Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study*

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. To effectively use the available processing resources on such platforms, scheduling methods must be aware of these caches. In this paper, we explore various heuristics that attempt to improve cache performance when scheduling real-time workloads. Such heuristics are applicable when multiple multithreaded applications exist with large working sets. In addition, we present a case study that shows how our best-performing heuristics can improve the end-user performance of video encoding applications.

1 Introduction

Multicore architectures, which contain multiple processing cores on a single chip, have been adopted by most chip manufacturers due to the thermal- and power-related limitations of single-core designs. Most chip manufacturers have released dual-core chips, Intel and AMD each have four-core chips on the market, and Sun's Niagara and more recent Niagara 2 processors are eight-core chips with multiple hardware threads per core. Furthermore, Intel has announced plans to release chips with as many as 80 cores within five years [10].

In most proposed multicore platforms, different cores share on-chip caches. To effectively exploit the available parallelism in these systems, such caches must not become performance bottlenecks. 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 consider this issue in the context of soft real-time applications. To reasonably constrain the discussion, we henceforth limit attention to the widely-studied multicore architecture shown in Fig. 1, where all cores are symmetric, single-threaded, and share an L2 cache. Both aforementioned Sun processors contain an L2 cache shared by all eight cores; also, Intel and AMD have announced quad-core machines where all cores share a cache (prior architectures had *pairs* of cores sharing a cache, even on quad-core chips).

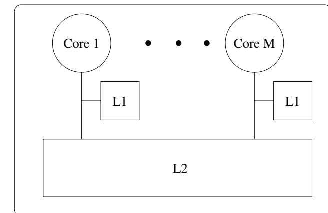

Figure 1: Multicore architecture.

The goal of this paper is to explore ways to improve shared cache performance by encouraging or discouraging the co-scheduling of groups of tasks based on their expected cache impact. We want to co-schedule task groups that reference a common region of memory, and avoid co-scheduling groups that will thrash the cache. It can be shown that the problem of encouraging or discouraging co-scheduling *while ensuring real-time constraints* is NP-hard in the strong sense.

In this paper, we assume that tasks are organized into *multithreaded tasks* (MTTs), where each MTT consists of periodic (sequential) tasks, which may have different execution costs but a common period. Each MTT has a working set (WS) that is shared by all tasks within it. The WS is the region of memory referenced by tasks within an MTT—the size of this region is known as the *working set size* (WSS) of the MTT. MTTs are useful for specifying groups of *cooperating* tasks. (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 MPEG video frame at the same rate, as given by the desired frame rate.

In recent work, we showed that it is possible to discourage high-cache-impact tasks from being co-scheduled while ensuring real-time constraints [2], and that it is possible to encourage the co-scheduling of MTTs [1]. Similar issues have also been explored in the context of throughput-oriented systems (*e.g.*, in [11, 12, 4, 16, 19]). Unfortunately, since the method in [2] places restrictions on parallelism that violate assumptions made in [1], prior work does not support both encouragement and discouragement simultaneously.

Fig. 2 presents an example for a three-core platform where encouraging and discouraging co-scheduling can be useful. Assuming the use of earliest-deadline-first (EDF) scheduling, jobs J and K have the highest priority at time 0, but would thrash the shared cache if co-scheduled. We can avoid thrashing by scheduling V instead of K. Additionally, since V and W are part of the same MTT, we want to encourage

*Work supported by grants from Intel and IBM Corps., by NSF grants CNS 0408996, CCF 0541056, and CNS 0615197 and by ARO grant W911NF-06-1-0425.

W to be scheduled when V is scheduled. To accomplish this, the priorities of V and W need to be increased. However, this may negatively impact real-time guarantees later in the schedule.

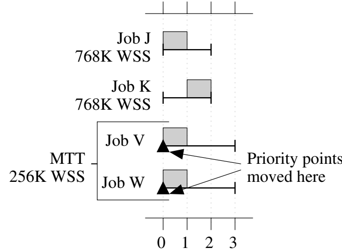

Figure 2: An example of where promoting jobs can improve cache performance, assuming the WSSs shown and a cache size of 1 MB.

Fortunately, recent work in [17] provides a mechanism to manipulate job priorities while still providing real-time guarantees. This work assigns *priority points* to every eligible job and prioritizes jobs by increasing priority point values. For example, when using EDF scheduling, 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 arbitrarily within its window*. (Such a guarantee is not possible under partitioned scheduling.) Knowing this, we can *promote* jobs V and W in Fig. 2 by moving their priority points to the current time. Doing so causes V and W to be scheduled next, thus improving cache performance, and still allows real-time guarantees to be made. This method of promoting jobs indirectly discourages the co-scheduling of certain groups of tasks by encouraging other groups to be co-scheduled instead. Related work on *symbiotic scheduling* [14, 18, 21] takes a similar "discouragement by encouragement" approach; however, this work lacks analysis for validating *real-time constraints*.

Improved cache performance can directly result in reduced execution costs for real-time tasks. We can take advantage of these reduced costs in several ways, such as using the same platform to support a larger workload, or performing additional computation that increases task utility. We elaborate on this in our case study, presented in Sec. 4.3.

While promoting jobs can lead to improved cache performance in the near term, it might result in poor cache performance later. For example, if we always promote jobs from MTTs with the smallest WSSs, then eligible jobs from MTTs with larger WSSs will be pushed later in the schedule. Thus, the choice of when to promote jobs, and which jobs to promote, can have a substantial impact on the effectiveness of this method, and is not always straightforward. For this reason, we propose and evaluate a large number of *heuristics* within this paper. Each heuristic represents a set of design decisions that dictate *when* to promote jobs and *which* jobs to promote. These heuristics are described in detail in Sec. 3.

Potential applications. Several types of applications could benefit from cache-aware real-time scheduling approaches. First, *MPEG-2 video encoding* applications have soft real-time constraints and are compute- and memory-intensive. Given the current multicore trend, these applications will need to become *multithreaded* as the demanded video quality increases. Video encoding requires a search as part of motion estimation—improving cache performance may allow for a more extensive search, thus improving video quality. Alternately, a digital content provider might be able to encode more videos using the same hardware.

Second, *high-performance computing* applications typically involve splitting large tasks into manageable pieces that can be handled by individual processors. As one might expect, such tasks are compute- and memory-intensive. It is often beneficial if all processors have made approximately the same amount of progress at any point in time, due to the need to periodically synchronize processors. If we represented such an application as one or more MTTs, then real-time constraints and MTT co-scheduling could help to maintain the same pace among all processors, improve cache performance, and reduce the overall time to complete a task.

Finally, real-time graphics applications are typically assisted by GPUs. Due to the nature of GPU hardware and graphics processing, such applications tend to be inherently parallel (*i.e.*, multithreaded) and memory-intensive. These applications also have real-time requirements (hence their name, and the need for GPU assistance), and are likely to run alongside other applications on the same platform. Our heuristics may be useful to improve cache performance and reduce execution requirements for such applications, and those with similar characteristics, which may become more common as interest in treating GPUs as co-processors for general computation continues to increase [13].

Summary of contributions. In this paper, we propose and evaluate cache-aware scheduling heuristics for real-time tasks on multicore platforms, assuming a tick-based global scheduling approach. We show that our heuristics result in better cache performance than global EDF (G-EDF) scheduling, yet still guarantee bounded tardiness. We demonstrate the performance impact of our method through experiments, including a video encoding case study.

The rest of this paper is organized as follows. In Sec. 2, we present an overview of our task model and other background information. Then, in Sec. 3, we describe our heuristics, and derive a tardiness bound for these heuristics. In Sec. 4, we present experimental results, and in Sec. 5, we conclude.

2 Background

In this section, we briefly introduce our task model as related to MTTs and some other background information.

For simplicity, we consider only periodic task systems, though our results apply to sporadic task systems as well.[1] 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. 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 $d(T_k) + B$.

Execution costs and periods are assumed to be integral. In this paper, we consider only *tick-based* scheduling, where scheduling decisions are made at each quantum boundary (*i.e.*, "tick"). Scheduling quanta can be any convenient size such that execution costs and periods are integral. Tick-based scheduling simplifies our heuristics by making it easier to predict cache performance. We briefly discuss the impact of allowing scheduling between ticks later.

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 make the simplifying assumption throughout this paper that each of an MTT's threads has the same execution cost (as well as period)—later, we briefly discuss how this restriction could be removed. We also assume that each MTT has at most M threads, the maximum parallelism achievable on M cores.

G-EDF scheduling. Under G-EDF scheduling, jobs are scheduled in order of increasing deadlines, with ties broken arbitrarily. All of our heuristics use G-EDF until a specified cache-utilization threshold is reached.[2] G-EDF is *not* optimal, so tasks may miss their deadlines; however, deadline tardiness under G-EDF is bounded [9].

3 Heuristics

The heuristics presented in this paper are used to make scheduling decisions at each tick in an effort to improve cache performance based on MTT information such as WSS. Note that our heuristics do not perform any scheduling, monitoring, or other activity *between* ticks—later in this section, we briefly consider allowing scheduling between ticks. Scheduling decisions are made iteratively over all cores—even when jobs are promoted, jobs that have already been scheduled on some core at the current tick are unaffected.

Several rules, stated below, are common to all heuristics.

```
MakeSchedulingDecisions(numCores, cacheSize)
      ▷ Initialize variable to track sum of MTT WSSs
 1    usedCache := 0;
      ▷ Make scheduling decisions by iterating over all cores
 2    for i := 1 to numCores do
         ▷ Assign each job a priority point equal to its deadline
 3       AssignJobPriorityPointsEqualToDeadlines();
         ▷ Promote job if applicable
 4       if (No eligible urgent jobs) then
 5            C := max(0, cacheSize − usedCache);
 6            N := numCores − i + 1;
 7            if (usedCache/cacheSize ≥ Lost-cause Threshold) then
 8              Promote job using Lost-cause Policy;
 9            elseif (usedCache/cacheSize ≥ Cache Utilization Threshold) then
10              Promote job using Cache-aware Policy;
11              if (Use phantom tasks ∨
                    Avoid scheduling partially-eligible MTTs) then
12                 Adjust job promotion policy accordingly;
                fi
            fi
            ▷ Else no job is promoted, use G-EDF
         fi
13       Schedule job on core i according to Priorities rule;
14       mtt := MTT of scheduled job;
15       if (scheduled job is first job of mtt scheduled at this tick) then
16            usedCache := usedCache + WSS(mtt);
         fi
17       Set urgent flags and promote/demote jobs according to
         Promoted Jobs and Urgent Jobs rules;
      od
```

Figure 3: Pseudo-code for all heuristics, invoked each tick.

- **Promoted Jobs.** A promoted job is given a new priority point that is equal to the minimum of its deadline and the current time (so that tardy jobs are not penalized). A promoted job is highly encouraged, but *not guaranteed*, to be scheduled next. This rule ensures that priority points are window-constrained and tardiness is bounded. Job promotions are only valid until the next scheduling decision is made, with the exception of urgent jobs, described next.
- **Urgent Jobs.** When a job T_i is scheduled, where task T corresponds to a thread within some MTT R, and T is the first thread in R to schedule its ith job at this tick, each job U_i, where U is also a thread of R and $U \neq T$, is flagged *urgent* and promoted until it too is scheduled. Note that this only occurs if T_i itself is not urgent. Urgent jobs remain promoted until they are next scheduled, and no non-urgent job can be promoted while eligible urgent jobs exist. This rule encourages jobs from the same MTT to be scheduled together.
- **Priorities.** Released jobs are scheduled in increasing order of their current priority points (including promotions). Ties are broken in favor of promoted jobs, since scheduling such jobs is expected to improve cache performance. (There is no tie-breaking rule related to urgent jobs, since urgent and non-urgent promoted jobs cannot both exist when making a scheduling decision.)

Fig. 3 presents pseudo-code that describes how scheduling

[1] In sporadic task systems, task periods *within* an MTT must still coincide.

[2] We use G-EDF since prior work has shown that it results in better schedulability for soft real-time systems than other approaches [7]. Note also that controlling co-scheduling under partitioned scheduling is problematic because processors are scheduled *independently*.

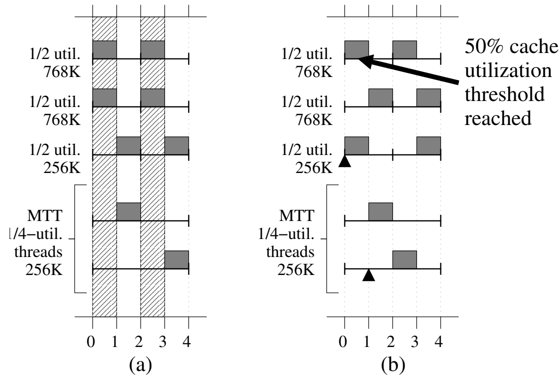

Figure 4: Two-core schedules for a set of five tasks (three tasks of util. 1/2, and one MTT with two 1/4-util. threads) with **(a)** G-EDF scheduling and **(b)** one of our heuristics. WSSs are shown along with the task utilizations. Thrashing occurs during "hatched" quanta, assuming a shared cache size of 1 MB, and black triangles indicate the new priority points of promoted jobs.

decisions are made by each heuristic. Note that some thresholds and policies used in the pseudo-code are undefined, as indicated by bold type. A heuristic is defined in terms of the thresholds and policies that it employs; however, similarities exist among all heuristics. First, all heuristics encourage jobs to be scheduled by promoting them. Second, all heuristics encourage the co-scheduling of MTTs in the same way. Third, all heuristics keep track of *cache utilization*, defined as the sum of the WSSs of all MTTs with jobs scheduled during this tick divided by the shared cache size. For example, if jobs from two MTTs with WSSs of 256K and 512K have been scheduled, then the current cache utilization of a 1 MB cache would be 75%. Fourth, all heuristics use G-EDF scheduling until the cache utilization reaches a *cache utilization threshold*, at which point a *cache-aware policy* is employed. Finally, if cache utilization reaches the *lost-cause threshold*, then a *lost-cause policy* is employed. Note that, while WSS is the cache utilization metric used in this paper, other metrics based on more sophisticated cache profiling techniques could be used instead. This might require the creation of policies that are suitable for the new metric.

Fig. 4 presents an example of how cache performance can be improved over G-EDF scheduling using our heuristics. The heuristic shown uses a 50% cache utilization threshold, a very high lost-cause threshold, and a cache-aware policy that promotes jobs from the MTT with the smallest WSS. At time 0, the job of the $1/2$-utilization task with the smallest WSS is promoted by the cache-aware policy, while at time 1, the second task of the MTT is promoted and flagged urgent by the *Urgent Jobs* rule. The *Priorities* rule causes the promoted jobs to be scheduled next (at times 0 and 2) in both cases.

Note that a large number of tardy jobs can make it very difficult to influence scheduling decisions through job promotion, but this is necessary if any real-time guarantees are to be made. By making intelligent scheduling decisions before these scenarios arise, we can minimize negative impacts on cache performance when we "lose control" of the system in this manner. In the discussion of policies that follows, we return to this issue.

3.1 Policies

We now present the policies that define a heuristic. While thresholds specify *when* to promote a job, policies specify *which* job to promote. In the discussion that follows, we speak of promoting *MTTs* instead of jobs. This means that we choose a single eligible job within that MTT to promote; however, by the *Urgent Jobs* rule, all other eligible jobs within that MTT will be flagged urgent and promoted as soon as the promoted job is scheduled. By the *Priorities* rule, co-scheduling of the MTT will be encouraged.

Cache-aware policy. This policy is employed when cache utilization reaches the cache utilization threshold. This policy is used for the current tick until all cores are scheduled or cache utilization reaches the lost-cause threshold. Each policy chooses an MTT to promote based on the remaining "un-utilized" cache C and "free" cores N (see Fig. 3). We present five different policies in the paper. Assume that the thread count and WSS of an MTT R are denoted $tc(R)$ and $\mathit{WSS}(R)$, respectively. Note that, for all policies, we only promote an MTT if it contains *eligible jobs*.

1. Promote R with the smallest $\mathit{WSS}(R)$.
2. Promote R with the largest $\mathit{WSS}(R)$ that does not exceed C, or exceeds it by the smallest amount if no such MTT exists.
3. Promote R with the smallest $\mathit{WSS}(R)/tc(R)$ ratio.
4. Promote R with the largest $\mathit{WSS}(R)/tc(R)$ ratio, where $\mathit{WSS}(R)$ does not exceed C, or exceeds it by the smallest amount if no such MTT exists.
5. Promote R with the largest $\mathit{WSS}(R)/tc(R)$ ratio that does not exceed C/N, or exceeds it by the smallest amount if no such MTT exists.

When $C = 0$, policies (2) and (4) become equivalent to (1), and policy (5) becomes equivalent to (3).

Insets (b) through (f) of Fig. 5 show the differences between policies when scheduling the task set in inset (a). Policies (1) and (3) make locally greedy decisions that improve cache performance in the near term, but result in cache thrashing later, since the remaining eligible jobs are from MTTs with large WSSs. Over time, these policies would result in periodic cache thrashing for this task set. Policies (2) and (4) attempt to minimize the impact of high-cache-impact MTTs by scheduling them whenever they will not cause thrashing. As a result, the jobs that are eligible later are from lower-cache-impact MTTs, cache utilizations are lower, and thrashing is less extreme. However, these policies differ in their definition of a "high-cache-impact" MTT. We

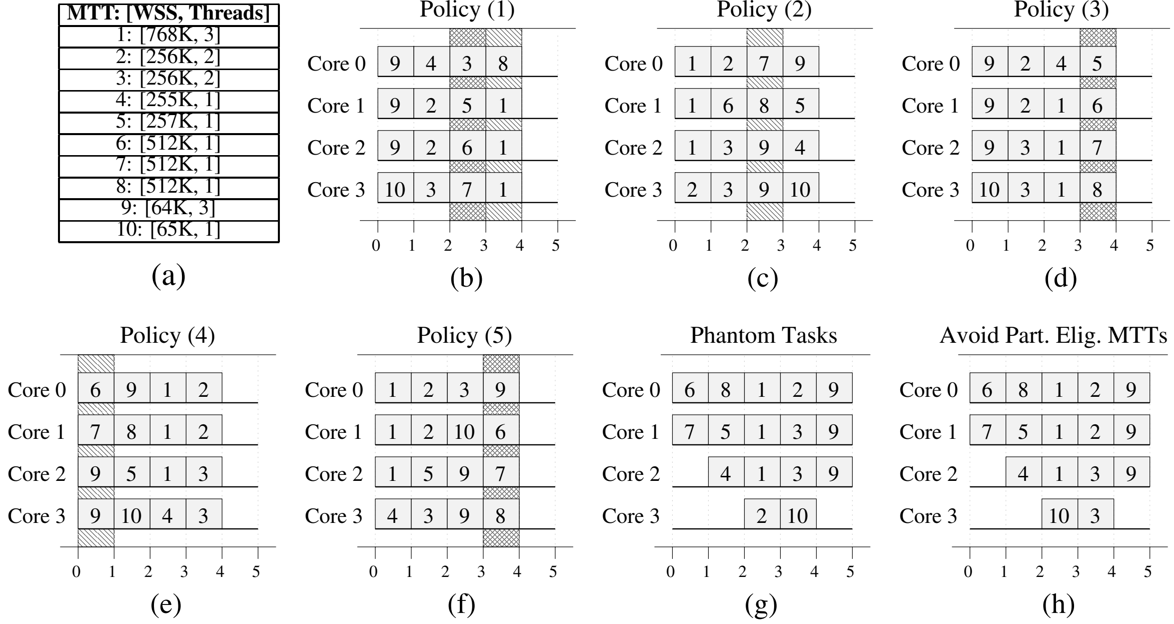

Figure 5: Four-core schedules demonstrating a variety of cache-aware policies, assuming a cache utilization threshold of 0%, an infinite lost-cause threshold, and a 1 MB shared cache. Each task has an execution cost of one and period of five. The numbers in boxes indicate the MTT that is scheduled on a core during a quantum—for example, in inset (b), three jobs of MTT 1 are scheduled on cores 1-3 between times 3 and 4. Hatching indicates cache thrashing, *i.e.*, cache util. exceeding 100%—cross-hatching indicates a cache util. exceeding 150%. Inset (a) shows the task set to be scheduled, and insets (b)-(f) show schedules when using cache-aware policies (1)-(5), respectively. Insets (g) and (h) show schedules when using policy (4) and phantom tasks—for inset (h), we also avoid scheduling partially-eligible MTTs.

believe that the definition used by policy (4) is more accurate, as MTTs with large $\frac{\text{WSS}}{\text{thread count}}$ ratios demand a large amount of the cache, yet are easily co-scheduled with jobs from other MTTs. This makes such MTTs quite difficult to schedule when cache performance is a concern. Finally, policy (5) is similar to (4), except that it tends to delay scheduling MTTs with the highest cache impact; this results in overall performance that is similar to policies (1) and (3).

While some policies were not very effective in this example, they may be more effective when cache utilization thresholds are higher, depending on which MTTs are scheduled before the threshold is reached. Therefore, it is not clear which policy will result in the best performance in all cases, or if such a policy exists—we use experiments to assist us in making recommendations.

Lost-cause policy. Each heuristic also employs a lost-cause policy when cache utilization reaches the lost-cause threshold. Once this threshold is reached, we assume that poor cache performance will be inevitable during the current quantum. We present three policies for when this occurs.

1. Revert to G-EDF.
2. Promote R with the largest $\mathit{WSS}(R)$.
3. Promote R with the largest $\mathit{WSS}(R)/tc(R)$ ratio.

Policy (1) attempts to reduce average tardiness, while policies (2) and (3) schedule high-cache-impact MTTs so that it is easier to avoid thrashing in future quanta, since near-term cache performance is essentially guaranteed to be poor. Note that policies (2) and (3) can backfire—since high-cache-impact MTTs will be most negatively affected by cache thrashing, and such MTTs generate the majority of memory references, the overall cache miss rate might increase rather than decrease.

Phantom tasks. If the system is not fully utilized, then it may be possible to idle one or more cores to prevent thrashing. A heuristic can choose to idle cores by promoting jobs of *phantom tasks*, which are single-threaded tasks that have a period equal to the hyperperiod of the real-time workload, an execution cost of one, and a WSS of zero. Phantom tasks represent the available idle time. A heuristic will only promote jobs of phantom tasks to avoid cache thrashing—if the number of eligible jobs of phantom tasks is at least the number of eligible jobs of the promoted MTT, and the WSS of the promoted MTT is larger than C, then jobs of the phantom tasks will be promoted instead. If phantom tasks are used, then a job of a phantom task must be scheduled whenever a core is idle, even if no "real" jobs are eligible. Fig. 5(g) shows the impact of using phantom tasks along with cache-aware policy (4), which allows cores 2 and 3 to be idle at time 0, and core 3 to be idle at time 1. As a result, cache thrashing is avoided entirely.

Partially-eligible MTTs. We consider an MTT R to be *partially eligible* either because fewer than $tc(R)$ jobs are eligible at the time that it is considered for promotion by the

cache-aware policy, or $tc(R)$ exceeds the current value of N. We may want to avoid scheduling such MTTs since their WSs will need to be referenced in at least one future quantum in addition to the quantum that follows the current tick. If a heuristic chooses to avoid scheduling partially-eligible MTTs, then such MTTs are promoted only when all other MTTs have WSSs that are greater than C. Fig. 5(h) shows how this is used along with policy (4) and phantom tasks. At time 2, we avoid scheduling MTT 2 in favor of MTT 10, which allows all jobs of MTT 2 to be co-scheduled at time 3. As a result, the amount of time that the WS of MTT 2 must be present in the cache decreases by 50%. Note that we choose to schedule a partially-eligible MTT to avoid cache thrashing before scheduling phantom tasks, as the former choice allows some real work to be accomplished.

Scheduling between ticks. We now consider allowing scheduling decisions to be made *between* ticks, *e.g.*, when jobs complete between ticks. In this case, by letting $N = 1$ and defining C based on the MTTs executing on the other $M - 1$ cores, we can use the same heuristics used at each tick. However, since job priorities change over time due to promotions, jobs scheduled between ticks may be quickly preempted at the next tick. Additionally, reliably predicting per-quantum cache performance becomes more difficult. These factors can decrease cache performance and may offset any utilization gains.

Removing the same-execution-cost restriction. If MTT threads are allowed to have different execution costs, then the policies described above that make decisions based on MTT thread counts must be changed. In particular, we can redefine $tc(R)$ for an MTT R to be the number of threads that have not completed their current job, and redefine a partially-eligible MTT with respect to this new definition. In the experiments that follow, we assume MTTs with the same execution costs for all threads, and leave further exploration of this issue as future work.

3.2 Tardiness Bound

Our tardiness bound follows directly from the work in [17], since job priorities are window-constrained for all heuristics. The tardiness for a task $T^k \in \tau$ scheduled using our heuristics is thus $x + e(T^k)$, where x is defined as follows (from [17], with minor changes for compatibility with our notation from Sec. 2).

$$x = \frac{E_L + A(k)}{M - U_L} \quad (1)$$

In (1), E_L (U_L) is the sum of the $M-1$ highest execution costs (utilizations) over all tasks in τ, and M is the number of processing cores. $A(k)$ is defined as follows (also from [17]).

$$A(k) = (M-1)\cdot\rho - e(T^k) + \sum_{T^j \in \tau \setminus T^k} \left(\lceil \frac{\psi_k + \phi_j}{p(T^j)} \rceil + 1\right) \cdot e(T^j)$$

ϕ_h (resp., ψ_h) indicates the amount by which the priority point of a job of T^h can be before its release time (resp., after its deadline), and $\rho = max_{T^h \in \tau}(\phi_h) + max_{T^h \in \tau}(\psi_h)$. Since our heuristics never allow a job's priority point to be less than its release time or greater than its deadline, $\phi_h = \psi_h = 0$ for all $T^h \in \tau$, and $\rho = 0$. Therefore, $A(k)$ becomes $\sum_{T^j \in \tau \setminus T^k} e(T^j) - e(T^k)$, and our tardiness bound is $\frac{E_L + \sum_{T^j \in \tau \setminus T^k} e(T^j) - e(T^k)}{M - U_L} + e(T^k)$. Note that scheduling phantom tasks will increase the tardiness bound, since such tasks must be treated as "real work" in this case.

4 Experimental Results

To assess the efficacy of our heuristics in improving both cache and user-perceived performance, we conducted experiments using the SESC Simulator [20], which is capable of simulating a variety of multicore architectures. The simulated architectures that we considered consist of eight or 32 3-GHz cores, each with a dedicated 4-way (resp., 2-way) set associative 16K L1 data (resp., instruction) cache with a random (resp., LRU) replacement policy; and a shared 8-way set associative 2 MB (8 MB on the 32-core machine) on-chip unified L2 cache with an LRU replacement policy. Each cache has a 64-byte line size. The 32-core architecture is a "scaled-up" version of the 8-core architecture that is used to gain a basic understanding of how our heuristics might perform on a large-scale multicore platform—we acknowledge that, in practice, 32 cores may not directly share a cache.

While 8-core architectures such as the Sun Niagara are available today, we chose to use a simulator for a number of reasons. First, the simulator allowed us to get detailed results on the performance of our heuristics in a more controlled environment where only a minimal operating-system layer exists. In future work, we plan to implement and evaluate the most promising heuristics within LITMUS$^{\text{RT}}$ [5]. Second, a simulator allows us to experiment with systems with more cores than commonly available today, such as our 32-core platform. Experimenting with both an 8-core and 32-core platform in simulation allows us to make reasonable comparisons between an architecture that is feasible today and one that may exist several years in the future, and allows us to determine if our heuristics will continue to have a performance impact.

We used CACTI 4.2 [15] to obtain realistic *cache access time* estimates for our simulations. (The cache access time is the time required to access a single block of the shared cache.) CACTI is a tool that uses analytical models based on empirical data to provide estimates of cache access times, energy consumption, and chip area, when given certain cache attributes such as size and associativity. The estimates pro-

Figure	Miss Rate	Per-MTT Improvement
Fig. 4(a)	23.99%	[0, 0, 0]%
Fig. 4(b)	15.41%	[11.02, **41.69**, **71.30**]%
Fig. 5(b)	9.07%	[0, 0, 0]%
Fig. 5(c)	7.90%	[-50.05, 21.26, 155.20]%
Fig. 5(d)	8.70%	[-64.93, 0.54, 76.97]%
Fig. 5(e)	7.94%	[-16.88, 15.41, 71.39]%
Fig. 5(f)	8.94%	[-64.22, 0.76, 68.39]%
Fig. 5(g)	6.60%	[-13.82, **37.45**, **166.12**]%
Fig. 5(h)	6.71%	[-13.51, **36.58**, **166.30**]%

Table 1: Shared cache performance for example task sets.

vided by CACTI were used in place of the default cache access times in the SESC configuration files. Since the default times in the configuration files assume different shared cache attributes (*e.g.*, 512K shared cache size), the use of CACTI improved the temporal accuracy of our simulations.

The following sections describe three sets of experiments involving example task sets, randomly-generated task sets, and multithreaded video encoding applications. In each set of experiments, G-EDF scheduling was compared to some subset of our heuristics. All task sets were scheduled and run for 20 (simulated) milliseconds. Previous experimental studies using SESC have indicated that this is long enough to observe performance differences [2]—we ran a subset of our randomly-generated task sets for 100 ms to confirm this.

MTTs are assigned memory regions according to their WSSs. In Sec. 4.1 and 4.2, each thread references the memory region of its MTT sequentially, looping back to the beginning of the region when the end is reached, for its entire specified execution time—thus, tasks are backlogged. The memory reference pattern for video-encoding MTTs (Sec. 4.3) is more complicated—each thread references its assigned video frame slice, plus some "nearby" slices. We accounted for scheduling, preemption, and migration costs in all simulations.

4.1 Example Task Sets

To demonstrate the performance impact of our heuristics, we first present results for the example schedules in the figures located in earlier sections of this paper. The results are categorized by figure number and presented in Table 1. Each task set was run using the heuristic, cache size, and core count indicated in the example figure. The "Per-MTT Improvement" column presents the minimum, average, and maximum percentage increase in number of per-quantum memory references for each MTT, relative to the first schedule in each figure. For example, the schedule in Fig. 5(c) resulted in a 21.26% increase in per-quantum memory references on average for each MTT when compared to the schedule in Fig. 5(b); for one MTT, per-quantum memory references increased by 155.20%, but for another MTT, per-quantum memory references decreased by 50.05%. The results show that, when suitable heuristics are employed, performance improves substantially (see the bold entries in Table 1). For example, the *overall* shared cache miss-rate decreases by over one third over G-EDF for the task set in Fig. 4 when one of our heuristics is used, and MTTs are able to perform an average of over 41% more memory references. For the schedules in Fig. 5, we see that different cache policies can influence performance in very significant ways. The best-performing heuristic results in a 37% increase in memory references on average, and a 166% increase in the best case, over the heuristic with the worst average-case performance. Note that even heuristics that perform better on average can result in worse performance for some MTTs, resulting in negative minimum performance improvement values; however, these values are often small when compared to the average- and best-case performance improvements when the best-performing heuristics are used. Of course, these experiments are not conclusive—the results in the following sections demonstrate the general applicability of our heuristics.

4.2 Randomly-Generated Task Sets

In this set of experiments, we evaluated many heuristics, representing different combinations of design decisions, on task sets representing a variety of task utilizations, system utilizations, and WSS distributions. Experiments were initially conducted on the 8-core architecture—the best-performing heuristics were then evaluated on the 32-core architecture. We considered heuristics representing the following thresholds and policies.

- **Cache utilization threshold:** 0%, 50%, or 75%.
- **Cache-aware policy:** all policies (1)-(5) considered.
- **Lost-cause threshold:** 110%.
- **Lost-cause policy:** all policies (1)-(3) considered.
- **Phantom tasks:** used and not used.
- **Avoid scheduling partially-eligible MTTs:** yes.

We chose to avoid scheduling partially-eligible MTTs in every experiment, since it should limit the amount of time that working sets of MTTs must be present in the cache.

Task-set generation methodology. When generating random task sets, we varied the following parameters.

- **System utilization:** 50% or 100% utilized.
- **MTT periods:** between 10 and 100 ms (some values removed to avoid arithmetic overflow), except for the the last generated MTT, which may have a larger period.
- **MTT utilizations:** uniform over [0.01, 0.1], [0.1, 0.4], [0.5, 0.9], or [0.01, 0.9].
- **MTT execution costs:** derived from periods and utils.
- **MTT thread counts:** uniform over [1, 8].
- **MTT WSSs**: uniform over [64 bytes, 2 MB]; or equal to the thread count multiplied by a size uniform over [64 bytes, 512K], and capped at 2 MB.

Task Set Parameters			Best-Performing Heuristic				L2 Miss Rate			Instrs. per Cycle (IPC)		
Sys. Util.	**MTT Util.**	**WSS Dist.**	**Cache Util. Thresh.**	**Cache Pol.**	**LC Pol.**	**Phant. Tasks**	G-EDF	**Heur.**	**% Impr.**	G-EDF	**Heur.**	**% Impr.**
50%	[0.01, 0.1]	TC Corr.	0	(1)	(1)	used	3.62	1.60	**55.88**	0.97	1.23	**26.46**
50%	[0.01, 0.1]	Uniform	0	(1)	(3)	used	7.14	3.16	**55.76**	0.80	1.17	**44.90**
50%	[0.1, 0.4]	TC Corr.	0	(1)	(1)	used	1.22	0.36	**70.62**	1.21	1.20	-1.07
50%	[0.1, 0.4]	Uniform	0	(3)	(1)	used	6.70	0.67	**90.00**	0.93	1.17	**25.19**
50%	[0.5, 0.9]	TC Corr.	0	(1)	(1)	used	1.07	0.28	**73.67**	1.03	1.01	-2.28
50%	[0.5, 0.9]	Uniform	0	(3)	(1)	used	15.38	0.98	**93.61**	0.77	0.92	**18.99**
50%	[0.01, 0.9]	TC Corr.	0	(3)	(1)	used	3.61	0.63	**82.68**	1.01	1.12	10.77
50%	[0.01, 0.9]	Uniform	0	(1)	(1)	used	7.92	0.78	**90.12**	0.97	0.95	-2.15
100%	[0.01, 0.1]	TC Corr.	0	(3)	(1)	N/A	5.30	1.67	**68.55**	0.85	1.16	**36.96**
100%	[0.01, 0.1]	Uniform	0	(3)	(2)	N/A	7.22	2.57	**64.38**	0.76	1.11	**45.26**
100%	[0.1, 0.4]	TC Corr.	0	(3)	(2)	N/A	3.75	1.35	**64.00**	0.96	1.18	**22.44**
100%	[0.1, 0.4]	Uniform	0	(3)	(3)	N/A	7.02	3.46	**50.71**	0.89	1.14	**28.20**
100%	[0.5, 0.9]	TC Corr.	0	(1)	(3)	N/A	3.81	2.83	25.66	1.05	1.13	7.20
100%	[0.5, 0.9]	Uniform	50	(1)	(1)	N/A	5.03	3.58	28.93	0.99	1.06	6.28
100%	[0.01, 0.9]	TC Corr.	0	(1)	(1)	N/A	2.49	0.88	**64.56**	1.09	1.23	13.29
100%	[0.01, 0.9]	Uniform	50	(1)	(1)	N/A	4.30	3.70	14.04	0.99	1.05	6.26

Table 2: The best-performing heuristics for random task sets.

Each heuristic was used to schedule 20 task sets for each combination of these parameters. In total, this resulted in nearly 30,000 experimental runs using SESC. Due to the large amount of time required for each experimental run, we were unable to run a larger set of experiments. Even with the assistance of a large research cluster, we were able to complete only a few thousand experimental runs per day. Like many architecture simulators, SESC is quite slow, especially when timing accuracy is required.

Task set justification. We believe our task periods represent a reasonable range of those observed in real applications. Our task utilization ranges are similar to those used in other work [6, 7, 3]. System utilizations were chosen so that scheduling flexibility was either substantial (at 50%) or very limited (at 100%). Finally, for half of the experiments, MTT WSSs were correlated with thread count. This seems realistic, since a larger number of threads could better process a larger memory region. WSSs were often large, but never exceeded the size of the L2 cache—otherwise, thrashing would be inevitable. Such large WSSs are realistic in practice; for example, the authors of [8] claim that the WSS for an HDTV-quality MPEG decoding task could be as high as 4.1MB, and statistics presented in [22] show that substantial memory usage is required for video-on-demand applications.

Results. Due to space constraints, it is impossible to present results for every explored heuristic in this paper; however, Table 2 presents the heuristics that performed best in the average case for each combination of task set generation parameters, and compares them to G-EDF in terms of average cache-miss rates and average *per-core* instructions per cycle (IPC), which typically correlates with memory references. We can make several observations from this data. First, in almost all cases, the best-performing heuristic outperforms G-EDF, often by a substantial margin (see the bold entries in Table 2). Second, heuristics that used cache-aware policies (1) or (3) performed best, with policy (3) performing better than (1) when system utilization was high and MTT utilizations were low. Third, the best-performing heuristics almost unanimously employed a cache utilization threshold of 0% and lost-cause policy (1), though lost-cause policies (2) and (3) sometimes performed best for the high-system-utilization, low-MTT-utilization task sets. Fourth, phantom tasks were employed by all best-performing heuristics, when applicable. Finally, performance improvements tended to be larger when MTT utilizations were lower. We therefore conclude that the overall best-performing heuristic employed a cache utilization threshold of 0%, cache-aware policy (1), lost-cause policy (1), and phantom tasks. In the cases where other heuristics performed best, the difference was not typically substantial. Phantom tasks only performed worse when they could not be employed effectively, *e.g.*, when the system was fully utilized—we could infer from this result that scheduling between ticks rather than idling a core (as done here) would often decrease performance.

Algorithm	Avg.	Max.
G-EDF	0.216	474
Heuristics	1.843	572
Best heur. only	3.711	493

Table 3: Tardiness for G-EDF and our heuristics (in quanta).

We next tabulated average and maximum observed deadline tardiness. These results are shown in Table 3. In this case, we ran each task set for 2,000 quanta rather than 20 quanta. Tardiness is higher with our heuristics than with G-EDF, but average tardiness is reasonable, and maximum tardiness is comparable to G-EDF with our best heuristic. The somewhat high maximum tardiness values are an artifact of our task generation methodology, which produces some tasks with very large execution costs. The average-case results suggest that tardiness will not significantly restrict the extent to which heuristics can be employed.

32-core architecture evaluation. We next ran a similar set of experiments for the 32-core architecture, where task sets were scheduled with both G-EDF and the best-performing

heuristic for the 8-core experiments (cache utilization threshold of 0%, cache-aware policy (1), lost-cause policy (1), and phantom tasks). The results are not shown here due to space constraints, but they are very similar to the 8-core case, with the heuristic outperforming G-EDF.

These experiments certainly should not be considered definitive; however, similar task sets have been effectively used in other published work [6, 7, 1, 2].

4.3 Video Encoding: A Case Study

We next evaluated the performance of real-time MPEG-2 video encoding applications when using our heuristics by emulating the motion estimation portion of the encoding within SESC. This is the most compute- and memory-intensive portion of MPEG video encoding. We achieved this emulation by mimicking a potential memory reference pattern for multithreaded motion estimation. As the core counts of multicore platforms increase, and the performance of individual cores remains similar (or decreases), most compute-intensive applications such as video encoding will be required to use a multithreaded approach to continue to achieve performance gains. Such performance gains are mandatory if the video quality demanded by users continues to increase.

Our memory reference pattern emulates multithreaded motion estimation where each video frame is split into identically-sized horizontal slices, each of which is processed by an individual thread. For example, 720p HDTV video contains frames of 1280 x 720 pixels (900K per frame assuming one byte per pixel); this video could be divided into 15 slices of size 1280 x 48 (60K per frame) or eight slices of size 1280 x 90 (112.5K per frame). For our heuristic, the latter division is more applicable so that the motion estimation task can be represented as an MTT on an 8-core machine.

Each thread processes the same slice of every video frame. All motion estimation requires a search—for each thread, this involves searching a memory region that includes both its assigned slice and several nearby slices, such as the slices immediately above and below its assigned slice. It is often desirable to search the largest space possible (to approximate an exhaustive search), so we assume that tasks are backlogged in that they continue the search for as long as time permits unless the search space is exhausted. The memory regions that are referenced by each thread in an MTT overlap more as the size of the region searched per-thread increases.

In our experiments, threads reference the memory region of their assigned slice first, and then search progressively more distant slices until they have exhausted their execution times. Both G-EDF scheduling and the best-performing heuristic from Sec. 4.2 were used to schedule task sets on the 8-core architecture. MTTs were generated according to the video quality level of the video that they represented. These levels define resolutions and frame rates that are typical for real applications, some more demanding than others. Table 4 presents these levels and their corresponding MTTs.

Level	Resolution	Frames/Sec.	WSS	Thr. Ct.	Exec. Cost	Period
1	1920 x 1080	30	2025K	8	1	33
2	1920 x 1080	30	2025K	5	1	33
3	1280 x 720	60	900K	8	1	16
4	1280 x 720	60	900K	4	1	16
5	720 x 480	30	338K	1	1	33
6	352 x 288	30	99K	1	1	33
7	320 x 240	24	75K	1	1	41
8	176 x 144	15	25K	1	1	66

Table 4: Video quality levels and their corresponding MTTs.

Video encoding task sets were randomly-generated according to the following methodology. System utilization was either 50% or 100%, and video quality levels for the MTTs in each task set were uniform over $[1, 8]$, $[1, 6]$, $[7, 8]$, or $[1, 4]$. Since there is little freedom when choosing task parameters for the MTTs in these task sets, only 10 task sets were generated for each combination of the above parameters. All results are shown in Table 5. In almost all cases, the best-performing heuristic (from Sec. 4.2) outperforms G-EDF, resulting in an average 10.65% increase in IPC over all experiments. Note that an increase in IPC can allow for a proportionate increase in the number of videos supported by the platform, an increase in the space searched for each video during motion estimation (to improve encoding quality), or upgrades in the quality level of some videos.

Other observations. We make several additional observations about the use of our heuristics for video encoding. First, if we shift all job releases and deadlines right by an amount equal to the tardiness bound B, and allow all jobs to be executed up to B time units before their actual release times, then deadlines will never be missed if we allow a *preprocessing interval* of length B before any job is released. Tardiness bounds are generally low enough (*e.g.*, hundreds of milliseconds) that such an interval will usually be acceptable to an end-user. This is similar to a method described in [1], but occurs in the opposite direction. Second, note that if execution costs drop due to cache performance improvements, then tardiness bounds should also decrease [17]. Finally, we could use our heuristics to ensure that the real-time workload typically occupies only a *portion* of the shared cache by setting the “cache size” artificially low. In this way, our heuristics allow us to create “soft” cache partitions.

5 Concluding Remarks

In this paper, we proposed heuristics to improve the performance of shared caches on multicore platforms while ensuring real-time guarantees in the form of bounded tardiness. We showed that when a suitable heuristic is employed for a real-time workload, cache performance significantly improves. Our case study showed that cache performance improvements can translate into performance improvements for higher-level metrics that can be perceived by an end user.

Parameters		L2 Miss Rate			Instrs. per Cycle (IPC)		
Sys. Util.	Video Qual.	G-EDF	Heur.	% Impr.	G-EDF	Heur.	% Impr.
50%	[1, 8]	25.97	17.12	**34.06**	1.36	1.30	-4.37
50%	[1, 6]	27.55	17.12	**37.86**	1.30	1.42	9.26
50%	[7, 8]	74.52	60.09	19.36	0.24	0.26	**10.87**
50%	[1, 4]	28.34	16.45	**41.94**	1.29	1.38	6.62
100%	[1, 8]	25.21	18.22	**27.75**	1.29	1.29	0.05
100%	[1, 6]	26.15	18.00	**31.16**	1.27	1.38	8.42
100%	[7, 8]	55.36	17.70	**68.04**	0.22	0.32	**44.44**
100%	[1, 4]	30.85	16.94	**45.10**	1.23	1.35	9.85

Table 5: Results for video-encoding MTTs (best results in bold).

Our results suggest a number of avenues for further research. First, our experiments currently use relatively simple memory reference patterns, due to our reliance on WSS as our cache-profiling metric. We plan to undertake significant work in an upcoming paper to determine a more accurate cache-profiling metric that is suitable for soft real-time applications. This metric should be obtainable dynamically, and should identify cooperating tasks that can be placed into MTTs. Second, we wish to investigate how system and synchronization overheads affect our heuristics. Finally, we plan to showcase our heuristics by implementing those that perform best within LITMUS^{RT} [5], and evaluating them for real (multimedia) applications on a multicore platform. This will include minimizing the scheduling overhead of our heuristics, and determining how to best measure and refine task execution costs when cache performance improves.

Acknowledgement: We would like to thank Ketan Mayer-Patel for discussions regarding MPEG-2 encoding.

References

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

[2] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symp.*, pp. 179–190. IEEE, 2006.

[3] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. Technical Report TR-051101, Department of Computer Science, Florida State University, 2005.

[4] G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. *Proc. of the 16th ACM Symp. on Parallelism in Algorithms and Architectures*. ACM, 2004.

[5] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS^{RT}: A status report. *Proc. of the 9th Real-Time Workshop*, pp. 107–123. Real-Time Linux Foundation, 2007.

[6] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: to block or not to block, to suspend or spin? *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symp.*, to appear. IEEE, 2008.

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

[8] H. Chen, K. Li, and B. Wei. Memory performance optimizations for real-time software HDTV decoding. *Journal of VLSI Signal Processing*, pp. 193–207, 2005.

[9] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. *Proc. of the 26th IEEE Real-time Systems Symp.*. IEEE, 2005.

[10] C. Farivar. Intel Developers Forum roundup: four cores now, 80 cores later. http://www.engadget.com/2006/09/26/intel-developers-forum-roundup-four-cores-now-80-cores-later/, 2006.

[11] 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.

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

[13] R. Fernando, editor. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Publishers, 2004.

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

[15] N. Jouppi. CACTI website. http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html.

[16] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning on a chip multiprocessor architecture. *Proc. of Parallel Architecture and Compilation Techniques*, 2004.

[17] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Proc. of the 28th IEEE Real-Time Systems Symp.*, 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] 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. *Proc. of 7th Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2004.

[20] J. Renau. SESC website. http://sesc.sourceforge.net.

[21] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. *Proc. of SIGMETRICS 2002*. ACM, 2002.

[22] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *IEEE Multimedia Systems*, pp. 197–208. IEEE, 1996.