K WSS estimates, then converges on 512K
1MB estimate replaced by 768K WSS estimate, schedule repeats and will converge on 768K

Figure 3: Two-core example schedule generated using the heuristic and task set from Fig. 2 when our profiler is used. As before, thrashing occurs during "hatched" quanta, assuming a shared cache size of 1 MB. WSS estimates over time for each MTT are noted.

Bootstrapping the profiler. Before any measurements have been obtained for an MTT, it is impossible to know when to discard job measurements due to thrashing, since we do not have the data necessary to compute a WSS estimate. This makes it difficult to guarantee assumption (2). To circumvent this issue, profiling begins with a *bootstrapping* process, illustrated with an example in Fig. 3 (which uses the same task set from Fig. 2). Assume for the sake of simplicity that a job that causes thrashing results in a WSS estimate of exactly 1 MB being generated by the profiler (for correct operation of the heuristic, all WSS estimates are capped at the size of the shared cache), while a job that does not cause thrashing results in an accurate estimate being generated, which is equal to its MTT WSS specified in Fig. 3.

In Fig. 3, all MTTs have a WSS of zero at time 0, which is the WSS assigned to an MTT until measurements are recorded for its first profiled job. At time 0, the heuristic (which is not very effective when all WSSs are zero) co-schedules jobs T_1 and U_1, resulting in thrashing; at time 1, thrashing also occurs when jobs V_1 and W_1 are scheduled. The promotion and scheduling of job W_1 also causes job X_1 to be promoted by the heuristic; both jobs remain promoted until they complete execution. Job W_1 completes execution at time 3, allowing job T_2 to be scheduled. By time 4, thrashing has occurred during the execution of all profiled jobs, resulting in a 1 MB WSS estimate being generated for every MTT.

This WSS overestimation causes MTTs to be scheduled in isolation from time 4 until time 8. When jobs are profiled during this time, thrashing does not occur, and we would expect accurate WSS estimates from our profiler. Due to this expectation, we initially only use measurements from the most recently profiled job to compute WSS estimates during bootstrapping, instead of computing an average WSS over all profiled jobs in the way described earlier. Thus, at times 5, 6, and 8, the profiler computes WSS estimates for tasks T, U, and V using measurements from completed jobs T_3, U_2, and V_2, respectively, and discards earlier measurements. This results in accurate WSS estimates for tasks T, U, and V.

WSS estimates for an MTT continue to be based solely on the most recently profiled job until its estimates have *converged*. This occurs when the difference between the estimates for two consecutive profiled jobs of an MTT drops below some threshold level. This first occurs at time 7 for task T, when the profiler generates the same 768K WSS estimate for jobs T_3 and T_4. From time 7 onward, WSS estimates for task T are computed as averages over all successive profiled jobs. Note that if WSS estimates are capped for consecutive jobs (*e.g.*, due to thrashing), we do not consider such estimates to have converged—this is why the WSS estimates converged for task T at time 7 instead of time 4.

The last eight time units of the schedule are identical to the schedule in Fig. 2(b). At time 9, jobs U_3 and V_3 complete execution, and since thrashing does not occur, the correct WSS estimates are generated a second time for both tasks U and V, resulting in converging estimates. At time 13, jobs W_2 and X_2, both from the same MTT, complete without thrashing, which allows the first accurate WSS estimate for that MTT to be generated. The last eight time units of the schedule repeat indefinitely; thus, thrashing is avoided from time 4 onwards. As a result, the WSS estimates of the MTT containing tasks W and X converge at time 21 (not shown), after the completion of jobs W_3 and X_3, at which time all WSS estimates have converged.

Related profiling work. Prior work has investigated the use of performance counters to improve cache performance for throughput-oriented tasks [16]. Additionally, other research in the real-time domain [19] has used performance counters to record per-task cache misses during execution; however, the results were used for WCET analysis rather than to evaluate the cache behavior of tasks for the purposes of online scheduling, as is the case in this paper.

4 Experimental Results

We now evaluate our cache-aware scheduler in terms of both profiler accuracy and its performance as compared to G-EDF. Our experimental platform consists of one quad-core Intel Core i7 processor running at 2.66 GHz. The Core i7 currently represents the state-of-the-art for released Intel multicore chips (it became publicly available in November 2008), and is the first general-purpose chip by Intel where four cores share a single low-level cache.

The Core i7 processor contains four cores. Each core contains private 32K L1 instruction and data caches, and a unified private 256K L2 cache. All cores also share an 8 MB on-chip L3 cache. The L1 instruction, L1 data, L2, and L3 caches are 4-way, 8-way, 8-way, and 16-way set associative, respectively, and all caches have a 64-byte line size. Hyper-threading is supported, but disabled in our experiments as it may result in timing anomalies related to when each hardware thread is allowed access to core resources. The machine has 4 GB of main memory.

As stated in Sec. 3, we implemented our scheduler within LITMUS$^{\mathrm{RT}}$. LITMUS$^{\mathrm{RT}}$ contains a G-EDF implementation, which was used in these experiments. Also, the default quantum length in LITMUS$^{\mathrm{RT}}$ is 1 ms, which we did not change. In both G-EDF and our scheduler, performance counters were programmed so that the total number of shared cache misses and references could be recorded for each MTT; this allowed cache miss rates to be determined.

The rest of this section is organized as follows. In Sec. 4.1, we determine the accuracy of our profiler for MTTs with known memory reference patterns and WSSs. Then, in Sec. 4.2, we compare our cache-aware scheduler to G-EDF in terms of cache performance, deadline tardiness, and scheduling overheads. Finally, in Sec. 4.3, we evaluate the performance of our scheduler as compared to G-EDF for a multimedia server workload.

4.1 Accuracy of WSS Estimates

We first determine how well our profiler estimates MTT WSSs. In these experiments, the per-job WSS is known for each MTT. To determine profiler accuracy for a given MTT, we compared the WSS estimate generated by our profiler to its known WSS. Since the known WSS does not account for instructions and bookkeeping variables, we would expect our estimates to be slightly higher than the known WSS values. (Similar reasoning was applied in an example in Sec. 3.2.)

We generated task sets with the following parameters.

- **System utilization:** Between 55% and 65%, assuming negligible scheduling overheads.
- **MTT periods:** Between 20 and 2,400 ms (some values removed to avoid arithmetic overflow). Larger periods were necessary to allow large per-job WSSs under certain memory reference patterns.
- **MTT utilizations:** Uniform over [0.01, 0.1], [0.1, 0.4], [0.5, 0.9], or [0.01, 0.9]; or bimodal, with a 50% probability of being distributed over [0.01, 0.1] or [0.5, 0.9]. Each task set generated with the bimodal distribution was required to have at least one MTT from each of the two utilization ranges.

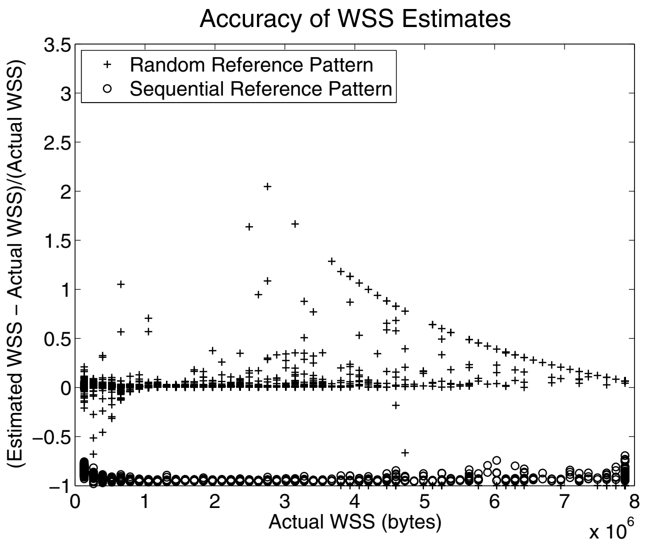


Figure 4: Profiler accuracy as a function of MTT WSS for random and sequential memory reference patterns, expressed as the proportionate error in the WSS estimates generated by the profiler.

- **MTT execution costs:** Derived from periods and utilizations. All tasks within an MTT have the same execution cost, which is at least 3 ms. Jobs are *not* backlogged, making exactly three passes over their working sets.
- **MTT thread counts:** Uniform over $[1, 4]$.
- **MTT per-job data WSSs**: Generated as a function of execution cost. For an MTT with execution cost e, its WSS was set to $min(\lfloor e/3 \rfloor \cdot 128\text{K}, 7.5\text{ MB})$. Due to the lower bound on execution cost, the minimum MTT WSS was 128K.

For each combination of the above parameters, we scheduled and profiled 100 task sets, where either a sequential or random memory reference pattern was employed by all jobs. All task sets were executed for one minute, and for each MTT, the WSS estimate produced at the end of the minute was compared to the known WSS. Overall, this resulted in 2,700 MTTs being profiled under each memory reference pattern.

Results. Fig. 4 shows the proportionate error in the WSS estimates generated by the profiler, when compared to the known WSS of each MTT. Since we found little difference in profiler accuracy when varying MTT utilizations, results are presented as a function of the actual WSS of each MTT.

For random reference patterns, our profiler is typically accurate. Exceptions are clearly shown in Fig. 4, and become more frequent at larger WSSs. This is because, as WSS increases, it becomes more difficult for the bootstrapping process (from Sec. 3.2) to result in WSS estimates that converge, due to the large number of discarded measurements resulting from shared cache thrashing; however, note that error again decreases as WSSs approach the shared cache size, since WSS estimates are capped at that size by the profiler. In any case, the exceptions are extreme outliers; the maximum observed error is under 10% and 5% for over 91% and 78%, respectively, of the MTTs that were profiled.

Hardware limitations. For sequential reference patterns, our profiler is extremely inaccurate, producing estimates that are close to zero for all profiled MTTs. This suggests that cache misses are being underreported by the performance counters. This is due to the hardware prefetcher, which attempts to anticipate the data needs of a core and fetch data earlier than it is needed; this reduces the perceived latency of referencing data and improves core utilization. The prefetcher is very likely to be triggered frequently by a sequential memory reference pattern, while a random pattern will trigger it rarely. Per-core shared cache misses related to this prefetcher are *not included* as part of any performance event in the Core i7, and the prefetcher cannot be disabled for experimental purposes. Thus, such underreporting cannot be avoided on our Core i7 processor.

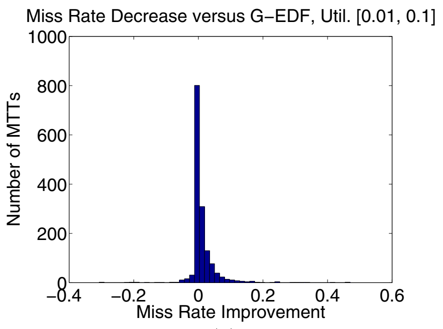
Interestingly, both the ability to count prefetching-related events (via a non-architectural performance event) and disable the prefetcher *were* available in most Core 2 chips, which preceded the Core i7. (The ability to disable prefetching is also available in some in-house versions of the Core i7 at Intel, but the feature has been locked, if present at all, in the commercial versions of the chip.) The existence of these features in earlier Intel processors suggests that it is not unreasonable to expect them. We believe that the loss of these features is a step in the wrong direction, as it makes shared cache management more difficult (and not just for real-time applications).

Regardless of these hardware limitations, the experimental results for random reference patterns show that *our concepts are sound*—given sufficient *hardware support* to accurately count shared cache misses, our profiler is often quite accurate. We shall see next that system performance can be improved over G-EDF when our cache-aware scheduler, which includes the profiler, is used.

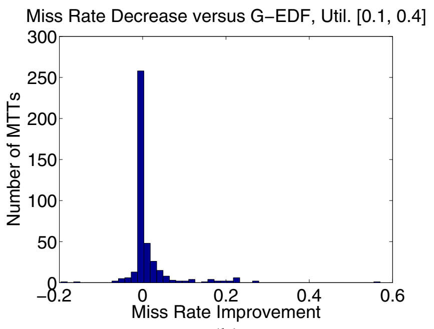
4.2 Performance Versus G-EDF

In this next set of experiments, the same task sets generated in Sec. 4.1 were scheduled under both our scheduler and G-EDF, again for one minute. In these experiments, only the random reference pattern was employed, since the profiler was inaccurate for sequential reference patterns due to the hardware limitations described earlier. MTTs were compared on the basis of several performance metrics under both our scheduler and G-EDF. These performance metrics are: cache miss rate, deadline tardiness, and scheduling overheads. Results related to each metric are presented next.

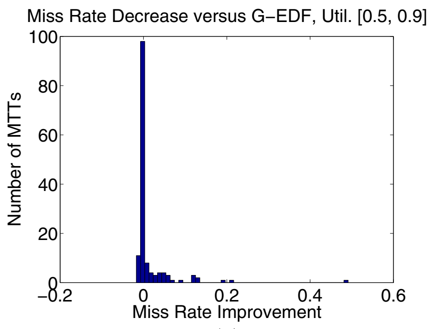
Average-case cache performance. Fig. 5 presents differences in cache performance under G-EDF and our sched-



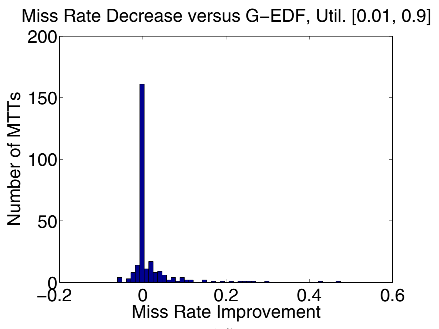
(a)



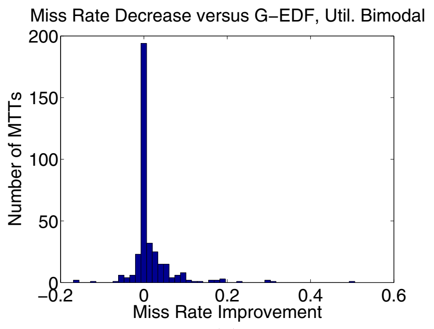
(b)



(c)



(d)



(e)

Util. Dist.	Min.	Avg.	Max.
[0.01, 0.1]	-0.308	0.013	0.470
[0.1, 0.4]	-0.196	0.016	0.570
[0.5, 0.9]	-0.017	0.015	0.491
[0.01, 0.9]	-0.061	0.019	0.475
Bimodal	-0.169	0.016	0.506

Util. Dist.	$x < -0.01$	$\|x\| \leq 0.01$	$x > 0.01$
[0.01, 0.1]	5.41%	61.39%	33.20%
[0.1, 0.4]	8.45%	65.46%	26.09%
[0.5, 0.9]	4.83%	74.48%	20.69%
[0.01, 0.9]	8.21%	65.30%	26.49%
Bimodal	9.22%	60.06%	30.73%

(f)

Figure 5: The decrease in the shared cache miss rate over each utilization distribution, as compared with G-EDF. A positive value indicates a performance increase over G-EDF, whereas a negative value indicates a decrease. Insets **(a)** through **(e)** present histograms for each distribution, while inset **(f)** presents statistics for each histogram.

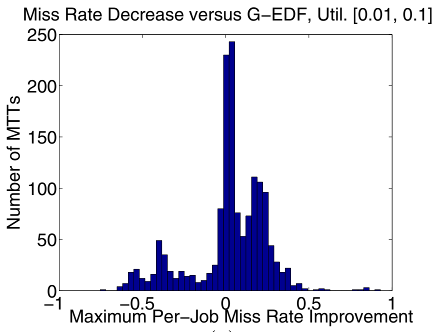
uler, for each utilization distribution. Insets (a) through (e) present histograms for each distribution indicating the (absolute, not relative) improvement in the average cache miss rate for all MTTs—thus, a miss rate decrease of 1.0 would imply a 100% cache miss rate under G-EDF and a 0% cache miss rate under our scheduler. Inset (f) presents additional statistics for each histogram: the minimum, average, and maximum miss rate decrease observed under our scheduler (with respect to G-EDF), and the percentage of MTTs that were positively impacted, not impacted, and negatively impacted by our scheduler (x represents the amount by which the miss rate decreased). Note that, under all distributions, our scheduler tends to outperform G-EDF, sometimes by a large margin, and rarely underperforms G-EDF. This is more easily observed in Fig. 5(f): cache performance improved for as many as 33.20% of MTTs, and worsened for at most 9.22% of MTTs. The impact of our scheduler is slightly higher when lower-utilization MTTs exist. This is because lower-utilization MTTs tend to have smaller WSSs, meaning that they execute less frequently and reference memory less frequently during execution, making it more difficult for their jobs to retain their working sets in the cache when thrashing occurs. This provides increased opportunities for our scheduler to improve performance by reducing thrashing.

Worst-case cache performance. Fig. 6 presents differences in cache performance, this time with respect to the *maximum* observed (*i.e.*, worst-case) per-job cache miss rate for every MTT. The differences are relative to G-EDF for each utilization distribution, and insets (a) through (f) present the results identically to Fig. 5. The results in this case are very similar to those in Fig. 5, except that the performance improvements are typically more significant, and our scheduler had a substantial impact on worst-case cache performance for the *majority* of MTTs in *every* distribution. This result has one important implication: under our scheduler, worst-case cache performance is generally better, which can directly result in a decrease in worst-case execution times. As a result, our scheduler has a much greater potential than G-EDF to efficiently utilize the system.

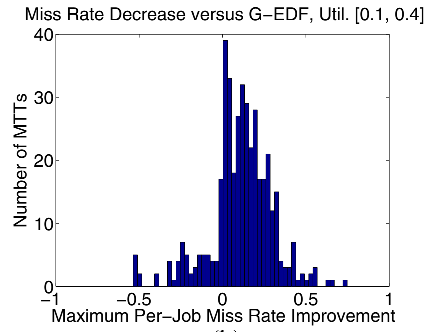
Algorithm	Avg.	Max.
G-EDF	0.00	18.76
CA-SCHED	12.26	1037.69

Table 1: Deadline tardiness results (in ms).

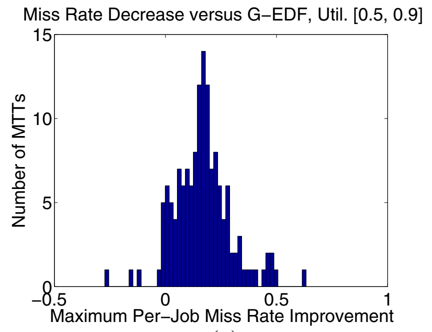
Deadline tardiness. Table 1 shows the actual deadline tardiness of jobs executing under both G-EDF and our scheduler (denoted CA-SCHED). Our scheduler results in higher tardiness than G-EDF, but the difference is not substantial (well under the smallest period of any MTT), with the exception of the worst case, which is artificially inflated by our large task periods. Nevertheless, if buffering can be employed to "hide" tardiness from an end user by shifting job releases and deadlines similarly to how it is done in [2, 7], then these values are reasonable, as a buffer



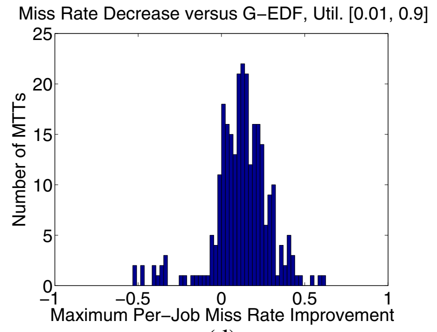
(a)



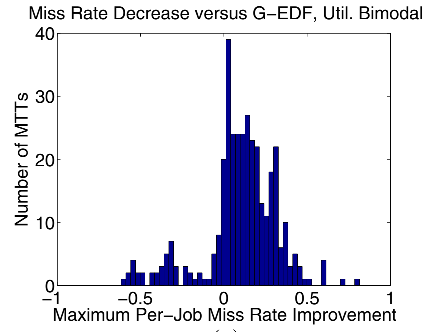
(b)



(c)



(d)



(e)

Util. Dist.	**Min.**	**Avg.**	**Max.**
[0.01, 0.1]	-0.754	0.023	0.930
[0.1, 0.4]	-0.536	0.110	0.748
[0.5, 0.9]	-0.273	0.167	0.633
[0.01, 0.9]	-0.532	0.122	0.626
Bimodal	-0.616	0.102	0.814

Util. Dist.	$x < -0.01$	$\|x\| \leq 0.01$	$x > 0.01$
[0.01, 0.1]	27.19%	8.84%	63.96%
[0.1, 0.4]	14.98%	5.07%	79.95%
[0.5, 0.9]	4.14%	4.83%	91.03%
[0.01, 0.9]	11.94%	4.85%	83.21%
Bimodal	16.76%	3.35%	79.89%

(f)

Figure 6: The decrease in the maximum (worst-case) per-job shared cache miss rate for each MTT over each utilization distribution, as compared with G-EDF. The insets are laid out identically to those in Fig. 5.

of roughly one second could be typical for many multimedia applications. Naturally, such a buffer should be designed to have minimal cache impact.

Scheduling overhead. Finally, we used *Feather-Trace* [5] to measure scheduling-related overheads under both G-EDF and our scheduler during the execution of a subset of the generated task sets. This subset consisted of five task sets from each of the five distributions: two containing the lowest and highest number of MTTs for that distribution, respectively, and the remaining three chosen randomly. We collected nearly five million overhead measurements (over 1 GB of raw data) during these experiments. Table 2 presents average- and worst-case overheads, discarding the top 1% of measurements for reasons discussed in [6]. Tick overhead represents the overhead of scheduling activities that occur during periodic timer interrupts (at quantum boundaries), scheduling overhead is the time to make a scheduling decision (that is not made within the timer interrupt), context switch overhead measures the cost of a context switch *not* including the cost of a preemption or migration, and release overhead is the cost of releasing a job (again, not within the timer interrupt). (A more detailed discussion of overheads is also available in [6].) Release overhead and scheduling overhead are both part of the tick overhead for our scheduler, as releases and most scheduling decisions occur at quantum boundaries. This is not true in G-EDF, where the times at

Algorithm	**Tick AVG**	**Sched. AVG**	**Context Sw. AVG**	**Release AVG**
G-EDF	0.916	3.358	0.953	4.592
CA-SCHED	9.221	2.052	0.972	—

Algorithm	**Tick WC**	**Sched. WC**	**Context Sw. WC**	**Release WC**
G-EDF	1.498	11.478	1.532	8.922
CA-SCHED	17.547	6.278	1.507	—

Table 2: Average and worst-case kernel overheads, in μs.

which jobs are released, and scheduling decisions are made, are independent of when timer interrupts occur—timers are armed to release jobs, and scheduling decisions occur as a result of both job releases and completions. Note that our scheduler does not result in significantly larger combined overheads than G-EDF, implying that overheads will not offset cache-related performance gains.

4.3 Multimedia Application Performance

We next present the results of an experiment conducted to compare our cache-aware scheduler to G-EDF in terms of job execution times when executing a multimedia server workload. This workload was simulated with multiple instances of the *mplayer* application, modified so that one frame is processed per job. Thus, the period of the *mplayer* application, as specified to LITMUS$^{\text{RT}}$, determined the frame rate.

Multiple copies of the same segment of high-quality video (taken from [4]) were processed by different applications so that they appeared to be from different sources. All copies were resident in main memory using RAM disks, to avoid issues with hard disk accesses. The resolution of the video is 1920x1080 with a 24 fps frame rate, resulting in a period of 41 ms. Based on our observations, the execution cost that we specified for an application processing this video is about 14 ms (though execution costs vary widely per frame), and we expected up to a 2.5 to 3 MB WSS per frame. Each application was represented as a single-threaded MTT.

In our experiments, we ran one, six, and twelve *mplayer* applications under both G-EDF and our scheduler. Table 3 shows the measured worst-case job execution times for each case. (Note that, in the case of twelve applications, the system is technically overloaded, and due to RAM disk space constraints, pairs of *mplayer* applications were forced to share the same video copy.) When one video is scheduled in isolation, the execution time under both algorithms is about 14 ms. Under G-EDF, this increases to roughly 15 ms when scheduling six and twelve applications, while it remains about 14 ms under our scheduler. Thus, through the use of our cache-aware scheduler, we are able to ensure that worst-case execution costs do not increase as a result of thrashing. In doing so, the worst-case cost is roughly 6-7% less than G-EDF in the case of six and twelve applications. In the case of twelve applications, this allows about 30% of the utilization of one core to be recovered, which may be used to support one additional *mplayer* application.

Algorithm	1 Video	6 videos	12 videos
G-EDF	14.004	14.947	14.956
CA-SCHED	13.771	13.935	13.972

Table 3: Worst-case execution times for *mplayer* applications (in ms).

Interestingly, the memory reference patterns of these *mplayer* applications are arguably more sequential than random, yet we still see a significant performance improvement when using our scheduler. We would expect that, if sufficient hardware support was available to allow more accurate profiling for sequential reference patterns, the performance benefit of our scheduler would be even greater.

5 Concluding Remarks

In this paper, we have presented the design and implementation of a cache-aware real-time scheduler for multicore platforms within the Linux kernel. Our specific focus has been to overcome two major obstacles to using cache-aware real-time schedulers in practice: automatic cache profiling and implementation efficiency. Our results show that MTTs can be automatically and accurately profiled to determine their per-job WSSs at run time, with reasonable overhead. Further, we have shown that by allowing the scheduler to take profiling information into consideration, performance in practice can be substantially improved.

To enable such performance improvements to be attained across a wide spectrum of applications, chip makers *must provide needed hardware support*. The support required by our profiler is not complex: we merely need performance counters that can be used to accurately count shared cache misses. As remarked earlier, the effectiveness of such counters has declined in moving from the Intel Core 2 chips to the new i7 chip. We view this as a serious mistake. If chip makers are really serious about tackling the issue of effective shared cache usage, then they need to re-think which performance monitoring hardware features they provide, and determine a *standard* set of supported features—this set should remain *stable* as chip architectures evolve, regardless of production deadline pressures. (The same can be said more generally for hardware support related to managing caches.)

Our results suggest a number of avenues for further research. First, we want to determine if cache reuse can be better estimated through the use of more sophisticated hardware monitoring features, such as Precise Event-Based Sampling on Intel platforms. Second, we want to extend our scheduler so that preemption overheads are minimized when a task resumes, by looking into ways of preventing preempted jobs from losing cache affinity while they are not executing. Third, we want to design more scalable run queue data structures that make frequent job priority changes more feasible; doing so would enable results such as [17] to have a greater practical impact, and would make a greater set of the policies from [7] feasible in practice. Finally, we want to provide better support for handling hierarchies of shared caches.

References

[1] F. Abazovic. Intel showcases 80-core CPU. http://www.fudzilla.com/index.php?option=com_content&task=view&id=10107&Itemid=1, 2008.

[2] J. Anderson and J. Calandrino. Parallel real-time task scheduling on multicore platforms. *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pp. 89–100. IEEE, 2006.

[3] G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. *Proceedings of the Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 235–244. ACM, 2004.

[4] Blender Foundation. Big Buck Bunny. http://www.bigbuckbunny.org/.

[5] B. Brandenburg and J. Anderson. Feather-Trace: A lightweight event tracing toolkit. *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 19–28. IEEE, 2007.

[6] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pp. 157–169. IEEE, 2008.

[7] J. Calandrino and J. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pp. 299–308. IEEE, 2008.

[8] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pp. 111–123. IEEE, 2006.

[9] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

[10] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Throughput-oriented scheduling on chip multithreading systems. Technical Report TR-17-04, Division of Engineering and Applied Sciences, Harvard University, 2004.

[11] A. Fedorova, M. Seltzer, and M. Smith. Cache-fair thread scheduling for multicore processors. Technical Report TR-17-06, Division of Engineering and Applied Sciences, Harvard University, 2006.

[12] J. Hennessy and D. Patterson. Memory hierarchy design. *Computer Architecture: A Quantitative Approach*, pp. 390–525. Morgan Kaufmann Publishers, 2003.

[13] Intel Corporation. Intel 64 and IA-32 architectures software developer's manuals. http://www.intel.com/products/processor/manuals/, 2009.

[14] R. Jain, C. Hughs, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pp. 134–145. IEEE, 2002.

[15] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning on a chip multiprocessor architecture. *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pp. 111–122. IEEE, 2004.

[16] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[17] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pp. 413–422. IEEE, 2007.

[18] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors. http://www.cs.washington.edu/research/smt/.

[19] R. Pellizzoni, B. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pp. 221–231. IEEE, 2008.

[20] L. Peng, J. Song, S. Ge, Y.-K. Chen, V. Lee, J.-K. Peir, and B. Liang. Case studies: Memory behavior of multithreaded multimedia and AI applications. *Proceedings of Seventh Workshop on Computer Architecture Evaluation using Commercial Workloads*, pp. 33–40. IEEE, 2004.

[21] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. Modeling cache sharing on chip multiprocessor architectures. *Proceedings of the IEEE International Symposium on Workload Characterization*, pp. 160–171. IEEE, 2006.

[22] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *IEEE Transactions on Embedded Computing Systems*. To appear.

[23] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. *SIGMETRICS Performance Evaluation Review*, 30(1):66–76. ACM, 2002.

[24] J. Stokes. Sun: Can you smell what the Rock is cookin'? http://arstechnica.com/news.ars/post/20080204-sun-can-you-smell-what-the-rock-is-cookin.html, 2008.

[25] UNC Real-Time Group. LITMUS$^{\mathrm{RT}}$ project. http://www.cs.unc.edu/~anderson/litmus-rt/.