ON THE DESIGN AND IMPLEMENTATION OF A CACHE-AWARE SOFT REAL-TIME SCHEDULER FOR MULTICORE PLATFORMS

John Michael Calandrino

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2009

Approved by:

James H. Anderson

Sanjoy Baruah

Scott Brandt

Kevin Jeffay

Ketan Mayer-Patel

Frank Mueller

ABSTRACT

JOHN MICHAEL CALANDRINO: On the Design and Implementation of a Cache-Aware Soft Real-Time Scheduler for Multicore Platforms
(Under the direction of James H. Anderson)

Real-time systems are those for which timing constraints must be satisfied. In this dissertation, research on multiprocessor real-time systems is extended to support multicore platforms, which contain multiple processing cores on a single chip. Specifically, this dissertation focuses on designing a cache-aware real-time scheduler to reduce shared cache miss rates, and increase the level of shared cache reuse, on multicore platforms when timing constraints must be satisfied. This scheduler, implemented in Linux, employs: (1) a scheduling method for real-time workloads that satisfies timing constraints while making scheduling choices that reduce shared cache miss rates; and (2) a profiler that quantitatively approximates the cache impact of every task during its execution.

In experiments, it is shown that the proposed cache-aware scheduler can result in significantly reduced shared cache miss rates over other approaches. This is especially true when sufficient hardware support is provided, primarily in the form of cache-related performance monitoring features. It is also shown that scheduler-related overheads are comparable to other scheduling approaches, and therefore overheads would not be expected to offset any reduction in cache miss rate. Finally, in experiments involving a multimedia server workload, it was found that the use of the proposed cache-aware scheduler allowed the size of the workload to be increased.

Prior work in the area of cache-aware scheduling for multicore platforms has not addressed support for real-time workloads, and prior work in the area of real-time scheduling has not addressed shared caches on multicore platforms. For real-time workloads running on multicore platforms, a decrease in shared cache miss rates can result in a corresponding decrease in execution times, which may allow a larger real-time workload to be supported, or hardware requirements (or costs) to be reduced. As multicore platforms are becoming ubiquitous in many domains, including those in which real-time constraints must be satisfied, cache-aware scheduling approaches such as that presented in this dissertation are of growing importance. If the chip manufacturing industry continues to adhere to the multicore paradigm (which is likely, given current projections), then such approaches should remain relevant as processors evolve.

To Liz, who will always be my constant.

ACKNOWLEDGMENTS

Neither this dissertation nor my graduate school career would have been possible without the help of a lot of people. I would first like to thank my committee: James Anderson, Sanjoy Baruah, Scott Brandt, Kevin Jeffay, Ketan Mayer-Patel, and Frank Mueller, for their help and feedback along the way. I would especially like to thank Jim, who was also my advisor and committee chair, for transforming a graduate student that had a less-than-perfect experience at Cornell into one that can both "do research" and really enjoy it at the same time. I would also like to thank the former and current real-time graduate students at UNC that I have known for their contributions to my transformation: Aaron Block, Björn Brandenburg, Uma Devi, Hennadiy Leontyev, Cong Liu, and Glenn Elliott; I would thank those with whom I have co-authored papers a second time. The real-time community as a whole also deserves credit—they have made most of my conference experiences a lot of fun, and I have been excited to be part of their research community over the last few years. My collaborators at Intel also deserve a large amount of credit for shaping the direction of the research that is presented in this dissertation: Scott Hahn, Dan Baumberger, Tong Li, and Jessica Young. Their advice was invaluable, and without their help and assistance during both my internship in the summer of 2006 and the visits that followed, I think that this dissertation would have been less interesting. I would also like to thank the UNC Department of Computer Science as a whole, which in my opinion offers an unusually positive and friendly environment in which to be a graduate student in computer science. I have also had some great teachers during my time in the department, where I believe that I learned more about computer science during two years of taking courses at UNC than I had learned during my previous six years as a student.

There are also those outside of the field of computer science that deserve credit for this dissertation. I would like to thank everyone at the UNC Institute of Marine Sciences (IMS), located on the North Carolina coast, where my wife was a graduate student. The people there were very supportive of me, especially during the summer of 2008, when I implemented a large portion of the cache-aware scheduler that is described in this dissertation while sitting at a desk at IMS. While I was at the coast, I essentially became an honorary IMS graduate student, thanks in part to my wife's roommates at the coast: Sandra Mesquita and Angie Coulliette. Finally, I would like to thank my wife, Liz, for (at least) the following: **(1)** an incredible amount of support; **(2)** tolerating someone that wanted to spend eleven years as a student; and **(3)** making sure that I always had plenty of homemade baked goods. The ongoing scavenger hunt that we have for National Park stamps made this dissertation possible in its own way as well.

Thanks again everyone, and enjoy the dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Real-time systems are those for which timing constraints must be satisfied. The goal of this dissertation is to extend research on multiprocessor real-time systems to support multicore platforms, or platforms containing multiple processing cores on a single chip. This research is important as multicore platforms are quickly becoming ubiquitous in the desktop, server, and embedded domains—including in settings where real-time constraints must be satisfied. In this dissertation, our focus is on reducing the miss rates of shared caches, and increasing the level of reuse for these caches, within such platforms while ensuring the timing constraints of a real-time workload. In multicore platforms, reducing shared cache miss rates can result in decreased execution times, which may allow a larger real-time workload to be supported or hardware requirements (or costs) to be reduced. To this end, we developed a *cache-aware real-time scheduler* for Linux that employs: (1) a scheduling method for real-time workloads that satisfies timing constraints while making scheduling choices that reduce shared cache miss rates; and (2) a profiler that quantitatively approximates the cache impact of every task during its execution. The design and implementation of this scheduler is described herein. Note that the scheduling overheads that are associated with our implementation are of critical importance—otherwise, such overheads could offset any reduction in cache miss rates.

Prior to the research in this dissertation, no cache-aware real-time scheduling algorithm had been proposed that addresses these practical concerns—prior work on scheduling in the presence of shared caches on multicore platforms had not addressed support for real-time workloads, and prior work in the area of real-time scheduling had not addressed shared caches on multicore platforms. It is at the intersection of these two research areas that this dissertation makes its primary contribution.

In the sections that follow, we first provide an introduction to real-time systems. We then motivate the need for cache-aware real-time scheduling with an introduction to multicore architectures. This is followed with a discussion of target applications. We then state the thesis and contributions of this dissertation. Finally, we conclude by outlining how the remainder of this dissertation is organized.

1.1 Real-Time Systems

Real-time systems include a notion of temporal correctness in addition to logical correctness. That is, we not only want such systems to perform correct operations, but to perform them at the correct times. In other words, timing constraints must be ensured. For example, a video application might need to decode and display a video frame every 33 ms, in order for playback to look "smooth" to an end user when a (roughly) 30 frame-per-second frame rate is desired.

Most real-time workloads contain tasks that require computation time at recurring intervals. For example, the workload of the aforementioned video application could be represented as a sequence of consecutive 33 ms intervals, where a certain amount of computation time is required during each such interval. Such timing constraints would result in a natural division of the workload into recurring *jobs*, each with its own *deadline*. The scheduler must ensure that jobs are allocated sufficient processing capacity so that their timing constraints are met.

In recent years, interest in techniques for effectively scheduling real-time workloads on *multiprocessor systems* has been increasing for several reasons. First, recent research on this topic has led to the development of new scheduling approaches and analytical results that remove some of the theoretical constraints imposed by prior work. This research includes Pfair scheduling and the notion of bounded tardiness as an acceptable type of real-time guarantee; these concepts will be elaborated upon in Chapter 2. Second, multimedia applications, perhaps the most ubiquitous class of real-time applications, are increasing in complexity, and it may soon be impossible to achieve acceptable performance from such applications without explicitly providing real-time support within general-purpose operating systems. Third, as mentioned earlier, chip makers have been shifting to multicore processor designs.

More recently, a desire has been expressed to provide better real-time support within general-purpose operating systems such as Linux. These efforts have involved supporting recurrent task models natively within Linux, rather than attempting to emulate this support using the Portable Operating System Interface for Unix (POSIX) real-time extensions [68], which provide no support for recurrent workloads. Native support for recurrent task models would be especially useful for many applications that naturally exhibit such a pattern of execution, or that require a proportionate share of the available processing resources.

Given this convergence of events, the time is now ripe to extend prior work on techniques for scheduling real-time workloads on multiprocessors, particularly in ways that will benefit multicore platforms. The specific challenges associated with such platforms that are of concern to us in this dissertation are outlined next.

1.2 Multicore Architectures

In multicore architectures, multiple processing cores are placed on the same chip. Most major chip manufacturers have adopted these architectures due to the thermal- and power-related limitations of single-core designs [50, 53]. Dual-core chips are now commonplace, and numerous four- and eight-core options exist. Further, per-chip core counts are expected to increase substantially in the coming years [53]. For example, Intel has claimed that it will release 80-core chips as early as 2013 [1]. Additionally, Azul, a company that creates machines for handling transaction-oriented workloads, currently offers the 54-core Vega 3 processor, which is used in systems with 864 total cores [7]. The shift to multicore technologies is a watershed event, as it fundamentally changes the "standard" computing platform in many settings to be a multiprocessor.

1.2.1 Shared Caches

On most current multicore platforms, different cores share on-chip caches. Without effective management by the scheduler, such caches can become performance bottlenecks if cache *thrashing* is allowed to take place. Thrashing occurs when the demand for space in the shared cache (*e.g.*, the amount of cache desired by all tasks scheduled at that time) outpaces the

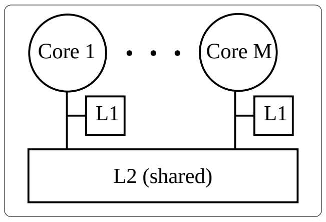


Figure 1.1: Multicore architecture with a private L1 cache per core, and an L2 cache shared by all M cores.

cache size—as a result, tasks that are executing at that time experience high cache miss rates since their data is frequently evicted from the cache before it can be reused. In the case of a lower-level cache (*i.e.*, a cache that is closer to main memory), this can result in high average memory reference times, due to a need to frequently access main memory, which is much slower than the cache. This, in turn, results in a severe overall degradation in system performance. For these reasons, the issue of efficient cache usage on multicore platforms is considered by chip makers to be one of the most important problems with which they are currently grappling. By effectively addressing this issue, the parallelism within these systems can be better exploited.

In this dissertation, we address this issue in the context of *real-time systems* implemented on a multicore platform where all cores are symmetric and share the lowest-level cache (*i.e.*, the level of cache that is most distant from the cores, and closest to main memory), as shown in Figure 1.1. This architecture is fairly common—the Sun UltraSPARC T1 and T2 processors have a lowest-level L2 cache shared by eight cores, and the recently-released Intel Core i7 chip contains a lowest-level L3 cache shared by four cores (in this case, each core contains private L1 *and* L2 caches).

In prior work pertaining to non-real-time systems, Fedorova *et al.* [32] showed that shared cache miss rates affect overall system performance to a much greater extent than the performance of many other processor components. For example, for the chip shown in Figure 1.1, shared L2 misses would be expected to affect performance considerably more than private L1 misses or pipeline conflicts. This is precisely due to the severe performance degradation that would be expected when miss rates are high or thrashing occurs with respect to the lower-level caches, as noted earlier. At any given time, the miss rate of a shared cache at

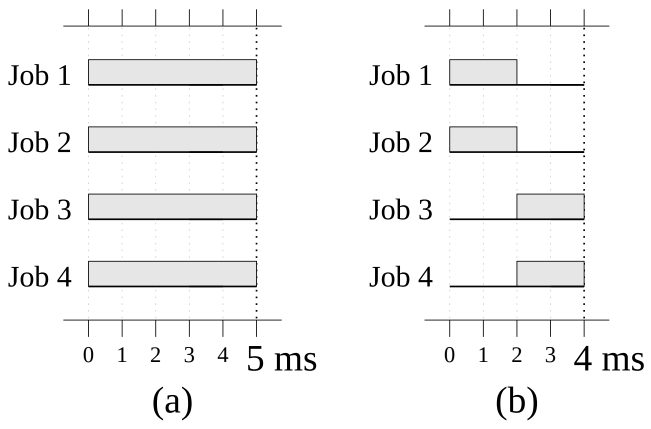


Figure 1.2: The impact of co-scheduling on shared cache thrashing and response times.

time t is strongly determined by the set of applications co-scheduled at that time: a set of applications is said to be *co-scheduled* at some time t if they are scheduled concurrently at time t. Fedorova *et al.* showed that shared cache miss rates can be reduced, and system performance improved, by *discouraging* a set of applications from being co-scheduled when doing so would cause thrashing in the L2 cache.

Example (Figure 1.2). To demonstrate the impact of co-scheduling on shared cache thrashing, consider the example in Figure 1.2, where four jobs need to be scheduled on a four-core platform, and all four cores share a cache. Each job requires 2 ms of execution time when no cache thrashing occurs, or 5 ms of execution time if the cache is thrashed. Further, assume that at most two jobs can be co-scheduled without thrashing the cache. If all four jobs are co-scheduled, each on a different core, then thrashing occurs, and all jobs complete after 5 ms (inset (a)). However, if only two jobs are co-scheduled, then two jobs execute concurrently, followed by the other two jobs, resulting in all jobs completing after 4 ms (inset (b)). This example demonstrates that there are cases where we can achieve better system performance by *reducing* parallelism, if doing so avoids cache thrashing, instead of maximizing parallelism at all costs. An additional implication is that a cache management policy that is cognizant of these issues is likely to have a significant impact on system performance. □

The problems addressed in this dissertation were initially motivated by the work of Fedorova *et al.*—we want to show that, in real-time systems, co-scheduling can be influenced in ways that reduce shared cache miss rates *while ensuring real-time constraints*. It is our focus on real-time constraints that distinguishes our work from that of Fedorova *et al.* To the best of our knowledge, we are the first to consider mechanisms for influencing co-scheduling choices when analysis *validating* real-time constraints is required.

Additionally, our methods take this work one step further by *encouraging* the co-scheduling of applications when it would *reduce* shared cache miss rates, typically because the applications (or the individual tasks associated with such applications) reference the same data in memory. The benefit of encouraging co-scheduling in these cases is that it increases opportunities for data in the cache to be reused.

The general problem of influencing co-scheduling in these ways while respecting real-time constraints is NP-hard in the strong sense. (This is unlikely to be a surprise to the reader, as many multi-dimensional scheduling problems are similarly difficult.) In Chapter 3, we present a formal statement of this co-scheduling problem, and prove that it is NP-hard in the strong sense, by a transformation from 3-PARTITION.

1.2.2 Related Work

Substantial additional prior work exists that is related to this co-scheduling problem, especially in the non-real-time domain. We briefly present only a few examples here—a larger body of work is considered in Chapter 2.

Batat and Feitelson [10] found that memory requirements should be taken into account during the co-scheduling of a multithreaded application—specifically, that it is more efficient to delay the execution of some threads when memory requirements cannot be met than to execute all threads concurrently, even if this results in a loss of processing capacity. This study (which did not consider real-time requirements) is particularly interesting considering that we take a similar approach to co-scheduling, which we elaborate upon further later in this dissertation.

In work on parallel computing, Peng *et al.* [55] found that the memory-reference patterns of threads can lead to co-scheduling choices that are either *constructive* or *disruptive*. Constructive choices, such as co-scheduling applications that share data, decrease shared cache miss rates, while disruptive choices increase miss rates. Scenarios should be avoided where the benefits of parallelization are offset by disruptive co-scheduling choices—in our case, those that lead to shared cache thrashing or high shared cache miss rates. To this end, the authors of [55] conducted a study to determine the memory reference patterns for multimedia and artificial intelligence applications. The results of this study can assist in making better co-scheduling decisions involving these large application classes.

Cache fairness. In another paper by Fedorova *et al.* [33] (with significant overlap with the authors of [32]), a cache-aware scheduling approach is proposed that encourages *cache fairness*, or fair allocation of a shared cache among threads. Quality-of-service requirements, such as those related to thread completion times, are considered only experimentally, and are primarily motivated by cache fairness; no real-time analysis is presented. Other research in the area of cache fairness includes work of Kim *et al.* [42], who quantified cache fairness using several different metrics, and presented a cache-partitioning scheme that uniformly distributes the impact of cache contention among co-scheduled threads.

Symbiosis-aware scheduling. Additional related work that lacks real-time analysis includes work on *symbiosis-aware scheduling* [40, 60, 52] in (non-multicore) systems where multiple hardware threads contend for shared resources within the same single-core chip. In symbiosis-aware scheduling, the goal typically is to maximize the overall "symbiosis factor," which indicates how well various thread groupings perform when co-scheduled, where performance is primarily determined by per-thread execution times.

Worst-case execution time analysis. Work also exists that is related to shared-cache-aware real-time *worst-case execution time* (WCET) analysis, where an upper bound on the execution time of a single job of a task is derived (*e.g.*, in [57, 54]). Such work may be considered to be more in line with the goals of this dissertation; however, this work only tangentially addresses cache-aware real-time *scheduling*, and does not address the issue of efficiently profiling the cache behavior of real-time tasks *during* execution.

1.3 Application Domains

In this section, we describe several application domains that may benefit from the cache-aware real-time scheduling support described in this dissertation.

1.3.1 Multimedia Applications

Multimedia applications are the most obvious beneficiary of this work. Such applications comprise one of the most ubiquitous types of real-time workloads, especially within general-purpose operating systems such as Linux, and are very likely to run on multicore platforms for that reason. Given the current multicore trend, these applications will need to become *multithreaded* as the demanded video quality increases, and there will exist a need to correctly coordinate the timing of these threads.

Video encoding applications have real-time constraints and are both compute- and memory-intensive. Most video encoding requires a search as part of motion estimation. Within a multimedia server that encodes live media streams (each encoded by a separate application that corresponds to a one or more real-time tasks), a reduction in cache miss rates may allow more memory to be referenced and a more extensive search to be conducted in the same amount of execution time, thus improving video quality. Alternately, our cache-aware scheduler may reduce thrashing and result in lower execution-time requirements. This, in turn, would allow a greater selection of media streams, or a greater number of clients, to be supported without upgrading hardware.

One envisioned application of larger multicore platforms is to act as *multi-purpose home appliances*, where the computing requirements of a household are pooled into a single powerful general-purpose machine [38]. The cost savings of such an approach could be substantial. Such computing requirements may include:

- Supporting a wide variety of multimedia applications (*e.g.*, streaming from multiple live and stored video sources to displays throughout the home, recording video for playback later, or supporting videoconferencing sessions). Video sources could include live broadcast television, on-demand services, or the Internet.

- Periodically monitoring various conditions of the home, such as temperature, and responding appropriately.

- Providing general-purpose computing capability as it is needed, perhaps through one or more user terminals that would emulate the functionality of a desktop system.

Naturally, the multimedia demands on such an appliance could be considerable, requiring a large number of high-quality (*e.g.*, HDTV) video encoding and decoding applications to be concurrently supported. In such an environment, providing real-time guarantees for such applications would be a more direct way of supporting their computing needs, potentially resulting in more efficient use of the underlying general-purpose hardware. This, in turn, could make multi-purpose home appliances more viable, or increase their capabilities. Ultimately, reductions in cache miss rates could directly translate into an improved experience for the end users of the services provided by such appliances.

1.3.2 Gaming and Real-Time Graphics

Real-time graphics applications are typically assisted by graphics processing units (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. By reducing the execution-time requirements for such applications through the use of our cache-aware scheduler (which could be used to manage any similar "cache-like" memory within GPUs), the processor-intensive applications typically associated with real-time graphics and gaming could be made viable on less-powerful platforms.

Two additional trends are of note. First, applications with similar characteristics to those commonly supported for real-time graphics may also see a performance benefit as interest in treating GPUs as co-processors for general computation continues to increase [34]. Second, if the core counts of CPUs increase as projected, and cores become simpler or more specialized, then the research within this dissertation might enable high-quality real-time graphics to be supported without the need for a specialized GPU or other hardware. Similarly, the cost of specially-designed hardware (*e.g.*, within video game consoles) could be reduced, as a smaller cache or less processing capacity may be acceptable when the provided resources are better utilized. Overall, the benefit would be cost reduction, whether those costs are purely monetary or related to chip area or energy.

1.3.3 High-Performance Computing

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. When running a single high-performance computing application on multiple processors, 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 every high-performance computing application were represented as one or more real-time tasks, then a real-time scheduler that attempts to make constructive co-scheduling choices could assist with the synchronization requirements of the application. This might allow such an application to finish earlier, or could allow multiple such applications to be supported, since an application may no longer require exclusive access to the processors that it is assigned.

1.4 Thesis Statement

The major limitation of prior approaches for multiprocessor real-time scheduling is that they are they are cache-agnostic; this can result in ineffective use of shared caches, either due to cache thrashing, missed opportunities for cache reuse, or an unnecessary loss of cache affinity (*e.g.*, due to preemptions). The result is that task execution-time requirements may need to be higher than if the cache were effectively used. We attempt to address this limitation in this dissertation. The thesis statement to be supported is the following.

> *Multiprocessor real-time scheduling algorithms can more efficiently utilize multi-core platforms when scheduling techniques are used that reduce shared cache miss rates. Such techniques can result in decreased execution times for real-time tasks,*

thereby allowing a larger real-time workload to be supported using the same hardware, or enabling costs (related to hardware, energy, or chip area) to be reduced.

1.5 Contributions

In this section, we present an overview of the contributions of this dissertation.

1.5.1 Cache-Aware Real-Time Scheduling Methods

The first major contribution of this dissertation is the creation of cache-aware scheduling heuristics that influence co-scheduling choices so that cache miss rates are reduced. As part of this contribution, we introduce the *multithreaded task* (MTT) abstraction. MTTs were first introduced in prior work leading to this dissertation [3] as a first step towards representing concurrency within task models that typically handle only the sequential execution of tasks (*e.g.*, the periodic and sporadic task models that are introduced in Chapter 2 and used throughout this dissertation). In MTTs, multiple (sequential) real-time tasks work together to perform the same operation. MTTs arise naturally in many settings. For example, multiple tasks might perform different functions on the same video frame, or the same function on overlapping portions of the same frame, with a common period implied by the desired frame rate (Section 1.3 provided additional examples). The benefits of co-scheduling tasks within the same MTT are taken into account in the design of our cache-aware scheduler.

It is projected that as per-chip core counts increase, the processing power of individual cores is likely to remain the same (or even decrease if cores become simpler) [53]. As a result, exploiting parallelism will be essential to achieving performance gains. This fact suggests the need to investigate abstractions for parallel execution in real-time systems (and most other sub-disciplines of computer science), and it is for this reason that MTTs are an important contribution of this dissertation.

In our cache-aware scheduler, co-scheduling is *encouraged* for tasks within the same MTT, and *discouraged* for tasks (within different MTTs) when it would cause shared cache thrashing. Note that devising methods to support co-scheduling while ensuring real-time constraints is quite difficult; as stated earlier, the problem of influencing co-scheduling in these ways *while* *ensuring real-time constraints* is NP-hard in the strong sense. As this dissertation progresses, we will find that designing even a sub-optimal approach that imposes few constraints on the types of task sets that can be supported is difficult. Further, an additional constraint on any scheduling heuristic that we choose to implement within a real operating system is the scheduling overheads associated with the heuristic—this issue is also addressed herein.

1.5.2 Online Cache Profiling of Real-Time Workloads

The second major contribution of this dissertation is an automatic cache profiler that determines the cache behavior of real-time tasks during execution, and supplies this information to the employed scheduling heuristic. The metric of interest to us is *MTT per-job working set size* (WSS), which for the purposes of this dissertation is the size of the per-job cache footprint of an MTT. The size of this footprint is estimated online using hardware performance counters. By profiling the cache behavior of tasks during their execution, the need to profile tasks offline before execution is eliminated. This would make profiling less of an inconvenience or impossibility for certain types of workloads.

In prior work, Knauerhase *et al.* [43] investigated the use of performance counters to reduce cache miss rates and improve system performance for throughput-oriented tasks. Additionally, in the real-time domain, Pellizzoni *et al.* [54] 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 dissertation.

1.5.3 Implementation and Evaluation

As a "proof of concept" that our scheduler and profiler are viable in practice, we implemented our cache-aware scheduler within Linux. This scheduler includes both a scheduling heuristic (as described in Section 1.5.1) and our cache profiler (as described in Section 1.5.2). This implementation is empirically evaluated on several multicore platforms under both synthetic and multimedia workloads. Demonstrating the practicality of such a scheduler within a general-purpose operating system is crucial, as many users would prefer to run certain real-time applications (*e.g.*, video players) in Windows or Linux (environments that may be more familiar and comfortable) instead of a specialized real-time operating system. We show that our scheduler often achieves a substantial reduction in shared cache miss rates over other multiprocessor real-time scheduling approaches, and since overheads are low, this translates into better overall system performance.

Our experimental evaluation, presented in Chapter 6, was conducted under both Linux and a hardware (processor and memory) architecture simulator. The use of an architecture simulator allowed us to get detailed results on the performance of our scheduling heuristics in a controlled environment, and to experiment with systems that are not commonly available today, to assess if our heuristics are likely to continue to have a performance impact as multicore architectures evolve. An initial evaluation of heuristics within the simulator, combined with some practical aspects of implementing these heuristics, guided our selection of which scheduling heuristic to use within Linux.

The presented evaluation suggests that eliminating thrashing and reducing miss rates in shared caches should be first-class concerns when designing real-time scheduling algorithms for multicore platforms with shared caches. 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.

1.6 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we review relevant prior work on real-time systems and cache-aware scheduling. In Chapter 3, we formally state the problem that this dissertation seeks to address, prove its intractability, and discuss early attempts at a solution. In Chapters 4 and 5, we describe the design and implementation of our cache-aware scheduling heuristics and cache profiler, respectively. In Chapter 6, we present an experimental evaluation of our scheduler under both an architecture simulator and Linux. Finally, we conclude in Chapter 7.

CHAPTER 2

PRIOR WORK

In this chapter, we review prior work that is related to this dissertation. We begin with an overview of real-time scheduling, including multiprocessor scheduling. This is followed by research on the scheduling and cache profiling of non-real-time tasks in the presence of shared caches, which are common on multicore platforms. We then address other relevant real-time systems research, including work on real-time operating systems and real-time scheduling in the presence of hardware multithreading.

2.1 Real-Time Scheduling Overview

In this section, we present work related to real-time scheduling. We begin with an overview of concepts in real-time systems, and then discuss the additional challenges presented by multiprocessor scheduling.

2.1.1 Recurrent Task Model

As indicated in Chapter 1, most real-time workloads require computation time at recurring intervals. Within the real-time research community, one of the simplest and most common ways of specifying recurrent workloads is the *periodic task model*, which was introduced by Liu and Layland [46]. In this model, the scheduling of a system of tasks τ is considered. Each task $T \in \tau$ is specified by a worst-case per-job *execution cost* $e(T)$ and *period* $p(T)$. The *utilization* of a task T is its execution cost divided by its period, or $u(T) = e(T)/p(T)$. Tasks release jobs T_1, T_2, $\ldots$, at the start of each period beginning at time zero—the release time of job T_i is denoted as $r(T_i)$. Such a released job T_i is considered to be *eligible* at time t if all jobs T_1 through T_{i-1} have completed by time t, and T_i has not completed execution at time

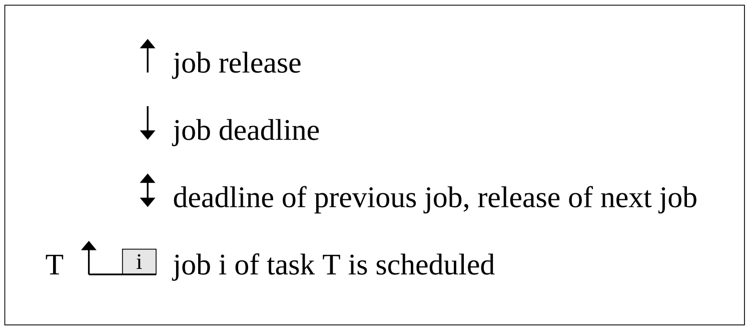

job release
job deadline
deadline of previous job, release of next job
T
i
job i of task T is scheduled

Figure 2.1: Conventions used in example schedules throughout this dissertation.

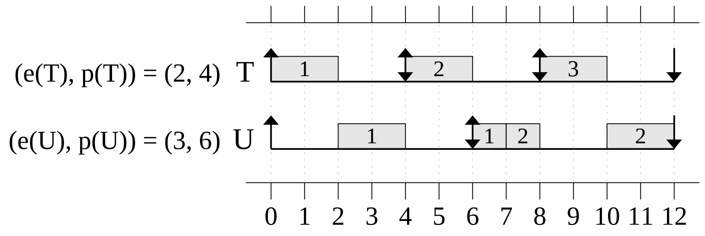


Figure 2.2: An example schedule for a periodic task system.

t. The *absolute deadline* $d(T_i)$ of a job T_i is $r(T_i) + D(T)$, where $D(T)$ is the *relative deadline* of T. If job T_i completes its execution after time $d(T_i)$, then it is *tardy*. Throughout this dissertation, we assume that deadlines are *implicit*, that is, $D(T) = p(T)$. Assuming implicit deadlines, the absolute deadline of a job T_i coincides with the release time of job T_{i+1}, or $d(T_i) = r(T_{i+1})$. As such, we do not explicitly specify relative deadlines for any task system considered in this dissertation, and we refer to the absolute deadline of a job as simply its *deadline*.

In this dissertation, schedules are shown for a wide variety of scheduling policies and task sets. Figure 2.1 describes several conventions that are used for these schedules unless otherwise noted. Unless otherwise noted, all schedules in this dissertation assume zero scheduling-related overheads. We also assume that all task sets are *synchronous*, meaning that all tasks in a task set release their first job at the same time.

Example (Figure 2.2). Consider the example in Figure 2.2, which consists of two periodic real-time tasks (with implicit deadlines) scheduled on a single processor. The release time and deadline of each job T_i is $(i-1) \cdot p(T)$ and $i \cdot p(T)$, respectively. Note that job releases and deadlines are typically combined, since the release time of job T_{i+1} coincides with the deadline of job T_i. In this example, task T is statically prioritized over task U. The schedule shown is for a single *hyperperiod* of the task set, whose length is the least common multiple of all task periods. For a given task set, the schedule associated with that task set will repeat across each hyperperiod, as long as all deadlines are met (or the last job of every task released during the hyperperiod meets its deadline). All jobs meet their deadlines in this example except for U_1, which misses its deadline by one time unit. □

A natural extension of the periodic task model, proposed by Mok [49], is the *sporadic task model*. In this model, $p(T)$ specifies the *minimum* separation between job releases of T, rather than the *exact* separation as is the case in the periodic task model. This means that $d(T_i)$ may not be equal to $r(T_{i+1})$, but it must be the case that $d(T_i) \leq r(T_{i+1})$. For example, in Figure 2.2, under the sporadic task model, T_2 could be released at time 5 instead of time 4, but could *not* be released at time 3. Once a job is released, its period still exactly determines its deadline if implicit deadlines are assumed, just as in the periodic task model; that is, for any T_i, $d(T_i) = r(T_i) + p(T)$. Although we present the sporadic task model here for the sake of completeness, throughout the remainder of this chapter and dissertation, *the periodic task model is assumed* unless stated otherwise.

2.1.2 Hard vs. Soft Real-Time Systems

Broadly, real-time systems can be divided into two categories: *hard* and *soft*. In hard real-time systems, missing a deadline is catastrophic; in such systems, *all* deadlines must be met. Examples of hard real-time systems exist in safety-critical domains including transportation, aviation, and defense. In soft real-time systems, deadline misses can be tolerated in some cases. Further, in hard real-time systems, worst-case execution costs must be carefully determined, whereas in soft real-time systems, such costs can be determined less precisely (*e.g.*, by executing a large number of jobs of a task and using the maximum observed execution time as the worst-case execution cost). Examples of soft real-time systems include multimedia applications for which buffering may be employed or an occasional missed frame is permissible, certain types of real-time transactions (*e.g.*, those that react to changes in the stock market), and applications for which guaranteeing a *share* of the system over time is of greater importance than meeting explicit deadlines.

While the primary objective of a real-time scheduling algorithm is to ensure timing constraints, recent results by Leontyev and Anderson [44] and Devi and Anderson [30] imply that soft real-time constraints provide sufficient flexibility to consider secondary objectives. In this dissertation, reducing miss rates in *shared caches* that are present in multicore platforms is the secondary objective. As explained in Chapter 1, such caches are a key bottleneck that must be addressed within scheduling approaches to achieve good overall system performance.

Several different notions of soft real-time guarantees exist. We now provide an overview of three different types of soft real-time guarantees.

Percentage of deadlines met. This type of soft real-time guarantee concerns the percentage of deadlines met by each task in a given sampling window, and is a function of the *miss ratio*, which is the number of deadline misses divided by the number of jobs in that same sampling window. This type of guarantee was considered by Lu *et al.* [47] and by Jain *et al.* [40]. In [47], feedback control mechanisms are employed to achieve a desired miss ratio for each task or an entire task system, whereas in [40], scheduling in the presence of hardware multithreading is considered, where a task set is considered "schedulable" if at most 5% of the deadlines of any task are missed.

(m, k)-firm guarantees. Alternatively, rather than being concerned with a percentage, we can make a guarantee related to the number of jobs m that must meet their deadlines within any window of k consecutive jobs. This type of guarantee was first introduced by Hamadoui and Ramanathan [36] as an *(m, k)-firm guarantee.* Such a guarantee tends to be stronger than a simple percentage-based guarantee, since the guarantee must hold for *any* window of k jobs rather than just the sampling window, but exceptions do exist. The differences between these two types of guarantees are better shown in Figure 2.3. In insets (a) and (b), the percentage-based guarantee of 50% is met; however, a $(1, 2)$-firm guarantee is not met in inset (b). In fact, even a seemingly weaker $(1, 3)$-firm guarantee is not met. Interestingly, inset (c) presents a case where the $(1, 2)$-firm guarantee is met without meeting the 50% guarantee; however, this is due to the size and position of the sampling window. If the $(1, 2)$-firm guarantee continued

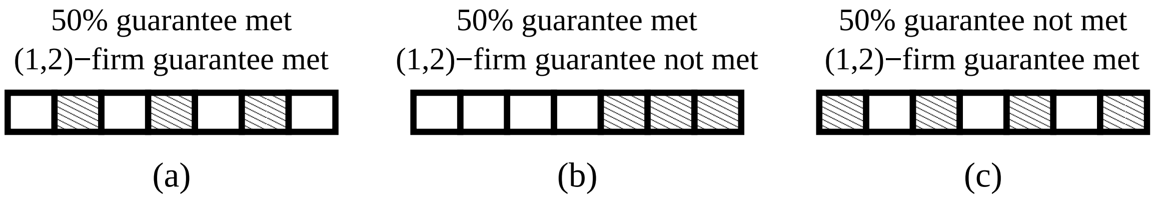


Figure 2.3: Comparison of percentage-based and (m, k)-firm soft real-time guarantees for seven consecutive jobs of the same task, where each job is indicated by a box. A hatched box indicates a deadline miss for the corresponding job.

to be met, then increasing the size of the sampling window by one job, or shifting it left or right by one job, would result in the 50% guarantee being met as well.

Bounded tardiness. The final notion of soft real-time computing considered in this dissertation is bounded tardiness. With this notion, soft real-time guarantees are met if the amount by which deadlines are missed, known as *tardiness*, is bounded. For some scheduling algorithms, tardiness may be bounded by some amount B, meaning that for a task T, any job T_i must complete execution no later than time $d(T_i) + B$, where B is a *tardiness bound.* As an example, in the schedule shown in Figure 2.2, hard real-time constraints cannot be met, since deadlines are missed. Since tardiness is at most one time unit, and therefore bounded, we *can* meet soft real-time constraints if this bound is suitable. Often, this bound is different for different tasks; if a bound on the entire system is desired, then a maximum value over all tasks can be computed. The first major research effort to determine such bounds for a real-time scheduling algorithm was by Srinivasan and Anderson [62]. This effort was later improved upon by Devi and Anderson [29], whose work paved the way for additional studies that provide tardiness bounds for broad classes of scheduling approaches [30, 44]. Further discussion of this work is best preceded by an overview of multiprocessor and global scheduling, which are provided in Sections 2.1.4 and 2.1.5.

Comparison. In this dissertation, bounded tardiness is the notion of soft real-time computing that we consider, since it has a number of advantages over the other notions of soft real-time guarantees. First, the work on bounded tardiness cited earlier allows us to conclude that a wide variety of scheduling approaches *automatically* provide bounded tardiness.

As a result, significant scheduling flexibility exists to influence co-scheduling decisions in an attempt to reduce cache miss rates. Second, bounded tardiness may be more compatible than other notions of soft real-time constraints for applications that can employ buffering, for example, multimedia applications. (An explanation of exactly how buffering can be employed to "hide" tardiness in such applications will be provided in Chapter 4.) Finally, when seeking to guarantee a share of the system to a task over time, rather than meet every deadline, a bound on tardiness is more natural than percentage-based or (m, k)-firm guarantees—as long as tardiness is bounded so that the desired share is maintained over time, the proportion of deadlines missed is unimportant. For these reasons, in the chapters that follow, bounded tardiness is the only notion of soft real-time guarantee that we consider. Henceforth, when discussing soft real-time constraints or guarantees, we mean bounded tardiness unless otherwise noted.

It is worth nothing that, since we implement our cache-aware scheduler within Linux, a number of sources of non-determinism exist (related to running a real operating system on real hardware) that are beyond our control. In such an environment, we interpret "soft real-time" to mean that deadline tardiness on average remains bounded, even if some tasks occasionally misbehave due to effects beyond our control. There are now many advocates of using Linux to support this notion of soft real-time execution. We will revisit this issue when presenting scheduling overheads in Chapter 6.

2.1.3 Uniprocessor Scheduling

In the schedule in Figure 2.2, a uniprocessor is assumed. This schedule is an example of *static-priority* scheduling, in which tasks are statically assigned priorities and scheduled in priority order. Often, such policies are simple to implement, but result in system under-utilization, especially when hard real-time guarantees are desired. In particular, a task set may be feasible, but is not schedulable. A task set is *feasible* when a schedule exists that would allow all timing constraints to be met. For hard real-time systems, this means meeting all deadlines, while for soft real-time systems, this (in our case) means ensuring bounded tardiness. A feasible task set is *schedulable* under a particular scheduling algorithm when it

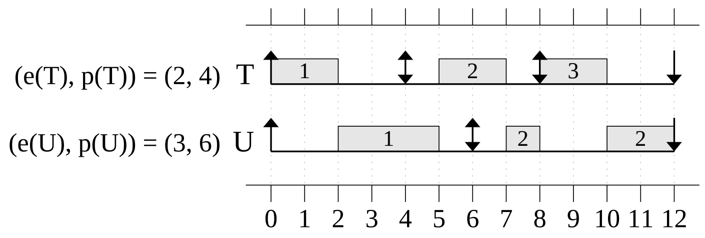


Figure 2.4: An EDF schedule for the task set in Figure 2.2.

is possible to meet the same timing constraints when that algorithm is used. For example, the task set in Figure 2.2 is feasible for hard real-time systems, but not schedulable using static-priority scheduling.

Alternately, we could allow task priorities to change during scheduling, otherwise known as *dynamic-priority* scheduling. One such scheduling algorithm is *earliest-deadline-first* (EDF), where jobs are scheduled in increasing order of their deadlines. Under EDF, all jobs will meet their deadlines as long as the system is not over-utilized—that is, the total utilization of all tasks is at most one [46]. In other words, any task set that is feasible for hard real-time systems on a uniprocessor is schedulable under EDF, since all deadlines will be met. This implies that, for uniprocessors, EDF is an *optimal* scheduling algorithm.

Example (Figure 2.4). Figure 2.4 depicts an EDF schedule for the same task set shown in Figure 2.2. Note that, in this example, all deadlines are met; in particular, U_1 has higher priority than T_2 at time 4, which allows its deadline to be met. Since the total utilization of all tasks is at most one, such a result is guaranteed. □

2.1.4 Multiprocessor Scheduling

Multiprocessor scheduling algorithms employ either a *partitioned* or *global* scheduling approach (or hybrids of the two). Under partitioned scheduling, tasks are statically assigned to processors, and tasks are scheduled on each processor using uniprocessor scheduling policies. An example of such an approach is *partitioned* EDF (PEDF), wherein the partitioned tasks are scheduled on each processor using EDF.

The advantages of a partitioned approach are practical in nature. First, most uniprocessor scheduling policies can be easily converted into multiprocessor scheduling policies by

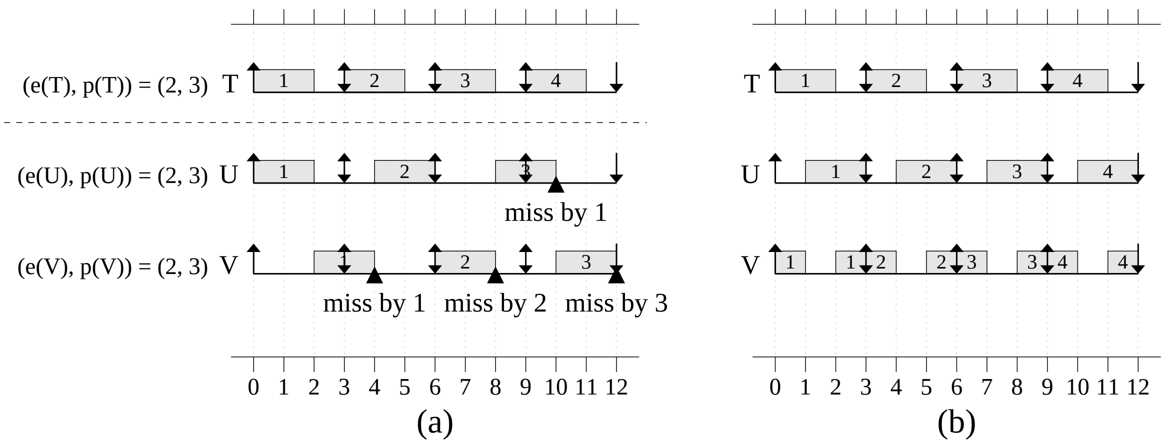


Figure 2.5: An example of how bin-packing affects schedulability under partitioning.

employing partitioning, which makes multiprocessor scheduling (superficially) no harder than uniprocessor scheduling. Second, scheduling overheads associated with such approaches are generally low: scheduling decisions only concern a single processor and a fraction of the tasks in the task set, contention for shared data structures such as run queues is minimal or non-existent, and migrations (which can result in a loss of cache affinity and an increase in cache coherency traffic) cannot occur. (The impact of these overheads will be discussed in greater detail later in Section 2.1.9.)

Partitioning does have some disadvantages, however. First, and most importantly, the management of a globally-shared resource such as a shared cache can become quite difficult under partitioning—this is precisely because each processor is scheduled independently, which was previously argued to be an advantage. Second, partitioning requires solving a bin-packing problem: on an M-processor system, each task with a size equal to its utilization must be placed into one of M bins of size one. The bin in which a task is placed indicates the processor to which it is assigned. There are many feasible task systems for which a bin packing does not exist.

Example (Figure 2.5). A simple example is shown in Figure 2.5, and concerns partitioning three tasks onto a two-core platform. For this and future examples, a dotted line as shown in inset (a) indicates how the tasks are partitioned onto cores. Since each task has a utilization of 2/3, each core can accommodate only one task without being overloaded. As a result, a bin packing does not exist, resulting in one overloaded core and unbounded tardiness, as can be seen in the schedule in inset (a), where PEDF is employed. Therefore, no real-time guarantees can be made; yet, the task set is feasible if one task is allowed to migrate between the two cores, executing half of the time on each, as seen in inset (b). □

In the worst case, bin packing is impossible for certain task sets that require only slightly more than 50% of the available processing resources—that is, their total utilization is slightly greater than $M/2$. Thus, partitioning approaches can result in inferior schedulability, especially when overheads are negligible and *soft* real-time schedulability is the primary concern. For these reasons, we give global scheduling approaches more consideration. These approaches are described next.

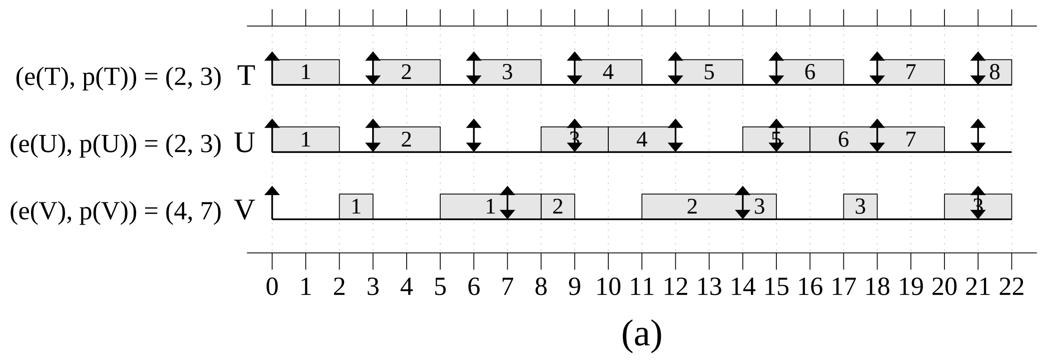
2.1.5 Global Scheduling

In global scheduling algorithms, all processors select jobs to schedule from a single run queue. As a result, jobs may migrate among processors, and contention for shared data structures is likely. Our cache-aware scheduler, in which co-scheduling decisions are influenced to reduce shared cache miss rates, is an example of a global scheduling algorithm.

In this dissertation, we are primarily concerned with two global scheduling policies in addition to our own: *preemptive global* EDF (GEDF) and *non-preemptive global* EDF (NP-GEDF). Under both GEDF and NP-GEDF, jobs are scheduled in order of increasing deadlines, with ties broken arbitrarily (*e.g.*, by task identifier). The difference between GEDF and NP-GEDF is that newly-released jobs can preempt scheduled jobs under GEDF, while under NP-GEDF, a scheduled job runs to completion without preemption.

Example (Figure 2.6). Consider Figure 2.6, which depicts two-core GEDF and NP-GEDF schedules for the same task set. At time 3, job V_1 is preempted in inset (a), while it continues to execute without being preempted in inset (b). As a result, the schedules deviate significantly from that time onwards. V_2 and V_3 are also preempted in inset (a), resulting in similar differences in the two schedules. □

Prior work has shown that a hybrid scheduling approach known as *clustered* EDF (CEDF), first proposed in [19], typically results in better schedulability for soft real-time systems than



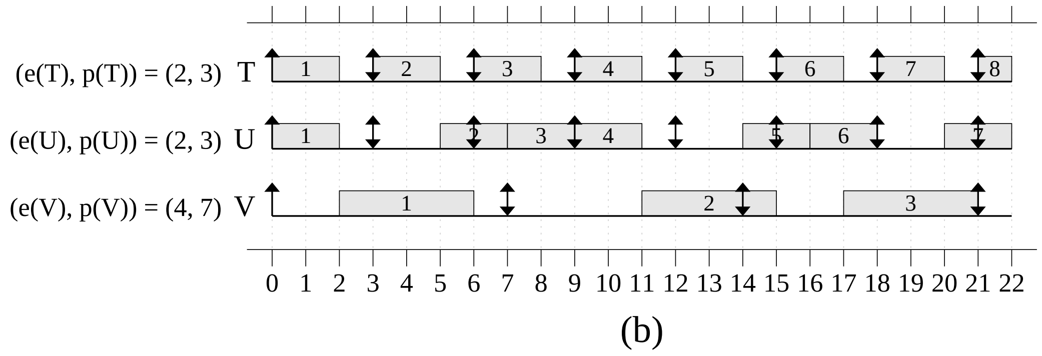


Figure 2.6: Two-core schedules under **(a)** GEDF and **(b)** NP-GEDF for the same task set.

other approaches [15, 22]. Under CEDF, tasks are statically assigned to clusters of cores (preferably clusters that share a cache), and tasks are scheduled on each cluster using GEDF. For platforms with multiple lowest-level shared caches (*e.g.*, a machine with two quad-core chips, where each chip contains a single shared lowest-level cache), these caches could be managed independently using a similar approach, where our cache-aware scheduler (described in Chapters 4 and 5) is used within each cluster instead of GEDF. It is for these reasons that in Chapter 6, GEDF is used as a baseline for evaluating the performance of our cache-aware scheduler. We also introduce NP-GEDF in this dissertation as an example of a non-preemptive policy, since our cache-aware scheduler attempts to emulate such a policy when feasible by encouraging (but not forcing) the non-preemptive execution of jobs.

Note that, in Figure 2.6, deadlines are missed under both approaches. Neither GEDF nor NP-GEDF is optimal; that is, for each algorithm, feasible task sets exist that will result in missed deadlines when scheduled using that algorithm. However, deadline tardiness under both GEDF and NP-GEDF is bounded [30] for all feasible task sets. Thus, we could consider such algorithms to be optimal when *soft* real-time constraints are desired. The tardiness bounds associated with GEDF and NP-GEDF are introduced next.

2.1.6 Tardiness Bounds Under Global Scheduling

We now provide a more detailed discussion of work related to bounded tardiness under global scheduling algorithms. Srinivasan and Anderson [62] undertook the first major effort to determine tardiness bounds for a global multiprocessor real-time scheduling algorithm—in this case, the *earliest-pseudo-deadline-first* algorithm, a type of Pfair algorithm (Pfair scheduling is introduced in Section 2.1.7). Devi and Anderson [29] extended the work of [62] by disproving the tardiness bound of one that was claimed in that work. Devi and Anderson followed this work with tardiness bounds for GEDF and NP-GEDF [30], and Leontyev and Anderson extended that work by providing a generalized tardiness-bound result for global multiprocessor scheduling algorithms [44]; these bounds are stated next.

The tardiness for a task $T \in \tau$ scheduled using GEDF is $x + e(T)$, where x is defined as follows (with minor changes from [30] to maintain consistency with our notation specified earlier).

$$x = \frac{\mathcal{E}_x - min_{U \in \tau}(e(U))}{M - \mathcal{U}_x} \tag{2.1}$$

In (2.1), $\mathcal{E}_x$ is the sum of the Λ highest execution costs over all tasks in τ, and $\mathcal{U}_x$ is the sum of the $\Lambda - 1$ highest utilizations over all tasks in τ, where Λ is defined as follows.

$$\Lambda = \left\lceil \sum_{U \in \tau} u(U) \right\rceil - 1 \tag{2.2}$$

The tardiness for a task $T \in \tau$ scheduled using NP-GEDF is $y + e(T)$, where y is similar to x and defined as follows.

$$y = \frac{\mathcal{E}_y + \mathcal{B}_y - min_{U \in \tau}(e(U))}{M - \mathcal{U}_y} \tag{2.3}$$

In (2.3), $\mathcal{E}_y$ is the sum of the $\Lambda + 1$ highest execution costs over all tasks in τ, $\mathcal{B}_y$ is the sum of the $M - \Lambda - 1$ highest execution costs over all tasks in τ (a term required to fully account for non-preemptive execution), and $\mathcal{U}_y$ is the sum of the Λ highest utilizations over all tasks in τ, where Λ is again defined as in (2.2).

Observe that deadlines are missed by at most one time unit in both schedules shown in Figure 2.6. Using the above formulas to calculate the tardiness bounds under each approach, we would get $\Lambda = \lceil 2/3 + 2/3 + 4/7 \rceil - 1 = 1$, $\mathcal{E}_x = 4$, $\mathcal{U}_x = 0$, $\mathcal{E}_y = 4 + 2 = 6$, $\mathcal{B}_y = 0$, $\mathcal{U}_y = 2/3$, $x = (4-2)/2 = 1$, and $y = (6-2)/(2-2/3) = 3$. The largest execution cost for any task in Figure 2.6 is 4; thus, the resulting tardiness bounds are 5 under GEDF and 7 under NP-GEDF. Typically, it is the case that observed tardiness bounds are considerably lower than any analytical bound; however, analytical bounds are *necessary* in order to make soft real-time guarantees.

As noted earlier, Leontyev and Anderson [44] extended the tardiness-bound proofs in [30] to apply to a wide variety of global scheduling algorithms. In this work, a *priority point* is assigned to each eligible job, with earlier priority points denoting a higher priority (job preemptions may occur). For example, under GEDF, 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 [44] that under any global scheduling algorithm with window-constrained priorities, deadline tardiness is bounded provided that 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.)

Example (Figure 2.7). To demonstrate the use of priority points, consider Figure 2.7, which depicts a two-core schedule for the task set shown assuming the EVEN-ODD policy, created specifically for this example. In the EVEN-ODD policy, the priority point of a job at any time t is its release time if $\lfloor t/4 \rfloor$ is odd and its deadline if $\lfloor t/4 \rfloor$ is even. As we can see in Figure 2.7, this policy results in deadline misses for jobs T_{11} and U_6. However, even though job priorities frequently change, such priorities are window-constrained, and the task set does not over-utilize the system; thus, tardiness is bounded. $\square$

As the reader might expect from the above example, many dynamic-priority multiprocessor real-time scheduling policies are window-constrained and therefore have bounded tardi-

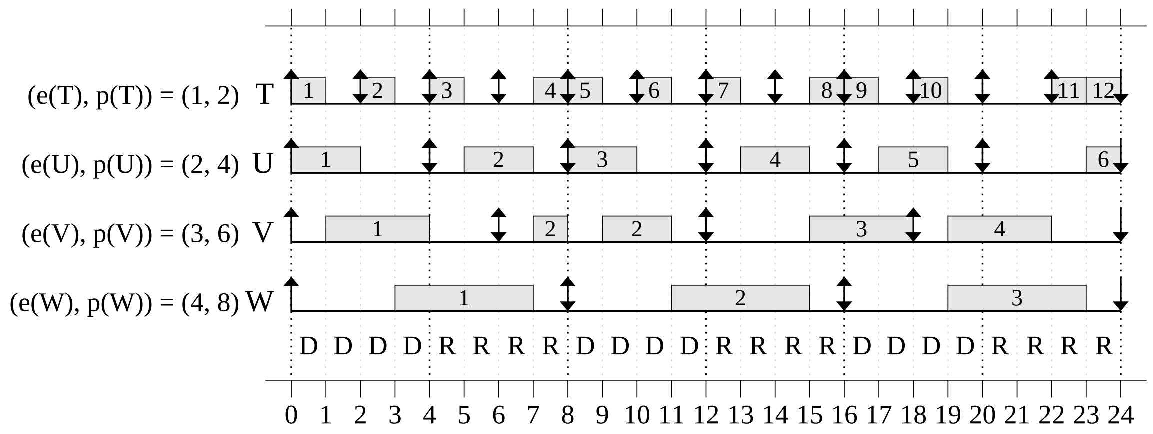


Figure 2.7: A two-core schedule under policy EVEN-ODD, where the power of priority points is demonstrated by allowing tardiness to be bounded in spite of frequent job priority changes. A "D" (respectively, "R") indicates that jobs are prioritized by deadline (respectively, release time) during that time interval.

ness. This includes the cache-aware scheduling heuristics that are described in Chapter 4, where the priority point of a job is moved to the current time to influence co-scheduling decisions, also known as a job *promotion*.

The tardiness bound for a scheduling algorithm with window-constrained priorities is similar in nature to those specified for GEDF and NP-GEDF. The tardiness for a task $T \in \tau$ scheduled using a window-constrained algorithm is $z + e(T)$, where z is defined as follows (as before, with minor changes from [44] to maintain consistency with our notation).

$$z = \frac{\mathcal{E}_z + A(T)}{M - \mathcal{U}_z} \tag{2.4}$$

In (2.4), $\mathcal{E}_z$ is the sum of the $M-1$ highest execution costs over all tasks in τ, and $\mathcal{U}_z$ is the sum of the $M-1$ highest utilizations over all tasks in τ. $A(T)$ is defined as follows.

$$A(T) = (M-1)\cdot\rho - e(T) + \sum_{U\in\tau\setminus T}\left(\left\lceil \tfrac{\psi(T)+\phi(U)}{p(U)}\right\rceil + 1\right)\cdot e(U) \tag{2.5}$$

$\phi(T)$ (respectively, $\psi(T)$) indicates the amount by which the priority point of a job of T can be before its release time (respectively, after its deadline), and $\rho = max_{T\in\tau}(\phi(T)) + max_{T\in\tau}(\psi(T))$.

For the cache-aware scheduling heuristics that are described in Chapter 4, the priority point of a job is never less than its release time or greater than its deadline; thus, $\phi(T) = \psi(T) = 0$ for all $T \in \tau$, and $\rho = 0$. The same is true under GEDF, as the priority point of a job is always its deadline, and under the EVEN-ODD policy used in Figure 2.7. Thus, in all three cases, $A(T)$ becomes $\sum_{U \in \tau \setminus T} e(U) - e(T)$, and the tardiness bound for a task T is $\frac{\mathcal{E}_z + \sum_{U \in \tau \setminus T} e(U) - e(T)}{M - \mathcal{U}_z} + e(T)$. The computed tardiness bound for GEDF, EVEN-ODD, and our cache-aware scheduling heuristic for the task set in Figure 2.6 would be $\frac{4+(4+2)-2}{2-2/3} + 2 = 8/(4/3) + 2 = 8$ for tasks T and U and $\frac{4+(2+2)-4}{2-2/3} + 4 = 4/(4/3) + 4 = 7$ for task V, or a tardiness bound of 8 for the task set. For the task set in Figure 2.7, the tardiness bound is largest for task T, and is $\frac{4+(2+3+4)-1}{2-1/2} + 1 = 12/(3/2) + 1 = 9$.

2.1.7 Pfair Scheduling

Our early attempts at cache-aware scheduling, presented in Chapter 3, rely on Pfair scheduling [6, 9, 61]. Under Pfair scheduling, first proposed by Baruah *et al.* [9], tasks are scheduled in discrete time units called *quanta*; execution costs and periods are required to be integral multiples of the quantum length. Note that, in this dissertation, it is assumed that scheduling quanta are both *uniformly-sized* and *aligned* across all cores in any quantum-based scheduler. With Pfair scheduling algorithms, all deadlines can be met for any task set with integral execution costs and periods that does not over-utilize the system.

A periodic task T with utilization $u(T)$ is scheduled one quantum at a time in a way that approximates an *ideal* allocation in which it receives $L \cdot u(T)$ time over any interval of length L.[1] This is accomplished by sub-dividing each task into a sequence of quantum-length *subtasks* $T_{(1)}$, $T_{(2)}$, ..., each of which must execute within a certain time *window*, defined by the subtask release time and deadline. Note that each job T_i consists of $e(T)$ subtasks under Pfair scheduling—subtasks $T_{(e(T)*(i-1)+1)}$ through $T_{(e(T)*i)}$. The release time and deadline of each subtask of a periodic task T are computed as follows (from [6]).

[1]If T is sporadic, then for any job T_i, zero allocations are received between the deadline of T_i and the release time of T_{i+1}. This must be taken into account when computing the ideal allocation to T.

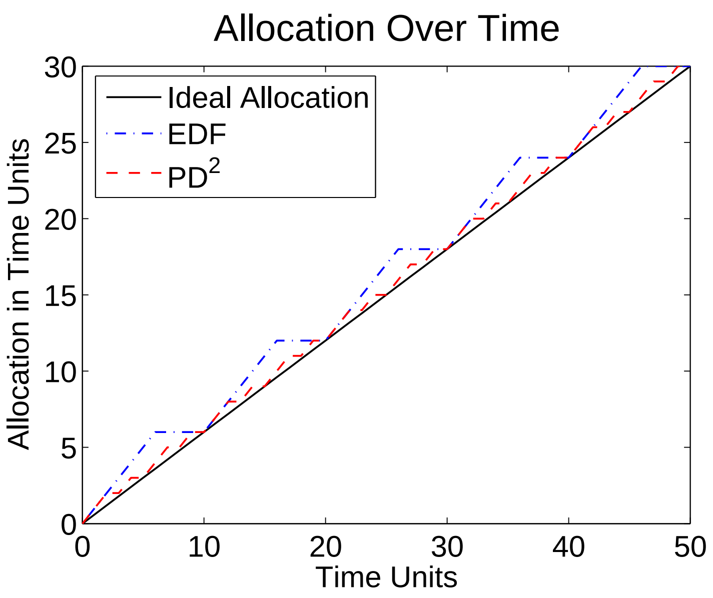


Figure 2.8: Allocation over time for a single task under an ideal allocation, EDF, and PD^2.

$$r(T_{(i)}) = \left\lfloor \frac{i-1}{u(T)} \right\rfloor \quad (2.6)$$
$$d(T_{(i)}) = \left\lceil \frac{i}{u(T)} \right\rceil \quad (2.7)$$

All subtasks are scheduled on an EDF basis, and tie-breaking rules are used in case of a deadline tie. The subtasks of a task may execute on any processor, but not at the same time (*i.e.*, tasks must execute sequentially).

Example (Figures 2.8 and 2.9). An example of how Pfair schedulers approximate an ideal allocation as compared to EDF is shown in Figure 2.8 for a task with an execution cost of six and period of ten, assuming processor allocations are received at the earliest allowable time under both algorithms. (PD^2 is a Pfair algorithm, described next.) In Figure 2.9, we can see the subtask window layout of this task that resulted in the processor allocations shown in Figure 2.8. □

The most efficient known optimal Pfair algorithm, first proposed by Srinivasan and Anderson, is PD^2 [6, 61], which uses two tie-breaking rules. These rules depend on the *successor bit* and *group deadline* of each subtask. The successor bit $b(T_{(i)})$ is set whenever the window

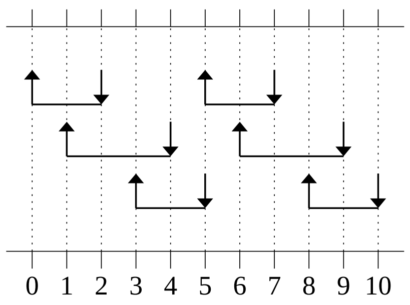


Figure 2.9: Pfair window layout for a task with an execution cost of six and period of ten. The up- and down-arrows in this case indicate *subtask* releases and deadlines instead of job releases and deadlines.

of a subtask overlaps with the window of its immediate successor, and is defined for a subtask $T_{(i)}$ as follows (from [6]).

$$b(T_{(i)}) \quad = \quad \left\lceil \frac{i}{u(T)} \right\rceil - \left\lfloor \frac{i}{u(T)} \right\rfloor \tag{2.8}$$

If we refer to the task in Figure 2.9 as T, then $b(T_{(1)}) = b(T_{(2)}) = 1$, and $b(T_{(3)}) = 0$. If subtask deadlines are the same, then the first PD^2 tie-break consists of checking this bit. A subtask that has its successor bit set has higher priority than a subtask that does not have it set. The reasoning behind this decision is that not scheduling a subtask that has its successor bit set may reduce the number of quanta available for scheduling its successor—this is not the case when the bit is not set.

The group deadline is a tie-break that is needed for "heavy" tasks, or those where $u(T) \geq 1/2$. For such tasks, cascading subtask windows of length two can arise. Such a group of subtask windows could be seen as having a single deadline at the end of the group—if any subtask in the group is scheduled in the last quantum of its window, then all successive subtasks in the group must also be scheduled in the last quantum of their windows to meet all subtask deadlines. Thus, a larger group deadline implies a larger cascading subtask group and a greater urgency to schedule that group. Therefore, if subtask deadlines and successor bits are the same, then the second PD^2 tie-break favors the subtask with the largest group deadline. The group deadline $D(T_{(i)})$ of a subtask $T_{(i)}$ can be computed as follows, assuming the periodic task model (from [6]).

$$D(T_{(i)}) = \begin{cases} 0, & \textbf{if } u(T) < 1/2 \\ \left\lceil \frac{\left\lceil \left\lceil \frac{i}{u(T)} \right\rceil \times (1 - u(T)) \right\rceil}{1-u(T)} \right\rceil, & \textbf{if } u(T) \geq 1/2 \end{cases} \tag{2.9}$$

Again referring to the task in Figure 2.9 as T, $D(T_{(1)}) = 3$, while $D(T_{(2)}) = D(T_{(3)}) = 5$. The second and third subtasks have the same group deadline, since scheduling the second subtask in the last quantum of its window will force the third subtask to also be scheduled in the last quantum of its window. Thus, a greater sense of urgency is implied by the need to schedule T for two consecutive quanta in order for all subtask deadlines to be met.

Under PD^2, if a task is allocated a quantum when it requires less execution time, the unused portion of that quantum is "wasted." In contrast, under the EDF schemes considered earlier, such a task would relinquish its assigned quantum "early," allowing another task to be scheduled. As a result, there is a utilization loss due to quantum-based scheduling, since a task with an execution cost slightly greater than x will need to be allocated $x + 1$ quanta of execution time, even though most of the last quantum that is allocated to the task will be wasted. Despite this utilization loss, quantum-based scheduling is often still appealing since it results in increased predictability, which can be useful for scheduling purposes. In fact, we argue in Chapter 4 that quantum-based scheduling is useful for our cache-aware scheduling heuristics, even though our heuristics do not employ PD^2 scheduling.

Example (Figure 2.10). To see some of the differences in the described EDF and PD^2 algorithms, consider Figure 2.10, which consists of three tasks scheduled on two cores. There are several things worth noting here. First, if the execution cost of either task U or V is increased by one, then a suitable bin packing would not exist, and the task set would not be schedulable under PEDF, even though the total utilization of the task set would be less than two and thus schedulable under the other algorithms (assuming that soft real-time guarantees are sufficient). Second, note that under PEDF and PD^2, no deadlines are missed, whereas deadlines are missed under GEDF and NP-GEDF. Third, note that NP-GEDF is the only

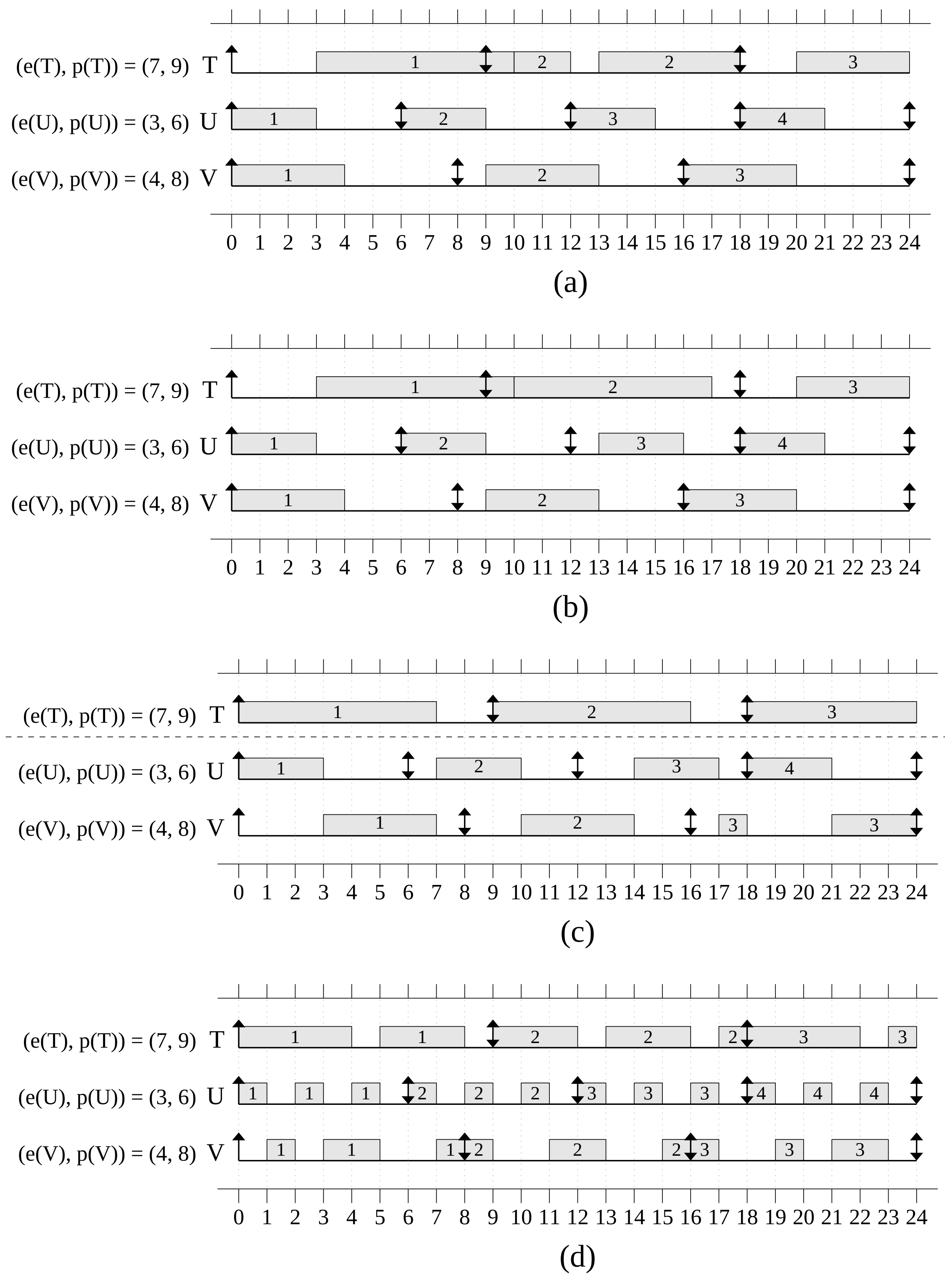


Figure 2.10: **(a)** GEDF, **(b)** NP-EDF, **(c)** PEDF, and **(d)** PD^2 schedules for the same task set.

scheduling approach where jobs are never preempted (for the sake of this example, in the case of a deadline tie under EDF or a group deadline tie under PD^2, we assume that ties are broken in favor of the task with the smaller period). Fourth, PD^2 schedules the tasks U and V at a rate of allocation that more closely approximates an "ideal" schedule than the other algorithms. The rates of allocation over time for tasks U and V are also more similar under PD^2 than the other algorithms. This is because the subtask windows of a task depend on its utilization, and although execution costs and periods differ for U and V, their utilizations are identical. $\square$

2.1.8 Early-Releasing

Our early attempt at MTT co-scheduling, first presented in [3] and discussed in Chapter 3, relies on the ability to release jobs or subtasks *early*; that is, a job or subtask can become eligible for execution before its actual release time. Early-releasing was first considered in [5] in work on Pfair scheduling. In all global, deadline-based scheduling methods known to us, the ability to meet timing constraints is not compromised if jobs or subtasks (as the case may be) are allowed to become eligible for execution before their designated release times. In other words, allowing early-releasing does not cause deadline misses (in hard real-time systems) or increase tardiness bounds (in soft real-time systems). Further, when a job or subtask is allowed to be released early, it is entirely *optional* as to whether the scheduler considers it for execution until the "official" release time of the job or subtask. This fact is exploited in the approach described in Chapter 3.

2.1.9 Impact of Overheads

When overheads are negligible, for soft real-time systems, GEDF, NP-GEDF, and PD^2 are clearly the superior algorithms in terms of schedulability, as they allow for full system utilization; however, overheads can drastically alter schedulability, and it is not immediately obvious which algorithm will prevail in every scenario. Overheads are typically accounted for by inflating task execution costs appropriately—an extensive discussion of inflation techniques can be found in [28].

In real systems, there are many types of overheads to consider. At the beginning of a scheduling quantum, *tick overhead* is incurred, which is the time needed to service a periodic timer interrupt. When a job is released, *release overhead* is incurred, which is the time needed to service the interrupt routine that is responsible for releasing jobs at the correct times, if such releases do not occur during the timer interrupt. Whenever a scheduling decision is made, *scheduling overhead* is incurred, which is the time taken to select the next job to schedule. This may include synchronization overheads related to accessing shared kernel data structures, such as a global run queue. Further, note that the majority of scheduling decisions may occur within the timer interrupt for certain algorithms. Whenever a job is preempted, *context-switching overhead* is incurred, as is either *preemption or migration overhead*; the former term includes any non-cache-related costs associated with the preemption, while the latter two terms account for any costs due to a loss of cache affinity. Preemption (respectively, migration) overhead is incurred if the preempted job later resumes execution on the same (respectively, a different) processor. In the case of migrations, additional overheads may be incurred, which are related to ensuring cache coherency. When inflating execution costs to determine hard real-time schedulability, worst-case overheads should be used; otherwise, we assume that for soft real-time schedulability, average overheads suffice.

To determine the impact of overheads on schedulability, we conducted a series of empirical evaluations of a variety of partitioned and global scheduling approaches in LITMUS$^{\mathrm{RT}}$, a Linux-based real-time testbed produced at UNC that has been developed over a series of studies [15, 16, 22] and is publicly available [69]. For soft real-time systems, these studies contained several findings. First, partitioned approaches (*e.g.*, PEDF) tend to be preferable in terms of soft real-time schedulability for task sets with exclusively low-utilization tasks, where the bin-packing problem is less difficult and overheads are considerably lower than other approaches. Global or hybrid approaches are preferable in most other scenarios. In particular, it was found that CEDF, a hybrid scheduling approach described earlier in Section 2.1.5, often performed best in terms of schedulability. Second, when comparing GEDF to NP-GEDF, schedulability tends to be approximately the same unless preemption or migration overheads are high, as NP-GEDF does not incur such overheads.

Third, with respect to global and hybrid approaches, we have found that allowing migrations tends to drive up overheads (due to an increase in bus and memory subsystem contention) for both preemptions and migrations [15]. However, in this case, preemption overheads tend to be *larger* than migration overheads, since the *length* of a preemption is typically longer when a task resumes on the same CPU (instead of a different CPU), and longer preemption lengths cause a greater loss of cache affinity and hence higher overheads. This makes sense under GEDF and CEDF: a job that is preempted will only migrate to another CPU if one becomes available before its current CPU is again available. Thus, when a job migrates, the total length of its preemption is reduced. Since these longer preemption lengths were relatively rare, the overall *average* cost of a preemption or migration was found to be similar across all variants of EDF. As a result, increases in scheduling and release overheads under global and hybrid approaches, due to an increase in the number of CPUs and tasks that must be considered, typically accounted for the soft real-time schedulability differences—partitioned approaches exhibited better schedulability when these overheads offset any bin-packing difficulties.

In this dissertation, it is vital that we show that the overheads associated with our cache-aware scheduler are comparable to other global scheduling approaches—otherwise, any reduction in cache miss rates could be offset by such overheads.

2.2 Cache-Aware Non-Real-Time Scheduling and Profiling

In this section, we begin with a discussion of the fundamentals of caches, so that the reader might better understand the work that follows. Next, we present metrics that have been proposed to assess the cache behavior of tasks. This is followed by a discussion of methods developed for non-real-time tasks to determine the cache behavior of groups of tasks when they are co-scheduled, so that better scheduling decisions can be made on multicore and hardware-multithreaded platforms, where resources (*e.g.*, caches) are often shared.

2.2.1 Caches: An Introduction

A *cache* is a memory unit that exists between the processor and off-chip main memory. This memory unit is faster and smaller than main memory. For example, a lower-level cache is likely to be an order of magnitude faster (*e.g.*, 50 processor cycles per access instead of hundreds of cycles) and several orders of magnitude smaller (*e.g.*, several megabytes as opposed to several gigabytes). By bringing the correct set of data from main memory into this cache, the speed gap that exists between main memory and processors can be alleviated.

2.2.1.1 The ABCs of Caches

Caches are often described in terms of three attributes: *associativity*, *block size*, and *capacity*, sometimes referred to as "the ABCs" of caches.

Block size. The block size of a cache is the size of a single datum ("block") in the cache, each of which is assigned a unique location. For example, a cache for which each location refers to a 64-byte data block would have a block size of 64 bytes. Today, this is more commonly known as *line size*, with each datum referred to as a cache line.

Associativity. The associativity of the cache determines how cache lines are assigned to locations in the cache. If a cache line may reside in any location in the cache, then the cache is *fully associative*. Fully associative caches result in the fewest number of misses, because any line can be stored in any location, and the cache can thus be treated as a large set containing some number of locations; however, they are typically slower since a search must be performed to find cache lines when they are referenced. At the other end of the spectrum are *direct mapped* caches, where each line can be stored in only one location in the cache. In this case, the number of misses increases, since two lines might map to the same location, forcing one to be evicted—this is known as a *conflict miss*, discussed in greater detail in Section 2.2.1.2. However, no search is necessary, so cache accesses are typically faster. Between these two extremes are *set associative* caches, where each line maps to a set of locations, and the size of each set is greater than one and less than the total number of locations in the cache. Such caches are specified in terms of some number of *ways* indicating the size of each set. For example, in a 4-way set associative cache, each cache line maps to four locations. In this case, a search is required, but on a much smaller scale, so access times are much closer to those of direct mapped caches. By allowing a line to map to four locations instead of one, the number of misses would typically be lower than in direct mapped caches.

Capacity. The capacity of a cache is its total size, either in terms of bytes (most commonly) or cache lines. The number of cache sets multiplied by the set associativity of the cache gives the capacity of the cache in lines; multiplying by the line size gives the capacity in bytes. In the case of a fully-associative cache, the capacity of the cache determines whether all of the cache lines that are needed by a task will fit in the cache.

Example (Figure 2.11). Figure 2.11 demonstrates how cache-line-sized pieces of main memory map to locations in a cache, depending on various cache attributes. All mappings to locations are indicated by arrows, and the mappings shown continue for the remainder of main memory; that is, the next cache-line-sized piece of main memory (which would be labeled "8") will map to the same location as the piece labeled "0". □

2.2.1.2 Cache Misses

A *cache miss* occurs when data that is requested by the processor is not present in the cache. (If the data is present in the cache, the request results in a *cache hit*.) Such misses are generally categorized as *compulsory* (or cold), *capacity*, or *conflict* misses; this categorization is known as "the three Cs" of cache misses. Compulsory misses occur when data is first referenced, at which point the data is brought into the cache. Since data cannot exist in the cache without first being brought into the cache, these misses cannot be avoided. Capacity misses occur when lines that are brought in by a task (and would be reused in the future) must be evicted from the cache, since the cache is not large enough to hold all of the cache lines that are needed by that task (or, in the case of a shared cache, it cannot hold the lines needed by all concurrently executing tasks). As a result, a miss will occur when data from an evicted line is next referenced. Finally, conflict misses occur when multiple cache lines map to the same set, and the associativity of the cache is too low to allow all lines to fit in this

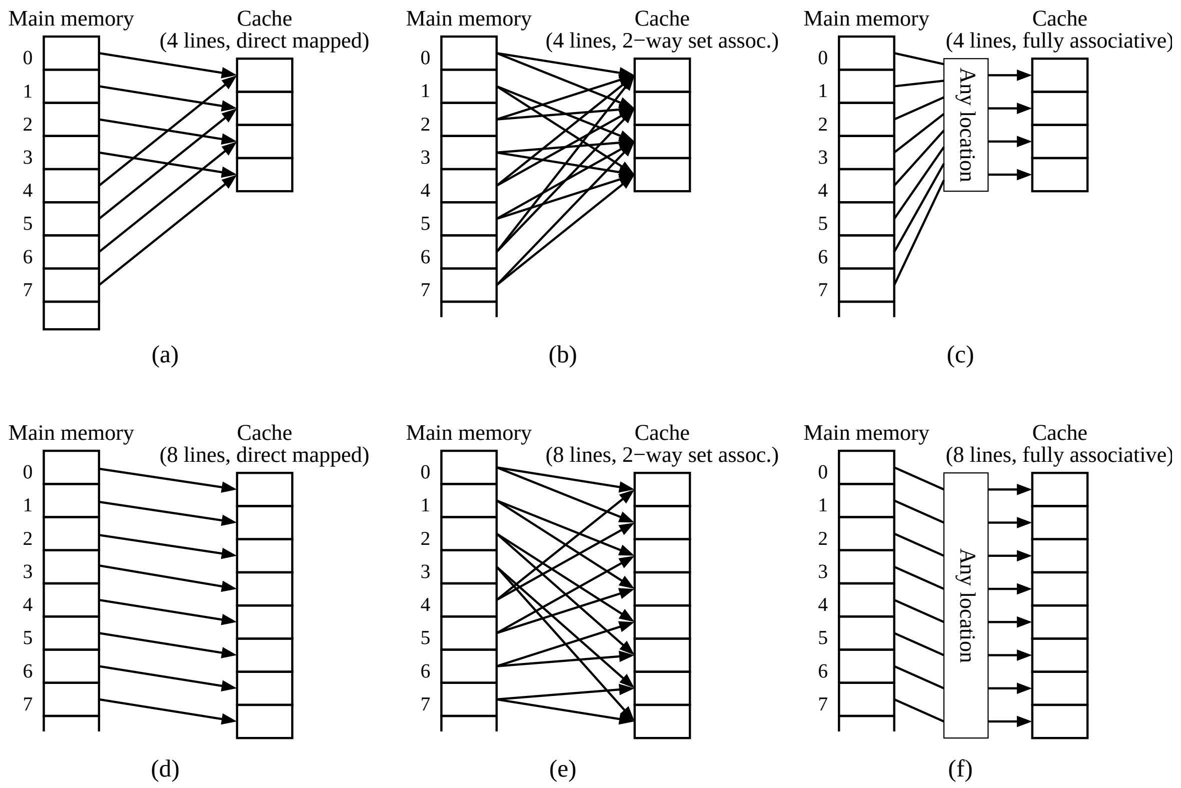


Figure 2.11: Mappings from cache-line-sized pieces of main memory to cache locations, based on the capacity and associativity of the cache. A 64-byte line size is assumed for all caches depicted. In the top row (insets (a) through (c)), the capacity of the cache is 256 bytes (four lines), while it is 512 bytes (eight lines) in the bottom row (insets (d) through (f)). In each column, from left to right, the caches are direct mapped (insets (a) and (d)), 2-way set associative (insets (b) and (e)), and fully associative (insets (c) and (f)).

Inset	Categorization of Misses	Miss Rate	Access Time	Total Ref. Time
(a)	C1 C1 C1 C1 C1 H C3 C3 C3	8/9	10	1610
(b)	C1 C1 C1 C1 C1 H C3 C3 C3	8/9	20	1620
(c)	C1 C1 C1 C1 C1 H H C2 H	6/9	40	1320
(d)	C1 C1 C1 C1 C1 H H C3 C3	7/9	10	1420
(e)	C1 C1 C1 C1 C1 H H C3 H	6/9	20	1260
(f)	C1 C1 C1 C1 C1 H H H H	5/9	80	1320

Table 2.1: Categorization of cache misses for the reference pattern 0, 2, 4, 6, 8, 6, 2, 0, 8, for a variety of caches. In the categorization, "H" indicates a cache hit, "C1" indicates a compulsory miss, "C2" indicates a capacity miss, and "C3" indicates a conflict miss. Cache miss rates and estimated total reference times are also shown, assuming the access time shown for each cache (all times are in cycles) and a main memory access time of 200 cycles.

set, resulting in evictions. Note that a miss is only categorized as a conflict miss when the eviction would not have had to occur in a fully-associative cache; otherwise, it is a capacity miss. As the set associativity of a cache increases, conflict misses become increasingly rare; only highly-unusual reference patterns, such as referencing single lines in very large strides across memory, will trigger them on a frequent basis.

Replacement policies. The *replacement policy* of a cache dictates which cache line will be evicted when it is necessary to do so. Regardless of the type of cache miss, the replacement policy will typically be invoked (perhaps to evict or replace lines that were brought in during the boot process, or during the execution of a previously-running program). One of the most common replacement policies is *least-recently-used* (LRU), wherein the line that has been used least recently (where "recently" is typically measured in terms of references to unique memory locations) is the one that is evicted. The LRU policy attempts to approximate an "ideal" replacement policy that would evict the cache line that will not be reused for the largest amount of time, in an attempt to preserve lines that will be reused in the immediate future. Since we do not have an oracle that can exactly predict future reference patterns, we instead use the past to predict the future by assuming that the LRU line is also least likely to be reused in the immediate future, and therefore evict it when necessary.

Example (Table 2.1). In Table 2.1, cache misses are categorized for the reference pattern 0, 2, 4, 6, 8, 6, 2, 0, 8, where each location refers to a cache-line-sized piece of data, as labeled in Figure 2.11. These misses are categorized for each of the caches in insets (a) through (f) of Figure 2.11, assuming the mappings as depicted and an LRU replacement policy. Additionally, the cache miss rate and an estimated total time to reference all data is provided. We assume that the access time of each cache is a function of its associativity. The total time to reference all data is computed as the number of cache hits multiplied by the access time of the cache (as indicated in Table 2.1), plus the number of cache misses multiplied by the main memory access time (200 cycles). For example, for the cache in inset (a) of Figure 2.11, the total reference time is $1 \cdot 10 + 8 \cdot 200 = 1610$ cycles. For all caches, the first five references are compulsory misses, as data is brought into the cache for the first time. For the fully associative caches, the last four references result in four cache hits for the eight-line cache (since the cache is sufficiently large to hold all referenced lines) and three hits and one capacity miss for the four-line cache (since line "0" is evicted by line "8" before it can be reused). For the direct mapped caches, half of the cache lines are never used due to the mappings of these caches. As a result, in the eight-line cache, two conflict misses occur since lines "0" and "8" map to the same location, and in the four-line cache, three conflict misses occur since all lines except line "6" are evicted due to conflicts before they can be reused. For the 2-way set associative caches, only half of each cache is again used; however, the conflicts are less severe for the eight-line cache, allowing for one additional cache hit, since the second reference to line "0" does not evict line "8" as it did in the direct mapped cache.

Overall, as might be expected, increasing cache capacity decreases the miss rate, and fully associative caches have lower miss rates than direct mapped or 2-way set associative caches when cache capacity is the same. However, for the eight-line cache, the 2-way set associative cache results in the lowest total reference time, since lower cache access times offset the additional cache miss when compared to the fully associative cache. Note also that the eight-line, 2-way set associative cache performs very similarly to the four-line, fully associative cache, but since access times are lower, better performance is achieved. □

2.2.1.3 Physically and Virtually Addressed Caches

In most general-purpose CPUs, virtual addressing is used so that each program runs in its own address space—addresses are translated from a virtual address used within the program into a physical address in main memory during a memory reference. As the cache resides between the CPU and main memory, it may be either physically or virtually addressed. Often, higher-level caches, which are closer to the CPU, are virtually addressed, while lower-level caches that are closer to main memory are physically addressed. There are numerous issues to consider when deciding between a physically or virtually addressed cache. For example, a virtually addressed cache does not require address translation and therefore typically results in lower access times, but if multiple virtual addresses are used by different programs to refer to the same physical address in main memory, then multiple copies of the data located at that address may be present in the cache, which can reduce cache reuse or increase the amount of cache space needed by all running programs to minimize cache misses. In our cache-aware scheduler, we make no assumptions either way about whether physical or virtual addressing is used within the cache—the issues associated with a physically or virtually addressed cache will persist regardless of whether our cache-aware scheduler is used. We note, however, that the lower-level shared caches of concern in this dissertation are typically physically addressed, since handling problems associated with multiple virtual addresses mapping to the same physical address becomes more costly as cache size increases.

2.2.2 Assessing the Cache Behavior of Tasks

A number of metrics exist to profile the cache behavior of tasks in non-real-time (throughput-oriented) systems. These methods can be used to determine the locality exhibited by an application, in order to determine how a task will perform in the presence of a cache of a certain size (and associativity). The results of such methods can then be used to determine how a *group* of tasks will perform when co-scheduled in the presence of a *shared* cache. For soft real-time systems, we can often use these same methods since precise WCET analysis is not necessary. (Even if it were necessary, tools for performing such analysis on multicore platforms are in their infancy.)

2.2.2.1 Working Set Size

The simplest and most popular cache profiling metric is *working set size* (WSS), or the size of the working set (WS) of a task. In [26, 27], the working set model was first proposed by Denning. In this model (as originally proposed), a WS for a process (or in our case, MTT) is defined as the set of memory pages that must be loaded and present in main memory at a given time to achieve the maximum performance impact from the use of main memory by that process. If sufficient main memory exists so that the WS of every running process can fit in main memory, then thrashing due to "unnecessary" page faults and evictions can be avoided. More recently, the notion of WSS has been extended to caches by Agarwal *et al.* [2], by defining the WS in terms of the set of *cache lines* that must be loaded and present in the cache to maximize process performance, or to minimize process execution times.

The WSS metric is fairly intuitive—a task executing in isolation should perform well in the presence of a private cache that is larger than its WSS. Similarly, a group of tasks should perform equally well in the presence of a shared cache that is larger than the sum of the WSSs of all tasks. Note that this assumes that the cache has a high set associativity, so that the cache can be treated as fully associative with little loss of accuracy; thus, conflict misses that could cause thrashing *within* a set are assumed not to occur. This simple model provides a fairly accurate estimate of cache locality and performance (in terms of miss rates and potential for thrashing) when the memory references of a task are distributed uniformly over its WS, or when WSS is computed separately for small, fixed-length intervals of execution (*e.g.*, for every job of a real-time task). Otherwise, it could be the case that a task with a large WSS references only a small amount of its WS very frequently, in which case a single value for WSS may not accurately capture cache behavior. In such cases, it may be necessary to use alternate metrics.

2.2.2.2 Reuse Distance

One alternate metric is *reuse distance*, first proposed by Berg and Hagersten [11] as part of a probabilistic cache model. Reuse distance is defined as the number of memory references between references to the same memory location, where a memory location refers to a cache-line sized piece of data. For example, if a task sequentially references memory locations A, B, C, B, and A, then it has a reuse distance of three for memory location A. Note that *all* references between the references to A are counted, rather than just the unique ones; otherwise, the reuse distance would be two.

The reuse distance can be calculated for all memory references of a task, and used to create a reuse distance histogram. In [11], this histogram is analyzed to estimate the cache miss ratio of a task, as follows. First, the authors define a function $f(n)$ as the probability that a cache line has been evicted from the cache after n cache misses, assuming a random replacement policy and a capacity of L cache lines (from [11]).

$$f(n) = 1 - \left(1 - \frac{1}{L}\right)^{n} \tag{2.10}$$

For a given reuse distance histogram h, $h(n)$ indicates the number of memory references with a reuse distance of n. Given $f(n)$ and $h(n)$, the authors specify the following equation for a task that makes N memory references, which can be solved to estimate the cache miss ratio R (also from [11]).

$$R \cdot N \approx h(1) \cdot f(R) + h(2) \cdot f(2R) + h(3) \cdot f(3R) + \dots \tag{2.11}$$

The intuition here is that the left side of the equation, or the total number of references N multiplied by the miss ratio R, should be approximately equal to the right side of the equation, which represents a summation over all reuse distances in the histogram. For each reuse distance n, the number of references with that reuse distance, or $h(n)$, is multiplied by the probability of a cache miss for a reference with a reuse distance of n, or $f(n \cdot R)$. Note that $f(n \cdot R)$ is the probability of a cache line being evicted (before it can be reused) after $n \cdot R$ misses (the expected number of misses over n references with a miss rate of R is $n \cdot R$).

Note that, while reuse distance is a better metric than WSS when reuse varies widely among the memory references of a task (rather than being uniform), it still requires the execution of a task to be divided into fixed-length intervals, each of which is analyzed independently. For some tasks where sufficiently small interval lengths are used, the assumption of uniformly-distributed references may not have a dramatic impact on cache miss rate estimates. Further, the sampling and computation overheads of using the reuse-distance method during execution, such as the need to monitor memory locations and to solve for R in (2.11) using numerical methods, may be too high when scheduling real-time tasks, where the scheduler is often invoked at least once every millisecond. If several hundred microseconds or more of computation time is required whenever the scheduler is invoked for the reuse-distance method to be used effectively, then the amount of real work that can be performed by the system will decrease substantially, and this in turn may impact task-set schedulability.

An effort to reduce these overheads was proposed by Berg and Hagersten in later work [12]. This approach involves monitoring only a random sampling of referenced memory locations. This is achieved by randomly generating a value, and triggering an interrupt in hardware when an instruction counter (that is, a performance counter that counts completed instructions) has increased by an amount that is equal to that value; the process is repeated after each sampling. During sampling, the instruction to which the program counter points is decoded; if it is a memory-referencing instruction, then the location being referenced is monitored using a watchpoint mechanism. This mechanism is provided by the Solaris operating system that the authors used for their implementation, and sends a signal when the monitored location is again referenced, so that reuse distance information can be collected. Note that using this approach requires significant hardware and operating system support, and still results in overheads that are likely too high for use within a real-time scheduler (*e.g.*, running times that are 40% higher, on average, than when the approach is not used).

Like WSS, the reuse distance metric assumes a fully-associative cache. If the set associativity of the cache is not high enough for such an assumption to be reasonable, then it may be more appropriate to consider the *stack distance* metric, presented next.

2.2.2.3 Stack Distance

The stack distance metric was introduced by Mattson *et al.* [48] as part of a general method for evaluating the performance of storage hierarchies. For a set of memory pages in a buffer, the *stack distance* of a page is its position in an ordering based on when pages would be

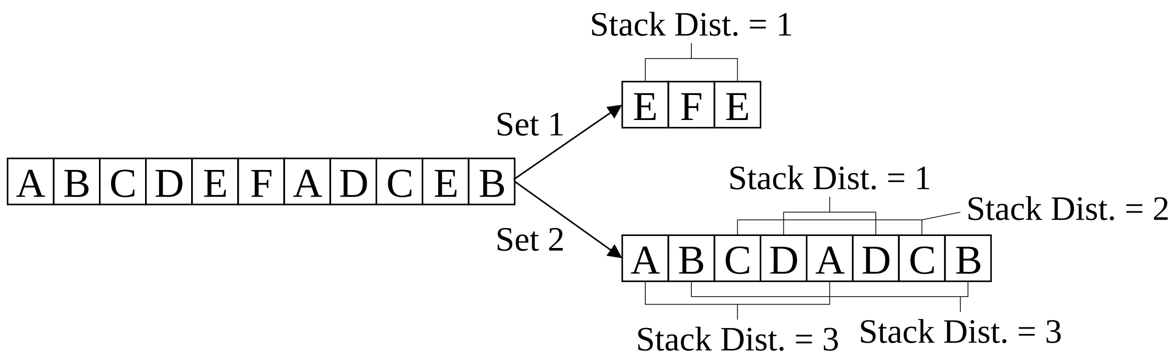


Figure 2.12: Computation of stack distances for a given reference pattern, when references map to multiple buffers.

evicted by a replacement policy. For example, if an LRU replacement policy is employed, then pages are ordered from most recently used to least recently used; the page with the highest stack distance would be replaced next if the buffer is at capacity. Assuming an LRU replacement policy, we can instead think of stack distance in much the same way as reuse distance, except that the stack distance is defined as the number of *unique* pages referenced between references to the same page. For example, if a task sequentially references pages A, B, C, B, and A, then it has a reuse distance of three for page A, but a stack distance of two for page A.

Stack distance can be used to determine if a page that is brought into the buffer will be available on subsequent references to that page. If the entire buffer is considered as a single set, then reused pages will always be available during subsequent references if the stack distance immediately preceding any reference is always less than the buffer size (in pages); note that immediately following the reference, the stack distance of that page will be zero. Stack distances are not computed for pages that are never reused, as their presence in the buffer presents no opportunity to reduce cache miss rates. If different pages map to different sets, then the same analysis can be performed independently for each set.

Example (Figure 2.12). Figure 2.12 demonstrates how stack distance is calculated for the reference pattern depicted when different pages map to different sets. The leftmost sequence represents the entire reference pattern, referenced sequentially starting with page A. The rightmost sequences represent the pattern divided according to how the pages map to buffers; in this case, we assume that references to pages E and F map to buffer set 1, and all other references map to buffer set 2. Note that no stack distance is computed for page F since it is never reused. Given the stack distances from Figure 2.12, and assuming that both buffers must be the same size, then each buffer must have a capacity of at least four pages to ensure that pages will always be available for reuse. However, if each buffer could hold only two pages, then a page would be available if the stack distance for references to that page is at most one; pages D and E satisfy this requirement. $\square$

Note that, if we consider cache lines instead of memory pages, then this model can be easily used to represent a set associative cache. If the set associativity of the cache (which determines the size of each set) is greater than the maximum stack distance for any cache line immediately before any reference, then no capacity *or conflict* misses should occur. Otherwise, we can maintain a stack distance *profile* for a task, which is very similar to a reuse distance histogram: for a given stack distance profile p, $p(n)$ indicates the number of memory references with a stack distance of n. Upon reuse of a cache line with a stack distance of s, $p(s)$ is incremented. For example, given the reference pattern in Figure 2.12, $p(1) = 2$, $p(2) = 1$, and $p(3) = 2$. If this example concerned a 2-way set associative cache rather than two-page buffers, we could calculate the hit rate of the cache as the total number of references with a stack distance less than two, over the total number of references, or 2/11; the miss rate would therefore be $1 - 2/11 = 9/11$. Thus, the stack distance metric allows us to determine the impact of set associativity on cache miss rate for caches where set associativity is low enough that conflict misses are still non-negligible.

While this method may be more accurate than those previously proposed, the overheads associated with creating a stack distance profile are similar to those of creating a reuse distance histogram. These overheads make the creation of a stack distance profile during execution impractical for real-time tasks, since such overheads are likely to significantly reduce the amount of real work that can be performed by the system in the worst case, and this worst-case scenario can impact task set schedulability. Suh *et al.* [66] showed that stack distance profiles can be created more efficiently if the necessary hardware performance counters exist; however, the appropriate counters do not appear to be "standard" on many platforms. Additionally, when generating a stack distance profile for a task, it is necessary to run that task in isolation.

Thus, even when overheads are not a concern, the need to run tasks in isolation makes the generation of the profile during execution only marginally more convenient than using offline profiling tools to create the profile, such those described by Cascaval *et al.* [23]. Unlike stack distance, the WSS-like profiling metric that we consider in Chapter 5 is well-suited to the profiling of tasks during their execution, even when these tasks are not run in isolation. Our method requires only a single performance counter to record shared cache misses; such counters exist on many multicore platforms today.

2.2.2.4 Comparison

A comparison of the WSS, reuse distance, and stack distance metrics is provided in Figure 2.13, wherein three different reference patterns are represented using each of the three metrics. Similarly to the example in Figure 2.12, assume that cache lines E and F map to cache set 1, and all other cache lines map to set 2. Inset (a) presents reference patterns (1) through (3). Insets (b), (c), and (d) present the WSS, reuse distance histogram, and stack distance profile, respectively, for each of the reference patterns. In insets (c) and (d), the notation previously introduced for a reuse distance histogram h and a stack distance profile p are used. For all reference patterns, exactly four unique lines are referenced, for a WSS of four lines. For reference pattern (1), three unique lines are referenced between each pair of references to the same line, so the reuse distance histogram and stack distance profile are identical. For reference pattern (2), reuse distances are twice stack distances, since each unique line referenced between two identical lines is referenced twice. Finally, reference patterns (2) and (3) produce identical reuse distance histograms, but since lines E and F map to a different set, the stack distance histogram is the combination of evaluating patterns A, B, B, A, and E, F, F, E independently, resulting in the profile shown.

To summarize, the ability of these metrics to differentiate between different reference patterns is substantially different for the three reference patterns considered; all three patterns result in the same WSS, and only the stack distance profile indicates differences between all three reference patterns. However, as previously discussed, any increase in accuracy as a result of using the stack distance metric is achieved at the cost of overheads that are prohibitive for

(1) A B C D A B C D	(1) WSS: 4 lines	(1) h(3) = 4	(1) p(3) = 4
(2) A B C D D C B A	(2) WSS: 4 lines	(2) h({6, 4, 2, 0}) = 1	(2) p({3, 2, 1, 0}) = 1
(3) A B E F F E B A	(3) WSS: 4 lines	(3) h({6, 4, 2, 0}) = 1	(3) p({1, 0}) = 2
(a)	(b)	(c)	(d)

Figure 2.13: Comparison of different metrics for representing the cache impact of tasks, based on their memory reference patterns. Inset (a) presents three different reference patterns, and insets (b), (c), and (d) present the WSS, reuse distance histogram, and stack distance profile, respectively, for each reference pattern. In inset (c), $h(\{X, Y, Z\}) = a$ has a meaning equivalent to $h(X) = h(Y) = h(Z) = a$; the same is true for p in inset (d).

real-time systems.

2.2.3 Shared Cache Behavior: Task Co-Scheduling

We now review methods, developed in prior work, to determine shared cache miss rates for a group of co-scheduled tasks. Many of these methods employ variants of either the reuse distance or stack distance metrics discussed earlier. Note that for the WSS metric, keeping shared cache miss rates low is simple if we observe that a group of tasks should avoid shared cache thrashing if the size of the cache is at least the sum of the WSSs of all tasks. While the other methods to be described may be more accurate at estimating shared cache miss rates or the potential for cache thrashing for a group of co-scheduled tasks, such methods are likely to incur greater overheads when employed during task execution or scheduling. Thus, with one exception, the methods described below are provided here for completeness (and since they inspired our work), but are not otherwise addressed in this dissertation. The exception is the method described in Section 2.2.3.4, which is similar in many ways to the methods that we will describe in Chapters 4 and 5, except that it is for non-real-time systems.

2.2.3.1 Combining Reuse Distance Histograms

Fedorova *et al.* [32] devised two methods to *combine* the reuse distance histograms of multiple tasks in ways that allow the shared cache miss rate for a group of co-scheduled tasks to be determined. For individual tasks, reuse distance histograms were found to result in considerably more accurate estimates of cache behavior than WSS, and therefore were chosen over WSS in this work.

The first method of combining reuse distance histograms involves summing the counts in each "bucket" across all histograms to create a single histogram for the co-scheduled tasks, and multiplying the reuse distances in the resulting histogram by the number of co-scheduled tasks, to account for the worst-case reuse distances that could result from concurrent execution. This combined reuse distance histogram could then be analyzed as described in Section 2.2.2.2 to estimate the cache miss ratio for the co-scheduled tasks. This method was found to be accurate, but the overheads of such a method would be prohibitive within a real-time scheduler (especially as the number of cores increases, since more combinations of histograms, and more histograms in each combination, would have to be analyzed to make an appropriate cache-aware scheduling decision).

Example. Let L be the number of lines in the shared cache, and let h_T and h_U represent the reuse distance histograms for tasks T and U, respectively, where $h_T(1) = 1000$, $h_T(5) = 2000$, $h_U(1) = 3000$, and $h_U(10) = 500$. Using the first method, we would first create a histogram h_{TU} representing the sum of the counts in each bucket, where $h_{TU}(1) = 4000$, $h_{TU}(5) = 2000$, and $h_{TU}(10) = 500$. Next, we multiply all reuse distances by two, so now $h_{TU}(2) = 4000$, $h_{TU}(10) = 2000$, and $h_{TU}(20) = 500$. This histogram would then be used to estimate the miss ratio, assuming a cache with L lines. $\square$

The second method, which was found to be equally accurate, is considerably more feasible to implement in practice. In this method, a cache miss ratio is computed for each task using its reuse distance histogram and the method in Section 2.2.2.2, assuming a private cache equal to the shared cache size divided by the number of co-scheduled tasks. These miss ratios are then averaged to determine the miss ratio for the co-scheduled tasks. Thus, the miss ratio can be computed for each task offline and provided to the scheduler when making online co-scheduling decisions. (Note, however, that many combinations of co-scheduled tasks may still have to be analyzed online in order to make a scheduling decision, even if considering each combination requires less online computation.) In this case, to estimate the miss ratio when tasks T and U are co-scheduled, we would compute miss ratios R_T and R_U using h_T and h_U, respectively, each time assuming a cache with $L/2$ lines. The estimated miss ratio for when T and U are co-scheduled would then be $(R_T + R_U)/2$.

In [32], both methods resulted in estimates that are within 5-20% of the actual miss rates for benchmarks from the SPEC CPU benchmark suite; however, both methods still require a considerable effort to generate the reuse distance histogram for each task, even if that effort is performed offline and supplied to these methods of combining the histograms. Further, the authors note that both methods appear to underestimate cache miss ratios as cache contention increases. This may be because the combination of reuse distance histograms in these ways "dilutes" the primary benefit of using reuse distance over WSS: that the memory references of a task do not need to be distributed uniformly over the entire WS to ensure accuracy. The combination of reuse distance histograms in the ways described assume that each task is uniformly impacted by the cache interference that results from co-scheduling. If this is not the case, then the derived estimates could clearly deviate from reality. As cache contention increases, the effect of such interference would become more pronounced, resulting in increasingly inaccurate estimates.

2.2.3.2 Redefining Reuse Distance

Petoumenos *et al.* [56] (including an author from [11]) redefined reuse distance for shared caches. Specifically, they proposed to redefine reuse distance in terms of the number of cache replacements, or *Cache Allocation Ticks* (CATs), between references to the same memory location (*e.g.*, cache-line sized piece of data), rather than the number of *references* between references to the same location. By redefining reuse distance in this way, the authors can rely on the fact that the expected lifetime of a cache line is a number of CATs equal to the number of lines in the cache (recall that the reuse distance model assumes a fully-associative cache) when estimating shared cache miss rates. As before, reuse distance is calculated for all memory references of a task and used to create a reuse distance histogram, and this histogram can be analyzed to estimate the cache miss ratio of the task.

The model in [56] can also incorporate a notion of *cache decay*, in which cache lines are marked decayed after not being referenced for a certain period of time (measured in CATs), and decayed lines have statically lower priority than non-decayed lines when a cache line must

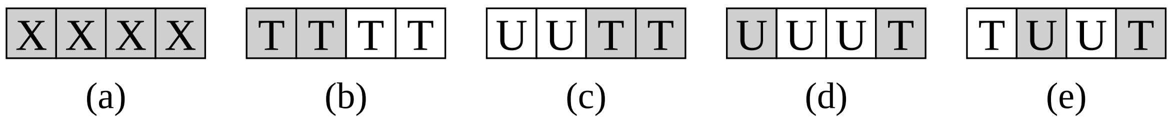


Figure 2.14: Cache decay can control the amount of cache used by each task, as shown. The data in each line is marked by the task that owns the data (T or U) or an X if the line belonged to a previously-executing task. Decayed lines are shaded.

be replaced; that is, decayed lines are replaced first. Decaying cache lines requires hardware support, but can be employed to dynamically control the amount of shared cache that a task is allowed to use without the need for explicit cache partitioning.

Example (Figure 2.14). In Figure 2.14, the impact of decaying lines can be seen when two tasks are co-scheduled, assuming that cache lines are marked decayed after being inactive for two CATs. Assume that initially a four-line cache is filled with decayed data from tasks that previously executed, as shown in inset (a). Next, task T references four cache lines of data, as shown in inset (b). Since four replacements were made, two of the four lines in the cache must be decayed. When task U references two cache lines of data, it will replace the decayed lines first, as shown in inset (c). This process will continue: if U references another cache line of data, then one line of U *must* be decayed, as shown in inset (d), after which T can replace the line with its own data, restoring the balance, as shown in inset (e). $\square$

Unfortunately, from our perspective, there are several problems with the approaches in [56] discussed here. First, even with this new definition of reuse distance, an approach for generating reuse distance histograms that is very similar to that in [12] for the original reuse distance model must be employed. As might be expected, since the approach is fundamentally the same, the overheads in this case are no better than they were in [12], and therefore this method cannot be effectively used within a real-time scheduler. Second, the ability to count cache replacements is more rare in multicore chips than the ability to count references; thus, hardware support for this new definition of reuse distance will be more limited than the original definition. Finally, hardware support is necessary if cache decay is employed, since it would require fundamental changes to the cache replacement policy to allow lines to be decayed and randomly replaced.

2.2.3.3 Using Stack Distance Profiles

Chandra *et al.* [24] presented several models for estimating shared cache miss rates by analyzing stack distance profiles. These methods vary significantly in terms of both accuracy and feasibility of implementation.

The *frequency of access* (FOA) model assumes that the cache space used by a task is proportional to its memory reference frequency; tasks that reference memory more frequently tend to maintain a larger cache footprint as a result. When using this model, each cache set (of a set associative cache) is proportionately allocated for the purpose of estimating miss ratios. The allocations are based on the memory reference frequency of each task, relative to the other tasks. For example, if two tasks are co-scheduled in the presence of an 8-way set associative shared cache, and the reference counts for these tasks are 750 and 250, then cache miss ratios will be estimated for these tasks assuming a 6-way and 2-way set associative cache, respectively, using the stack distance profile for each task as described in Section 2.2.2.3. This model was the least accurate since it makes many assumptions about the stack distance profiles of co-scheduled tasks (*e.g.*, that the "shapes" of the stack distance profiles of all tasks are similar—that is, for all tasks, the histogram indicating the frequencies of observed stack distances would have a similar shape), but it is the most straightforward to implement. This is because the stack distance profiles themselves do not need to be manipulated. Only the set associativity of the cache changes when estimating the miss ratio of each task; otherwise, tasks are evaluated as if they were running in isolation.

In the *stack distance competition* (SDC) model, the stack distance profiles of co-scheduled tasks are compared for each stack distance from zero to the set associativity of the shared cache minus one. This model proportionately allocates each cache set for the purposes of estimating miss ratios as in the FOA model, this time based on the reuse frequency of each task. The intuition behind this model is that, as the reuse frequency of a task increases, so does the size of its cache footprint. For each stack distance, the task with the highest count is noted, and cache sets are proportionately allocated based on the number of times that each task has the highest count for a particular stack distance—if the cache is 8-way set associative, and one task has the highest count for six of the eight considered stack distances

n	$p_T(n)$	$p_U(n)$
0	1000	0
1	1000	2000
2	1000	2000
3	0	1500
4	0	1500
5	15	50
6	50	10
7	5	1
8	0	500

Table 2.2: Stack distance profiles for tasks T and U.

in its profile, then its miss ratio is estimated assuming a 6-way set associative cache. The original stack distance profiles are then used to estimate the cache miss ratio of each task, as in the FOA model.

Example (Table 2.2). For example, assume that p_T and p_U represent the stack distance profiles for tasks T and U, respectively, where these profiles are defined as shown in Table 2.2. Assuming an 8-way set associative cache, we only compare counts for stack distances between zero and seven, thus $p_T(8)$ and $p_U(8)$ are ignored. Upon comparing values, we find that task T has the highest count for three stack distances, and task U has the highest count for the other five stack distances. Therefore, the cache miss ratios will be estimated for tasks T and U assuming a 3-way and 5-way set associative cache, respectively. □

While the SDC model proved to be more accurate than the FOA model, and is still relatively easy to implement, since we are simply comparing values and changing the set associativity of the cache as before, it is inaccurate in certain scenarios, particularly when the stack distance profiles of each task are very different for stack distances higher than the set associativity of the cache.

The last model presented by Chandra *et al.* [24] is the most accurate, but its computational complexity is too high to be effectively used within a real-time scheduler in practice. This *inductive probability* model uses probability theory to predict the cache impact of co-scheduling tasks. Briefly, this model estimates the miss count for each task by computing the probability of a cache miss for this task for each stack distance in its profile (note that this probability is always one for stack distances that are greater than the set associativity, since a stack distance that is greater than the set associativity of the cache implies that a line with that stack distance would be evicted before it could be reused even if the task ran in isolation), and then computing an expected miss count based on these probabilities and the counts at each stack distance.

2.2.3.4 Performance Counters

Knauerhase *et al.* [43] proposed and implemented a low-overhead, cache-aware scheduling policy for non-real-time systems. This scheduler consists of an *observation subsystem* that collects per-task performance information dynamically using performance counters, and a scheduling policy that uses this information. In the authors' prototype system, information about cache misses is obtained and used in a policy that attempts to reduce cache interference in the lowest-level (shared) cache of a multicore platform. This is done by using the information collected by the observation subsystem to compute a *cache weight* for each task—experimentation with several metrics led to the conclusion that computing cache weights based on *shared cache misses per clock cycle* provided the best results. For a set of cores sharing a cache, the policy ensures that multiple tasks with "heavy" cache weights are not co-scheduled. Experiments revealed that this policy results in significant speedup for SPEC CPU2000 benchmarks as compared with unmodified Linux and OS X operating systems.

This cache-aware scheduler is closest in nature to the scheduler that is described in this dissertation for real-time systems. The observation subsystem and scheduling policy described here are analogous to our cache profiler described in Chapter 5 and our scheduling heuristic described in Chapter 4. However, these components behave quite differently in our system due to the special needs of real-time workloads. In particular, the policies in our scheduling heuristic are considerably more elaborate so that timing constraints can be ensured, and since more is known about our workloads *a priori*, we can choose to idle cores to prevent cache thrashing when doing so will not violate timing constraints.

2.3 Real-Time Operating Systems

In this section, we provide a discussion of real-time operating systems (RTOSs), since we implement our cache-aware scheduler within LITMUS$^{\text{RT}}$, a Linux-based testbed that provides much of the functionality of a multiprocessor RTOS.

Most prior work on RTOSs has focused on *uniprocessor* systems—see [63] for a recent survey. In most such work, techniques for *scheduling* multiprocessor workloads are rarely discussed. The prevailing attitude seems to be that, on a multiprocessor platform, partitioning is the only viable choice, and therefore, scheduling reduces to a uniprocessor problem. Given this prior emphasis, we mostly limit our discussion of prior work to LITMUS$^{\text{RT}}$ and related research that pertains to Linux or that addresses multiprocessor systems more directly.

2.3.1 LITMUS$^{\text{RT}}$

LITMUS$^{\text{RT}}$ [22], or the **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems, is a Linux-based testbed that supports multiprocessor real-time scheduling and synchronization policies. The creation of LITMUS$^{\text{RT}}$ was crucial to demonstrate the potential viability of various scheduling methods being investigated by our group. Until the creation of LITMUS$^{\text{RT}}$, such a testbed did not exist, and no implementations of global scheduling policies existed in real systems. The author of this dissertation led the original effort to create LITMUS$^{\text{RT}}$ and conduct the first empirical evaluation of global multiprocessor real-time scheduling policies [22], and was also involved in later studies that used LITMUS$^{\text{RT}}$ to explore synchronization [16] and scalability [15] issues. During our studies, our empirical evaluations provided a better understanding of the "common case" for a variety of scheduling and synchronization problems. By designing algorithms to perform well in the common case rather than under a myriad of theoretical corner cases, the resulting approaches tended to be simpler and easier to analyze. Thus, a recurring theme of this work was that simpler approaches often resulted in better performance in practice, since they resulted in lower overheads or allowed for more accurate schedulability tests to be used. The most recent version of LITMUS$^{\text{RT}}$, which is a patch against Linux kernel version 2.6.24, is publicly available [69].

2.3.2 RTLinux

Yodaiken and Barabanov [71] are responsible for an early effort at creating an RTOS, known as RTLinux, which has evolved over time into a commercial product. RTLinux runs real-time tasks in a thin real-time kernel, with Linux itself running on top of this kernel as a low-priority *background task*. This strategy prevents the Linux kernel from disrupting real-time tasks, but at the same time, restricts the ability of such tasks to invoke Linux kernel services, and may severely impact the performance of non-real-time tasks by forcing them to be scheduled at the lowest priority at all times. In contrast, LITMUS$^{\mathrm{RT}}$ incorporates the scheduling algorithms that are required directly into Linux itself. RTLinux supports periodic threads, but scheduling is limited to FIFO, round-robin, and fixed-priority schemes. While various multiprocessor scheduling algorithms could potentially be incorporated within RTLinux, this would preclude supporting in a straightforward manner tasks that require the services of the Linux kernel.

2.3.3 Multiprocessor RTOSs

To our knowledge, multiprocessor-based RTOSs were first considered as part of work on the Spring kernel, created by Stankovic and Ramamritham [64]. The scope of Spring extended beyond stand-alone multiprocessor systems and encompassed distributed systems comprised of several multiprocessing nodes and tasks with synchronization requirements; however, Spring predated almost all of the recent advances in multiprocessor scheduling. In particular, this includes advances in global multiprocessor scheduling algorithms, such as PD^2, and results related to soft real-time systems.

In other recent work concerning multiprocessors, Stohr *et al.* [65] presented the **RE**altime with **C**ommercial **O**ff-The-Shelf **M**ultiprocessor **S**ystems (RECOMS) software architecture, which is a framework for running a general-purpose OS and an RTOS on the same multiprocessor machine. This framework partitions the system by placing the general-purpose OS on its own processor and preventing I/O accesses from interfering with the RTOS. RECOMS was designed as an extension to the **R**eal**T**ime **A**pplication **I**nterface, or RTAI [31], which is closely related to RTLinux, and therefore the work in [65] is different from LITMUS$^{\mathrm{RT}}$

in the same ways. The static partitioning of the system may allow tasks to perform better with the RECOMS architecture than RTLinux, especially if the non-real-time and real-time workloads are assigned to cores that share different sets of resources (*e.g.*, caches), but is still less flexible than LITMUS$^{\mathrm{RT}}$. Additionally, the work in [65] suffers from the problem that real-time tasks cannot invoke Linux kernel services.

2.3.4 Linux Real-Time Preempt Patch

The Real-Time Preempt Patch [51] is a patch against mainline Linux, and represents a large-scale effort to create a version of Linux that is more suitable for real-time applications. The main feature that is provided as a part of this project is a fully preemptible kernel. The features to support a fully preemptible kernel include preemptible spin locks and critical sections with support for priority inheritance, representation of interrupt handlers as preemptible kernel threads, and a new Linux timer API that supports user-space high-resolution POSIX timers. One goal of this project is to merge many of these features into the mainline kernel, so that more sophisticated real-time support can be provided natively—in fact, merging is in progress for some of the features listed here. This would eliminate the need for patches that must "keep up" with mainline Linux changes independently (meaning that the latest mainline Linux kernel features cannot be used until a patch is created for that kernel release).

2.4 Real-Time Scheduling on Multithreaded Platforms

A number of publications exist on scheduling in systems employing simultaneous multithreading (SMT), where multiple hardware threads, all on the same single-core chip, contend for shared hardware resources. The goal of SMT is to allow a single core to be treated as multiple logical CPUs (one per hardware thread) by an operating system, particularly within the scheduler; thus, threads can contend for shared chip resources much in the same way that multiple cores may contend for resources on a multicore chip.

Jain *et al.* [40] considered scheduling for SMT systems in the context of schedulability for soft real-time workloads (in this case, percentage of deadlines met was the metric of interest). This was done by comparing a variety of scheduling policies for task sets representing either synthetic or multimedia workloads. Approximately half of these policies were *symbiosis-aware*, meaning that they attempted to increase the overall "symbiosis factor" of the system when making scheduling decisions. The symbiosis factor of the system represents the impact of co-scheduling a set of jobs as the ratio of instructions-per-cycle (IPC) achieved by each job when co-scheduled over its IPC achieved in isolation (during single-threaded execution), defined for a set of N co-scheduled jobs as follows (from [40]).

$$\text{symbiosis factor} = \sum_{i=1}^{N} \frac{\text{realized IPC of job } i}{\text{single-threaded IPC of job } i} \tag{2.12}$$

In this study, the primary system of concern contained a single processor with two hardware threads. When considering the percentage of schedulable task sets, the best performing policy was a global approach where the job J with the earliest deadline is co-scheduled with a set of jobs that will maximize symbiosis when co-scheduled with J. Additionally, an exception is made to this policy that allows slightly better schedulability when tasks with high utilizations exist. This policy outperformed partitioning policies and those that did not consider symbiosis. The downside to the policies considered in [40] is that most of them, including all symbiosis-aware policies, do not provide a way to *analytically guarantee* schedulability—in this work, schedulability was determined through empirical observations of deadline misses during experiments. Another downside of this work is that determining the symbiosis factor for every possible set of co-scheduled jobs may be infeasible in many scenarios, especially as the number of hardware threads increases.

Snavely *et al.* [60] considered symbiosis-aware and symbiosis-oblivious scheduling policies in experiments on a simulated SMT platform where jobs may have different priorities that entitle them to some share of the processing resources of the system. The policies examined in this work resulted in system throughput increases of up to 40% and a reduction in average job response times; however, these policies do not directly consider real-time schedulability and result in even weaker guarantees than those in [40]. Further, these approaches still require the computation of a symbiosis factor for each possible set of co-scheduled jobs, which can quickly become infeasible as the number of hardware threads increases. Additionally, these policies rely on the assumption that maximizing parallelism within an SMT system is typically the best approach, but such an approach is often *not* best for multicore platforms with shared caches.

2.5 Conclusion

In this chapter, we reviewed several different areas of prior work. This review included an overview of real-time scheduling concepts, methods for scheduling and profiling tasks in the presence of a shared cache (for non-real-time systems), RTOSs, and SMT-aware real-time scheduling. In the remainder of this dissertation, we focus our attention at the intersection of the multiprocessor scheduling research being conducted within the real-time community (discussed in Section 2.1) and the multicore (*i.e.*, shared-cache-aware or resource-constrained) scheduling research being conducted outside of the real-time community (discussed in Section 2.2). It is at this intersection that this dissertation makes its primary contribution.

CHAPTER 3

PROBLEM STATEMENT

In this chapter, we will formally state the problem that this dissertation addresses. We then show that the problem, as formally stated, is NP-hard in the strong sense. Therefore, a practical, efficient, and exact solution to the problem is unlikely to exist, and we must rely on less perfect approaches. Finally, we review two of our early approaches to solving parts of the problem and discuss their shortcomings, which warranted the additional research that led to the work presented in Chapters 4 and 5.

3.1 The Problem: Cache-Aware Real-Time Co-Scheduling

In this section, we begin with a more detailed introduction to MTTs, first mentioned in Chapter 1, as a deeper understanding of MTTs is necessary before stating the problem that this dissertation addresses. Next, we formally present the problem and prove that it is NP-hard in the strong sense.

3.1.1 Multithreaded Tasks

In Chapter 1, the MTT abstraction was stated as a contribution of this dissertation. MTTs are groups of tasks that reference a common set of data, and are intended to specify groups of *cooperating* tasks. To use the terminology from [55], MTTs specify groups of tasks for which co-scheduling would be constructive. (Note that an ordinary periodic or sporadic task is just a “single-threaded” MTT.) For example, each real-time task of an MTT could be represented within an operating system as a single thread of the same process; thus, all tasks of the MTT would share resources.

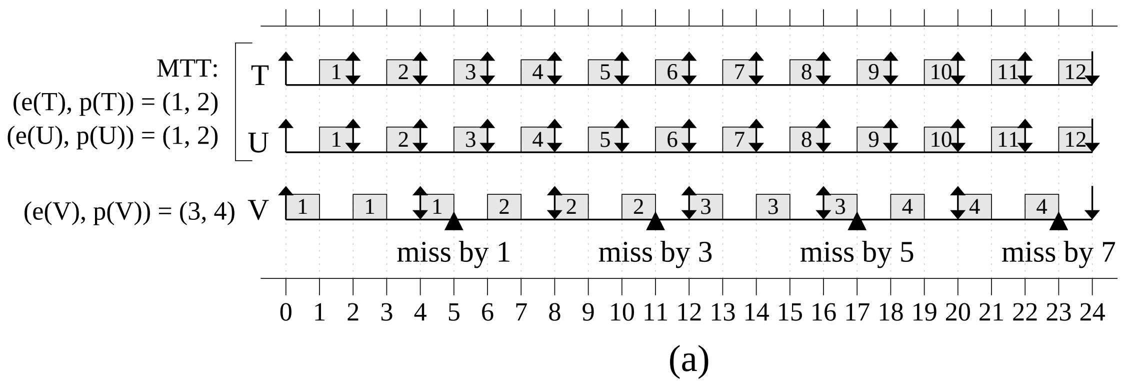
The (periodic or sporadic) tasks within an MTT may have different execution costs, but have a common period. Further, in sporadic task systems, task periods *within* an MTT must coincide. We consider this to be reasonable since we intend an MTT to represent a single recurrent real-time computation, and the tasks within an MTT are assumed to be cooperating to perform different portions of that computation. In this dissertation, we assume that, on a machine with M cores, each MTT has at most M tasks, representing the maximum achievable level of parallelism.

If cache miss rates are not a concern, then the tasks of an MTT can be treated as individual independent tasks during scheduling; however, since the tasks within an MTT reference a common set of data (and thus are assumed to share data), we would expect to see cache miss rate reductions from co-scheduling such tasks, for two reasons. First, co-scheduling the tasks within an MTT increases the likelihood of cache reuse, which can reduce cache miss rates for such tasks. Second, co-scheduling the tasks within an MTT minimizes the amount of time that the set of data referenced by that MTT, or its working set, must be available in the shared cache. This frees space in the cache for other MTTs. As a result, the cache miss rates of the tasks within those MTTs may improve, especially if shared cache thrashing can be avoided as a result.

We would like to devise mechanisms that encourage all tasks within an MTT to be co-scheduled.[1] Thus, the essence of the problem at hand is to encourage *parallelism* within an MTT: when one task of an MTT is scheduled, *all* tasks of that MTT should be scheduled to increase cache reuse and reduce the possibility of cache thrashing. Unfortunately, this is not always possible in the presence of real-time constraints. Due to this limitation, we do not *force* co-scheduling to occur, but rather strongly *influence* co-scheduling choices so that, in the absence of pressing timing constraints (*e.g.*, tardy jobs), co-scheduling is achieved.

Example (Figure 3.1). Consider Figure 3.1, where the schedules shown in this example are not intended to reflect any particular scheduling *policy*—indeed, for inset (b), any policy that

[1]Note that for certain types of applications, it may be more beneficial to schedule MTT tasks differently, *e.g.*, in a *pipelined* manner. (Liu and Anderson [45] have recently explored issues related to the pipelined execution of real-time tasks.) While this issue is not considered further in this dissertation, we believe that allowing MTTs to specify their co-scheduling preferences would make such an abstraction considerably more powerful, and thus plan to consider this important issue in future work.



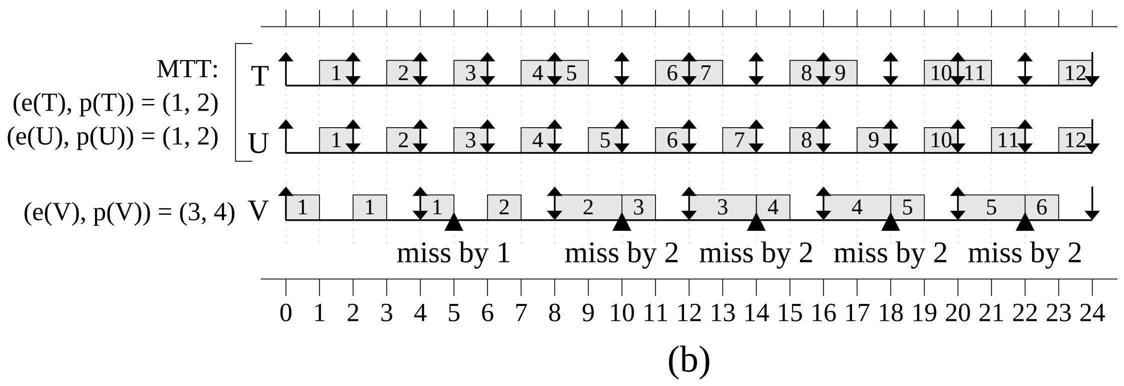


Figure 3.1: Example two-core schedules demonstrating **(a)** how forcing the co-scheduling of the tasks in an MTT can result in unbounded deadline tardiness for other tasks in the task set; but **(b)** merely influencing co-scheduling can result in bounded tardiness.

could generate the schedule shown would have to be clairvoyant. Instead, this example intends to prove a point related to co-scheduling MTTs. In Figure 3.1(a), forcing the co-scheduling of the jobs of tasks T and U, both of which belong to the same MTT, results in *unbounded* deadline tardiness for task V. (Continuing this job execution pattern into the future causes the tardiness of V to increase with each successive job.) Thus, even soft real-time guarantees cannot be made. In inset (b) of Figure 3.1, the jobs of T and U are only co-scheduled when a tardy job of V does not need to execute. This results in less co-scheduling, but allows soft real-time guarantees to be made due to bounded tardiness. (Note that, in the latter half of the schedule, there is a consistent job execution pattern and the tardiness of V does not increase.) $\square$

When global scheduling approaches such as our cache-aware scheduler are employed, scheduling-related data structures (*e.g.*, run queues) are shared. As a result, encouraging co-scheduling becomes relatively straightforward; since the tasks of an MTT have a common period, we can encourage co-scheduling by changing how jobs are prioritized within run queues. Such approaches are described further in Section 3.2.2 of this chapter and in Chapter 4.

Note that this method of encouraging the co-scheduling of MTTs to reduce cache miss rates is similar to the gang scheduling approach taken by Batat and Feitelson [10] for non-real-time systems. Their work found that, for a set of parallel jobs that would benefit from co-scheduling, delaying the execution of some of the jobs in the set when the memory requirements of all jobs cannot be satisfied is more efficient than co-scheduling all jobs regardless of memory requirements. In this dissertation, the sets of jobs to schedule would be within MTTs, which are intended to represent a similar notion of parallelism in *real-time systems*, particularly in recurrent task models that traditionally handle only the sequential execution of tasks. As we will discuss further in Chapter 4, we also take a similar approach to co-scheduling, though it is somewhat more coarse-grained, and is instead concerned with cache requirements: we avoid scheduling *any* job of an MTT in a quantum unless the cache requirements of *all* jobs can be satisfied, as determined by the size of the cache footprint of the MTT. Further, we do not force the desired co-scheduling to occur, but instead influence co-scheduling so that it occurs whenever it will not cause timing constraints to be violated—the addition of real-time constraints adds a (non-trivial) dimension to the problem that has not been considered before.

In light of the work on symbiosis-aware scheduling by Jain *et al.* [40] and Snavely *et al.* [60] that was described in Chapter 2, one could think of an MTT as a group of tasks with an infinite symbiosis factor. In other words, co-scheduling the tasks of an MTT should always be encouraged, and it is assumed that every MTT benefits equally when its tasks are co-scheduled (which may be seen as an oversimplification). However, note that MTTs avoid some of the downsides to the symbiosis-aware work discussed in Chapter 2. These downsides include: **(1)** a lack of techniques for analytically guaranteeing schedulability, **(2)** the need to determine a symbiosis factor for every possible set of co-scheduled jobs, and **(3)** in many cases, an assumption that maximizing parallelism is the best strategy, even when doing so (instead of idling cores) would result in shared cache thrashing.

3.1.2 Problem Statement

We now formally state the problem that we address in this dissertation. Given a task set τ, a set $\mathcal{M}$ of MTTs where every task $T \in \tau$ belongs to exactly one MTT $\mathcal{T} \in \mathcal{M}$ (to specify how tasks are grouped into MTTs), a WSS $\mathcal{W}_{\mathcal{T}}$ for each MTT $\mathcal{T}$, and M cores sharing a cache of size C, we first define what it means for a set of jobs within an MTT to execute with *maximal concurrency*; this is followed with the decision problem itself.

Maximal Concurrency: Assume that a set of n jobs $J^1, J^2, \ldots, J^n$ exists, and these jobs are ordered such that the execution cost of J^i is at most the execution cost of J^{i+1}, and all jobs have a common release time and deadline. Such a set of jobs executes with *maximal concurrency* if, for each job J^i, J^i executes only when job J^{i+1} executes. (This definition implies that job J^n has no restrictions on when it executes, and job J^1 must execute concurrently with every other job in the set.)

The decision problem is as follows.

Cache-Aware Real-Time Co-scheduling Problem (CARTCP): Does there exist an M-core schedule (with a length equal to the hyperperiod of the task set, that can be repeated indefinitely to generate longer schedules) for the tasks in τ so that: **(i)** no job deadline is missed; **(ii)** for any set $\mathcal{S}$ of co-scheduled MTTs, $\sum_{\mathcal{T} \in \mathcal{S}} \mathcal{W}_{\mathcal{T}} \leq C$, and **(iii)** co-scheduling within MTTs occurs such that, for any i and MTT $\mathcal{T}$, the set of jobs consisting of each T_i, where $T \in \mathcal{T}$, executes with maximal concurrency?

We next show that CARTCP is NP-hard in the strong sense.

3.1.3 NP-Hardness Proof for CARTCP

We now show that CARTCP is NP-hard in the strong sense by a polynomial-time transformation from 3-PARTITION, a problem that is already known to be NP-hard in the strong sense [35], to CARTCP. 3-PARTITION is defined as follows.

3-PARTITION **(from Garey and Johnson** [35]**):** Given a finite set A of $3m$ elements, a bound $B \in Z^+$, and a "size" $s(a) \in Z^+$ for each $a \in A$, such that each $s(a)$ satisfies $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$, can A be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ such that, for $1 \leq i \leq m$, $\sum_{a \in S_i} s(a) = B$? (Notice that the above constraints on the item sizes imply that every such S_i must contain *exactly* three elements from A.)

We transform 3-PARTITION to CARTCP by showing that an arbitrary instance of 3-PARTITION returns the answer "yes" if and only if an equivalent instance of CARTCP returns the answer "yes". The equivalent instance is created as follows.

- Each task represents a unique element in the set A (there are $3m$ tasks total). Each task has an execution cost of one and a period of m.
- Each task belongs to a different MTT; in other words, all tasks are single-threaded. Therefore, (iii) in CARTCP is trivially satisfied.
- The WSS of each task is $s(a)$ for the equivalent element $a \in A$, with the same restrictions on its value as in 3-PARTITION.
- Three cores share a cache of size B. The tasks scheduled at any time will map to the same set in 3-PARTITION. If (i) in CARTCP is satisfied and no deadlines are missed, then exactly three tasks will be scheduled at any time. All sets will be disjoint, and there will be m such sets, due to task execution costs and periods.
- If (ii) in CARTCP is satisfied, then the sum of the size of all "co-scheduled" elements (as determined by the tasks that represent them) is at most B. Further, given the restrictions imposed on task WSSs from 3-PARTITION, this size must be *exactly* B at all times in the schedule; otherwise, the sum of the size of all co-scheduled elements would have to exceed B at some time, which would violate (ii).

Since this transformation can be accomplished in polynomial time, CARTCP is NP-hard in the strong sense.

3.2 Early Approaches

In this section, we review two of our earlier approaches to deriving partial solutions to CARTCP. We first describe a method to enforce (i) and (ii), and then describe a method to enforce (i) and (iii). We conclude the presentation of each method by stating its limitations, and conclude this section by explaining why combining the two methods (to create a method that satisfies all three conditions of CARTCP) is problematic, and warranted additional research.

3.2.1 Preventing Thrashing With Megatasks

In [4], co-scheduling a group of (single-threaded) tasks was discouraged when it would cause shared cache thrashing. This was accomplished by grouping tasks into *megatasks*. A megatask is a set of tasks that is treated as a single schedulable entity by a top-level scheduler. Each task of a megatask is referred to as a *component task*; if a megatask is scheduled on M processors at time t, then (up to) M of its component tasks may be scheduled at time t. PD^2 is the scheduling policy used at both levels of this scheduling hierarchy.

Unlike MTTs, where parallelism is encouraged within each MTT, megatasks prevent cache thrashing by *restricting* parallelism, as it is assumed that cache thrashing could occur when "too many" of the component tasks of a megatask are co-scheduled. Let γ be a megatask comprised of component tasks with total utilization u. The component tasks of γ will require between $\lfloor u \rfloor$ and $\lceil u \rceil$ processors for their deadlines to be met. A megatask is allocated $\lfloor u \rfloor$ allocations every quantum, and an additional allocation whenever a "special task" associated with it is scheduled by the top-level scheduler. This special task has a utilization of $u - \lfloor u \rfloor$, corresponding to the fractional portion of the total utilization of the megatask. Therefore, this scheme ensures that at most $\lceil u \rceil$ tasks in γ are ever co-scheduled. Megatasks prevent cache thrashing when, for each megatask γ, co-scheduling at least $x \geq \lceil u \rceil$ tasks within that megatask is necessary to cause thrashing. As a result, when tasks are effectively grouped into megatasks, restricting parallelism in this way prevents groups of tasks that could cause cache thrashing from being co-scheduled.

Our approach for preventing thrashing by using megatasks is a two-step process: **(i)** combine tasks into groups, where we want to discourage co-scheduling among the tasks within each group—each group becomes a megatask; and **(ii)** at runtime, use a scheduling policy that reduces concurrency within megatasks, to prevent thrashing.

Example (Figure 3.2). Consider a four-core system with a 1 MB shared cache. Naturally, we would like to avoid thrashing by ensuring that the combined WSSs of any co-scheduled tasks does not exceed the size of this shared cache. As shown in Figure 3.2, the task set is comprised of three tasks of utilization 0.6 and a WSS of 400K (Group A), and four tasks of utilization 0.3 and with a WSS of 100K (Group B). The total utilization of the task set is three, so co-scheduling at least three tasks during any quantum is unavoidable. However, since the combined WSS of the tasks in Group A exceeds the shared cache capacity, it is desirable that the three co-scheduled tasks not all be from this group; otherwise thrashing can occur, as shown in inset (a) when PD^2 is used without megatasks. In inset (b), megatasks are employed; because the total utilization of Group A is 1.8, we can combine the tasks in Group A into a single megatask, and ensure that at most two tasks from it are ever co-scheduled. The same is also done with Group B to create a megatask of utilization 1.2. $\square$

Sub-optimality of using megatasks. Unfortunately, the hierarchical scheduling policy required for megatasks may result in component task deadline misses, even though PD^2 is used at both levels of the hierarchy. This is because there may be a misalignment between when processing time is allocated to a megatask, and when the component tasks of that megatask are able to consume that time. As a result, the processing time that is assigned to a megatask may sometimes be unused, and sufficient processing time will be unavailable when it is needed, resulting in missed deadlines.

Example (Figure 3.3). As an example, consider the case where five tasks are placed within a single megatask, as shown in Figure 3.3, with a total utilization of 7/5. The component tasks receive two processor allocations during any quantum that the special task S is scheduled (as shown), and one allocation otherwise. As usual, PD^2 is employed. From time 6 to time 9, the special task receives no allocations. As a result, sufficient processing capacity does not exist to allow all deadlines to be met, resulting in a deadline miss for job W_3. $\square$

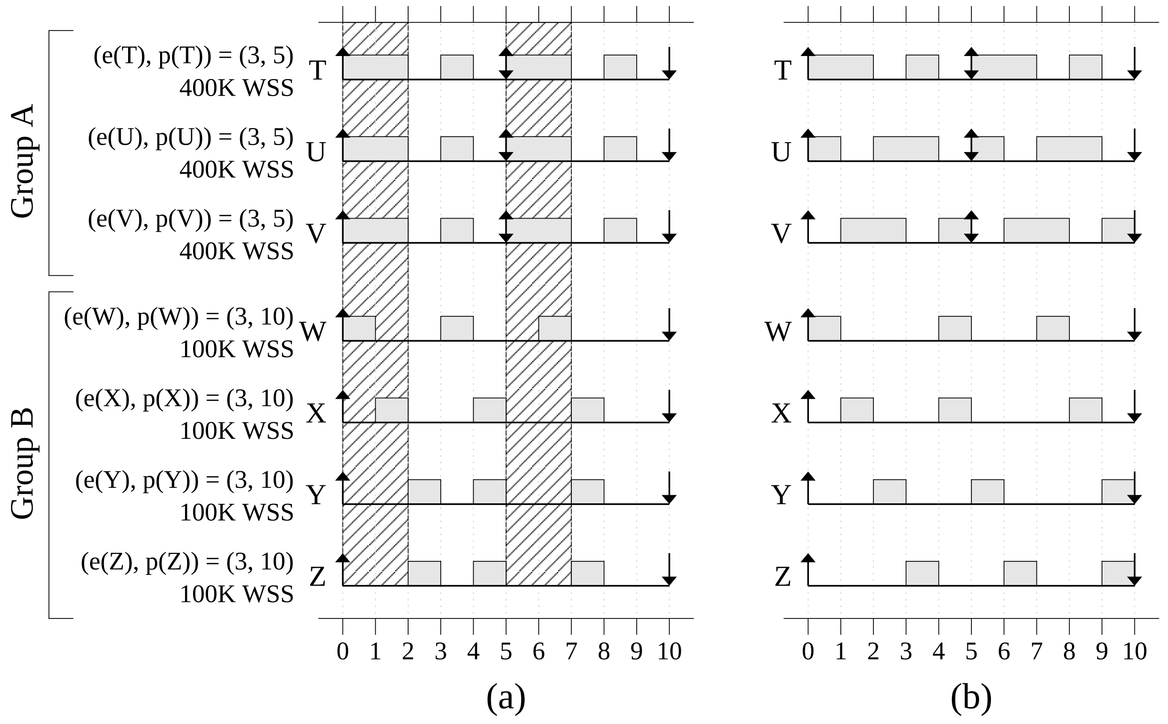


Figure 3.2: An example of how megatasks can prevent shared cache thrashing. Both insets show schedules for the same task set when PD^2 is employed **(a)** without megatasks and **(b)** with megatasks. Thrashing occurs during "hatched" quanta.

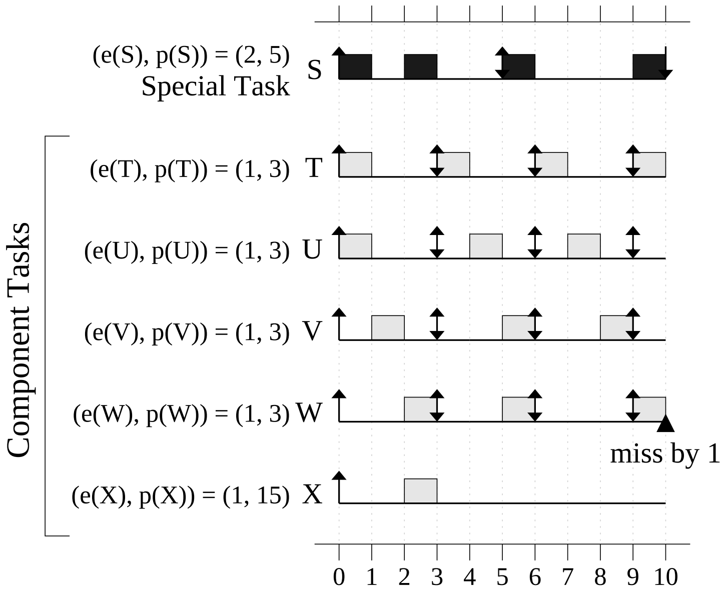


Figure 3.3: An example of how deadline misses can occur when using megatasks, due to a mis-alignment between when processing time is allocated to the special task S, and when the component tasks need that processing time.

We can handle this issue in two ways. First, we note that the amount by which deadlines are missed in these scenarios is bounded, since the megatask is receiving a sufficient share of the system to eventually complete all tardy jobs; therefore, soft real-time constraints can be ensured. Second, we can avoid these misses altogether by slightly inflating the utilization of each megatask, a process that we call *reweighting*, during which we increase the utilization of the megatask so that it is slightly larger than the total utilization of its component tasks. This approach allows processing time to be available when it is needed by providing allocations to the megatask at a more frequent rate, even if some of those allocations are wasted. We have found that the benefits of using megatasks often far outweigh any utilization loss due to reweighting.

Packing strategies. The *packing strategy* determines how tasks are grouped into megatasks. In the strategy that we proposed, tasks are considered in decreasing order of WSS. One megatask is created at a time. If the current task could be added to the current megatask without pushing the utilization of the megatask beyond the next integer boundary, then this is done, because if the megatask could prevent thrashing among its component tasks before, then it could do so afterwords (recall that tasks are considered in decreasing order of WSS). Otherwise, a check is made to determine whether creating a new megatask would be better than adding to the current one—adding to the current one is preferred if allowing an extra task to be scheduled would still prevent thrashing. While this is an easy packing strategy, it does not necessarily result in the best possible packing for avoiding cache thrashing. For example, a better packing might be possible by allowing a new task to be added to a megatask generated prior to the current one.

Example. Consider a task set with seven tasks: two with utilization 1/2 and a WSS of 300K, two with utilization 2/10 and a WSS of 299K, two with utilization 9/10 and a WSS of 298K, and one with utilization 3/10 and WSS of 297K. Assume an eight-core platform with a 1 MB shared cache, so no more than three of these tasks can be co-scheduled without causing thrashing. Tasks are added to the current megatask in decreasing order of WSS while the utilization of the megatask remains at most three. This results in all tasks with utilization 1/2 and 2/10, and one task with utilization 9/10, being added to one megatask, and the remaining two tasks being placed into their own megatask. As a result, our packing strategy creates two megatasks with utilizations 23/10 and 12/10, and at most five tasks will be co-scheduled, reducing thrashing. Note that, if the task with utilization 3/10 could be placed into the first megatask, then the utilization of that megatask would be 26/10, which is still less than three, and the utilization of the second megatask would be 9/10, ensuring that no more than four tasks were ever co-scheduled, reducing thrashing further. □

Experimental results. To assess the efficacy of using megatasks in reducing cache contention, we conducted experiments comparing its performance to both PD^2 and PEDF using the SESC architecture simulator [58]. The simulated architecture we considered consists of a variable number of cores, each with dedicated 16K L1 data and instruction caches (4- and 2-way set associative, respectively) with random and LRU replacement policies, respectively, and a shared 8-way set associative 512K on-chip L2 cache with an LRU replacement policy. Each cache has a 64-byte line size. Each scheduled task was assigned a utilization and WS. A task references its WS sequentially, looping back to the beginning when the end is reached. All scheduling, preemption, and migration costs were accounted for in these simulations. For PEDF, tasks were placed onto cores in decreasing order of WSS using a first-fit approach. If successful, this ensures that the largest possible combined WSS of all tasks running concurrently is small, and thrashing is likely to be avoided. Therefore, in these experiments, we believe that PEDF is treated more fairly than megatasks, for which a less optimal packing strategy is used.

We performed experiments with both hand-crafted and randomly-generated task sets. In this overview, we will discuss only the hand-crafted task sets, as they most clearly demonstrate the impact of using megatasks.

Hand-crafted task sets. The hand-crafted task sets that we created are listed in Table 3.1. Each was run on either a four- or eight-core machine, as specified, for the indicated number of quanta (assuming a 1-ms quantum length). Table 3.2 shows for each case the L2 (shared) cache miss rates that were observed.

Task Set	**No. Tasks**	**Task Properties: ($e(T)$, $p(T)$, WSS)**	**No. Cores**	**No. Quanta**
BASIC	3	(3, 5, 250K)	4	100
SMALL_BASIC	5	(7, 20, 250K)	4	60
ONE_MEGA	5	(7, 10, 120K)	8	50
TWO_MEGA	6	Half (3, 5, 190K) and half (3, 5, 60K)	8	50

Table 3.1: Properties of example task sets.

Task Set	PEDF	PD^2	**Megatasks**
BASIC	89.12%	90.35%	2.20%
SMALL_BASIC	17.24%	28.84%	2.89%
ONE_MEGA (1 megatask)	11.07%	11.36%	0.82%
ONE_MEGA (2 megatasks of utilizations 2.1 and 1.4)	11.07%	11.36%	1.79%
TWO_MEGA (1 megatask, all tasks included)	10.94%	10.97%	5.67%
TWO_MEGA (1 megatask, only 190K WSS tasks)	10.94%	10.97%	5.52%
TWO_MEGA (2 megatasks, one each for 190K and 60K tasks)	10.94%	10.97%	1.02%

Table 3.2: L2 cache miss ratios for example task sets.

BASIC consists of three high-utilization tasks. Running any two of these tasks concurrently will not thrash the L2 cache, but running all three will. The total utilization of all three tasks is less than two, but the number of cores is four. Both PD^2 and PEDF use more than two cores, causing thrashing. By combining all three tasks into one megatask, thrashing is eliminated. In fact, the difference here is quite dramatic. SMALL_BASIC is a variant of BASIC with tasks of smaller utilization. The results here are similar, but not quite as dramatic.

ONE_MEGA and TWO_MEGA give cases where one megatask is better than two and vice versa. In the first case, one megatask is better because using two megatasks of utilization 2.1 and 1.4 allows an extra task to run in some quanta, during which thrashing may occur. In the second case, using two megatasks ensures that at most two of the 190K WSS tasks and two of the 60K WSS tasks run concurrently, thus guaranteeing that their combined WSS is under 512K. Packing all tasks into one megatask ensures that at most four of the tasks run concurrently; however, it does not allow us to specify *which* four. Thus, all three tasks with a 190K WSS could be scheduled concurrently, which is undesirable. Interestingly, placing just these three tasks into a single megatask results in little improvement—thrashing is less likely, but still possible.

Limitations. While this method of packing tasks into megatasks to avoid thrashing is often highly effective at reducing cache miss rates, such a method is also significantly limited in certain ways, as follows. Most of these limitations are related to the fact that the scheduler itself does not make any cache-aware scheduling decisions online, and as a result, the packing strategy must often be conservative.

- A task can be part of only *one* megatask. This means that if two tasks would thrash the cache if co-scheduled, then both tasks must be within the same megatask if thrashing is to be avoided. For example, one task T in a task set with n tasks may cause thrashing when co-scheduled in isolation with any of the other $n - 1$ tasks. While we only need to ensure that T always runs in isolation, we cannot achieve this without placing all n tasks within the same megatask. If the utilization of this megatask exceeds one as a result, then the guarantee provided by megatasks would be too weak, and no megatask packing would be sufficient to avoid thrashing. In practice, it may be possible to meet timing constraints by scheduling T in isolation and co-scheduling the other $n - 1$ tasks whenever T is not scheduled.

- Similarly, note that it is impossible to guarantee that fewer than $\lceil u \rceil$ of the tasks in a megatask γ with a utilization of u will execute at any time. If co-scheduling all possible sets of $\lceil u \rceil$ tasks in γ will thrash the shared cache, then the system simply must be re-designed (*e.g.*, by finding a better assignment of tasks to megatasks, or failing that, actually modifying the tasks in the system). If thrashing will only occur when specific $\lceil u \rceil$-sized groups within the megatask are co-scheduled, rather than *any* $\lceil u \rceil$-sized group, then better results might be obtained if we could dynamically reduce co-scheduling below the threshold guaranteed by the megatask (*e.g.*, by delaying one or more processor allocations to the megatask) when it can be shown that thrashing will occur otherwise.

- Alternately, the maximum amount of concurrency that can be supported within the megatask without thrashing may be considerably higher than $\lceil u \rceil$. As a result, parallelism is restricted unnecessarily. For example, consider a system with ten tasks of utilization 1/10, each with a WSS equivalent to one-third of the shared cache size. Three such tasks can be co-scheduled without causing thrashing; however, the utilization of the megatask will restrict parallelism so that at most one processor is utilized. This may not appear to be a major concern unless we want to introduce MTTs into our task model, and allow some degree of co-scheduling within MTTs in addition to using megatasks to avoid thrashing. In that case, restricting parallelism in this way will keep one processor continually busy, limiting the parallelism available to MTTs. More generally, unnecessarily restricting parallelism may not allow other scheduling constraints to be satisfied.

- In dynamic systems, this method can result in high scheduling overheads, as new task arrivals or changes to task WSSs may require a complete repacking of tasks into megatasks. As a result, this method may be infeasible to use in such systems.

- Megatasks as described rely on Pfair scheduling. The methods described in Chapters 4 and 5 instead consider GEDF, as it has been shown to perform better in terms of schedulability for soft real-time systems when overheads are considered.

3.2.2 Encouraging MTT Co-Scheduling Through Early-Releasing

In [3], the notion of an MTT described earlier is introduced, and a method to encourage the co-scheduling of tasks within MTTs is proposed. It was found that, by selectively allowing jobs or subtasks to be released early (that is, before their designated release times as described in Chapter 2), the amount by which higher-priority jobs or subtasks interfered with MTT co-scheduling can be significantly reduced. As a result, co-scheduling guarantees could be made based on how early jobs or subtasks were allowed to be released. This method often resulted in significantly lower cache miss ratios for MTTs, especially those containing low-utilization tasks.

The focus of the method proposed in [3] is on minimizing a factor (also introduced in [3]) called *spread*. Assume that each MTT is comprised of tasks that have the same execution cost (and a common period, as required by definition). An MTT has a spread of k if for all i, the i^{th} unit of computation for each task of the MTT is scheduled within the interval

$[t, t+k)$ for some t (where a "unit of computation" can be defined as the size of the scheduling quantum).

While the method in [3] does not explicitly require each MTT to be comprised of tasks that have the same execution cost, we make such an assumption here for several reasons. First, it allows for a more straightforward presentation of the co-scheduling method in [3], as both the definition of spread and the scheduling rules, presented later, are simplified. Second, when all tasks in an MTT have the same execution cost, we can make analytical spread guarantees. When tasks have varying execution costs, our method still reduces spread; however, no guarantees can be made. In this case, the method is more of a heuristic than an exact approach for which co-scheduling guarantees can be made.

In the method in [3], MTTs are scheduled so that real-time constraints are met and spread is minimized to the extent possible. When spread is one, condition (iii) of problem CARTCP is satisfied; otherwise, by minimizing spread, we get as close as possible to satisfying (iii). Note that, even when spreads exceed one but remain small, a potential for cache reuse exists.

Proposed approach. The approach for minimizing spread while meeting real-time constraints is based upon two observations.

1. Global scheduling algorithms are more naturally suited to minimizing spread than partitioning approaches. This is particularly the case when using deadline-based scheduling methods. This is because "work" with a common deadline submitted at the same time will occupy consecutive slots in the run queue of the scheduler, and thus, such work will be scheduled in close proximity over time, unless disrupted by later-arriving, higher-priority work.

2. As discussed in Chapter 2, allowing jobs or subtasks to be released early does not affect timing guarantees, and the use of early-releasing is entirely optional for each job or subtask. We can exploit this fact to minimize disruptions to the co-scheduling of an MTT that are caused by higher-priority work.

Example (Figure 3.4). As an example, consider the PD^2 schedules in Figure 3.4, where tasks W and X are in the same MTT. Inset (a) shows a schedule without early-releasing. In

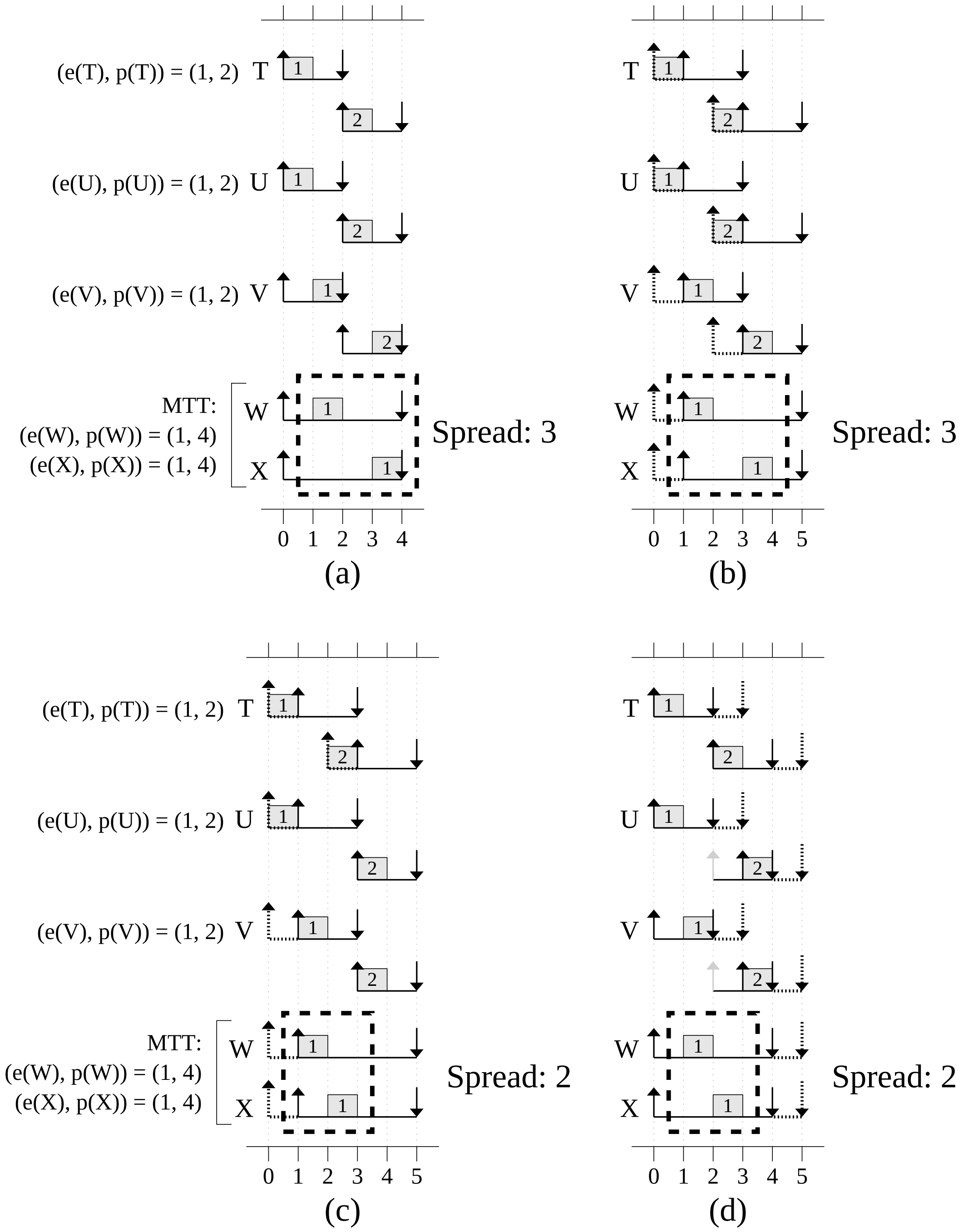


Figure 3.4: A two-core PD^2 schedule of a set of five tasks with **(a)** no early-releasing; **(b)** early-releasing by one quantum, where all windows are right-shifted by one quantum; **(c)** similarly-shifted windows, but selective early-releasing; **(d)** no shifting or early-releasing, but subtasks are considered optional within the first quantum after their release, and deadlines can be missed by one quantum.

inset (b), all subtask windows are shifted right by one quantum, and all subtasks are early-released by one quantum (indicated by dotted lines before each official release), producing the same schedule as in (a). (All deadline comparisons are the same.) We refer to a schedule in which all subtask windows are right-shifted by k quanta and all subtasks are early released by k quanta as a *k-shifted schedule*. In inset (b), $k = 1$. The schedules in (a) and (b) both result in a spread of three for the MTT. In inset (c), we show that selectively allowing early-releasing can reduce spread to two. Alternately, instead of shifting the schedule and early-releasing subtasks, as in (b) and (c), we can instead consider a subtask to be optional for scheduling for the first k quanta after its release, and allow it to miss its deadline by up to k quanta, as shown in inset (d). Here, the dotted lines after each window indicate by how much each deadline could be missed (though no misses occur here). In general, if subtasks can be early-released to the extent we require, then no deadlines will be missed; otherwise, deadlines may be missed, but by bounded amounts only. □

Scheduling rules. Based upon the above observations, we have devised a set of rules for *guaranteeing* low spreads in deadline-based, global scheduling approaches. We have applied these rules to both PD^2 and GEDF, assuming an $(X-1)$-shifted schedule is used (or alternately, that deadlines can be missed by up to $X-1$ quanta—see Figure 3.4(d)).

Three rules are required, and are outlined in detail for PD^2 below.

- **Urgent Tasks.** When subtask $T_{(i)}$ is scheduled, where task T corresponds to a task within some MTT $\mathcal{T}$, and T is the first task in $\mathcal{T}$ whose i^{th} subtask is scheduled, each subtask $U_{(i)}$, where U is also a task of $\mathcal{T}$ and $U \neq T$, is flagged "urgent" until it is also scheduled.

- **Early-Release Eligibility.** A *non-urgent* subtask $T_{(j)}$ at time t is "early-release eligible" at t if all of the following hold:

  1. $r(T_{(j)}) - (X-1) \leq t < r(T_{(j)})$ (*i.e.*, time t is within $X-1$ quanta of the actual release time of subtask $T_{(j)}$).

  2. All subtasks $T_{(k)}$ of T, where $k < j$, have already been scheduled prior to time t.

3. $|\mathcal{U}| + |\mathcal{H}| < M$, where M is the number of cores and at time t, $\mathcal{U}$ is the set of eligible urgent subtasks, and $\mathcal{H}$ is the set of non-urgent eligible subtasks $T_{(k)}$, where $r(T_{(k)}) \leq t$, such that each subtask in $\mathcal{H}$ has higher priority than at least one subtask in $\mathcal{U}$. Note that tasks in $\mathcal{H}$ are (by definition) not early-release eligible at time t.

4. Subtask $T_{(j)}$ is one of the $e = M - (|\mathcal{U}| + |\mathcal{H}|)$ highest-priority subtasks at time t satisfying (i) and (ii) above.

A subtask $T_{(j)}$ that is *urgent* at time t is "early-release eligible" at time t if conditions (i) and (ii) hold for it.

- **Priorities.** Eligible subtasks (early-released or not) are scheduled using the same priority rules as in PD^2. In the case of a tie, urgent subtasks have higher priority, with the MTT identifier used as a tie-break. (This ensures that MTTs achieve the lowest possible spread when nothing in the PD^2 priority rules would prevent it.)

Example (Figure 3.5). These rules are illustrated in Figure 3.5, where a 1-shifted schedule is used. With regular PD^2 (inset (a)), the task set achieves maximum spreads of two and six for MTTs 1 and 2, respectively. Our rules reduce the spread of MTT 2 to two (inset (b)), without changing the spread of MTT 1. This reduction happens because at time 4 in (b), the scheduling of subtask $W_{(1)}$ results in subtask $X_{(1)}$ being favored over all other subtasks by the *Urgent Tasks* rule. Note that the *Early-Release Eligibility* rule only allows $T_{(4)}$ to become early-release eligible at time 5, so that urgent subtask $X_{(1)}$ can be scheduled. Note also that, if the task set included some additional tasks with a deadline of 11 that were eligible at time 5, then the *Priorities* rule would ensure that the urgent subtask was scheduled first, and MTT 2 would still have a spread of two. □

Analytical guarantees. When these rules are applied to PD^2, and each MTT is comprised of tasks that have the same execution cost, we can achieve the following spread guarantees for a task set τ [3].

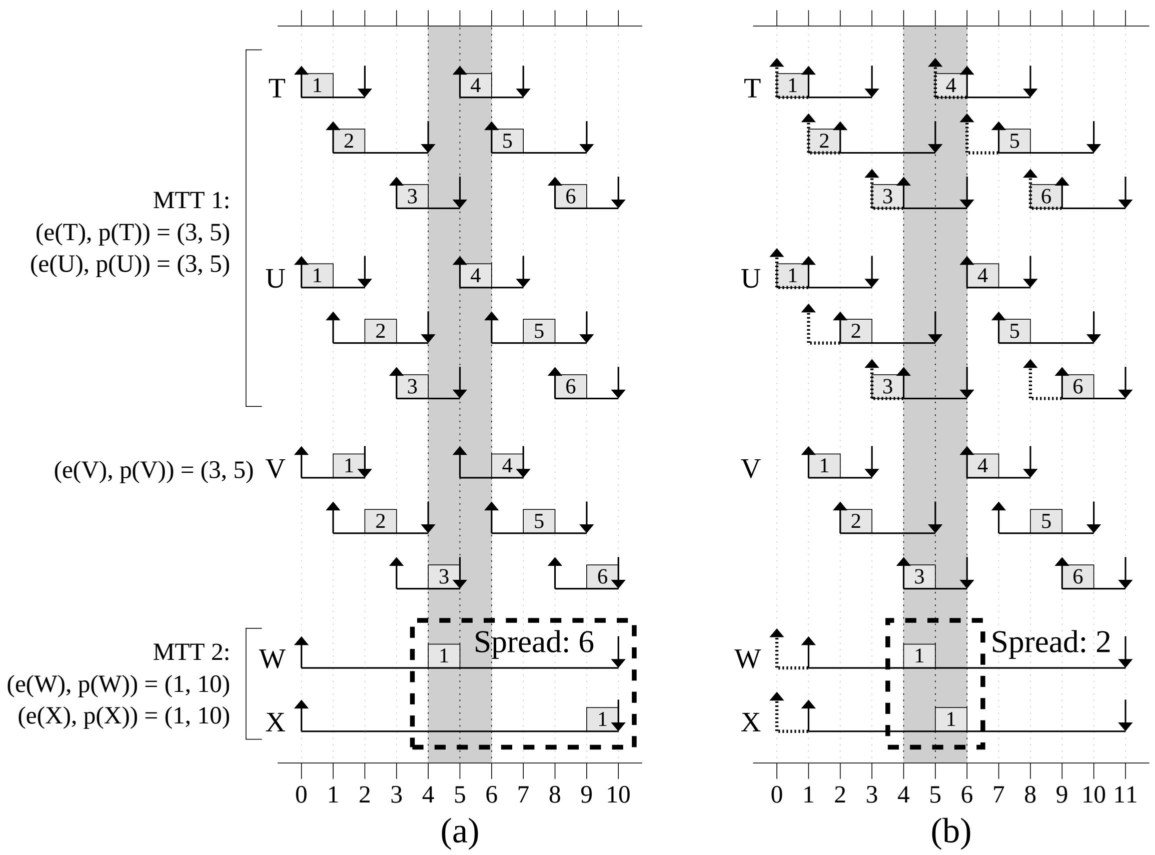


Figure 3.5: An example two-core schedule demonstrating how our scheduling rules reduce spread. Both insets show schedules for the same task set when PD^2 is employed **(a)** without our rules and **(b)** with our rules. The shaded quanta indicate where the two schedules deviate so that spread is reduced in inset (b). In this figure, the number within each scheduled piece of work denotes the corresponding *subtask* index, rather than job index.

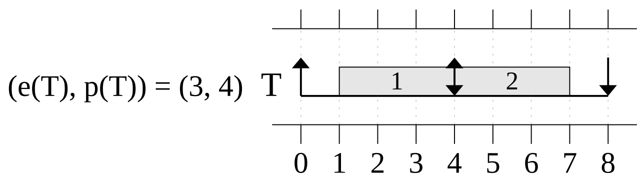


Figure 3.6: An example demonstrating the maximum execution time of a task under GEDF in the absence of deadline tardiness, wherein T_1 begins execution exactly $e(T)$ time units prior to its deadline, T_2 begins execution exactly at its release time, and both jobs are not preempted.

Theorem 3.1. *If* PD^2 *is modified as described above, subtasks are eligible for early-releasing* $X - 1$ *quanta before their actual release times, and all tasks in the same MTT have the same execution cost, then the spread of any MTT is no greater than* X *as defined in* (3.1), *where* $u_{\max} = \max_{T \in \tau} u(T)$.

$$X = \begin{cases} 3, & \textbf{if } u_{\max} \leq 1/3 \\ 4, & \textbf{if } 1/3 < u_{\max} \leq 1/2 \\ 2 \times \left\lceil \frac{1}{1-u_{\max}} \right\rceil - 1, & \textbf{if } u_{\max} > 1/2 \end{cases} \tag{3.1}$$

For GEDF, we can state a similar theorem. In this case, assuming that all tasks have a utilization less than one, a task T can execute for at most $2 \cdot e(T)$ consecutive time units in the absence of tardy jobs, as seen in Figure 3.6. Therefore, the maximum amount of time that *any* task in the task set can execute is $2 \cdot e_{max}$ consecutive time units, where $e_{\max} = \max_{T \in \tau} e(T)$.

Assume that task T is the first task in some MTT $\mathcal{T}$ to schedule its i^{th} unit of computation, and this occurs at time t. As a result, all other tasks $U \in \mathcal{T}$, where $U \neq T$, will be flagged urgent. Given the upper bound on the number of consecutive time units that any task may execute, we argued in [3] that any task that interferes with the scheduling of these urgent tasks must be ineligible for at least one time unit in the interval $[t, t + 2 \cdot e_{max} + 1)$. From this, it can be shown that sufficient processing capacity exists so that all other tasks $U \in \mathcal{T}$ will have their i^{th} units of computation scheduled in the interval $[t, t + 2 \cdot e_{max} + 1)$, resulting in a spread of $2 \cdot e_{max} + 1$. This result is stated formally in the following theorem.

Theorem 3.2. *Consider a task set* τ *for which tardiness is at most* Δ *under* GEDF, *and let* e_{max} *denote the largest job execution cost in* τ. *If* GEDF *is modified as described above for*

Alg.	Util. Constr.	X	ER	Spread: MTT Size = 2 Min	Avg	Max	MTT Size = 3 Min	Avg	Max	MTT Size = 4 Min	Avg	Max
PD^2	$(0,1/3]$	–	0	1	1.35	**41**	1	1.66	**40**	1	1.99	**41**
[3]	$(0,1/3]$	3	2	1	1.27	**2**	1	1.52	**2**	1	1.77	**3**
PD^2	$(0,1/2]$	–	0	1	1.40	**37**	1	1.78	**41**	1	2.18	**37**
[3]	$(0,1/2]$	4	3	1	1.28	**2**	1	1.53	**2**	1	1.77	**3**
PD^2	$(0,3/4]$	–	0	1	1.39	**25**	1	1.83	**33**	1	2.29	**41**
[3]	$(0,3/4]$	7	6	1	1.29	**2**	1	1.57	**2**	1	1.81	**3**

Table 3.3: Spread under PD^2 with and without the method in [3]. Each entry represents 50,000 task sets.

PD^2, *but instead jobs (rather than subtasks) are allowed to become eligible for early-release up to* $2 \cdot e_{max}$ *time units before their actual release times, all tasks in the same MTT have the same execution cost, and* $p(T) \geq e(T) + 1 + \Delta$ *for each* $T \in \tau$*, then the spread of any MTT is at most* $2 \cdot e_{max} + 1$.

Note that, if the method is employed by allowing deadline tardiness instead of shifting the schedule and early-releasing, then any tardiness arising from our method must be added to the tardiness bound for GEDF.

Spread experiments. To assess the effectiveness of our approach, we conducted two experimental evaluations. In the first, we randomly generated 50,000 task sets in several categories, and simulated the scheduling of these task sets on a four-core system. We measured the spread for each MTT both with and without our method under PD^2 scheduling. An upper bound of 1/3, 1/2, or 3/4 was enforced on task utilizations, depending on the experiment. In this and all of the other experiments presented in [3], the utilization of every task in an MTT is assumed to be the same. All simulations were run up to the hyperperiod of each task set. Results are shown in Table 3.3. These experiments show that our rules often have an impact on both average and maximum spreads so that they are closer to one (*i.e.*, perfect parallelism)—in fact, spreads are much lower than might be expected from the analytical spread guarantees presented earlier. Most significantly, note that our method *always prevents extremely high spreads*, as shown in the boldface columns of Table 3.3.

Shared cache miss rates. We also conducted experiments that demonstrate the effectiveness of our method in reducing shared cache miss rates. We first estimated miss ratios for the same 50,000 task sets considered earlier using a simple (hand-coded) cache model. This model assumes a "best-case" scenario where the cache was fully associative. Each task was specified to sequentially reference 10,000 cache lines, or 640K of memory, every quantum. The region of memory referenced was dependent on the subtask—equivalent subtasks of tasks in the same MTT referenced the same unique region of memory. Thus, the only opportunities for cache reuse existed when tasks of the same MTT referenced the same region of memory. We assumed that the amount of data referenced per subtask is the same every quantum regardless of cache miss rates: if a task finishes early, the rest of the quantum is wasted. We further assumed that the shared cache can hold exactly four working sets of data, and employs an LRU replacement policy. Cache lines that could be reused during the current quantum were reused before being replaced (an idealistic assumption). Cache lines that were not reused were eventually replaced per the LRU policy. This admittedly simplistic cache model allowed all tasks to be scheduled up to the task set hyperperiod, as done in the prior experiment. This was not possible with the experiments discussed below, where SESC was used (SESC is very exact, but also quite slow). The results of experiments conducted assuming our simple cache model, shown in Figure 3.7, are quite dramatic. Note that the best achievable cache miss ratio for an MTT $\mathcal{T}$ (with $|\mathcal{T}|$ tasks) is $1/|\mathcal{T}|$, and most MTTs approach this miss ratio with our method.

We next ran more realistic and complex experiments using SESC. SESC executes *actual* task and scheduling code, and therefore scheduling, preemption, and migration costs were accounted for in these simulations. In order to examine the benefits of our method per MTT, the simulator was modified so that each memory reference could be "tagged" with a value indicating the MTT with which it was associated. The simulated architecture consisted of four cores, each with dedicated 16K L1 data (4-way set associative) and instruction (2-way set associative) caches with random and LRU replacement policies, respectively, and a shared 2048K 8-way set associative on-chip L2 cache with an LRU replacement policy. Each cache has a 64-byte line size. The memory reference pattern of all tasks remained the same as in

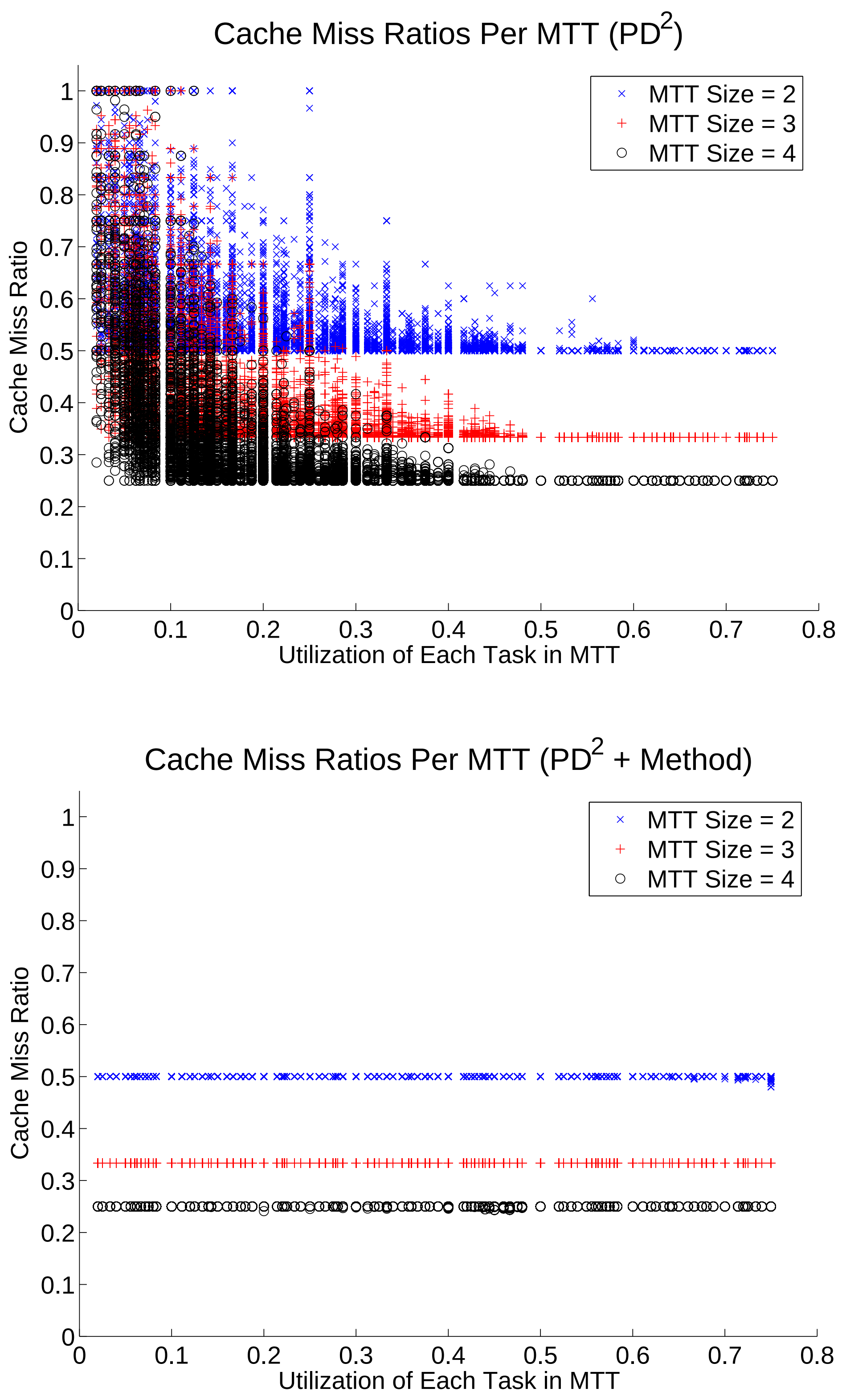
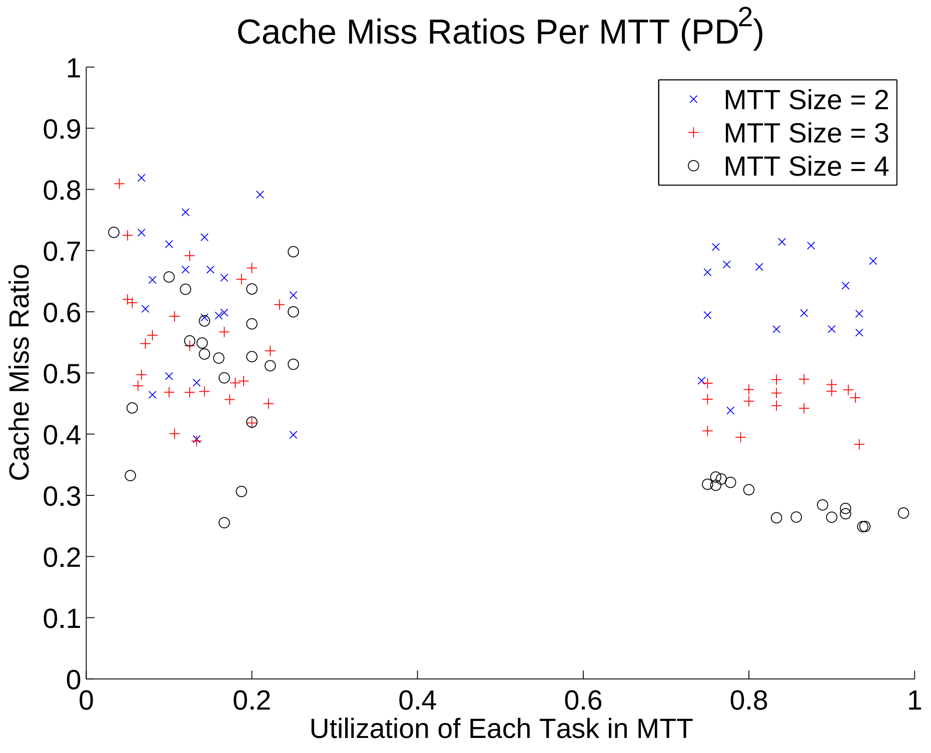


Figure 3.7: Cache miss ratios for PD^2 with and without the method in [3], using a simulated simple cache model. Each scatter plot represents the same 50,000 task sets from Table 3.3 with utilization constraint $(0, 3/4]$.

the simpler experiments, and thus the L2 cache could hold approximately three working sets of data. Additionally, all tasks in the same MTT reference the same memory region, but starting from different locations, wrapping if necessary. This better utilizes the cache and prevents tasks from proceeding in "lock step" while waiting for lines to be loaded into the cache from main memory, resulting in virtually no cache benefit.

In these experiments, we simulated 50 randomly-generated task sets for 20 quanta (instead of up to the hyperperiod) assuming a 0.75-ms quantum. While SESC is very accurate, it comes at the cost of being quite slow. Therefore, longer and more detailed results could not be obtained because of the length of time that it took the simulations to run. In order to demonstrate the substantial impact of our method on MTTs with low-utilization tasks, we required all tasks to either have utilization at most 1/4 or at least 3/4. (This creates opportunities for high-utilization tasks to disrupt the co-scheduling of low-utilization tasks that belong to the same MTT.) Task periods varied from three to 100, and all task sets fully utilized all four cores, with MTTs containing between two and four tasks. In all cases, we early-release by only six quanta—we would need to early release by much more in order to make spread *guarantees*, but we can still see substantial benefits with limited early-releasing. Results are shown in Figure 3.8. These results, while not as dramatic, still demonstrate a significant benefit for lower-utilization tasks. Note the especially large benefit for MTTs with three and four tasks, where cache miss ratios in the range of 50% to 75% decrease to at most 50% with few exceptions. Note also that opportunities for cache reuse are limited by our memory reference pattern, and therefore all miss ratios are quite high. However, our method shows a substantial overall improvement with these task sets.

Limitations. While the method in [3] is effective at reducing spread and shared cache miss rates for MTTs, it encourages co-scheduling on a per-subtask or per-unit-of-execution basis rather than per-job (as in problem CARTCP). As a result, job priorities change considerably during execution, and cache affinity may be lost due to frequent job preemptions or migrations. By using a per-job metric, many jobs may avoid losing cache affinity during execution, resulting in more significant cache miss rate reductions. Additionally, when this method is used in conjunction with GEDF (and most other non-Pfair policies), the spread guarantee is



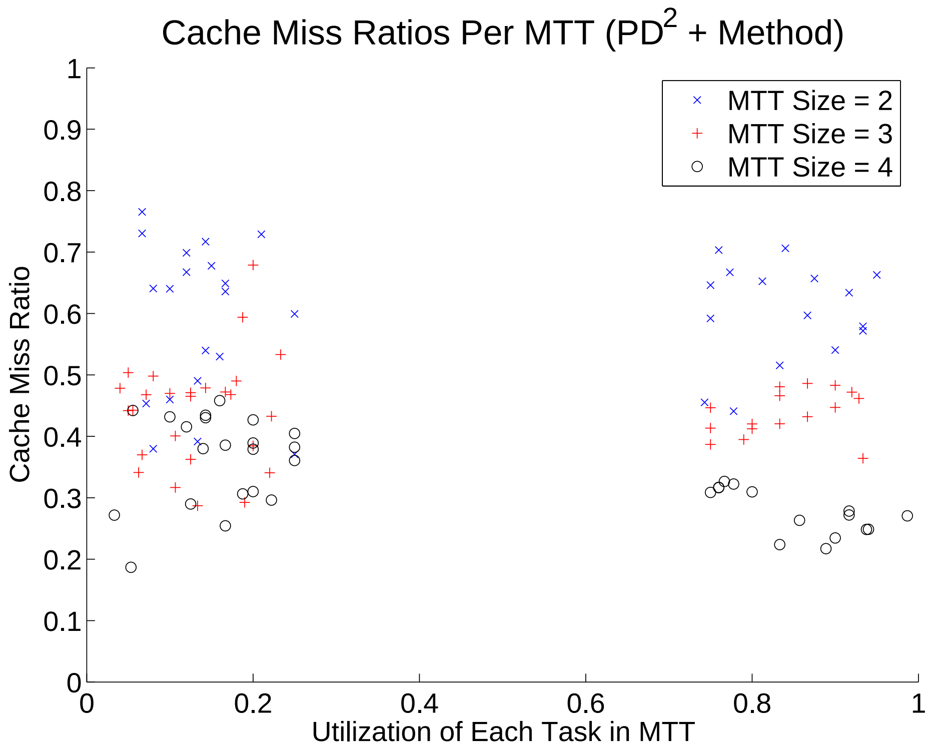


Figure 3.8: Cache miss ratios for PD^2 with and without the method in [3], using the cache model within SESC. Each scatter plot represents the same 50 task sets.

relatively weak, and only applies in specific cases. As a result, this method, like the use of megatasks, is heavily reliant on Pfair scheduling—the schedulability benefits of using GEDF as the baseline policy could be offset by weaker spread guarantees. Optimally, it would be desirable to have both the practical schedulability benefits of GEDF and the co-scheduling benefits that can be achieved when using this method with PD^2.

3.2.3 Problems With Combining These Methods

Unfortunately, the methods described in Sections 3.2.1 and 3.2.2 cannot be combined effectively. This was hinted at when describing the limitations of megatasks, since megatasks limit the parallelism available to MTTs.

More generally, megatasks achieve performance gains by discouraging parallelism (Section 3.2.1), while MTT co-scheduling achieves gains by encouraging parallelism (Section 3.2.2). There is no straightforward way to combine these mechanisms, since encouraging and discouraging are opposite goals. The static restrictions on parallelism that are imposed by megatasks make it very difficult to encourage co-scheduling, even if such co-scheduling would actually contribute towards the goal of reducing cache miss rates and preventing thrashing. Upon further reflection on this point, we observed it is necessary only to discourage co-scheduling of tasks from *different* MTTs—encouraging co-scheduling of tasks within the *same* MTT implicitly satisfies this goal. This observation led to the work in [17], which will be discussed further in Chapter 4.

3.3 Conclusion

In this chapter, we have formally introduced the cache-aware co-scheduling problem that we seek to address in this dissertation, and proven that finding an exact solution to the problem is NP-hard in the strong sense. We also described two earlier attempts to address aspects of the problem and discussed limitations that make them both individually restrictive and impractical to combine. In the remaining chapters, we will present a more complete solution to the problem that also has low enough scheduling overheads to be used in a real operating system in practice.

CHAPTER 4

CACHE-AWARE REAL-TIME SCHEDULING

In this chapter,[1] we present our methods for cache-aware real-time scheduling. These methods influence co-scheduling through the use of a technique, called a *job promotion*, that we introduce to reduce cache miss rates. This chapter is organized as follows. First, we discuss how job promotions can be used to influence co-scheduling, and present an example of how certain co-scheduling choices can affect the cache miss rates experienced by real-time workloads. We then introduce a large number of cache-aware real-time scheduling policies that dictate when and how job promotions are used; these policies are employed within *heuristics*. A large number of heuristics are evaluated in Chapter 6 based on their ability to reduce cache miss rates. Next, we present the tardiness bounds of our method, and explain how buffering can sometimes be used to hide tardiness from an end user. Finally, we present a detailed example of our best-performing heuristic, and discuss implementation concerns that guided our selection of the heuristic to implement within LITMUS$^{\text{RT}}$, and then conclude.

4.1 Influencing Co-Scheduling: Job Promotions

Co-scheduling decisions can have a significant impact on the cache miss rates of real-time tasks. Our scheduler is concerned with reducing cache miss rates for periodic tasks within soft real-time workloads, where tasks are grouped into MTTs.

[1]Contents of this chapter previously appeared in preliminary form in the following papers:
Calandrino, J., and Anderson, J. (2008). Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Proceedings of the 20st Euromicro Conference on Real-Time Systems*, pages 299–308.
Calandrino, J., and Anderson, J. (2009). On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 194–204.

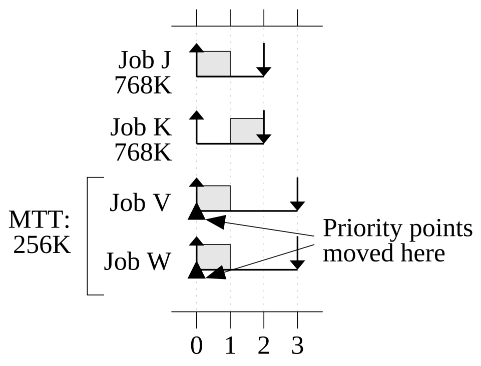


Figure 4.1: An example of where promoting jobs can reduce cache miss rates, assuming the WSSs shown and a cache size of 1 MB.

As stated in Chapter 1, our cache-aware scheduler attempts to reduce cache miss rates by encouraging the co-scheduling of tasks within the same MTT, and discouraging the co-scheduling of tasks within different MTTs when doing so would cause shared cache thrashing. This is achieved through job promotions, wherein a job is given a temporary increase in priority by moving its priority point, originally located at the job deadline, to the current time. 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 accurate WSS information already exists for each MTT. In practice, the profiler described in Chapter 5 is responsible for providing this information.

Example (Figure 4.1). Figure 4.1 presents an example for a three-core platform where influencing co-scheduling in the ways previously stated can be useful. Assuming the use of GEDF scheduling, jobs J and K have the highest priority at time 0, but would thrash the shared cache if co-scheduled, since the sum of the WSSs of jobs J and K is $768 + 768 = 1536$K, which exceeds the shared cache size of 1 MB, or 1024K. We can avoid thrashing by scheduling job V instead of job K. Additionally, since jobs V and W are part of the same MTT, we want to encourage job W to be scheduled when job V is scheduled. To accomplish this, the priorities of jobs V and W need to be increased, which has the potential to negatively impact real-time guarantees later in the schedule. However, if we were to use job promotions, then priority points would remain window-constrained, as defined in Chapter 2; that is, the priority point of each job would remain between its release time and its deadline, and as a result, tardiness would remain bounded as it is under GEDF. Knowing this, we can promote jobs V and W in Figure 4.1 at time 0. Doing so causes V and W to be scheduled next, which should reduce cache miss rates, while still allowing soft 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. $\square$

While promoting jobs can lead to reduced cache miss rates in the near term, it might result in higher cache miss rates, or cache thrashing, 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, which may be problematic. Thus, the choice of when to promote jobs, and which jobs to promote, can have a substantial impact on the effectiveness of our scheduler. Moreover, this choice is not always straightforward. For this reason, we propose (in this chapter) and evaluate (in Chapter 6) a large number of heuristics within this dissertation. Each heuristic represents a set of policies that dictate *when* to promote jobs and *which* jobs to promote. These heuristics, and the policies that they employ, are described next.

4.2 Promotion Heuristics

The heuristics presented in this chapter are used to make scheduling decisions at every quantum boundary in an effort to reduce cache miss rates, based on the WSSs of the MTTs that must be scheduled. Note that our heuristics do not perform any scheduling, monitoring, or other activity *between* quantum boundaries; later in this chapter, we briefly discuss why allowing scheduling between quantum boundaries can be problematic.[2] Scheduling decisions are made iteratively over all cores—even when jobs are promoted, jobs that have already been

[2]Note that our profiler, discussed in Chapter 5, *does* perform monitoring activities between quantum boundaries, which are necessary for generating accurate WSS estimates.

scheduled on some core at the current quantum boundary are unaffected.

4.2.1 Common Rules

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

- **Promoted Jobs.** A promoted job is given a new priority point that is equal to the current time. Tardy jobs are never promoted, as doing so might move the priority point of a tardy job from its deadline to a time after its deadline. As a result, the job would actually experience a priority *decrease*, and priority points would no longer be window-constrained, resulting in potentially unbounded deadline tardiness. Instead of promoting tardy jobs, such jobs are simply scheduled first, in EDF order, after which non-tardy promoted jobs can be scheduled. This allows priority points to remain window-constrained, and therefore tardiness is bounded. Note that this means that a promoted job is highly encouraged, but *not guaranteed*, to be scheduled next. The duration of a job promotion is determined by the *promotion-duration* policy, described in Section 4.2.2.

- **Urgent Jobs.** When a job T_i is scheduled, where task T corresponds to a task within some MTT $\mathcal{T}$, and T is the first task in $\mathcal{T}$ to schedule its i^{th} job at this quantum boundary, each job U_i that has not yet executed to completion, where U is also a task of $\mathcal{T}$ and $U \neq T$, is flagged *urgent* and promoted. Note that this only occurs if T_i itself is not urgent. Regardless of the promotion duration policy that is employed, all urgent jobs remain promoted until at least the time that they are next scheduled, and no non-urgent job (from a different MTT) can be promoted while eligible urgent jobs exist. This rule encourages jobs from the same MTT to be scheduled together, in order to increase the level of shared cache reuse.

- **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 reduce cache miss rates. Additionally, urgent promoted jobs have priority over non-urgent promoted jobs, in the event that both types of jobs exist when making a scheduling decision.

Note that these rules allow us to speak of promoting *MTTs* instead of jobs when presenting the policies in Section 4.2.2. Promoting an MTT means that we choose a single eligible job within that MTT to promote; however, when the i^{th} job of some task in that MTT is promoted, then by the *Urgent Jobs* rule, the (incomplete) i^{th} jobs of all other tasks in 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.

Figure 4.2 presents pseudo-code that describes how scheduling decisions are made by each heuristic. Note that some heuristic-specific 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, and all heuristics encourage the co-scheduling of MTTs based on the rules stated earlier and the *promotion-duration policy* that is employed. Second, all heuristics maintain the current *cache utilization* for the shared cache, which is defined as the sum of the WSSs of all MTTs with jobs scheduled thus far at the current quantum boundary 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%. Third, all heuristics use GEDF scheduling until the cache utilization reaches a *cache utilization threshold*, at which point a *cache-aware policy* is employed to promote jobs. Finally, if cache utilization reaches the *lost-cause threshold*, or a threshold at which we assume that cache thrashing is inevitable during the current quantum, then a *lost-cause policy* is employed to promote jobs. Heuristics might also employ the use of phantom tasks, or avoid the scheduling of partially-eligible MTTs; both of these policies are described in Section 4.2.2.

Example (Figure 4.3). Figure 4.3 presents an example of how cache miss rates can be reduced over GEDF scheduling using our heuristics. The heuristic shown uses a 50% cache utilization threshold, an infinite lost-cause threshold, a cache-aware policy that promotes jobs from the MTT with the smallest WSS, and a promotion-duration policy where, for non-urgent jobs, promotions persist until the next scheduling decision is made (regardless of the job that

```
MakeSchedulingDecisions($numCores$, $cacheSize$)
     ▷ Initialize variable to track sum of MTT WSSs
 1   $usedCache := 0$;
     ▷ Initially, assign each job a priority point equal to its deadline
 2   AssignJobPriorityPointsEqualToDeadlines();
     ▷ Make scheduling decisions by iterating over all cores
 3   for $i := 1$ to $numCores$ do
         ▷ Promote job if applicable
 4       if (No eligible urgent jobs) then
                 ▷ Compute C and N for use by policies
 5               $C := \max(0, cacheSize - usedCache)$;
 6               $N := numCores - i + 1$;
                 ▷ Apply policies as necessary
 7               if ($\frac{usedCache}{cacheSize} \geq$ Lost-cause Threshold) then
 8                   PromoteJobUsingLostCausePolicy(Lost-Cause Policy, $C, N$)
 9               elseif ($\frac{usedCache}{cacheSize} \geq$ Cache Utilization Threshold) then
10                   PromoteJobUsingCacheAwarePolicy(Cache-Aware Policy, $C, N$);
11                   $J$ := Promoted job;
12                   if (Avoid scheduling partially-eligible MTTs $\wedge$
                         MTT of $J$ is partially eligible) then
13                       AdjustPromotedJobToFullyEligibleMTT($J, C, N$)
                     fi;
14                   $J$ := Promoted job (may have changed);
15                   if (Use phantom tasks $\wedge$
                         WSS of $J$ is greater than $C$) then
16                       AttemptToPromotePhantomTasks($J$)
                     fi
                 fi
                 ▷ Else no job is promoted, use GEDF
         fi;
         ▷ Schedule highest-priority job on core $i$, using Priorities rule
17       ScheduleHighestPriorityJob($i$);
18       $J$ := Scheduled job;
19       $mtt$ := MTT of $J$;
20       $jobIndex$ := Job index of $J$;
21       if ($J$ is first job of $mtt$ with job index $jobIndex$ scheduled at this quantum boundary) then
22               $usedCache := usedCache + \mathrm{WSS}(mtt)$
         fi;
         ▷ Set urgent flags and promote/demote jobs by Promoted Jobs and Urgent Jobs rules
23       SetUrgentAndPromotionStatus(Promotion Duration Policy)
     od
```

Figure 4.2: Pseudo-code for all heuristics, invoked at every quantum boundary.

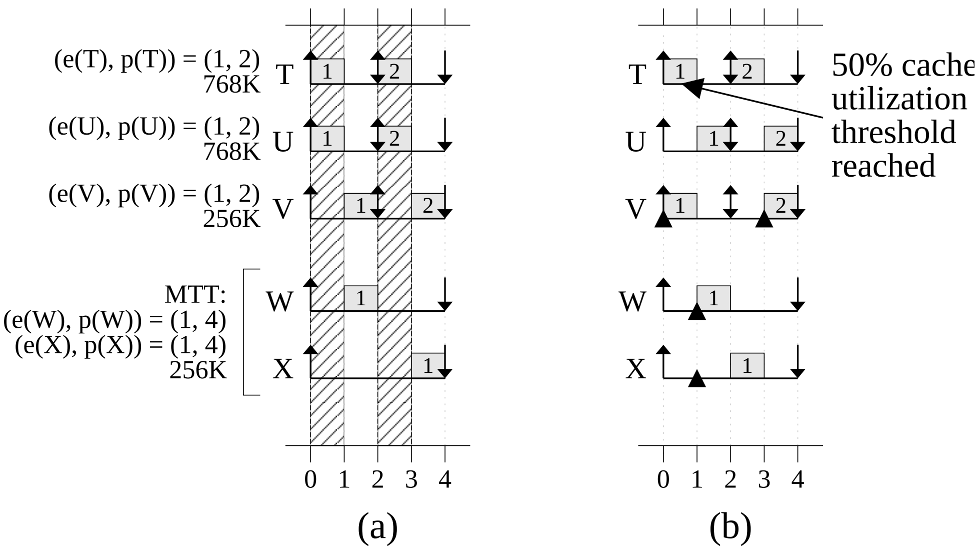


Figure 4.3: Two-core schedules for a set of five tasks. Tasks are scheduled using **(a)** GEDF scheduling and **(b)** one of our heuristics. WSSs are shown along with execution costs and periods. Thrashing occurs during "hatched" quanta, assuming a shared cache size of 1 MB, and black triangles indicate the new priority points of promoted jobs.

is scheduled next), and for any urgent job, promotions persist until that job is scheduled. At time 0, job V_1 is promoted by the cache-aware policy after the 50% cache utilization threshold is reached by scheduling T_1, since V_1 has the smallest WSS (and ties are broken by task identifier), and the *Priorities* rule causes V_1 to be scheduled. At time 1, the threshold is again reached by scheduling U_1, and W_1 is promoted and scheduled; this causes X_1 to be promoted and flagged urgent by the *Urgent Jobs* rule. The *Priorities* rule X_1 to be scheduled at the next available time (at time 2). □

Tardy jobs. A large number of tardy jobs can make it difficult to influence scheduling decisions through job promotions, but this is necessary if any real-time guarantees are to be made. By making appropriate scheduling decisions before such problematic scenarios arise, we can reduce the impact on cache miss rates when we "lose control" of the system in this manner. In the discussion of policies that follows, we return to this issue.

4.2.2 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 for the reasons discussed Section 4.2.1.

Promotion-duration policy. This policy dictates the duration of a job promotion. Two options exist:

1. For non-urgent jobs, promotions persist until the next scheduling decision is made; this means that a job may be promoted and demoted without being scheduled. For urgent jobs, promotions persist until the job is scheduled; that is, the job will *not* be demoted until it is scheduled, at which time the urgent flag for that job is also cleared. The rationale is that non-urgent promoted jobs are promoted to influence the current scheduling decision; in future scheduling decisions, it may make more sense to promote a different job, regardless of whether the currently-promoted job is actually scheduled. However, urgent promoted jobs are promoted so that they are scheduled as close as possible in time (preferably, co-scheduled) with another recently-scheduled job in the same MTT. As such, until an urgent promoted job is scheduled, it must remain promoted so that it is scheduled at the earliest possible time that will not cause timing constraints to be violated.

2. For all jobs, promotions *and urgency* persist until the job completes—the urgent flag for a job is also cleared upon its completion. Thus, in this case, the set of promoted jobs is often identical to the set of urgent jobs. As a result of this policy, only tardy jobs can preempt the execution of promoted jobs or interfere with MTT co-scheduling. In the absence of tardy jobs, promoted jobs execute non-preemptively, which preserves cache affinity and increases cache reuse among jobs belonging to the same MTT.

Note that policy (1) was employed for all heuristics in [17], while policy (2) was employed in [18]—many of the heuristics described in this chapter previously appeared in preliminary form in these papers (the author of this dissertation is a co-author of both papers).

Cache-aware policy. This policy is employed when cache utilization reaches the cache utilization threshold. Once this threshold is reached, we assume that cache utilization is high enough that thrashing is a concern for any remaining scheduling decisions that are made at this quantum boundary. Therefore, cache miss rates are given higher priority than timing constraints (as long as tardy jobs do not exist) when this threshold is reached. This policy is used for the current quantum boundary 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 Figure 4.2). We present five different cache-aware policies. Assume that the WSS of an MTT $\mathcal{T}$ is denoted $\mathit{WSS}(\mathcal{T})$. If all tasks within an MTT have the same execution cost, then $tc(\mathcal{T})$ indicates the task count of the MTT. Otherwise, $tc(\mathcal{T})$ is the number of tasks that *have not completed* job i, where i is the lowest job index such that, for at least one task $T \in \mathcal{T}$, T_i has not completed execution (this definition works for same-execution-cost MTTs as well, but the prior definition is simpler). This definition of $tc(\mathcal{T})$ has an implicit time parameter, as $tc(\mathcal{T})$ must be defined with respect to the time at which we are making scheduling decisions. For all policies, we only promote an MTT if it contains eligible jobs.

1. Promote $\mathcal{T}$ with the smallest $\mathit{WSS}(\mathcal{T})$.
2. Promote $\mathcal{T}$ with the largest $\mathit{WSS}(\mathcal{T})$ that does not exceed C, or exceeds it by the smallest amount if no such MTT exists.
3. Promote $\mathcal{T}$ with the smallest $\mathit{WSS}(\mathcal{T})/tc(\mathcal{T})$ ratio.
4. Promote $\mathcal{T}$ with the largest $\mathit{WSS}(\mathcal{T})/tc(\mathcal{T})$ ratio, where $\mathit{WSS}(\mathcal{T})$ does not exceed C, or exceeds it by the smallest amount if no such MTT exists.
5. Promote $\mathcal{T}$ with the largest $\mathit{WSS}(\mathcal{T})/tc(\mathcal{T})$ 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), as both policies promote the MTT $\mathcal{T}$ where $\mathit{WSS}(\mathcal{T})$ exceeds $C = 0$ by the smallest amount, which defaults to promoting $\mathcal{T}$ with the smallest $\mathit{WSS}(\mathcal{T})$ (even if a $\mathcal{T}$ does exist where $\mathit{WSS}(\mathcal{T}) = 0$, the MTT $\mathcal{T}$ with the smallest $WSS(\mathcal{T})$ is still being promoted). Additionally, when $C = 0$, policy (5) becomes equivalent to (3).

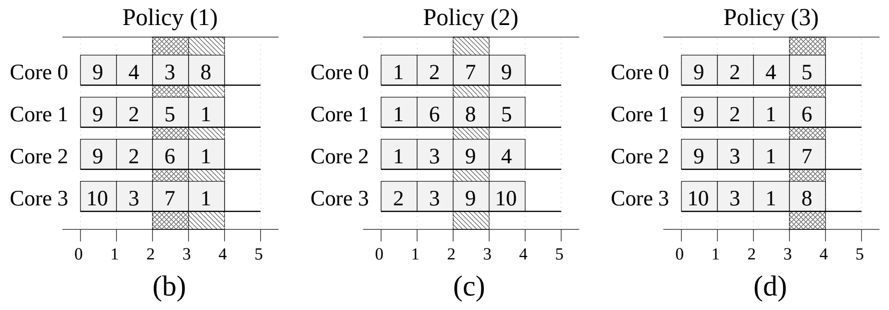
Example (Figure 4.4). Insets (b) through (f) of Figure 4.4 show the differences between policies when scheduling the task set in inset (a). Policies (1) and (3) make locally greedy decisions that should reduce cache miss rates in the near term, but can result in cache thrashing later, since the remaining 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 reduce 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 believe that the definition used by policy (4) is most accurate, as an MTT $\mathcal{T}$ with a large $WSS(\mathcal{T})/tc(\mathcal{T})$ ratio demands a large amount of the cache, yet is easily co-scheduled with jobs from other MTTs. This makes $\mathcal{T}$ quite difficult to schedule when cache miss rates are 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 periodic cache thrashing as was seen in policies (1) and (3).

Note that, while some policies are not very effective in this example, they may be more effective when cache utilization thresholds are higher (a threshold of zero is assumed here), depending on which MTTs are scheduled before the threshold is reached. Moreover, certain cache-aware policies may perform better when used in conjunction with the other types of policies that are discussed in this section. 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 (in Chapter 6) to assist us in making recommendations. $\square$

Lost-cause policy. Each heuristic also employs a lost-cause policy when cache utilization reaches the lost-cause threshold. This threshold is typically greater than 100% cache utilization. Once this threshold is reached, we assume that cache thrashing, or high cache miss rates, are inevitable during the current quantum. We present three policies for when this occurs.

1: [768K, 3]
2: [256K, 2]
3: [256K, 2]
4: [255K, 1]
5: [257K, 1]
6: [512K, 1]
7: [512K, 1]
8: [512K, 1]
9: [64K, 3]
10: [65K, 1]

(a)



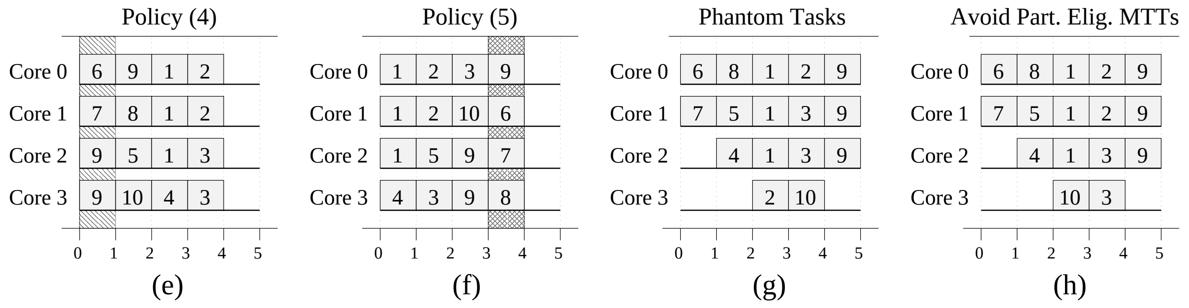


Figure 4.4: Four-core schedules demonstrating a variety of cache-aware policies, assuming a cache utilization threshold of 0%, an infinite lost-cause threshold, and a 1 MB shared cache. Inset (a) shows the task set to be scheduled, consisting of ten MTTs, one specified per line using the format “MTT identifier: [WSS, task count]”. Each task has an execution cost of one and period of five. 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. 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 utilization exceeding 100%—cross-hatching indicates a cache utilization exceeding 150%.

1. Revert to GEDF.
2. Promote $\mathcal{T}$ with the largest $WSS(\mathcal{T})$.
3. Promote $\mathcal{T}$ with the largest $WSS(\mathcal{T})/tc(\mathcal{T})$ 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 miss rates are essentially guaranteed to be high. 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, divided into quantum-length intervals (as each task has an execution cost of one) so that one job of a single phantom task can be scheduled on a core to idle that core during a quantum. A heuristic will promote jobs of phantom tasks only to avoid cache thrashing. Let $\mathcal{T}$ be the MTT that would be promoted if phantom tasks were not used. If $WSS(\mathcal{T}) > C$, and therefore, it is assumed that promoting $\mathcal{T}$ would cause thrashing, then we can promote jobs of the phantom tasks instead if the total number of eligible jobs of all phantom tasks is at least $tc(\mathcal{T})$. Doing so avoids thrashing by reducing parallelism in scenarios where better system performance is achieved by reducing parallelism to avoid cache thrashing, rather than maximizing parallelism at all costs (recall the example in Figure 1.2 presented in Chapter 1). If phantom tasks are employed, then a job of a phantom task must be scheduled whenever a core is idle and at least one such job is eligible, even if no "real" jobs are eligible, as phantom tasks are intended to account for available idle time, whether it is used to prevent thrashing or not.

Example (Figure 4.4(g)). Figure 4.4(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 $\mathcal{T}$ to be *partially eligible* either because fewer than $tc(\mathcal{T})$ jobs are eligible at the time that it is considered for promotion by the cache-aware policy (in which case, at least one tardy job T_{i-1}, where $T \in \mathcal{T}$, is scheduled in the current quantum, so that job T_i is not eligible until the next quantum) or $tc(\mathcal{T})$ exceeds the current value of N. We may want to avoid scheduling such MTTs since their working sets will need to be referenced in at least one future quantum in addition to the quantum for which scheduling decisions are being currently made. If a heuristic chooses to avoid scheduling partially-eligible MTTs, then such MTTs are promoted only when all other eligible MTTs have WSSs that are greater than C. In this case, we would choose to schedule a partially-eligible MTT *before* scheduling phantom tasks, as both choices avoid thrashing in the current quantum, but the former choice also allows some real work to be accomplished and "saves" eligible phantom tasks for times when thrashing cannot be avoided without scheduling them.

Example (Figure 4.4(h)). Figure 4.4(h) shows how avoiding the scheduling of partially-eligible MTTs impacts scheduling when employed along with policy (4) and phantom tasks. At time 2, we avoid scheduling MTT 2 in favor of MTT 10 (when comparing insets (h) and (g), respectively, of Figure 4.4). This 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 (again, as compared to inset (g) of Figure 4.4) decreases by 50%. □

Scheduling between quantum boundaries. We now consider allowing scheduling decisions to be made *between* quantum boundaries—this would most commonly occur when jobs complete between quantum boundaries. By letting $N = 1$ and defining C based on the MTTs executing on the other $M - 1$ cores (assuming an M-core platform), we can use the same heuristics used at each quantum boundary. However, since job priorities change over time due to promotions, jobs scheduled between quantum boundaries may be quickly preempted at the next quantum boundary. For example, if the next quantum boundary is at time t, and a job completes at time $t - \epsilon$, then a job scheduled at time $t - \epsilon$ could be preempted at time t

when its priority changes. If ϵ is not large relative to scheduling overheads, then the utilization gains resulting from scheduling a job at time $t - \epsilon$ may be negligible. Additionally, by scheduling jobs between quantum boundaries, reliably predicting the potential for thrashing in a quantum becomes more difficult. As a result, scheduling between quantum boundaries may decrease overall system performance and offset any utilization gains. Thus, in all of the heuristics considered in this dissertation, scheduling is not performed (and neither is any other activity except profiling) between quantum boundaries.

4.3 Tardiness Bound

As discussed in Chapter 2, our tardiness bound follows directly from the work of Leontyev and Anderson [44], since job priority points are window-constrained for all heuristics. Therefore, the tardiness bound for a task T when using our heuristics, also presented in Chapter 2, is $\frac{\mathcal{E}_z + \sum_{U \in \tau \setminus T} e(U) - e(T)}{M - \mathcal{U}_z} + e(T)$. Note that scheduling phantom tasks will increase the tardiness bound, since such tasks must be treated as "real work" when this bound is calculated. Specifically, for every phantom task that is added to the task set, the tardiness bound must be calculated as if another "real" task P were added, where $e(P) = 1$ and $p(P)$ is equal to the hyperperiod of the real-time workload without phantom tasks. As a result, each phantom task will add one to the summation term when calculating the tardiness bound for any "real" task. However, while it is necessary to include the impact of phantom tasks when computing tardiness bounds for any "real" task, we do *not* need to include the tardiness bounds for the phantom tasks themselves when computing a maximum tardiness bound for the entire task set; since phantom tasks are not real tasks, any "tardiness" experienced by the jobs of these tasks is irrelevant.

4.4 Hiding Tardiness Through Early Releasing and Buffering

We now describe a method that, in some cases, allows tardiness to be "hidden" from an end user. This method employs a combination of early-releasing jobs and buffering results. Specifically, we convert a schedule with a tardiness bound of B into a B-shifted schedule, where all jobs are early-released by B time units, and the scheduled is shifted right by B time units, as described in Chapter 3. Doing so creates a "preprocessing interval" of length B at the beginning of task set execution (assuming a synchronous task set where all tasks release their first job at the same time) during which no job is "officially" released. In the resulting preprocessing interval, work can be completed early (and the results of the work buffered if necessary), so that jobs always finish by their new shifted deadlines; thus, deadlines are never missed. One caveat to this method is that it requires an end user to wait for the duration of the preprocessing interval before any results are seen; however, long wait times upon launching an application are quite standard, and unless tardiness bounds are very large (*e.g.*, ten seconds), such waiting should not be excessive. Alternately, if the tardiness bounds that are analytically guaranteed are not sufficient, but we do not need to ensure hard real-time constraints, then we can choose an "intermediate" solution where a smaller amount of preprocessing is exchanged for an equivalent decrease in our tardiness bound. Finally, we note that this method of hiding tardiness might not be feasible in scenarios where early-releasing jobs is not possible; for example, when a job release is dictated by an externally generated event, so that performing work related to that job before its release time is not possible.

Example (Figure 4.5). Consider the two-core schedules in Figure 4.5, where tasks are scheduled using GEDF in all insets. In inset (a), no shifting or early-releasing is used (0-shifted schedule), and a maximum tardiness of two is observed. In inset (b), the schedule is 2-shifted, resulting in no deadline tardiness and a preprocessing interval of two time units, during which a user must wait, as shown. Finally, inset (c) shows an "intermediate" solution where a 1-shifted schedule is used, resulting in a maximum tardiness of one and a smaller preprocessing interval of one time unit. If necessary, work that is completed before the official release time of a job can be buffered until its release. □

Cache impact of buffering. If buffering is employed, then it will likely have an impact on cache miss rates. For example, consider a job for which 20K of data, representing the result of computations performed during job execution, must be buffered if the job completes before its release time. This 20K of data must be accounted for as part of the WSS of the job (even if

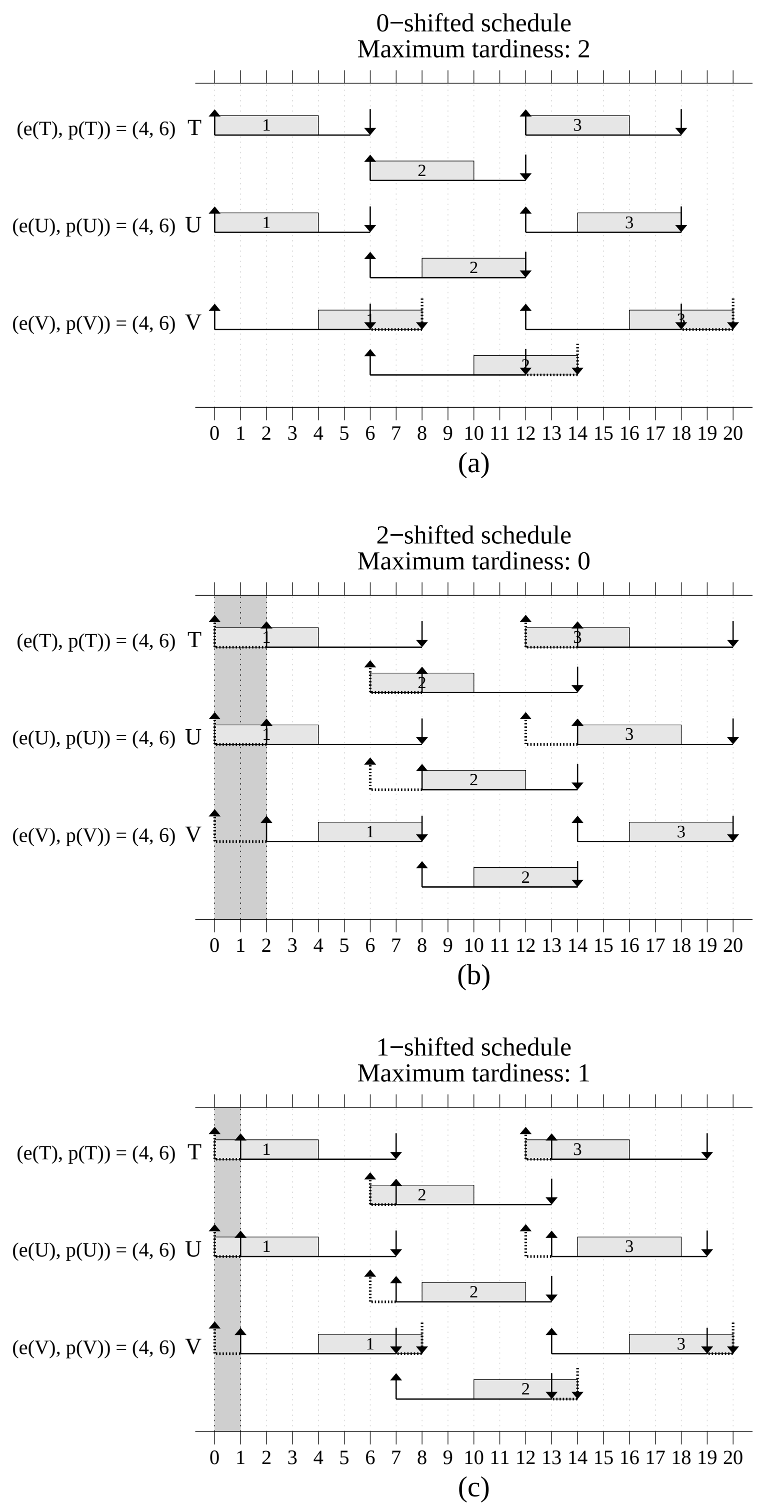


Figure 4.5: Two-core schedules demonstrating how tardiness can be hidden from an end user. All insets use GEDF scheduling. **(a)** No shifting or early-releasing, which results in deadline misses by up to two time units. **(b)** A 2-shifted schedule, which results in no deadline misses. **(c)** A 1-shifted schedule, which results in deadline misses by up to one time unit. In each schedule, the preprocessing interval, during which a user must wait, is shaded.

the buffering activities are not technically part of that job), so that the cache-aware scheduler knows the full cache impact of scheduling a job. Note that this implies that the amount of memory that is required to buffer results should be relatively small compared to the WSS of a job; otherwise, the cache requirements can rise significantly, perhaps offsetting any gains achieved by using a cache-aware heuristic. Alternately, we could prevent memory pages that belong to the buffer from being cached at all, if this is supported by hardware, in which case no additional accounting would be necessary, but the time required to retrieve buffered data could become non-negligible. As a result, a model where a certain portion of every job must be completed after its official release, and the remaining computation can occur before its release, might be appropriate. This type of execution model would be interesting to pursue in future work.

4.5 Implemented Heuristic

The previous sections imply that determining how and when to promote jobs to reduce cache miss rates is not straightforward. As such, we conducted evaluations of a large number of heuristics, representing different sets of scheduling policies, within the SESC architecture simulator, the results of which are shown in Chapter 6. One of these heuristics was found to be particularly effective at improving system performance for a wide variety of task sets. We hereafter refer to this heuristic as the "best-performing" heuristic, and it is this heuristic that we implemented within LITMUS$^{\text{RT}}$ along with the profiler described in Chapter 5.

The best-performing heuristic. The best-performing heuristic, based on the experiments presented in Chapter 6, employs a cache utilization threshold of zero (that is, the cache-aware policy is *always* used to promote and schedule jobs instead of GEDF, except when jobs become tardy), cache-aware policy (1), and phantom tasks. The heuristic does not support *any* lost-cause policy, nor does it avoid scheduling partially-eligible MTTs—both policies were found to be difficult to implement efficiently in practice, so that scheduling overheads remain low, for reasons discussed later in this section. Although we show in experiments in Chapter 6 that not supporting a lost-cause policy can result in *increased* cache miss rates (as

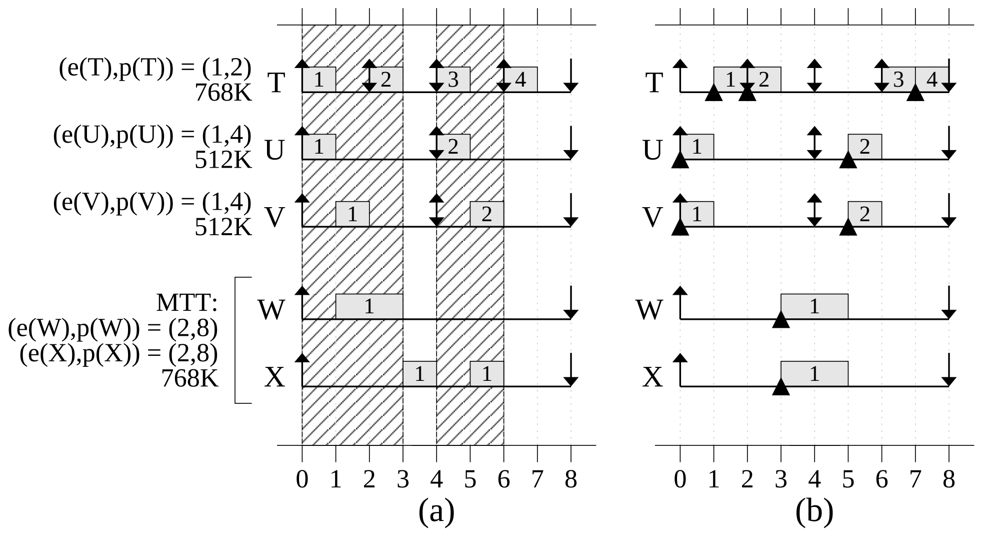


Figure 4.6: Two-core schedules for a set of five tasks. Tasks are scheduled using **(a)** GEDF and **(b)** the best-performing heuristic. WSSs are shown along with task execution costs and periods. Thrashing occurs during "hatched" quanta, assuming a shared cache size of 1 MB.

compared to a heuristic that supports a lost-cause policy), we believe that in practice, the additional complexity required to support such a policy would offset any decrease in miss rate resulting from using the policy. Interestingly, we also found that not avoiding the scheduling of partially-eligible MTTs actually *reduced* cache miss rates.

Our best-performing heuristic also employs promotion-duration policy (2). In Chapter 6, we show that promotion-duration policy (2) results in performance differences as compared to policy (1), some positive and some negative. However, policy (2) allows for a more efficient LITMUS^{RT} implementation, since the tracking of promoted and urgent jobs is more straightforward. As such, we would expect heuristics employing policy (2) to perform slightly better in practice.

For these reasons, we believe that our selection of the best-performing heuristic is justified—we elaborate upon this justification further in Chapter 6. For the convenience of the reader, we next provide a detailed example of how the best-performing heuristic, which we implement within LITMUS^{RT}, makes scheduling decisions.

Example (Figure 4.6). We illustrate the implemented heuristic with an example, shown in Figure 4.6—the schedule under GEDF is shown in inset (a), and the schedule generated by the heuristic is shown in inset (b). 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. As before, black triangles indicate the new priority points of promoted jobs, and a shared cache size of 1 MB is assumed. At time 0, jobs U_1 and V_1, each with a 512K WSS, are promoted (by cache-aware policy (1)) 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, so we idle the second core by scheduling a phantom task. (We assume that exactly four phantom tasks are eligible per hyperperiod, where each hyperperiod is eight time units long. This ensures that a sufficient number of phantom tasks exist to account for all of the available idle time in each hyperperiod.) 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. At time 4, jobs W_1 and X_1 are again scheduled—since promotion-duration policy (2) is employed, these jobs remain promoted until they complete, preventing jobs U_2 and V_2, which have smaller WSSs, from executing immediately following their release. At time 5, jobs U_2 and V_2 are promoted and scheduled, and job T_3 becomes tardy as a result, but is scheduled at time 6. (Note that T_3 is not promoted, but would have higher priority than any promoted job.) Finally, at time 7, the remaining job T_4 is scheduled. We can see from comparing insets (a) and (b) that the heuristic should result in reduced cache miss rates—thrashing occurs 62.5% of the time under GEDF, and is eliminated entirely by the heuristic. □

Implementation efficiency. Efficiency is very important in a real scheduler, so that scheduling overheads do not offset any reduction in shared cache misses. Interestingly, the best-performing 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 (so that they remain promoted until completion) 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, so that the amount of “un-utilized” cache C can be determined (see Section 4.2.2 and Figure 4.2). If the WSS of the job to schedule exceeds C, 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 idle a core without violating timing constraints (using phantom tasks), 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 from Section 4.2 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 Section 4.2, including those that employ *any* of the three lost-cause policies, choose a job to promote based on the current cache utilization or the current value of C, both of which were defined earlier. (Note that this is different from using the value of C to detect thrashing.) Scheduling decisions typically cause the cache utilization and C to change, requiring the promotion queue to be reordered. Alternately, when attempting to avoid the scheduling of partially-eligible MTTs, the number of utilized *cores* will impact job priorities; clearly, the number of utilized cores changes after every scheduling decision, and this will lead to frequent reordering of the promotion queue. The higher overheads associated with such heuristics make them less practical from an implementation standpoint, and therefore disqualify them as candidates for implementation within a real system (*e.g.*, within LITMUS$^{\mathrm{RT}}$).

4.6 Conclusion

In this chapter, we proposed heuristics to reduce shared cache miss rates, and avoid shared cache thrashing, 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 miss rates can significantly decrease and thrashing can be avoided. We also showed that bounded tardiness can be hidden from an end user in scenarios where early-releasing and buffering are possible, as could be the case for certain multimedia applications. Finally, we presented a detailed example of the best-performing heuristic, and discussed issues related to its implementation. As we will show in Chapter 6, cache miss rates often decrease for a wide variety of task sets when these heuristics are effectively used; such cache miss rate decreases can translate into performance improvements for higher-level metrics that are better perceived by an end user.

CHAPTER 5

CACHE PROFILING FOR REAL-TIME TASKS

In this chapter,[1] we describe our online profiler, which quantifies the cache impact of each MTT as a single value, and supplies these values to the cache-aware scheduling heuristics described in Chapter 4 so that scheduling decisions can be made that reduce cache miss rates. This profiler operates during execution, dynamically converging on accurate estimates for each MTT. We begin with an overview and justification of our profiling approach. Next, we state the assumptions required for accurate estimates by this profiler, and then describe the profiler in detail.

5.1 Overview

Our profiler provides a per-job WSS estimate for each MTT. We profile MTTs rather than tasks since for a particular job i, all tasks in the same MTTs reference a common working set.[2] As stated earlier, 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, in that it results in low scheduling overheads.

[1]Contents of this chapter previously appeared in preliminary form in the following paper:
Calandrino, J., and Anderson, J. (2009). On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 194–204.

[2]This does not mean that tasks reference *exactly* the same data, only that the sets of data referenced by all tasks overlap sufficiently to make co-scheduling the jobs of such tasks beneficial. As such, the WSS of the MTT is defined to be the size of the *union* of all of the sets of data referenced by all tasks in the MTT. For example, if tasks T and U belong to the same MTT and independently reference 200K and 300K of data, respectively, and 50K of the data referenced by each task is shared by both tasks, then the WSS of the MTT would be (200K−50K) + (300K−50K) + 50K= 450K.

Event Name	**Event Description**
UnHalted Core Cycles	Core clock cycles when core is not halted.
Instruction Retired	Number of retired instructions.
UnHalted Reference Cycles	Core reference cycles (fixed frequency, does not change when core frequency changes).
LLC Reference	Cache references for lowest-level on-chip cache. If shared, counts only events originating from core.
LLC Misses	Cache misses for lowest-level on-chip cache. If shared, counts only events originating from core.
Branch Instruction Retired	Number of retired branch instructions.
Branch Misses Retired	Number of branch mispredictions.

Table 5.1: The architectural performance events that most Intel processors are capable of monitoring, along with a brief description of each event, from [39].

5.1.1 WSS as a Cache Behavior Metric

WSS may be seen as an overly simplistic predictor of cache behavior; however, as discussed in Chapter 2, the WSS metric tends to work well for small intervals, such as the execution time of a single job of a real-time task, and it is typically the easiest metric (by far) to approximate efficiently given current hardware. Assuming a fully-associative shared cache (or high set-associativity, *i.e.*, eight ways or more—see [37]) so that conflict misses are avoided, our profiler can allow for significant reductions in cache miss rates over GEDF when used within our cache-aware scheduler, as we will see in Chapter 6.

5.1.2 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. In Chapter 6, we describe in detail the three machines on which we implemented and evaluated our profiler in LITMUS$^{\text{RT}}$; these machines contain Intel Core i7, Intel Xeon E5420, and Sun UltraSPARC T1 processors. On the Intel platforms, performance counters are programmed by writing to the *performance event select registers*. Most current and future Intel processors, including ours, are capable of monitoring any of the same seven *architectural performance events*, shown in Table 5.1, in addition to a large number of events that are specific to a particular type of processor [39]. On the Sun UltraSPARC T1 processor, a single counter can be programmed to monitor one of the eight events shown

Event Name	Event Description
SB_full	Number of cycles that a hardware thread is stalled due to a full store buffer.
FP_instr_cnt	Number of completed floating-point instructions (executed by the shared floating point unit).
IC_miss	Number of L1 instruction cache misses.
DC_ miss	Number of L1 data cache misses.
ITLB_miss	Number of instruction TLB misses.
DTLB_miss	Number of data TLB misses.
L2_imiss	Number of L2 misses due to instruction cache requests.
L2_dmiss_ld	Number of L2 misses due to data cache load requests (stores cannot be counted).

Table 5.2: The performance events that can be monitored on the UltraSPARC T1 processor by writing to the performance control register, along with a brief description of each event, from [67].

in Table 5.2 by writing to a *performance control register*; a second counter also exists that always counts the number of completed instructions [67].

Typically, a separate set of performance counters is available for each core, and can be programmed to track events originating from that core. On the UltraSPARC T1, there is actually a set of counters on each core, one for each of the four *hardware threads*, and events originating from a particular thread can be recorded. On the Intel Core i7 chip, a single counter exists for each core, even though there are two hardware threads per core; the counters can be programmed to track events from one or both threads. (In Chapter 6, we avoid issues related to multiple hardware threads sharing a single set of performance counters on the Core i7 machine, among other issues, by disabling hardware multithreading on that machine.)

Regardless of the hardware platform, we programmed a counter at each logical CPU (a single core for our Intel machines, or a single hardware thread for our UltraSPARC T1 machine) 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 performance counters and recording data are low-overhead operations, and computed WSS estimates are cached to reduce computation, the overhead of the profiler is relatively low, as will be shown in Chapter 6.

5.2 Assumptions

The profiler described in this dissertation requires several assumptions.

1. The i^{th} jobs of all tasks in the same MTT reference the same set of data, which is unique to this set of jobs, but of similar size to the sets referenced by other similarly-constructed sets of jobs in the same MTT. Further, each task of the MTT performs roughly the same operations during every job (even though those operations are performed on different 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, share significant data *only* if $i = j$ (even if $T = U$, significant data is not shared if $i \neq j$); 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.*, that of Ramaprasad and Mueller [57]), 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. As was the case in Chapter 4 when describing our heuristics, 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. 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, as one inaccurate measurement can result in inaccurate WSS estimates being generated for that MTT.

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. We believe this assumption to be reasonable, as the size of the lowest-level shared cache in multicore chips is on the order of megabytes and continues to increase (in fact, caches are sometimes so large that they occupy more space on the chip than all of the other processor components combined). Further, assumption (2) ensures that the number of capacity misses observed over non-discarded jobs is negligible—without preemptions or thrashing, data that is brought into the cache by a job should remain in the cache during the entire time that it executes.

5.3 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. We can conclude from our assumptions that the first reference to a particular line of data during the execution of a job should result in a compulsory miss, and future references should result in cache hits, since data brought into the cache should not be evicted during job execution. Thus, the vast majority of shared cache misses that are recorded by our performance counters should be compulsory misses. As such, we can compute the average per-job WSS for an MTT by dividing the total cache misses observed over all profiled jobs by the total number of profiled jobs for the MTT, and multiplying the resulting value 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.) This computation results in an estimate of the cache "footprint" of an MTT, which we use as an approximation of WSS.

Example (Figure 5.1). As an example, consider Figure 5.1, which depicts a two-core GEDF schedule for a task T and an MTT containing two tasks, U and V, as well as the profiling activities that occur. When jobs U_1 and V_1 of the MTT are profiled, cache miss counts of 997 and 1,242 are recorded for each job, respectively, for a total miss count of 2,239 for the first job of the MTT. Next, the data obtained for the second job of the MTT is discarded, since job V_2 is preempted—recall that measurements for the i^{th} job of an MTT (that is, the measurements obtained 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 is preempted. Finally, profiling jobs U_3 and V_3 results in counts of 1,072 and 1,203 being recorded for each job, respectively, for a total miss count of 2,275 for the third job of the MTT, and a total miss count of 4,514 over both profiled jobs. Assuming a 64-byte cache line size, the WSS estimate that our profiler would produce (over the two non-discarded MTT jobs) before the fourth job begins execution is $(4,514/2) \cdot 64 = 144,448$ bytes, or a WSS of roughly 141K. (Note that we divide by two in this computation since there

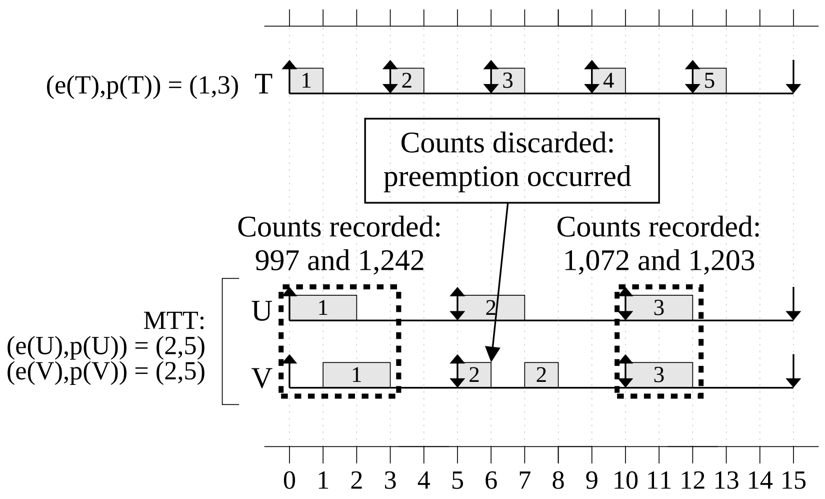


Figure 5.1: Two-core example schedule demonstrating how the cache profiler collects information during job execution. The recorded cache miss counts are shown for each job of the MTT, except the second job (which consists of jobs U_2 and V_2), where measurements are discarded due to the preemption of V_2.

are two non-discarded MTT jobs, *not* because there are two tasks in the MTT.) □

5.3.1 WSS Versus Cache Footprint

Thus far, we have defined WSS in this dissertation to be the size of the per-job cache footprint of an MTT. However, it is possible that many lines that are brought into the shared cache by an MTT, and thus are part of the cache footprint of that MTT, will not be reused. Such cache lines are not truly part of the "real" WSS of the MTT as defined by Denning [26, 27] and Agarwal *et al.* [2] and discussed in Chapter 2, since the absence of these lines in the cache would not have any (negative) impact on any aspect of MTT performance (*e.g.*, job execution times).

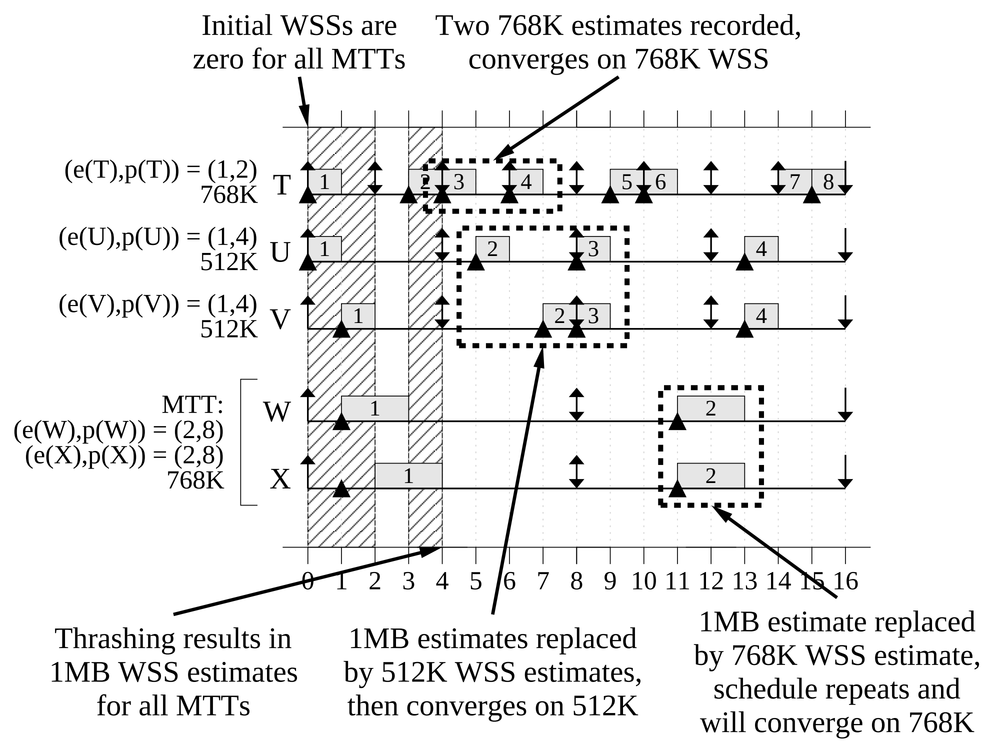
Given this distinction between cache footprint size and "real" WSS, we choose to use cache footprint size as our WSS estimate for two reasons. First, generating a WSS estimate that better approximates the "real" WSS of an MTT would require detailed information about cache reuse. This information could be collected online with additional hardware support, but otherwise is difficult to obtain during execution. Without reuse information, WSS will continue to be overestimated as the size of the cache footprint. Second, when using WSS estimates to avoid shared cache thrashing, the cache footprint of an MTT actually may be *more* useful than a "real" WSS. This is because shared cache interference may occur whenever a line is brought into the cache, since a line that is brought into the cache by one job, regardless of the frequency of reuse of that line, can evict a line of another co-scheduled job that would have otherwise been reused. The cache footprint size (conservatively) accounts for this fact by assuming that *all* cache lines that are brought into the cache by an MTT, and together represent the cache footprint of that MTT, may cause shared cache interference.

5.3.2 Bootstrapping the Profiler

Before any measurements have been obtained for an MTT, it is still necessary to make scheduling decisions and execute jobs. These jobs are then profiled during their execution. At such a time, the heuristics from Chapter 4 do not have sufficient information to make cache-aware scheduling decisions, yet decisions must be made. (This is an issue that we avoided in Figure 5.1, by using the profiler in conjunction with GEDF.) Such a situation poses problems for the profiler as well—before any measurements have been obtained, 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), in particular that profiled jobs do not cause shared cache thrashing.

Example (Figure 5.2). To circumvent these issues, profiling begins with a *bootstrapping* process, illustrated with an example in Figure 5.2 (which uses the same task set and heuristic from Figure 4.6). 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. A job that does not cause thrashing results in an accurate estimate being generated, which is equal to its MTT WSS, as specified in Figure 5.2.

At the beginning of execution, each MTT is assigned a WSS of zero, which is the WSS assigned to an MTT until measurements are recorded for its first profiled job. This approach provides the heuristic with sufficient information to make scheduling decisions (even if those decisions are likely to be poor) so that the profiler can begin collecting information. This policy is reflected in Figure 5.2, where at time 0, all MTTs have a WSS of zero. As a result,



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 5.2: Two-core example schedule generated using the heuristic and task set from Figure 4.6 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.

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. At time 4, all jobs profiled thus far have experienced thrashing, resulting in a 1 MB WSS estimate being generated for every MTT. In this case, *none* of the measurements obtained by the profiler are discarded due to thrashing; since each MTT is assigned a WSS of zero before execution, thrashing is not detected, resulting in miss counts that include a large proportion of capacity misses (in addition to compulsory misses) and overestimated WSSs for every MTT. However, note that these overestimates will result in very conservative scheduling by the heuristic, which ultimately leads to accurate WSSs being generated, as we will see next.

As a result of these overestimated WSSs, MTTs are 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. In light of this expectation, the profiler initially uses measurements from only the most recently profiled job of each MTT when computing its WSS, instead of computing an average WSS over all (non-discarded) profiled jobs in the way described earlier. This approach allows earlier inaccurate measurements to be discarded, rather than being incorporated into a running average WSS for the MTT. Therefore, 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, instead of estimates that are an average of earlier overestimates and the current accurate estimate.

Only once *converging* estimates are detected for an MTT do we begin using an average over all profiled jobs. Convergence is said to occur when the difference between the estimates for two consecutive profiled jobs of an MTT (not including those discarded due to preemptions or thrashing) drops below some threshold. This threshold can be set appropriately depending on when it is considered safe to use average estimates. In our implementation, we set this threshold to 100 cache misses, or 6,400 bytes, since the line size of the shared cache was 64 bytes on all platforms on which we implemented this profiler (these platforms are described in detail in Chapter 6). However, for simplicity in this example, we assume that convergence only occurs after observing two identical consecutive WSS estimates. 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. Thus, from time 7 onward, WSS estimates for task T are computed as averages over all successive profiled jobs. Note that, when the WSS estimates of two consecutive jobs must both be capped at the size of shared cache (*e.g.*, due to thrashing), we do not consider such estimates to have converged even though they are technically identical, since they are only identical because they were capped (and these estimates are unlikely to be accurate in any case)—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 Figure 4.6(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. Note that the last eight time units of the schedule repeat indefinitely; thus, thrashing is avoided from time 4 onwards. Therefore, 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. □

5.3.3 Profiler Pseudo-code

Figure 5.3 presents pseudo-code for our cache profiler. Procedure InitializeProfiler() initializes the profiler variables for a particular MTT before any of its jobs are scheduled. (We assume that a data structure exists for each MTT with the four fields shown in this procedure.) Procedure ProfileAfterJobCompletion() is invoked whenever a job of a task within an MTT completes, where ReadAndResetMissPerfCtr() reads the (per-CPU) performance counter that is counting cache misses for the shared cache, resets the counter to zero, and returns the result of reading the counter to the caller. Note that the correct operation of ReadAndResetMissPerfCtr() depends on each performance counter being correctly programmed at boot time, and a counter being reset to zero whenever a job begins execution (a counter might be non-zero at the beginning of job execution due to non-real-time or background task activity on the same logical CPU). Procedure WasPreempted() returns a boolean value that indicates whether the job passed to it was preempted, and procedure CausedThrashing() returns a boolean value that indicates whether the job passed to it caused thrashing during its execution, where 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, as stated in Section 5.2. ProfileAfterJobCompletion() also includes the bootstrapping process illustrated in Figure 5.2—the Converged() procedure checks whether the two measurements that are passed to it have converged, using the test described earlier. Finally, procedure ComputeWSS() is invoked by the heuristic whenever a WSS estimate is needed for an MTT. This procedure assumes that the shared cache line size can be obtained by calling procedure SharedCacheLineSize().

```
InitializeProfiler($mtt$)
    ▷ Initialize the profiler for an MTT
1   $mtt.JobMisses := 0$;
2   $mtt.TotalMisses := 0$;
3   $mtt.ProfiledJobs := 0$;
4   $mtt.ThrashingOrPreemptionOccurred := 0$

ProfileAfterJobCompletion($mtt$)
     ▷ Read cache miss performance counter, then reset to zero
 1   $counterMisses :=$ ReadAndResetMissPerfCtr();
     ▷ Add misses to running total for $i^{th}$ job of all tasks in MTT (job $i$ of MTT)
 2   $mtt.JobMisses := mtt.JobMisses + counterMisses$;
     ▷ If completed job caused thrashing or was preempted during execution, then set flag
 3   $J :=$ Completed job;
 4   if (WasPreempted($J$) $\vee$ CausedThrashing($J$)) then
 5       $mtt.ThrashingOrPreemptionOccurred := 1$
     fi;
     ▷ If job $i$ of every task in the MTT completed, then perform bookkeeping
 6   if (Job $i$ of every task in MTT completed) then
         ▷ Only keep measurements if thrashing or preemptions did not occur
 7       if ($mtt.ThrashingOrPreemptionOccurred = 0$) then
 8               if ($mtt.ProfiledJobs = 1 \wedge \neg$Converged($mtt.JobMisses$, $mtt.TotalMisses$)) then
                     ▷ No convergence of job measurements yet, replace old miss count with new
 9                   $mtt.TotalMisses := mtt.JobMisses$
10               else
                     ▷ Data is valid and has converged, or we are profiling the first job
11                   $mtt.TotalMisses := mtt.TotalMisses + mtt.JobMisses$;
12                   $mtt.ProfiledJobs := mtt.ProfiledJobs + 1$
                 fi
         fi;
         ▷ Reset job miss count and flag
13       $mtt.JobMisses := 0$;
14       $mtt.ThrashingOrPreemptionOccurred := 0$
     fi

ComputeWSS($mtt$)
    ▷ If no jobs have been profiled, then return a WSS of zero
1   if ($mtt.ProfiledJobs = 0$) then
2       return 0
    fi;
    ▷ Otherwise, return miss count divided by the number of profiled jobs, multiplied by line size
3   $estWSS := (mtt.TotalMisses / mtt.ProfiledJobs) *$ SharedCacheLineSize();
4   return $estWSS$
```

Figure 5.3: Pseudo-code for the cache profiler.

5.4 Conclusion

In this chapter, we have presented the design and implementation of an online automatic cache profiler for real-time tasks that are grouped into MTTs. The profiler was implemented within LITMUS$^{\text{RT}}$, along with the selected heuristic described in Chapter 4, as part of our cache-aware scheduler. We will show in Chapter 6 that this profiler is accurate across a variety of architectures and that, by allowing a real-time scheduler to take this profiling information into consideration, system performance can be substantially improved in practice.

CHAPTER 6

EVALUATION

In this chapter,[1] we present two sets of experiments. First, we present the results of experiments conducted within the SESC architecture simulator. In these experiments, our heuristics were evaluated in detail and a "best-performing" heuristic was identified, based on a combination of raw performance (in terms of cache miss rates and instructions per cycle) and implementation concerns—this is the same best-performing heuristic that is described in Chapter 4. Second, we discuss experiments conducted to evaluate the performance of a LITMUS^{RT}-based implementation of our cache-aware scheduler, which consists of both the best-performing heuristic and the cache profiler, on three multicore machines with substantially different architectures.

6.1 SESC-Based Experiments

In this first set of experiments, we assessed the efficacy of our heuristics in improving both cache and user-perceived performance by conducting experiments using the SESC architecture simulator [58], which is capable of simulating a variety of multicore architectures. The simulated architectures that we considered consist of eight or 32 3-GHz (single-threaded) cores, each with a dedicated 4-way (respectively, 2-way) set associative 16K L1 data (respectively, instruction) cache with a random (respectively, 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

[1]Contents of this chapter previously appeared in preliminary form in the following papers:
Calandrino, J., and Anderson, J. (2008). Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Proceedings of the 20st Euromicro Conference on Real-Time Systems*, pages 299–308.
Calandrino, J., and Anderson, J. (2009). On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 194–204.

"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 eight-core architectures are available today, such as the Sun UltraSPARC T1 processor that was used in the experiments described in Section 6.2, we chose to use SESC for this set of experiments for a number of reasons. First, SESC 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. This allowed us to evaluate and compare heuristics in the absence of operating system "noise" that might make our results a function of the underlying operating system characteristics, rather than a function of the scheduling policies that are employed by each heuristic. Second, SESC allowed us to experiment with systems containing more cores than are commonly available today—for example, our 32-core platform. Experimenting with both an eight-core and 32-core platform, both of which are quite similar in all ways except core count and shared cache size, allowed us to make reasonable comparisons between an architecture that is feasible today and one that may exist several years in the future. This, in turn, allowed us to make an educated prediction as to whether our heuristics will continue to have a performance impact as multicore architectures evolve.

We used CACTI 4.2 [41] to obtain realistic *cache access time* estimates for our simulations. (The cache access time is the time required to access a single line of the shared cache.) CACTI is a tool that provides estimates of cache access times, energy consumption, and chip area, when given as inputs certain cache attributes such as size and associativity. The estimates provided by CACTI are based on a detailed analytical cache model that has been calibrated, in part, by empirical data from real processors. These CACTI estimates were used in place of the default cache access times in the SESC configuration files. Since the default times in the SESC configuration files assume the default system attributes, such as a 512K shared cache size, the use of CACTI improved the accuracy of our cache modeling, and the overall accuracy of our simulations.

We now describe experiments involving example task sets, randomly-generated task sets, and a multithreaded video encoding application workload. In each set of experiments, GEDF

Figure	**Miss Rate**	**Per-MTT Improvement**
Figure 4.3(a)	23.99%	[0, 0, 0]%
Figure 4.3(b)	15.41%	[11.02, **41.69**, **71.30**]%
Figure 4.4(b)	9.07%	[0, 0, 0]%
Figure 4.4(c)	7.90%	[-50.05, 21.26, 155.20]%
Figure 4.4(d)	8.70%	[-64.93, 0.54, 76.97]%
Figure 4.4(e)	7.94%	[-16.88, 15.41, 71.39]%
Figure 4.4(f)	8.94%	[-64.22, 0.76, 68.39]%
Figure 4.4(g)	6.60%	[-13.82, **37.45**, **166.12**]%
Figure 4.4(h)	6.71%	[-13.51, **36.58**, **166.30**]%

Table 6.1: Shared cache performance for example task sets.

scheduling was compared to some subset of our heuristics. All task sets were scheduled and run for 20 (simulated) milliseconds (or 20 quanta, as the size of a quantum was 1 ms). Previous experimental studies using SESC have indicated that this is long enough to observe performance differences [4]—we ran a subset of our randomly-generated task sets for 100 ms to confirm this for our experimental setup.

Each task in an MTT references the same memory region—the size of this region is equal to the WSS of the MTT. In Sections 6.1.1 and 6.1.2, tasks reference data in memory sequentially, looping back to the beginning of the region when the end is reached, for their entire specified execution time; thus, tasks are backlogged. The data memory reference pattern for video encoding MTTs considered in Section 6.1.3 is more complicated: each task references its assigned video frame slice, plus some "nearby" slices, and the memory region referenced changes with every job. We accounted for scheduling, preemption, and migration costs in all simulations.

6.1.1 Example Task Sets

To demonstrate the performance impact of our heuristics, we first present results for some example schedules from Figures 4.3 and 4.4 (found in Chapter 4). The results are categorized by figure and presented in Table 6.1. Each task set was run using the heuristic, shared cache size, and core count indicated in its respective figure and inset—thus, in these experiments, the core count and shared (in this case, L2) cache size deviated from the simulated architectures described at the beginning of Section 6.1. The "Per-MTT Improvement" column presents the minimum, average, and maximum percentage increase in the number of per-quantum memory references per MTT, relative to the first schedule depicted in each figure (inset (a) in Figure 4.3 and inset (b) in Figure 4.4). For example, the schedule in Figure 4.4(c) resulted in a 21.26% increase in per-quantum memory references on average for each MTT when compared to the schedule in Figure 4.4(b); for one MTT, per-quantum memory references increased by 155.20%, but for another MTT, per-quantum memory references decreased by 50.05%.

These results show that, when suitable heuristics are employed, performance can improve substantially (see the bold entries in Table 6.1). For example, the *overall* shared cache miss rate decreased by over one third as compared with GEDF for the task set in Figure 4.3 when the specified heuristic was used, and MTTs were able to perform an average of over 41% more memory references. For the schedules in Figure 4.4, we see that different cache policies can influence performance in very significant ways. The heuristic that performs best in this case resulted in a 37% increase in memory references on average, and a 166% increase in the best case, compared to the heuristic that achieved the lowest number of per-quantum memory references (the schedule associated with Figure 4.4(b)). Note that heuristics that perform better on average can still result in worse performance for some MTTs, resulting in negative minimum per-MTT improvement values; however, these values are often small when compared to the average- and best-case increases in per-quantum memory references when better-performing heuristics are used.

Of course, these experiments are not conclusive—as stated in Chapter 4, the heuristics that perform well in these examples will not necessarily be the heuristics that perform best across a wide variety of task sets. The experiments that follow demonstrate the broader applicability of our heuristics. However, an implication of these results, particularly those related to Figure 4.4, is that a relative decrease in cache miss rate often results in at least a corresponding relative increase in memory references in the average case *even if the absolute decrease in cache miss rate is small.*

6.1.2 Randomly-Generated Task Sets

In the next set of experiments, we evaluated many heuristics, representing different combinations of policies, on task sets representing a variety of system utilizations, task utilizations, and WSS distributions. Experiments were initially conducted on the eight-core architecture—the heuristic that was found to be particularly effective at improving system performance for a wide variety of task sets was then evaluated on the 32-core architecture. We considered heuristics that employed the following thresholds and policies.

- **Promotion-duration policy:** (1).
- **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.

Note that, in these experiments, we sometimes chose to consider only one threshold or policy choice. This was done when we did not expect the choice to have a significant impact on the performance of the heuristics, especially for the purpose of making *relative* performance comparisons between various policies. This greatly reduced the number of policy combinations that needed to be evaluated; considering additional policy variations would have made the required number of experiments prohibitive. Later, in Section 6.1.4, we consider making changes to the heuristics that performed well in these experiments for the purpose of implementation efficiency, so that scheduling overheads are low. The changes considered there were evaluated by conducting experiments with additional threshold or policy choices beyond those considered here, particularly in categories where only one choice was considered.

Task-set generation methodology. When generating random task sets, we varied the following parameters. In this chapter, the *utilization* of an MTT indicates the utilization of every task within that MTT—the execution cost and period of an MTT are defined similarly. Note that this means that, in our experiments, all tasks within an MTT have the same utilization, execution cost, and period.

- **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 utilizations, and at least 1 ms.
- **MTT task counts:** Uniform over [1, 8].
- **MTT WSSs**: Uniform over [64 bytes, 2 MB]; or equal to the task count multiplied by a size uniform over [64 bytes, 512K], and capped at 2 MB.

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 and processing power that is required for each experimental run, running additional experiments is problematic (which is why, as stated earlier, considering additional policy combinations would have required a prohibitive number of experiments). Even with the assistance of a large research cluster, we were able to complete only a few thousand experimental runs per day in the best case (*i.e.*, when there is little contention for the cluster, which is shared across campus). Like many architecture simulators, SESC is quite slow, especially when timing accuracy is required.

Task set justification. We believe that our task periods represent a reasonable range of those observed in real applications, and our task utilization ranges are similar to those used in other work [8, 16, 22]. System utilizations were chosen so that scheduling flexibility was either substantial (at 50%) or very limited (at 100%). For half of the experiments, MTT WSSs were

Task Set Parameters			**Heuristic**				**L2 Miss Rate**			**IPC**		
%S	**MTT U**	**WD**	**T**	**CP**	**LP**	**PT**	GEDF	**H**	**%Im**	GEDF	**H**	**%Im**
50	[0.01, 0.1]	TC	0	(1)	(1)	used	3.62	1.60	**55.88**	0.97	1.23	**26.46**
50	[0.01, 0.1]	Uni	0	(1)	(3)	used	7.14	3.16	**55.76**	0.80	1.17	**44.90**
50	[0.1, 0.4]	TC	0	(1)	(1)	used	1.22	0.36	**70.62**	1.21	1.20	-1.07
50	[0.1, 0.4]	Uni	0	(3)	(1)	used	6.70	0.67	**90.00**	0.93	1.17	**25.19**
50	[0.5, 0.9]	TC	0	(1)	(1)	used	1.07	0.28	**73.67**	1.03	1.01	-2.28
50	[0.5, 0.9]	Uni	0	(3)	(1)	used	15.38	0.98	**93.61**	0.77	0.92	**18.99**
50	[0.01, 0.9]	TC	0	(3)	(1)	used	3.61	0.63	**82.68**	1.01	1.12	10.77
50	[0.01, 0.9]	Uni	0	(1)	(1)	used	7.92	0.78	**90.12**	0.97	0.95	-2.15
100	[0.01, 0.1]	TC	0	(3)	(1)	N/A	5.30	1.67	**68.55**	0.85	1.16	**36.96**
100	[0.01, 0.1]	Uni	0	(3)	(2)	N/A	7.22	2.57	**64.38**	0.76	1.11	**45.26**
100	[0.1, 0.4]	TC	0	(3)	(2)	N/A	3.75	1.35	**64.00**	0.96	1.18	**22.44**
100	[0.1, 0.4]	Uni	0	(3)	(3)	N/A	7.02	3.46	**50.71**	0.89	1.14	**28.20**
100	[0.5, 0.9]	TC	0	(1)	(3)	N/A	3.81	2.83	25.66	1.05	1.13	7.20
100	[0.5, 0.9]	Uni	50	(1)	(1)	N/A	5.03	3.58	28.93	0.99	1.06	6.28
100	[0.01, 0.9]	TC	0	(1)	(1)	N/A	2.49	0.88	**64.56**	1.09	1.23	13.29
100	[0.01, 0.9]	Uni	50	(1)	(1)	N/A	4.30	3.70	14.04	0.99	1.05	6.26

Table 6.2: The heuristics that performed best for random task sets. When specifying task set parameters, the columns labeled "%S", "MTT U", and "WD" correspond to percent system utilization, MTT utilization distribution, and WSS distribution, respectively. For WSS distributions, "Uni" means uniformly distributed and "TC" means correlated by task count. For the policies used by the heuristics, the columns "T", "CP", "LP", and "PT" stand for cache utilization threshold, cache policy, lost-cause policy, and phantom tasks, respectively. Finally, when presenting L2 miss rates and instructions per cycle (IPC), the column labeled "H" presents performance numbers for the heuristic indicated, and the column labeled "%Im" presents the relative percentage improvement in miss rate or IPC over GEDF.

correlated with task count. This seems realistic, since a larger number of tasks would be more capable of referencing and processing a larger memory region. WSSs were often large, but never exceeded the size of the L2 cache—otherwise, thrashing would be inevitable. Large WSSs are realistic in practice; for example, the authors of [25] claim that the WSS for an HDTV-quality MPEG decoding task could be as high as 4.1 MB, and statistics presented in [70] show that substantial memory usage is required for video-on-demand applications. Finally, note that while these experiments certainly should not be considered definitive, similar task sets have been used effectively in other published work [3, 4, 16, 22].

Results. Table 6.2 presents average cache miss rates and average *per-core* instructions per cycle (IPC)[2] for both GEDF and the heuristic that exhibited the best performance in terms

[2]In comparing this data to that presented in Section 6.1.1, note that IPC is often correlated with the number of memory references performed.

Algorithm	**Average**	**Maximum**
GEDF	0.216	474
Heuristics	1.843	572
Best heuristic only	3.711	493

Table 6.3: Tardiness for GEDF and our heuristics (in quanta).

of these two metrics, as indicated. We can make several observations from this data. First, in almost all cases, the heuristic that performed best for a particular combination of task-set generation parameters outperformed GEDF, often by a substantial margin (see the bold entries in Table 6.2). Second, heuristics that use cache-aware policies (1) or (3) performed best; however, as we will see in Section 6.1.4, when policy (3) performed better than policy (1), it was often by a negligible margin. Third, the use of phantom tasks was clearly beneficial, as every heuristic in the table employs their use when applicable; in fact, we believe that performance improvements tended to be larger at 50% system utilization solely because phantom tasks could be effectively employed. Fourth, the heuristics that performed best almost unanimously employed a cache utilization threshold of 0% and lost-cause policy (1), though lost-cause policies (2) and (3) sometimes performed best at 100% system utilization. This is probably because, at 100% system utilization, phantom tasks cannot be employed, and lost-cause policies (2) and (3) present another way of reducing the impact of MTTs that have the greatest potential to cause thrashing. Overall, we conclude that the heuristic that performed best for the widest variety of task sets employed a cache utilization threshold of 0%, cache-aware policy (1), lost-cause policy (1), and phantom tasks.

Deadline tardiness. We next tabulated average and maximum observed deadline tardiness. These results are shown in Table 6.3. In this case, we ran each task set for 2,000 quanta rather than 20 quanta. Tardiness is higher with our heuristics than with GEDF, but average tardiness is reasonable, and maximum tardiness is comparable to GEDF 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 our heuristics can be employed. Further, if tardiness is undesirable, then a combination of early-releasing

Task Set Parameters			**L2 Miss Rate**			**IPC**		
%S	**MTT U**	**WD**	GEDF	**H**	**%Im**	GEDF	**H**	**%Im**
50	[0.01, 0.1]	TC	3.10	3.30	-6.65	0.90	1.29	**43.72**
50	[0.01, 0.1]	Uni	4.72	5.02	-6.41	0.78	1.24	**58.87**
50	[0.1, 0.4]	TC	0.66	0.59	11.76	1.25	1.31	4.67
50	[0.1, 0.4]	Uni	1.40	0.94	**32.54**	1.15	1.30	12.63
50	[0.5, 0.9]	TC	0.22	0.23	-2.68	1.51	1.52	0.77
50	[0.5, 0.9]	Uni	0.51	0.37	27.75	1.50	1.52	1.84
50	[0.01, 0.9]	TC	0.35	0.33	7.77	1.41	1.44	1.72
50	[0.01, 0.9]	Uni	0.75	0.40	**47.28**	1.34	1.42	6.40
100	[0.01, 0.1]	TC	6.01	3.22	**46.37**	0.92	1.96	**113.47**
100	[0.01, 0.1]	Uni	6.62	6.04	8.84	0.91	1.87	**106.73**
100	[0.1, 0.4]	TC	1.19	0.60	**49.85**	1.13	1.41	**25.10**
100	[0.1, 0.4]	Uni	2.40	0.98	**59.09**	0.96	1.34	**39.18**
100	[0.5, 0.9]	TC	0.64	0.49	23.15	1.20	1.29	7.70
100	[0.5, 0.9]	Uni	1.69	0.79	**53.29**	1.08	1.26	**15.88**
100	[0.01, 0.9]	TC	0.75	0.60	20.03	1.18	1.31	10.31
100	[0.01, 0.9]	Uni	1.14	0.78	**31.78**	1.13	1.27	12.96

Table 6.4: Evaluation of one of our heuristics for the 32-core architecture. This heuristic employs a cache utilization threshold of 0%, cache-aware policy (1), lost-cause policy (1), and phantom tasks. The meaning of each column in the table is identical to its meaning in Table 6.2.

and buffering can be employed to "hide" tardiness from an end user, as described in Chapter 4.

32-core architecture evaluation. We next ran similar experiments for the 32-core architecture, where task sets were generated identically to those for the eight-core architecture (*i.e.*, same parameters, but many more tasks per task set, since the platform is considerably larger). Task sets were scheduled using GEDF and the heuristic that performed best over the widest variety of task sets in the eight-core experiments (cache utilization threshold of 0%, cache-aware policy (1), lost-cause policy (1), and phantom tasks). The results in Table 6.4 are similar to the eight-core results in Table 6.2, with the heuristic outperforming GEDF.

Interestingly, on the 32-core architecture, there were several instances where cache miss rates *increased* slightly when our heuristic was used; however, per-core IPC was always higher under our heuristic (as done earlier, entries representing large improvements are in bold in Table 6.4). In one of the cases where the cache miss rate increased (the fifth entry in Table 6.4), the IPC increase is somewhat small; therefore, we assume that performance differences between GEDF and our heuristic were not substantial. In the two other cases where the cache miss rate increased (the first and second entries in Table 6.4), *substantial* IPC increases were observed—these also happen to be cases where task utilizations are low, and MTT task counts are more likely to be high as a result. In these cases, the results may have less to do with cache miss rates (especially if thrashing was avoided under both GEDF and our heuristic), and more to do with memory bandwidth, and perhaps even contention for accessing lines of the shared cache itself. In this case, if MTTs are being co-scheduled more often when our heuristic is used, then there is more data being shared, and a smaller total set of data being referenced, at any point in time. This reduced pressure on the *entire* memory subsystem may, in turn, result in IPC improvements even if cache miss rates are relatively the same. Further, reducing pressure on the memory subsystem would be much more likely to have a noticeable impact when the number of cores in the system quadruples. In summary, these results give us reason to believe that the tested heuristic will continue to perform well as the core counts of multicore architectures increase.

6.1.3 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. Motion estimation is the most compute- and memory-intensive portion of MPEG video encoding. We emulated motion estimation in our experiments by mimicking its potential memory reference pattern. As discussed in Chapter 1, as the core counts of multicore platforms increase, and the processing power of individual cores remains similar (or even decreases), most compute-intensive applications such as video encoding will need to become multithreaded to continue to achieve performance gains. Such performance gains are mandatory if the video quality demanded by users continues to increase.

We mimicked a potential memory reference pattern of multithreaded motion estimation by splitting each video frame into identically-sized horizontal slices, each of which is processed by a different task of the same MTT. All motion estimation requires a search—for each task in an MTT, 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. In our experiments, tasks reference the memory region of their assigned slice first, and then search

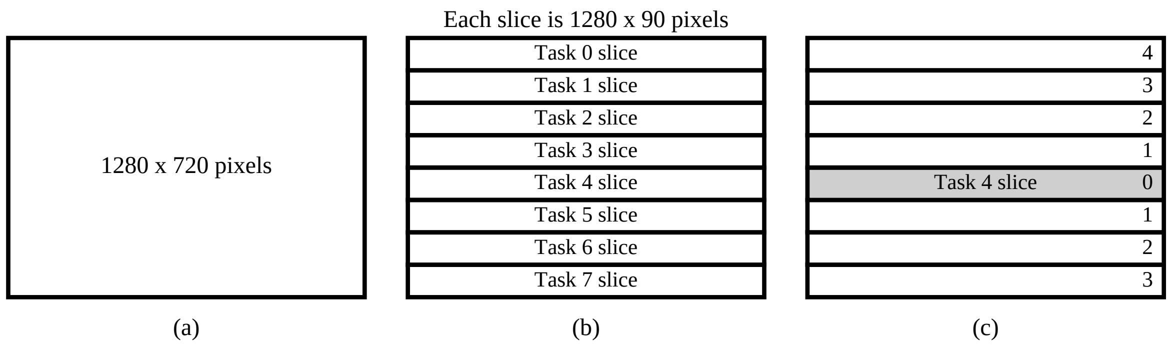


Figure 6.1: The memory reference pattern for multithreaded motion estimation. The insets show **(a)** a 720p HDTV video frame; **(b)** the same frame divided into eight slices, each processed by a different task of the same MTT; and **(c)** the search pattern for the task that is processing the fifth slice (the numbers on the right-hand side of each slice indicate the order in which the slices would be incorporated into the search).

progressively more distant slices. It is often desirable to search the largest space possible (to approximate an exhaustive search), so we assumed that tasks were backlogged in that they continue the search until either their execution time or search space is exhausted. The memory regions that are referenced by each task in an MTT overlap more as the size of the region searched per task increases.

Example (Figure 6.1). As an example, consider Figure 6.1, which concerns a 720p HDTV video, containing frames of 1280 x 720 pixels (900K per frame assuming one byte per pixel), as shown in inset (a). This video could be divided into eight slices of size 1280 x 90 (112.5K per frame), as shown in inset (b), each of which is processed by a different task in the same MTT. The search pattern for the task that is processing the fifth slice, assuming eight slices and a corresponding eight-task MTT, is shown in inset (c). In inset (c), the numbers on the right-hand side of each slice indicate the order in which the slices would be incorporated into the search. □

Both GEDF scheduling and the heuristic that performed best in Section 6.1.2 (cache utilization threshold of 0%, cache-aware policy (1), lost-cause policy (1), and phantom tasks) were used to schedule task sets on the eight-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 of which are more demanding than others. Table 6.5 presents these levels and their corresponding MTTs, along with a use

Level	Resolution	FPS	WSS	Task Count	Period	Use in Practice
1	1920 x 1080	30	2025K	8	33	1080p HDTV (high-quality), moderate frame rate
2	1920 x 1080	30	2025K	5	33	1080p HDTV (high-quality), moderate frame rate
3	1280 x 720	60	900K	8	16	720p HDTV (mid-quality), high frame rate
4	1280 x 720	60	900K	4	16	720p HDTV (mid-quality), high frame rate
5	720 x 480	30	338K	1	33	standard TV and DVD
6	352 x 288	30	99K	1	33	video conferencing
7	320 x 240	24	75K	1	41	high-end portable devices
8	176 x 144	15	25K	1	66	low-end portable devices

Table 6.5: Video quality levels and their corresponding MTTs. The column labeled "FPS" indicates the frames-per-second of the video, and the last column provides one use for each video quality level in practice. All tasks have an execution cost of one—jobs are expected to be backlogged.

for each video quality level in practice.[3] Note that the only difference between levels 1 and 2, and levels 3 and 4, is the number of tasks in the MTT that processes each video frame.

Video encoding task sets were randomly generated according to the following methodology. System utilization was either 50% or 100%, and the 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 system utilization and video quality level.

All results are shown in Table 6.6. In almost all cases, the tested heuristic outperformed GEDF, resulting in an average 10.65% increase in IPC over GEDF in all experiments. Note that an increase in IPC can allow for a proportionate increase in the number of videos or clients 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.

6.1.4 Implementation Concerns

In Chapter 4, we noted that **(i)** avoiding the scheduling of partially-eligible MTTs, **(ii)** employing lost-cause policies, and **(iii)** using promotion-duration policy (1) are all problematic

[3]This information is readily available on the Internet—one good resource for such information is the "list of common resolutions" page at Wikipedia.

Parameters		**L2 Miss Rate**			**IPC**		
System Util.	**Video Quality**	GEDF	**Heur.**	**% Impr.**	GEDF	**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 6.6: Results for video-encoding MTTs (best results in bold). As in Table 6.2, both the actual performance numbers for the tested heuristic and relative percentage improvements over GEDF are presented for each combination of parameters.

when implementation efficiency is a concern; that is, when we want to keep scheduling overheads low. Thus, we do not use these policies in the heuristic that is implemented within LITMUS$^{\text{RT}}$, even though these policies were employed in *all* of the heuristics that were evaluated in Section 6.1.2. The experiments presented in this section were conducted to determine the impact of that decision on cache miss rates and IPC. In these experiments, we took the heuristic that performed best in Section 6.1.2 for each combination of task-set generation parameters (sixteen heuristics in total) and modified it so that three new heuristics were created (48 heuristics in total). We henceforth refer to the set of heuristics that performed best for each combination of task-set generation parameters, before any modifications, as the *original* heuristics. Table 6.7 presents the cache miss rate and IPC for the new heuristics as compared to both each original heuristic (presented in the column labeled "H" in Table 6.7) and GEDF. For each combination of task-set generation parameters, the three new heuristics were created from the corresponding original heuristic as follows.

1. Each original heuristic was modified so that it does *not* avoid scheduling partially-eligible MTTs (note the double-negative), instead treating them as it would any other MTT. The column that presents the results for this heuristic is labeled "P" in Table 6.7.

2. Heuristic (1) was modified so that no lost-cause policy is employed. (This is also equivalent to employing a lost-cause policy, but with an extremely high or infinite lost-cause threshold.) The column that presents the results for this heuristic is labeled "PL" in Table 6.7.

Task Set Parameters			L2 Miss Rate					
%S	**MTT U**	**WD**	GEDF	**H**	**P**	**PL**	**PLD**	**B**
50	[0.01, 0.1]	TC	3.62	1.60	1.31	3.28	3.58	3.58
50	[0.01, 0.1]	Uni	7.14	3.16	2.53	17.65	11.12	11.12
50	[0.1, 0.4]	TC	1.22	0.36	0.36	1.14	1.31	1.31
50	[0.1, 0.4]	Uni	6.70	0.67	0.60	5.79	7.30	7.30
50	[0.5, 0.9]	TC	1.07	0.28	0.28	1.08	1.08	1.08
50	[0.5, 0.9]	Uni	15.38	0.98	0.85	15.13	15.48	15.48
50	[0.01, 0.9]	TC	3.61	0.63	0.61	2.60	3.27	3.27
50	[0.01, 0.9]	Uni	7.92	0.78	0.76	7.91	9.28	9.28
100	[0.01, 0.1]	TC	5.30	1.67	1.44	4.87	4.71	4.71
100	[0.01, 0.1]	Uni	7.22	2.57	1.97	24.86	14.17	14.17
100	[0.1, 0.4]	TC	3.75	1.35	1.14	9.66	4.99	4.99
100	[0.1, 0.4]	Uni	7.02	3.46	3.25	22.75	15.38	15.38
100	[0.5, 0.9]	TC	3.81	2.83	0.76	1.99	2.27	2.27
100	[0.5, 0.9]	Uni	5.03	3.58	3.57	4.51	5.05	5.08
100	[0.01, 0.9]	TC	2.49	0.88	2.70	6.16	3.43	3.43
100	[0.01, 0.9]	Uni	4.30	3.70	3.55	6.07	5.20	5.13

Task Set Parameters			IPC					
%S	**MTT U**	**WD**	GEDF	**H**	**P**	**PL**	**PLD**	**B**
50	[0.01, 0.1]	TC	0.97	1.23	1.27	1.00	1.00	1.00
50	[0.01, 0.1]	Uni	0.80	1.17	1.26	0.62	0.73	0.73
50	[0.1, 0.4]	TC	1.21	1.20	1.41	1.29	1.28	1.28
50	[0.1, 0.4]	Uni	0.93	1.17	1.34	0.98	0.90	0.90
50	[0.5, 0.9]	TC	1.03	1.01	1.58	1.46	1.46	1.46
50	[0.5, 0.9]	Uni	0.77	0.92	1.61	1.18	1.16	1.16
50	[0.01, 0.9]	TC	1.01	1.12	1.46	1.20	1.17	1.17
50	[0.01, 0.9]	Uni	0.97	0.95	1.60	1.09	1.08	1.08
100	[0.01, 0.1]	TC	0.85	1.16	1.22	0.91	0.93	0.93
100	[0.01, 0.1]	Uni	0.76	1.11	1.15	0.46	0.61	0.61
100	[0.1, 0.4]	TC	0.96	1.18	1.20	0.70	0.89	0.89
100	[0.1, 0.4]	Uni	0.89	1.14	1.17	0.51	0.68	0.68
100	[0.5, 0.9]	TC	1.05	1.13	1.25	1.12	1.11	1.11
100	[0.5, 0.9]	Uni	0.99	1.06	1.07	0.99	0.98	0.98
100	[0.01, 0.9]	TC	1.09	1.23	1.14	0.95	1.09	1.09
100	[0.01, 0.9]	Uni	0.99	1.05	1.06	1.00	1.03	1.03

Table 6.7: The cache impact of avoiding policies that are difficult to implement efficiently in LITMUS^{RT}. Task-set generation parameters are specified identically to Table 6.2. The column labeled "H" presents performance numbers for the heuristic that performed best (in Section 6.1.2) for the combination of task-set generation parameters indicated. The columns that follow then present results for the same heuristic, modified based on the following code: "P" indicates that the heuristic does *not* avoid scheduling partially-eligible MTTs; "L" indicates that the heuristic does not employ *any* lost-cause policy; and "D" indicates that promotion duration policy (2) was used instead of policy (1). Finally, the column labeled "B" presents performance numbers for the "best-performing" heuristic that was implemented within LITMUS^{RT}.

3. Heuristic (2) was modified so that promotion-duration policy (2) is used instead of policy (1). The column that presents the results for this heuristic is labeled "PLD" in Table 6.7.

Additionally, Table 6.7 presents results for the "best-performing" heuristic that is described in Section 4.5 of Chapter 4 and implemented within LITMUS$^{\mathrm{RT}}$. This heuristic does not use the policies specified by **(i)**, **(ii)**, or **(iii)** stated at the beginning of this section. These results are presented in the column labeled "B" in Table 6.7.

Table 6.7 shows that, for each combination of task-set generation parameters, significant performance differences exist between each original heuristic and the new heuristics. First, for each combination of task-set generation parameters, heuristic (1) (column "P") *always* results in lower cache miss rates and higher IPC than the corresponding original heuristic (column "H"). This result was quite unexpected, but upon further reflection, we believe that it is because avoiding the scheduling of partially-eligible MTTs does not really achieve its intended goal of reducing pressure on the system in the future. When we schedule a partially-eligible MTT in the current quantum, that MTT must still be scheduled in a future quantum; however, the same is true when we do *not* schedule the MTT in the current quantum, except that more cores will be needed to fully schedule the MTT. In this case, the additional demand may often result in less scheduling flexibility, especially for MTTs with many tasks. This is because such MTTs may have their scheduling delayed multiple times, which can result in tardy jobs that must immediately be scheduled. Additionally, if an MTT is partially scheduled in the current quantum, then in most cases, all remaining cores were used to schedule it, and it will be the last MTT that is scheduled in the current quantum. As such, when scheduling that MTT does not result in thrashing, the jobs of that MTT that are scheduled in the future will likely benefit from cache reuse, since in the absence of tardy jobs, the remaining jobs will be scheduled in the next quantum.

Second, for each combination of task-set generation parameters, heuristic (2) (column "PL") always results in higher cache miss rates and lower IPC than heuristic (1) (column "P"). We believe that employing a lost-cause policy impacted scheduling in the following ways. First, as discussed in Section 6.1.2, at 100% system utilization, phantom tasks cannot be employed, so lost-cause policies (2) and (3), which schedule high-cache-impact MTTs in the current quantum (since thrashing will occur anyway) sometimes result in better performance in future quanta. Second, we believe that lost-cause policy (1) did more than just reduce average tardiness—when thrashing could not be avoided, the use of GEDF allows the jobs that would otherwise become tardy soonest to be scheduled. Since tardy jobs have higher priority than all other jobs under all of our heuristics (in order to ensure timing constraints), such jobs can make cache-aware scheduling considerably more difficult, as they will be scheduled with no regard to their cache impact. Thus, it makes sense to schedule jobs before they become tardy when possible, to ensure that sufficient scheduling flexibility will exist to prevent cache thrashing in future quanta.

Third, switching from promotion-duration policy (1) to policy (2) (in comparing column "PL" to column "PLD") seems to have either a minor negative impact on miss rates and IPC, or a significant positive impact. We also believe that policy (2) is more natural for an EDF-based policy, especially when considering more than the first 20 quanta of execution, as it promotes a job for its entire execution rather than for a single quantum-sized unit of its computation (the latter may be more natural for Pfair-based policies that schedule quantum-sized subtasks).

Fourth, performance seems to improve across all heuristics (and GEDF) when the task count is correlated with MTT WSS—when WSS is not correlated with the task count, there is a greater potential that MTTs exist with low task counts and high WSSs. Such MTTs are often very difficult to schedule in a way that avoids thrashing. Regardless of the reason, a correlation between task count and MTT WSS is desirable, as we believe such a correlation to be the more realistic scenario, since a larger number of tasks should be capable of referencing a larger region of memory.

Finally, note that, when comparing heuristic (3) for each combination of task-set generation parameters (column "PLD") to the best-performing heuristic that is implemented in LITMUS$^{\mathrm{RT}}$ (column "B"), the performance differences were almost always negligible. Further, the best-performing heuristic typically outperforms GEDF except in several notable cases where system utilization is 100%. As discussed in Section 6.1.2, these tended to be the cases where lost-cause policies had the most significant impact, as such policies presented a way to avoid thrashing in scenarios where phantom tasks could not be employed. Since the best-performing heuristic does not employ *any* lost-cause policy to keep scheduling overheads low, and phantom tasks cannot be employed, the best-performing heuristic has considerable difficulty achieving performance gains in these cases.

Overall, it appears that not employing a lost-cause policy has the greatest negative impact on system performance. Therefore, we conclude that finding a way to support lost-cause policies in the LITMUS^{RT} implementation in a low-overhead manner should be our most pressing concern related to improving our cache-aware scheduler, as supporting such policies would likely result in significant reductions in cache miss rates, and increases in IPC. However, note that even without support for lost-cause policies, our cache-aware scheduler often performs very well when compared to GEDF, as we will see next.

6.2 LITMUS^{RT}-Based Experiments

We next evaluated our LITMUS^{RT}-based cache-aware scheduler, consisting of both the best-performing heuristic and the cache profiler, in terms of profiler accuracy and performance as compared to GEDF. To do so, we performed experiments on three different machines, the attributes of which are shown in Table 6.8.

Machine A contains the Intel Core i7 processor, which represents nearly the state-of-the-art for released Intel multicore chips (it became publicly available in November 2008), and is the first general-purpose chip released by Intel where four cores share a single low-level cache. While hyperthreading allows two hardware threads to be supported per core on machine A, we disable hyperthreading in our experiments—the entries in Table 6.8 marked (*) have been adjusted to account for disabling hyperthreading. Hyperthreading can result in timing anomalies related to when each hardware thread is allowed access to core resources, which can make it difficult to enforce timing constraints and thus difficult to support real-time workloads. Therefore, in machine A, four logical CPUs share an 8192K cache.

In machine B, pairs of cores within the same processor package share a low-level cache; there are two pairs of cores (and two low-level caches) in each package. As we are concerned

Attribute	**Machine A**	**Machine B**	**Machine C**
Physical Processors	1	2	1
Processor Type	Intel Core i7	Intel Xeon E5420	Sun UltraSPARC T1
Processor Frequency	2.66 GHz	2.5 GHz	1.2 GHz
Cores per Processor	4	4	8
Hardware Threads per Core	1 (*)	1	4
Logical CPUs per Shared Cache	4 (*)	2	32
On-chip Cache Levels	3	2	2
L1 Instruction Cache Size	32K	32K	16K
L1 Instruction Cache Set Assoc.	4-way	8-way	4-way
L1 Instruction Cache Line Size	64 bytes	64 bytes	32 bytes
L1 Data Cache Size	32K	32K	8K
L1 Data Cache Set Assoc.	8-way	8-way	4-way
L1 Data Cache Line Size	64 bytes	64 bytes	16 bytes
L2 Cache Size	256K	6144K	3072K
L2 Cache Set Assoc.	8-way	24-way	12-way
L2 Cache Line Size	64 bytes	64 bytes	64 bytes
L3 Cache Size	8192K	N/A	N/A
L3 Cache Set Assoc.	16-way	N/A	N/A
L3 Cache Line Size	64 bytes	N/A	N/A
Shared Cache Level	L3	L2	L2
Off-Chip Main Memory	4 GB	8 GB	16 GB

Table 6.8: Attributes of the three different machines on which experiments were performed.

with the performance of our cache-aware scheduler in the presence of a single shared cache,[4] real-time tasks are prevented from executing on all but a single pair of cores sharing a single L2 cache. Thus, we treat machine B as a single dual-core machine, where two logical CPUs share a 6144K cache. Machine B also has the most fully-featured performance monitoring for the shared cache—issues related to performance monitoring will be discussed in more detail in Section 6.2.1, where results related to the accuracy of our profiler are presented for each of the three machines.

Finally, in machine C, all 32 hardware threads share a single L2 cache. Unlike machine A, hardware multithreading is *enabled* on this machine. This is because multithreading is considerably more deterministic on this hardware platform—each of the four threads is "scheduled" on each core in a round-robin manner, and the cores themselves do not support out-of-order execution, prefetching, branch prediction, or other features that would increase the possibility

[4]As stated in Chapter 2, multiple shared caches can be supported if we use a clustered variant of our cache-aware scheduler, where each cache is managed independently within each cluster; however, this scheduler was not implemented. Since clusters are scheduled independently, demonstrating a performance improvement through the use of our scheduler within a single cluster under a wide variety of real-time workloads should imply that performance improvements will be observed when a larger real-time workload is partitioned across multiple clusters.

of timing anomalies due to hardware multithreading. Since multithreading is enabled, the four hardware threads on each core share an L1 cache. While there are multiple levels of cache sharing in this machine, we only focus on avoiding L2 cache thrashing, since we expect L2 cache miss rates to have the most significant impact on overall system performance. Thus, machine C is viewed as containing 32 logical CPUs sharing a 3072K cache. It is worth noting that the effective speed of each logical CPU in machine C is considerably slower than the logical CPUs in machines A and B.

As stated in Chapter 4, we implemented our scheduler within LITMUS$^{\text{RT}}$ on all three machines. LITMUS$^{\text{RT}}$ contains a GEDF implementation, which was used in these experiments. Also, the default quantum length in LITMUS$^{\text{RT}}$ is 1 ms, which we did not change. In both GEDF 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. Since machine C cannot record shared cache references for each MTT with its performance counters, we instead counted instructions, and calculated misses per instruction instead of miss rates.

The rest of this section is organized as follows. In Section 6.2.1, we determine the accuracy of our profiler for MTTs with known memory reference patterns and WSSs, noting the considerable differences in accuracy across the three machines. Then, in Section 6.2.2, we compare our cache-aware scheduler to GEDF in terms of shared cache miss rates, deadline tardiness, and scheduling overheads. Finally, in Section 6.2.3, we evaluate the performance of our scheduler as compared to GEDF for a multimedia server workload.

6.2.1 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 its known WSS to the WSS estimate generated by our profiler. Since the known WSS does not account for cache space taken by instructions and bookkeeping variables, we would expect our estimates to be slightly higher than the known WSS values in most cases.

We generated task sets with the following parameters. Recall from Section 6.1.2 that

MTT utilization indicates the utilization of every task within an MTT, with execution cost and period defined similarly, and therefore all tasks within an MTT have the same utilization, execution cost, and period.

- **System utilization:** Between 55% and 65%, assuming negligible scheduling overheads.
- **MTT periods:** Between 20 and 2,400 ms on machine A, and between 40 and 2,400 ms on machines B and C (some values removed to avoid arithmetic overflow). Larger periods were necessary to allow larger execution costs, which were needed to support large per-job WSSs under certain memory reference patterns. For machines A and C, the lower bound on MTT periods was raised to 40 ms to further ensure that execution costs would be high, thus ensuring higher MTT WSSs. This was necessary for interesting experiments on machine A, since only two cores share a cache. Similarly, it was necessary on machine C since it takes considerably longer to reference memory on that machine than on machines A and B.
- **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.
- **MTT execution costs:** Derived from periods and utilizations, and at least 3 ms. Jobs are *not* backlogged, making exactly three passes over their working sets.
- **MTT task counts:** Uniform over [1, 4] on machines A and C, and uniform over [1, 2] on machine B.
- **MTT per-job data WSSs**: Generated as a function of execution cost. For an MTT with execution cost e, its WSS was set to $min(\lfloor e/3 \rfloor \cdot 128\text{K}, 7.5\text{ MB})$ on machine A, $min(\lfloor e/3 \rfloor \cdot 128\text{K}, 5.5\text{ MB})$ on machine B, and $min(\lfloor e/3 \rfloor \cdot 8\text{K}, 1.5\text{ MB})$ on machine C. (Recall that it takes much longer to reference a memory location on machine C.) Due to the lower bound of 3 ms on execution cost, the minimum MTT WSS was 128K on machines A and B, and 8K on machine C. For every machine, the upper bound on WSS

was intended to increase the probability that thrashing can be avoided when MTTs are co-scheduled, assuming that some level of co-scheduling will be required in order to meet timing constraints. To this end, the upper bounds on WSS were within 0.5 MB of the shared cache size on machines A and B, rather than exactly the shared cache size. For machine C, a more conservative upper bound was required due to the large number of logical CPUs, which resulted in large task sets: the upper bound was half the size of the shared cache.

For each combination of the above parameters, we scheduled and profiled 100 task sets (which were different for each machine), where either a sequential or random memory reference pattern was employed by all jobs. Figure 6.2 presents pseudo-code for each MTT, where procedure RandomPattern() or SequentialPattern() is invoked once per job depending on whether a random or sequential reference pattern was employed by all jobs for a given experiment. In the case of the random memory reference pattern, different jobs in the same MTT employed entirely different random patterns, as can be seen in Figure 6.2 where GenerateRandomValue() is invoked independently by each job—this was in contrast to the sequential memory reference pattern, where the per-job reference pattern is the same. All task sets were executed for one minute.

Under all experiments, each MTT (including each single-threaded MTT) is launched as a process—this process is represented by the procedure LaunchMTT() in Figure 6.2, which takes as parameters both the number of tasks in the MTT and the MTT WSS. This process allocates a two-dimensional array representing the working set of all tasks in the MTT by invoking procedure AllocateIntegerArray(), after which one thread is launched (using the POSIX Threads, or *Pthreads*, API in Linux) for every real-time task that is associated with that MTT—in Figure 6.2, a thread is created using CreateThread(), which creates a thread executing the code of procedure TaskThread() with the arguments shown. In other words, an MTT maps to a process, and the tasks within an MTT map to individual threads; as such, the threads of the MTT process share an address space (and a working set) in the same way that we conceptually assumed that tasks within an MTT share a working set. Once each thread is launched, it continues to be available for execution for one minute, releasing

```
RandomPattern(jobNum, workingSet, WSS)
    ▷ Make three random passes over working set
1   for i := 1 to 3 ∗ WSS do
2       nextRandLoc := GenerateRandomValue(WSS);
3       tempVariable := workingSet[jobNum mod numberOfWorkingSets][nextRandLoc]
    od

SequentialPattern(jobNum, workingSet, WSS)
    ▷ Make three sequential passes over working set
1   for i := 1 to 3 ∗ WSS do
2       tempVariable := workingSet[jobNum mod numberOfWorkingSets][i mod WSS]
    od

TaskThread(workingSet, WSS)
    ▷ Reference memory until the task is told to stop
1   jobNum := 0;
2   while continueExecuting do
        ▷ Sleep until the next job is released
3       SleepUntilNextJobRelease();
        ▷ Reference memory appropriately
4       if (Random memory reference pattern employed) then
                ▷ Invoke random reference pattern
5               RandomPattern(jobNum, workingSet, WSS)
6       else
                ▷ Invoke sequential reference pattern
7               SequentialPattern(jobNum, workingSet, WSS)
        fi;
8       jobNum := jobNum + 1
    od

LaunchMTT(numTasks, WSS)
    ▷ Allocate memory for shared working set
1   workingSet := AllocateIntegerArray(numberOfWorkingSets, WSS);
    ▷ Launch a thread for each real-time task in the MTT
2   for i := 1 to numTasks do
3       CreateThread(TaskThread, workingSet, WSS)
    od
```

Figure 6.2: Pseudo-code for each MTT. Working sets are allocated as integer arrays, and the WSSs that are passed to procedures RandomPattern() and SequentialPattern() indicate the size of this integer array. A fixed number of such arrays, indicated by the global variable *numberOfWorkingSets*, are used so that working sets appear to be different for every job of a task. Variable *continueExecuting* is a conditional variable, initialized to `true`, which becomes `false` after one minute of execution (the length of an experiment).

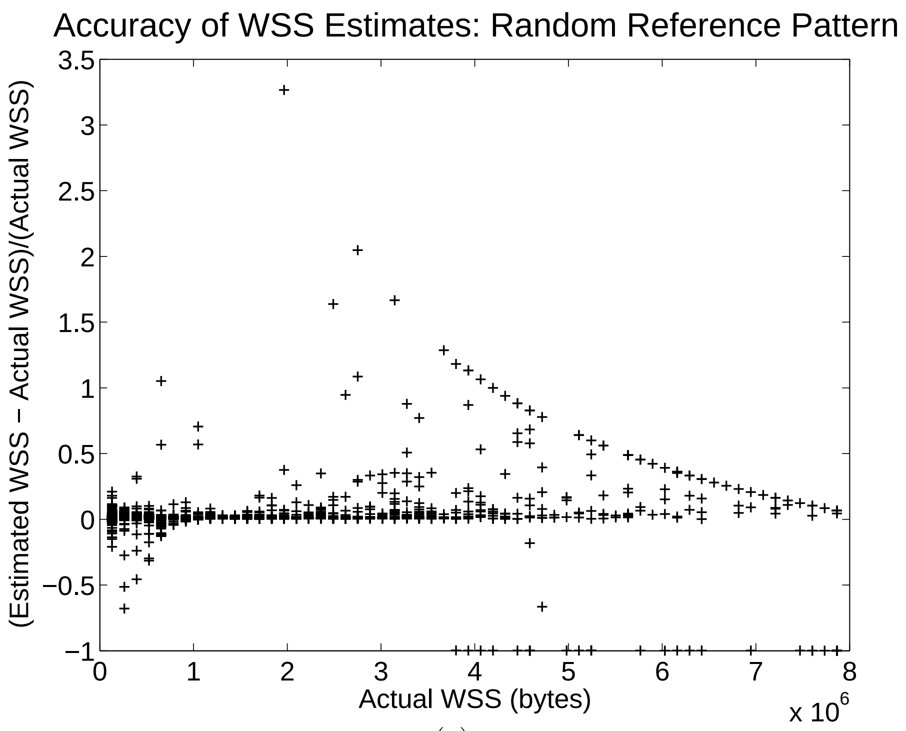
one job per period. After one minute (of wall clock time, not execution time), the conditional variable *continueExecuting* becomes `false`, and all threads in the MTT stop executing, at which point the MTT is considered to have completed execution. Since all MTTs begin execution at the same time (recall that we are concerned with synchronous task sets in this dissertation), each experiment lasts for one minute, after which all MTTs complete execution and data is logged before the next task set is launched. (This ensures that each task set is executed for one minute, as stated earlier.)

During thread execution, procedure SleepUntilNextJobRelease() is called so that the thread does nothing until the release time of the next job, as determined by the task with which the thread is affiliated. Upon awakening, each thread then references memory randomly or sequentially, as shown. Different working sets are referenced for every job, so that 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 $T = U$)—this allows the assumptions upon which our profiler relies, stated in Chapter 5, to be satisfied.

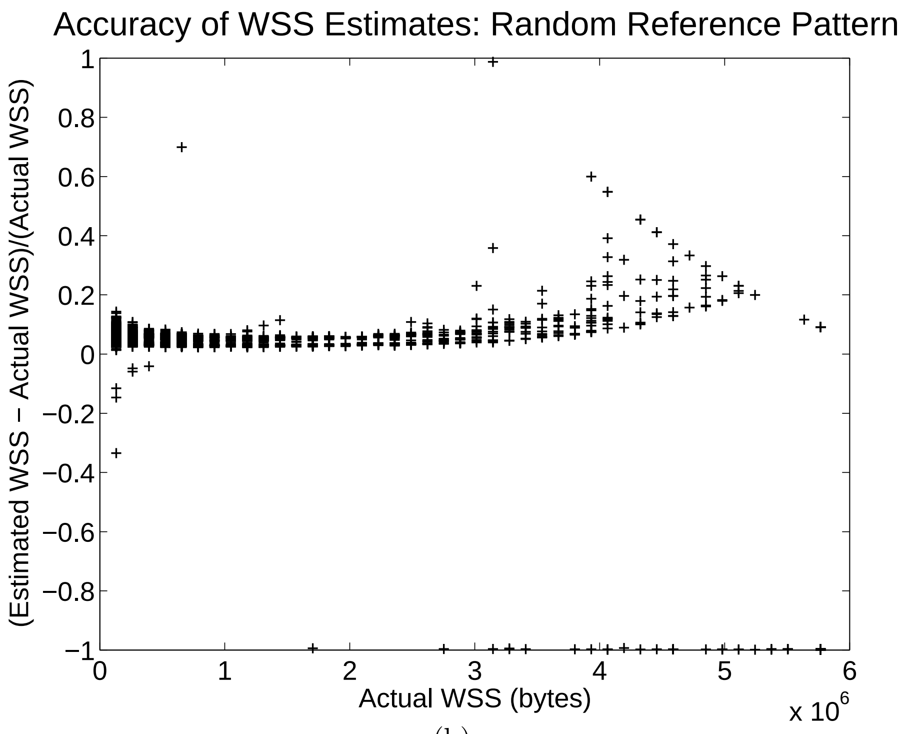
All pages that are associated with an MTT process are locked in main memory—this includes all pages that are mapped into the address space of the MTT process, including those pages that comprise the working set that is referenced by all tasks within the MTT. Therefore, during our experiments, real-time tasks do not generate page faults, and since background activity is negligible during our experiments, paging activity that would be associated with virtual memory has no significant impact on our shared cache management policy or job execution times.

For each MTT, the WSS estimate produced at the end of the minute of execution was compared to the known WSS. Overall, this resulted in 2,700, 2,309, and 19,263 MTTs being profiled under each memory reference pattern for machines A, B, and C, respectively.

Results. Figures 6.3 and 6.4 show the proportionate error in the WSS estimates generated by the profiler on each machine, when compared to the known WSS of each MTT, for both random and sequential memory reference patterns, respectively. Proportionate error is defined to be the difference between the WSS estimate generated by the profiler and the actual WSS of the MTT, divided by the actual WSS of the MTT. Since we found little difference in

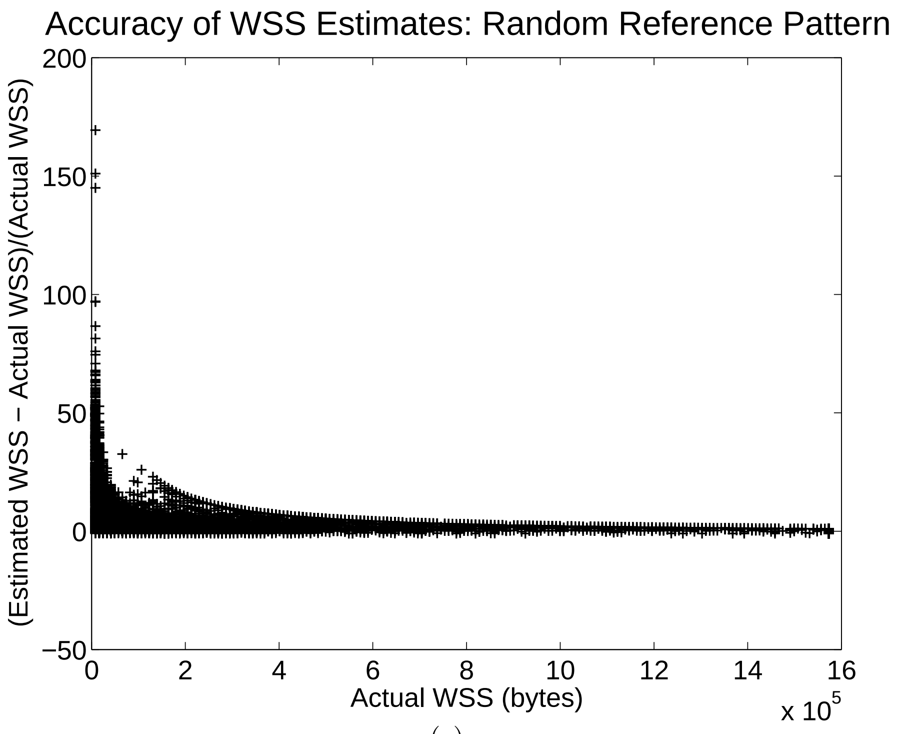


(a)

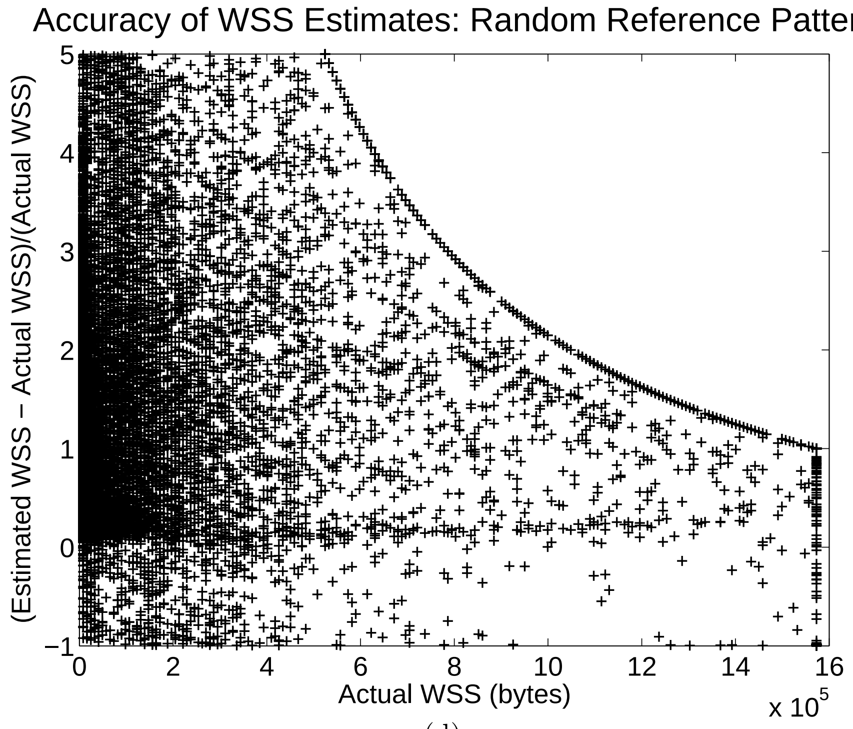


(b)

Figure 6.3: Profiler accuracy for **(a)** machine A, **(b)** machine B, **(c)** machine C, and **(d)** machine C with very large error values removed, when a random reference pattern is employed. (Continued on the next page.)

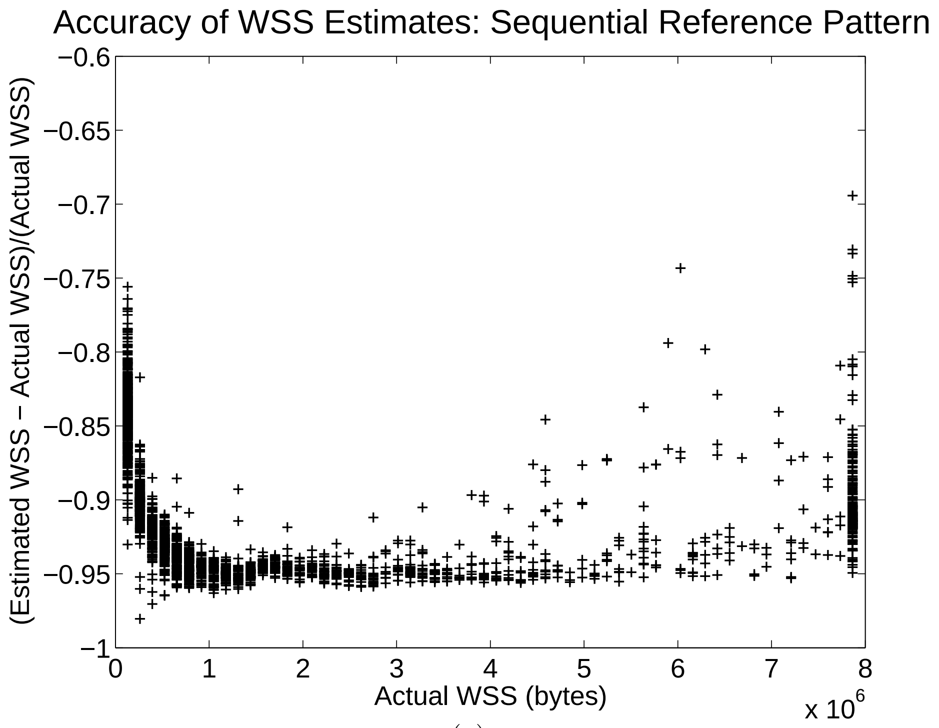


(c)

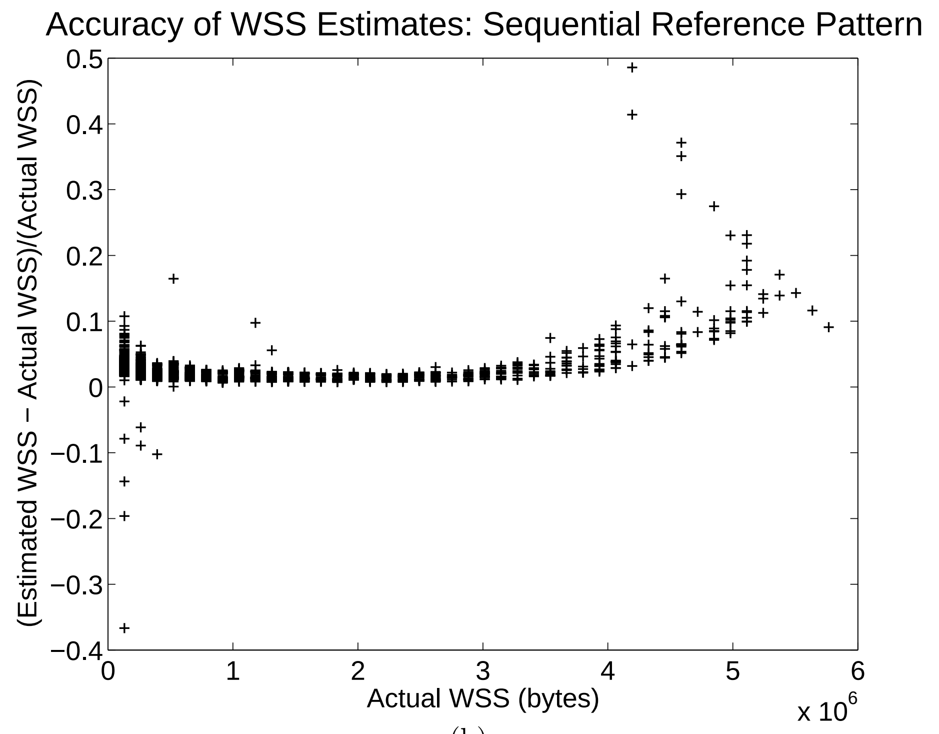


(d)

Figure 6.3: (continued) Profiler accuracy for **(a)** machine A, **(b)** machine B, **(c)** machine C, and **(d)** machine C with very large error values removed, when a random reference pattern is employed.

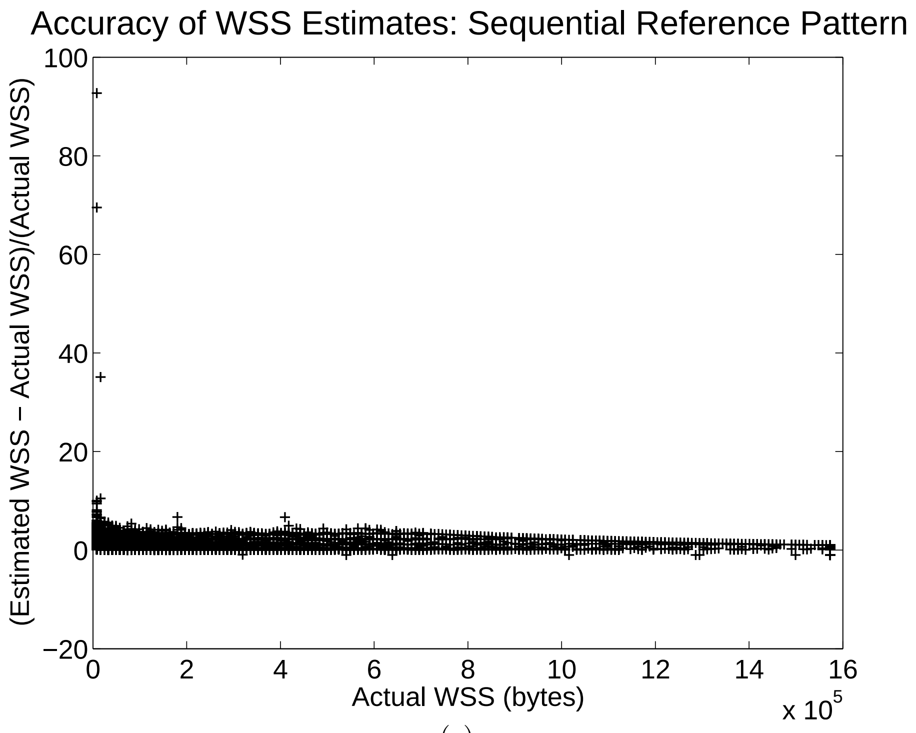


(a)

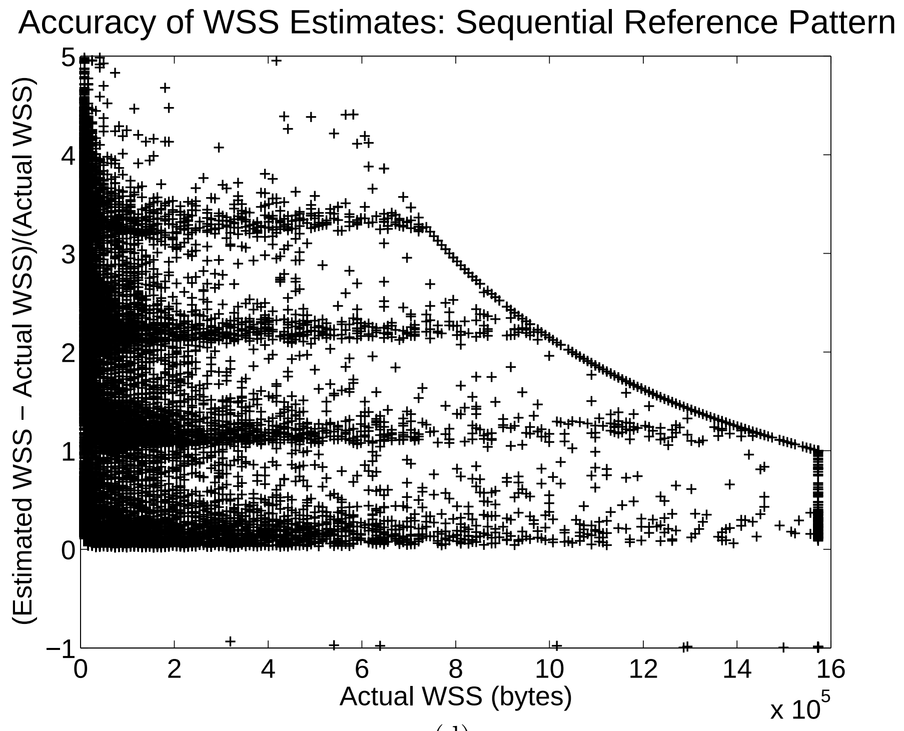


(b)

Figure 6.4: Profiler accuracy for **(a)** machine A, **(b)** machine B, **(c)** machine C, and **(d)** machine C with very large error values removed, when a sequential reference pattern is employed. (Continued on the next page.)



(c)



(d)

Figure 6.4: (continued) Profiler accuracy for **(a)** machine A, **(b)** machine B, **(c)** machine C, and **(d)** machine C with very large error values removed, when a sequential reference pattern is employed.

profiler accuracy when varying MTT utilizations, results are presented as a function of the actual WSS of each MTT. We now discuss the results for each machine.

6.2.1.1 Machine A

The results for machine A are presented in inset (a) of Figures 6.3 and 6.4. Our profiler is typically accurate on machine A for random reference patterns. Exceptions are clearly shown, and become more frequent at larger WSSs. This is because, as WSS increases, it becomes more difficult for the bootstrapping process (described in Chapter 5) to result in WSS estimates that converge, due to the large number of discarded measurements resulting from shared cache thrashing, and a threshold of convergence that may be too low for larger WSSs (100 lines, or 6,400 bytes). However, note that error then decreases as WSSs approach the shared cache size, since WSS estimates are capped at that size by the profiler. Therefore, to some extent, the upper bound on WSS partially "cancels out" this issue with the bootstrapping process, since as WSS increases, inaccurate measurements are much more likely to be capped. Overall, the exceptions were outliers, since the maximum observed error was under 10% and 5% for over 91% and 78%, respectively, of the MTTs (with random reference patterns) that were profiled on machine A.

Hardware limitations. For sequential reference patterns, our profiler is extremely inaccurate, producing estimates that are close to zero for all profiled MTTs (see inset (a) of Figure 6.4). This suggests that most cache misses were not counted by the performance counters in machine A. 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. While prefetching generally improves system performance, especially under sequential reference patterns, 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, poor WSS estimates cannot be avoided for sequential reference patterns on our Core i7 processor.

Interestingly, both the ability to count prefetching-related events (via a processor-specific performance event) and disable the prefetcher *were* available in most Core 2 chips, and the Xeon chip in machine B, both of 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 this feature has been locked, if present at all, in the commercial versions of the chip.) In particular, while the Core 2, Xeon, and Core i7 chips all have the ability to count lower-level cache misses, the Core 2 and Xeon chips allow prefetching events to be included in the counts by slightly changing the bits that are written to a performance event select register to program the counters. While the Core i7 does include additional "uncore" performance counters, which are intended to count events that are related to *off-core* (but on-chip) components such as the shared cache, these counters also do not provide any way to account for prefetching events, especially if events originating from different cores need to be counted separately, as is needed in this case. Therefore, the new "uncore" counters are not particularly useful for our purposes. In summary, the existence of the prefetching-related features in earlier Intel processors suggests that it is not unreasonable to expect them, and we believe that the loss of these features is a step in the wrong direction, since it makes shared cache management more difficult (and not just for real-time applications).

6.2.1.2 Machine B

The results for machine B are presented in inset (b) of Figures 6.3 and 6.4. As indicated earlier, the performance counters on this machine *can* account for prefetching effects. As a result, the profiler was more accurate for sequential reference patterns on machine B than on *any* other machine employing either reference pattern—the maximum observed error was under 10% and 5% for over 97% and 79%, respectively, of the MTTs (with sequential reference patterns) that were profiled. Interestingly, the results for random reference patterns were actually slightly worse than those for machine A—the maximum observed error was under 10% and 5% for over 89% and 39%, respectively, of the MTTs that were profiled when random reference patterns were used. However, trends appear to be quite similar on both machines, with accuracy being quite good until larger WSSs are reached, at which time error increases due to difficulties with the bootstrapping process.

We believe that, under random reference patterns, additional misses were counted by the prefetcher that were related to the additional logic required to generate a random reference pattern, which requires more instructions and bookkeeping than generating a sequential reference pattern. These misses were then factored into the estimated WSS, which, in turn, resulted in error values for random reference patterns that were slightly worse than machine A, where prefetching events were not counted. While we could claim that counting prefetching events was actually undesirable in this case, in reality, the effect of the additional logic will increase the size of the cache footprint of the MTT, and should be accounted for. Thus, we believe that the profiler was most accurate by far on machine B under *both* reference patterns, as compared to the other machines.

Another trend worth noting under machine B is a pronounced upswing in *minimum* error at the largest WSSs under both the random and sequential reference patterns, clearly seen in inset (b) of Figures 6.3 and 6.4—in fact, as WSS approaches the shared cache size, minimum errors close to zero are never observed, replaced instead by a minimum error that peaks at roughly 20% and 10% under the random and sequential reference patterns, respectively. In these cases, we believe that conflict misses, which are assumed to be negligible due to the high set associativity of the shared cache, may have occurred more frequently than expected. Such misses are more likely to occur during periods of high contention for the shared cache, such as when a job is scheduled with a WSS that approaches the shared cache size. This is especially true under the random reference pattern, where the potential for generating reference patterns that result in conflict misses increases, which would explain why we see a greater minimum error in this case.

6.2.1.3 Machine C

The results for machine C are presented in insets (c) and (d) of Figures 6.3 and 6.4. Due to a large number of extremely inaccurate estimates that make it difficult to see detail in inset (c) of each figure, inset (d), which shows the same results with the large error values removed, has been included in each figure (note the different y-axis range in each inset). Overall, profiler accuracy was very poor on machine C, though for very different reasons than those encountered on machine A. Instead, we believe that the results are mostly related to shared cache misses being counted multiple times when different jobs in the same MTT reference the same memory locations. When multiple jobs of the same MTT attempt to reference the same location in memory, we would assume that the first job to reference the location would bring data from that location into the cache, resulting in a cache miss, and the remaining jobs would then use the data in the cache; thus, the location would generate one cache miss. Unfortunately, in reality, this is not what happened when different jobs referenced the same location at approximately the same time, as was often the case in the sequential reference pattern. Instead, since the time to bring a piece of data from main memory into the cache is considerably higher on machine C than on machines A or B, the first job to reference the location was often still waiting for data to be brought into the cache when the remaining jobs referenced the data. As a result, some of the remaining jobs would be forced to wait, and the performance counters associated with those jobs would also record a cache miss, resulting in the miss being counted two or more times for the same location. In the worst case, this can result in a "lock-step" sequential reference pattern where *all* jobs must wait for *every* memory location to be brought into the cache, resulting in each miss being recorded multiple times.

The impact of this lock-step effect can be seen clearly for sequential reference patterns in inset (d) of Figure 6.4, where the proportionate error congregates around integer values between zero and three, corresponding to MTTs that contained between one and four tasks (in the worst case, misses would be recorded X times for an MTT with X tasks, resulting in an error of $X - 1$). Under random reference patterns, the lock-step effect is likely to occur much less frequently, since it will only occur when multiple tasks attempt to reference the same memory location (or two locations that are part of the same cache line); this is seen in inset (d) of Figure 6.3, where the effect occurs, but since it does not occur reliably throughout the reference pattern, errors do not congregate around integer values. There also appears to be additional error under machine C that cannot be fully accounted for by the lock-step effect, though we do expect that it is related to the large amount of time required to bring data from a memory location into the cache, during which any references to that data will result in cache misses being recorded by the performance counters.

Additionally, the counters themselves are somewhat limited, as they are unable to count events related to data stores, and can only count either events related to data loads or instruction cache requests. Since there is only one programmable counter per hardware thread, both events cannot be counted simultaneously, so we chose to count data loads, since we expected data loads to have a larger impact on WSS than instruction cache requests.

6.2.1.4 Summary

Overall, we found that, regardless of hardware limitations, the experimental results for random reference patterns on machine A, and for both random and sequential reference patterns on machine B, show that *our concepts are sound*—given sufficient *hardware support* to accurately count shared cache misses, our profiler is often quite accurate. Moreover, the needed hardware support *is not complicated*. We shall see next that system performance can be improved over GEDF when our cache-aware scheduler is used and the profiler is reasonably accurate.

6.2.2 Performance Versus GEDF

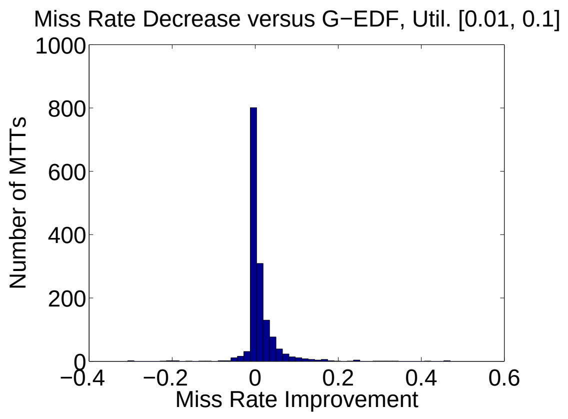
In the next set of experiments, the same task sets generated in Section 6.2.1 were scheduled under both our scheduler and GEDF, again for one minute. In these experiments, only the random reference pattern was employed under Machine A, since the profiler was inaccurate for sequential reference patterns due to the hardware limitations described earlier. (While the profiler was also inaccurate for both patterns in machine C, the estimates were typically *higher* than reality, rather than roughly zero, and we could therefore expect to see at least a marginal decrease in cache miss rates when our scheduler is used.) MTTs were compared on the basis of several performance metrics under both our scheduler and GEDF. These performance metrics are: cache miss rate, deadline tardiness, and scheduling overheads. Results related to each metric are presented next.

6.2.2.1 Average-Case Cache Miss Rate

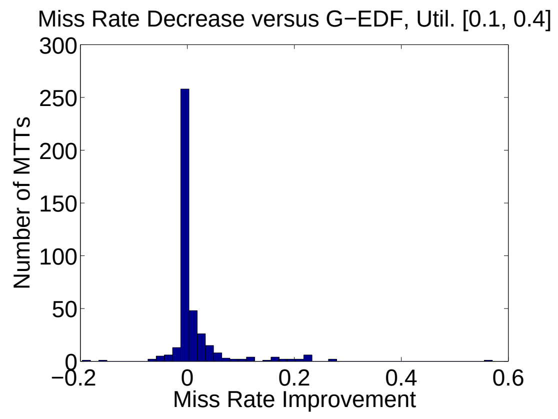
Figures 6.5–6.9 present differences in cache miss rates or misses per instruction under GEDF and our scheduler, for each utilization distribution on machines A (random reference pattern), B (random and sequential reference patterns), and C (random and sequential reference patterns), respectively. In all figures, insets (a) through (e) present histograms for each distribution indicating the (absolute, not relative) improvement in the average cache miss rate (or misses per instruction, for machine C) for each MTT over all of its jobs. For machines A and B, a value of 1.0 would imply a 100% cache miss rate under GEDF and a 0% cache miss rate under our scheduler; for machine C, a value of 1.0 would indicate a drop in average misses per instruction of one. Inset (f) presents additional statistics for each histogram: the minimum, average, and maximum miss rate (or misses per instruction) decrease observed under our scheduler (with respect to GEDF), and the percentage of MTTs that were positively impacted, not impacted, and negatively impacted by our scheduler (x represents the amount of the decrease). Finally, note that the range of the x-axis often varies substantially between figures and within insets of the same figure.

Machine A. On machine A (Figure 6.5), our scheduler tends to result in a reduction in cache miss rates over GEDF under all distributions, sometimes by a large margin, and rarely results in miss rate increases. This is more easily observed in Figure 6.5(f): for a given utilization distribution, cache miss rates improved (*i.e.*, decreased) for as many as 33.20% of MTTs, and worsened for at most 9.22% of MTTs. The positive 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 reduce cache miss rates by avoiding thrashing.

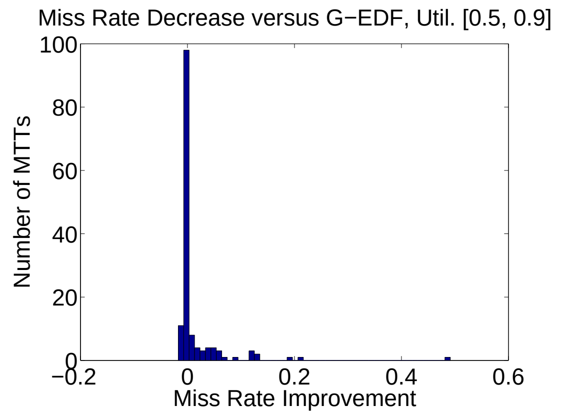
Machine B. Our scheduler demonstrated the greatest miss rate improvements on machine B, as expected since profiler accuracy was greatest on this machine. Under the random reference pattern (Figure 6.6), improvements were comparable to those seen on machine A.



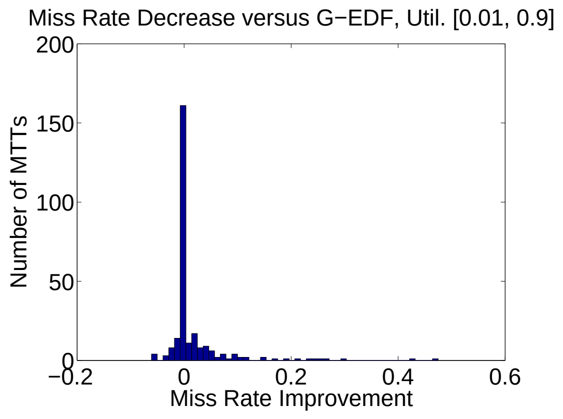
(a)



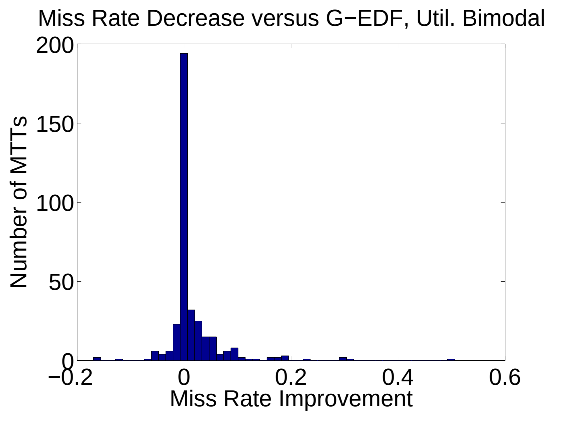
(b)



(c)



(d)



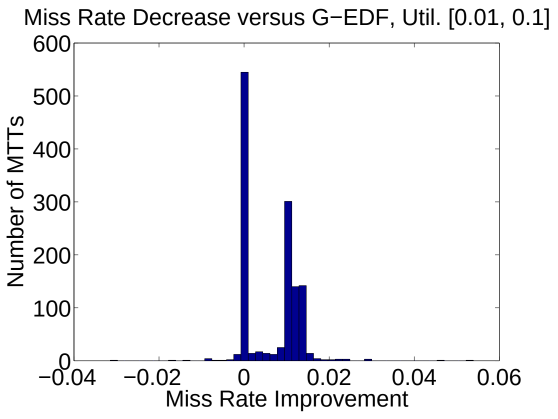
(e)

Util. Dist.	**Min.**	**Avg.**	**Max.**
[0.01, 0.1]	-0.307	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
Overall	**-0.307**	**0.014**	**0.570**

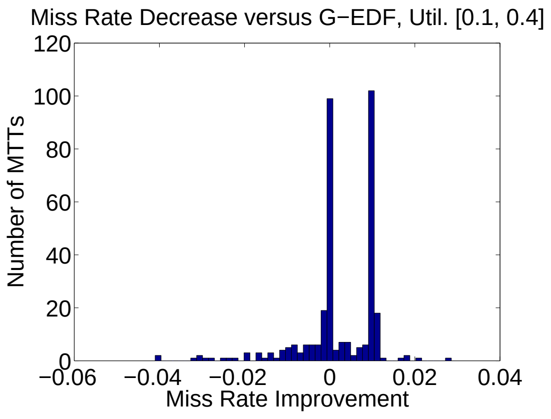
Util. Dist.	$x < -0.01$	$\lvert x \rvert \leq 0.01$	$x > 0.01$
[0.01, 0.1]	5.41%	61.39%	33.20%
[0.1, 0.4]	8.45%	65.46%	26.09%
[0.5, 0.9]	4.83%	74.48%	20.69%
[0.01, 0.9]	8.21%	65.30%	26.49%
Bimodal	9.22%	60.06%	30.73%
Overall	**6.63%**	**62.93%**	**30.44%**

(f)

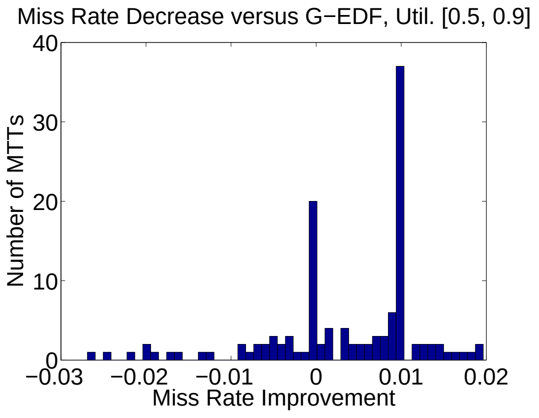
Figure 6.5: The decrease in the average shared cache miss rate over each utilization distribution, as compared with GEDF, for machine A (random reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.



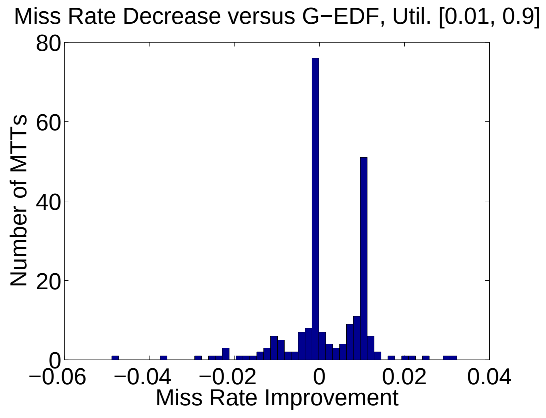
(a)



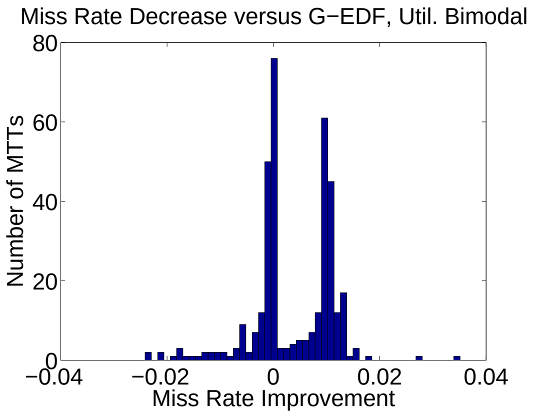
(b)



(c)



(d)



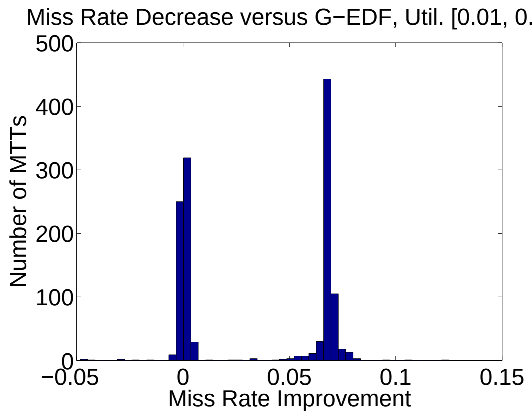
(e)

Util. Dist.	**Min.**	**Avg.**	**Max.**
[0.01, 0.1]	-0.032	0.006	0.054
[0.1, 0.4]	-0.041	0.002	0.029
[0.5, 0.9]	-0.027	0.004	0.020
[0.01, 0.9]	-0.049	0.002	0.032
Bimodal	-0.024	0.004	0.035
Overall	**-0.049**	**0.005**	**0.054**

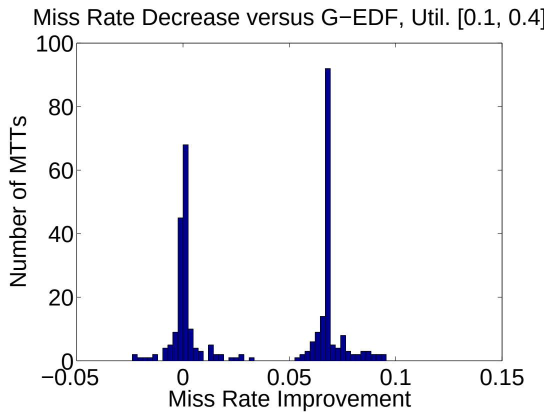
Util. Dist.	$x < -0.01$	$\|x\| \leq 0.01$	$x > 0.01$
[0.01, 0.1]	0.24%	57.11%	42.65%
[0.1, 0.4]	7.53%	69.88%	22.59%
[0.5, 0.9]	7.94%	76.19%	15.87%
[0.01, 0.9]	9.78%	72.89%	17.33%
Bimodal	4.72%	69.17%	26.11%
Overall	**3.33%**	**63.40%**	**33.26%**

(f)

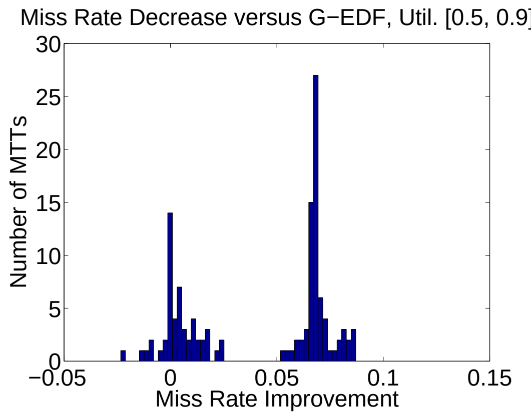
Figure 6.6: The decrease in the average shared cache miss rate over each utilization distribution, as compared with GEDF, for machine B (random reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.



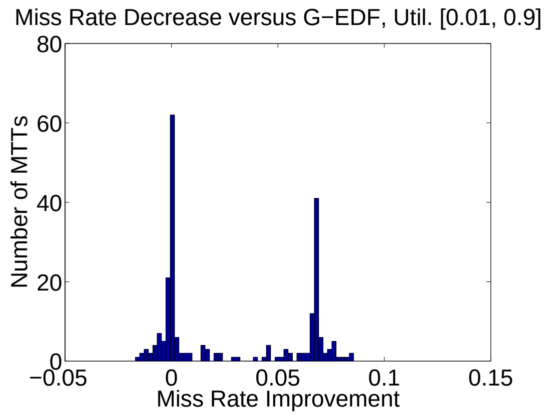
(a)



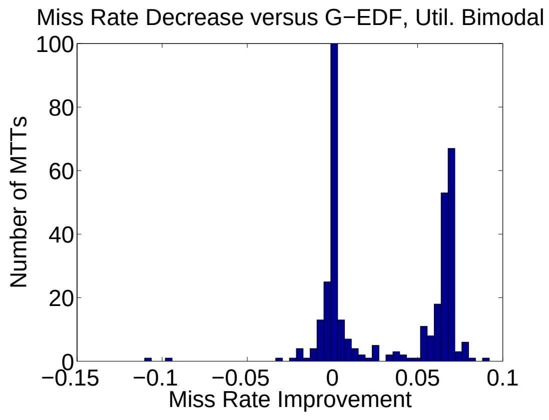
(b)



(c)



(d)



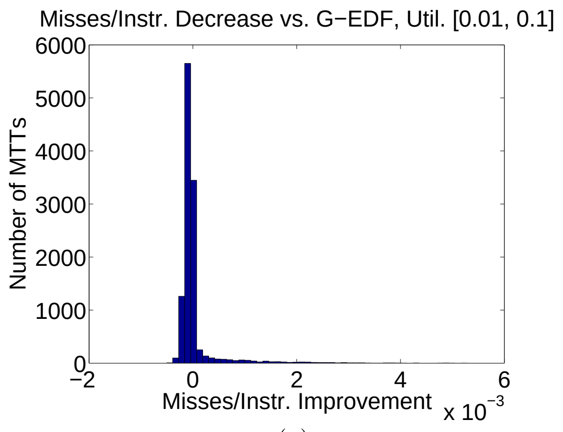
(e)

Util. Dist.	Min.	Avg.	Max.
[0.01, 0.1]	-0.048	0.035	0.125
[0.1, 0.4]	-0.024	0.035	0.096
[0.5, 0.9]	-0.023	0.042	0.087
[0.01, 0.9]	-0.017	0.028	0.086
Bimodal	-0.110	0.032	0.092
Overall	**-0.110**	**0.034**	**0.125**

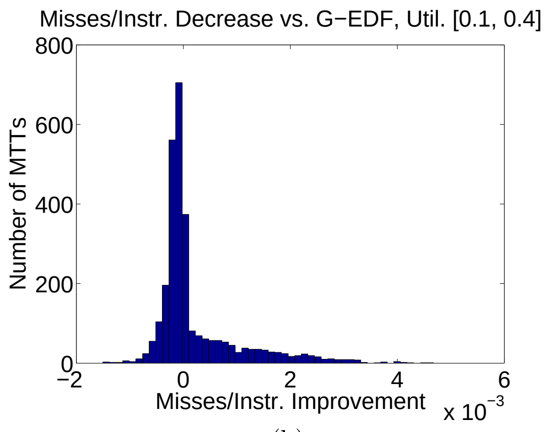
Util. Dist.	$x < -0.01$	$\lvert x \rvert \leq 0.01$	$x > 0.01$
[0.01, 0.1]	0.55%	47.95%	51.50%
[0.1, 0.4]	2.11%	44.58%	53.31%
[0.5, 0.9]	2.38%	28.57%	69.05%
[0.01, 0.9]	3.56%	49.33%	47.11%
Bimodal	3.61%	43.61%	52.78%
Overall	**1.65%**	**45.86%**	**52.49%**

(f)

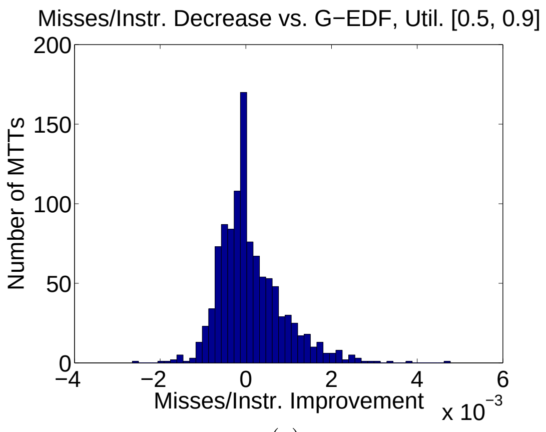
Figure 6.7: The decrease in the average shared cache miss rate over each utilization distribution, as compared with GEDF, for machine B (sequential reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.



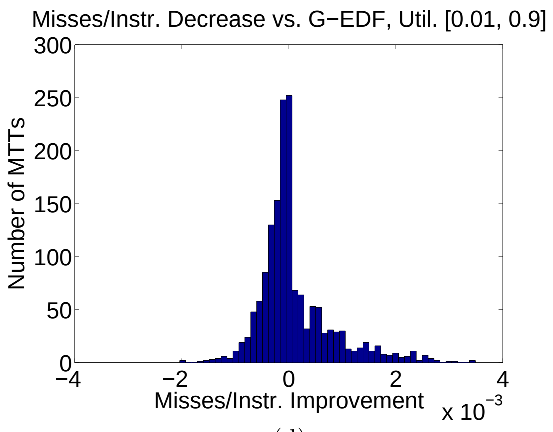
(a)



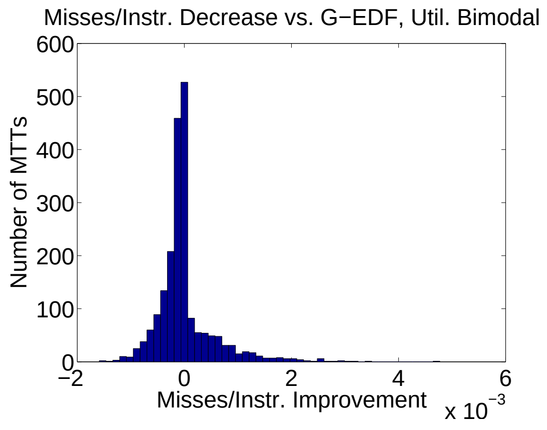
(b)



(c)



(d)



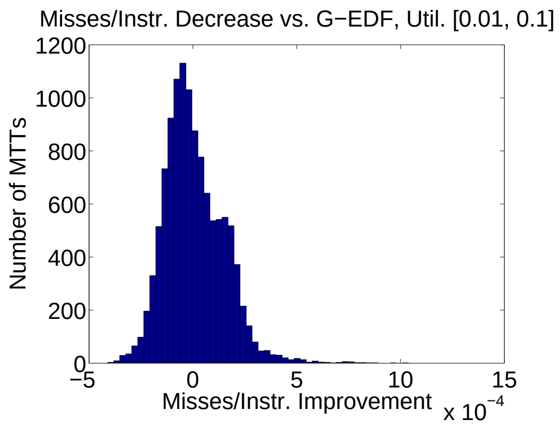
(e)

Util. Dist.	**Min.**	**Avg.**	**Max.**
[0.01, 0.1]	-0.0005	0.0000	0.0053
[0.1, 0.4]	-0.0015	0.0002	0.0047
[0.5, 0.9]	-0.0027	0.0001	0.0047
[0.01, 0.9]	-0.0020	0.0001	0.0035
Bimodal	-0.0016	0.0000	0.0048
Overall	**-0.0027**	**0.0001**	**0.0053**

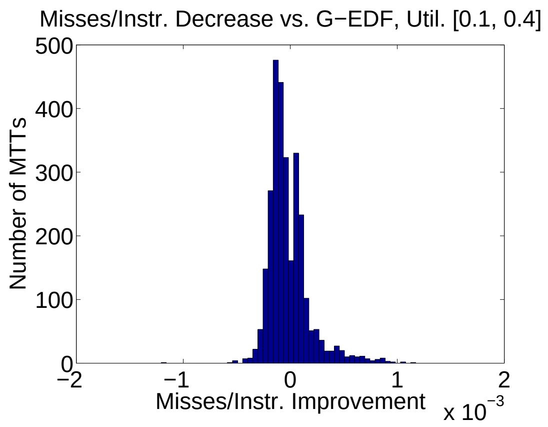
Util. Dist.	$x < -0.01$	$\lvert x\rvert \leq 0.01$	$x > 0.01$
[0.01, 0.1]	0.00%	100.00%	0.00%
[0.1, 0.4]	0.00%	100.00%	0.00%
[0.5, 0.9]	0.00%	100.00%	0.00%
[0.01, 0.9]	0.00%	100.00%	0.00%
Bimodal	0.00%	100.00%	0.00%
Overall	**0.00%**	**100.00%**	**0.00%**

(f)

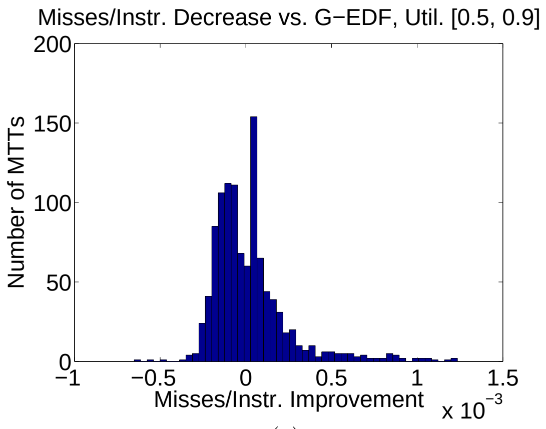
Figure 6.8: The decrease in the average shared cache miss rate over each utilization distribution, as compared with GEDF, for machine C (random reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.



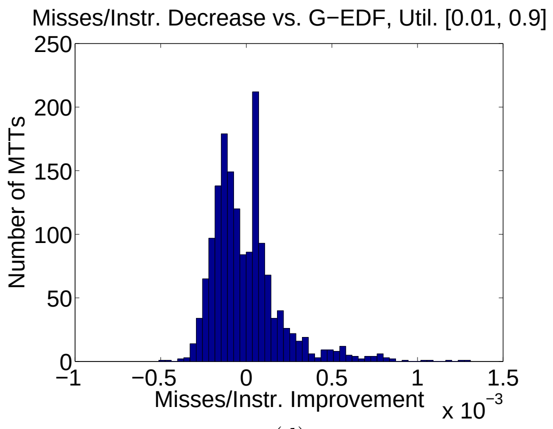
(a)



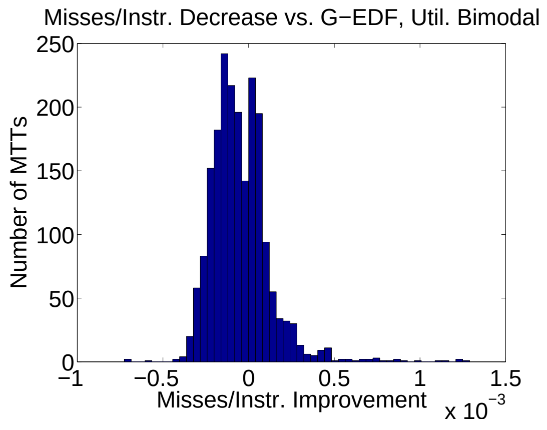
(b)



(c)



(d)



(e)

Util. Dist.	**Min.**	**Avg.**	**Max.**
[0.01, 0.1]	-0.0004	0.0000	0.0010
[0.1, 0.4]	-0.0012	0.0000	0.0012
[0.5, 0.9]	-0.0007	0.0000	0.0012
[0.01, 0.9]	-0.0005	0.0000	0.0013
Bimodal	-0.0007	0.0000	0.0013
Overall	**-0.0012**	**0.0000**	**0.0013**

Util. Dist.	$x < -0.01$	$\lvert x \rvert \leq 0.01$	$x > 0.01$
[0.01, 0.1]	0.00%	100.00%	0.00%
[0.1, 0.4]	0.00%	100.00%	0.00%
[0.5, 0.9]	0.00%	100.00%	0.00%
[0.01, 0.9]	0.00%	100.00%	0.00%
Bimodal	0.00%	100.00%	0.00%
Overall	**0.00%**	**100.00%**	**0.00%**

(f)

Figure 6.9: The decrease in the average shared cache miss rate over each utilization distribution, as compared with GEDF, for machine C (sequential reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.

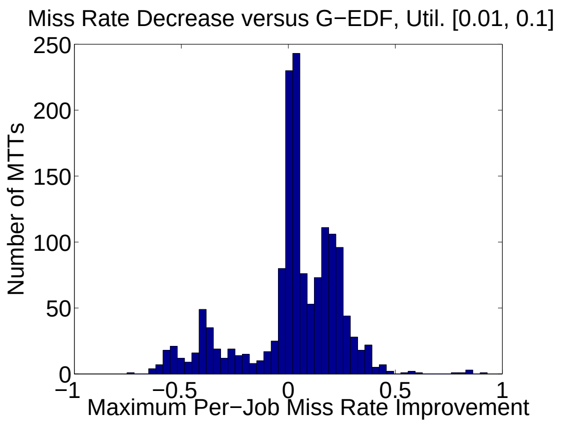
Under the sequential reference pattern (Figure 6.7), improvements were somewhat greater, and in most cases, the majority of MTTs experienced a reduction in cache miss rates when our scheduler was used. Note that, under both reference patterns, we also see a bimodal trend in the results. We believe that the two peaks correspond to MTTs with one or two tasks, where MTTs with two tasks benefit more due to co-scheduling. If this belief is true, then for MTTs with two tasks, a greater benefit is seen under the sequential reference pattern, which makes sense since under such a reference pattern, there is more predictable data sharing within MTTs.

Machine C. As was expected, due to the poor accuracy of the profiler on machine C, differences in misses per instruction were not very significant when our scheduler was used on machine C, as seen in Figures 6.8 and 6.9. Additionally, our performance metric was different on this machine, since shared cache references could not be counted, which may have made performance differences more difficult to see. If performance differences were *not* impacted by the change in metric, then in the average case, there was virtually no benefit to using our scheduler; however, as we will see next for all machines, there was some benefit to using our scheduler when considering *worst-case* per-job misses per instruction.

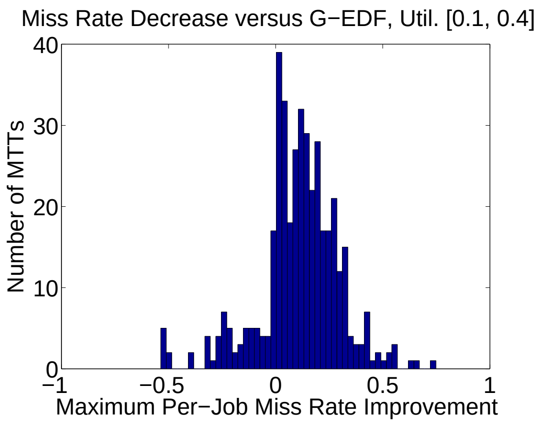
6.2.2.2 Worst-Case Cache Miss Rate

Figures 6.10–6.14 present differences in cache miss rates or misses per instruction on machines A (random reference pattern), B (random and sequential reference patterns), and C (random and sequential reference patterns), respectively, this time with respect to the *maximum* observed (*i.e.*, worst-case) per-job cache miss rate or misses per instruction for each MTT. The differences are relative to GEDF for each utilization distribution, and insets (a) through (f) present the results identically to the figures for average-case performance. As with the average-case results, note that the range of the x-axis often varies substantially between figures and within insets of the same figure.

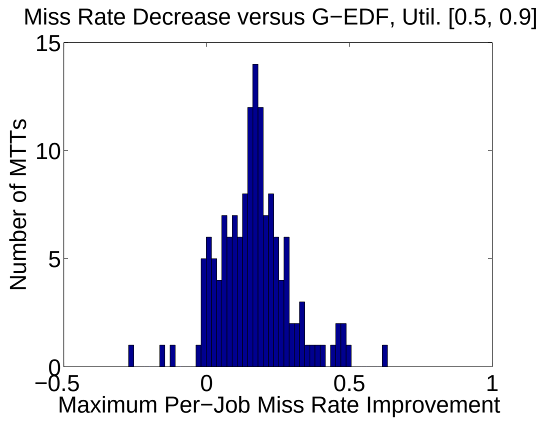
Machine A. On machine A (Figure 6.10), the results are similar to, but considerably more significant than, those observed in the average case. Our scheduler had a substantial impact



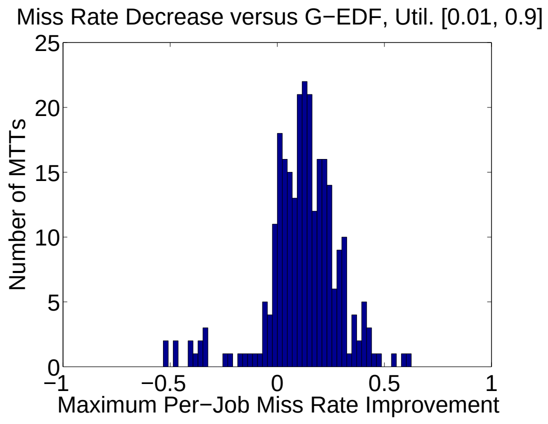
(a)



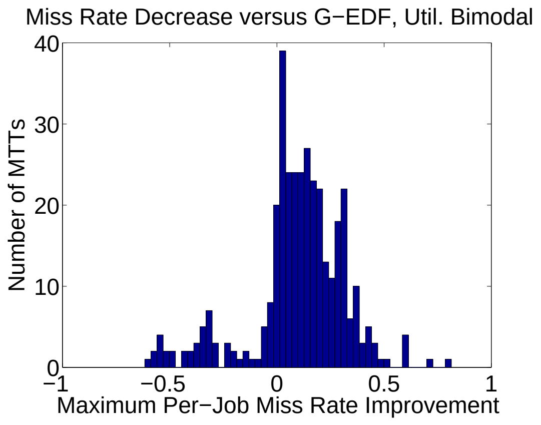
(b)



(c)



(d)



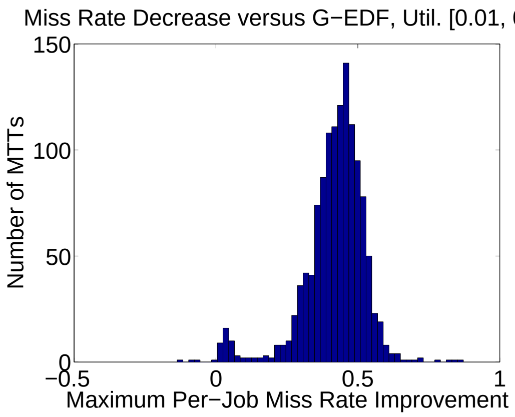
(e)

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
Overall	**-0.754**	**0.064**	**0.930**

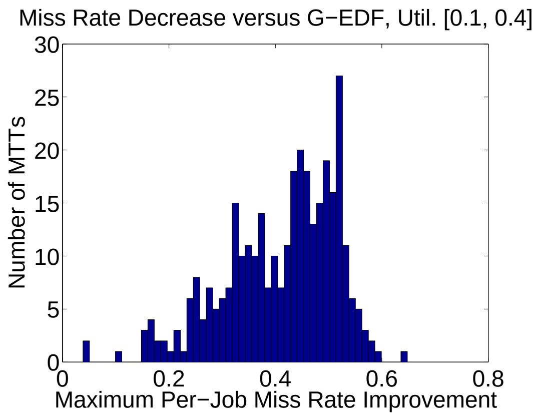
Util. Dist.	$x < -0.01$	$\|x\| \leq 0.01$	$x > 0.01$
[0.01, 0.1]	27.19%	8.84%	63.96%
[0.1, 0.4]	14.98%	5.07%	79.95%
[0.5, 0.9]	4.14%	4.83%	91.03%
[0.01, 0.9]	11.94%	4.85%	83.21%
Bimodal	16.76%	3.35%	79.89%
Overall	**21.19%**	**6.93%**	**71.89%**

(f)

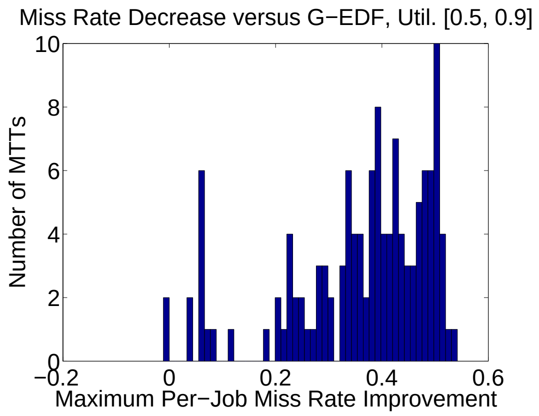
Figure 6.10: The decrease in the maximum (worst-case) per-job shared cache miss rate over each utilization distribution, as compared with GEDF, for machine A (random reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.



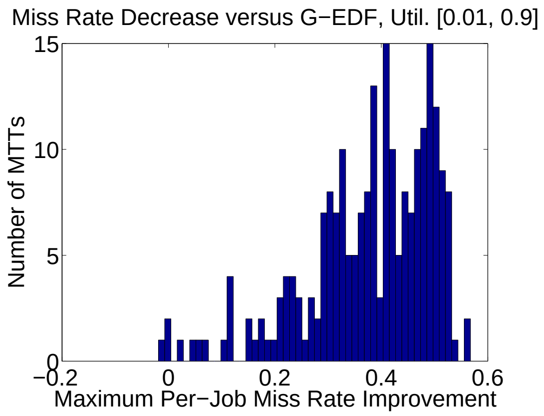
(a)



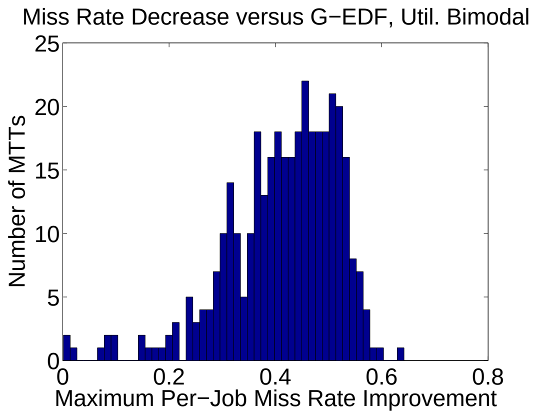
(b)



(c)



(d)



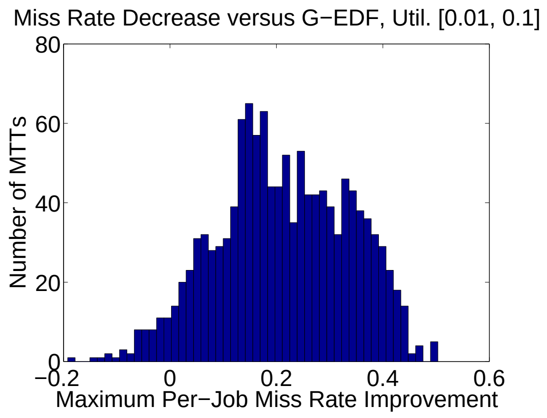
(e)

Util. Dist.	**Min.**	**Avg.**	**Max.**
[0.01, 0.1]	-0.136	0.417	0.872
[0.1, 0.4]	0.039	0.410	0.648
[0.5, 0.9]	-0.011	0.359	0.542
[0.01, 0.9]	-0.019	0.378	0.568
Bimodal	0.001	0.415	0.642
Overall	**-0.136**	**0.409**	**0.872**

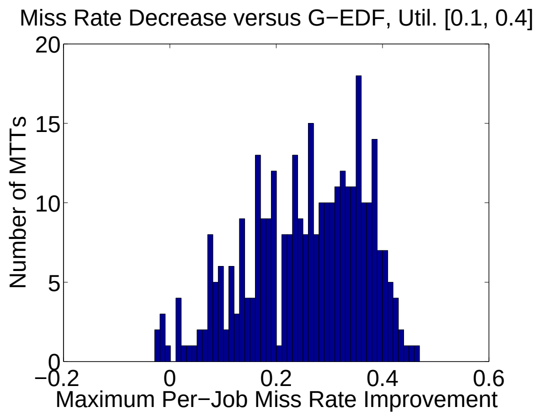
Util. Dist.	$x < -0.01$	$\|x\| \leq 0.01$	$x > 0.01$
[0.01, 0.1]	0.24%	0.08%	99.68%
[0.1, 0.4]	0.00%	0.00%	100.00%
[0.5, 0.9]	0.79%	0.79%	98.41%
[0.01, 0.9]	0.44%	0.89%	98.67%
Bimodal	0.00%	0.56%	99.44%
Overall	**0.22%**	**0.26%**	**99.52%**

(f)

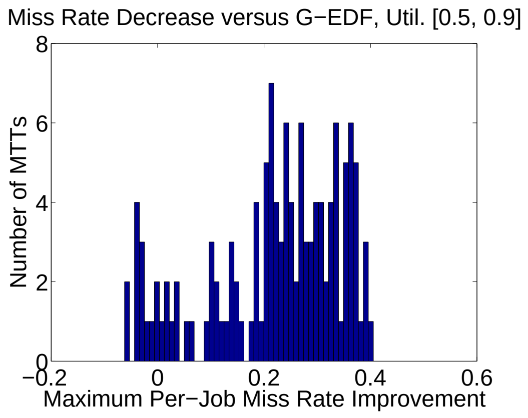
Figure 6.11: The decrease in the maximum (worst-case) per-job shared cache miss rate over each utilization distribution, as compared with GEDF, for machine B (random reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.



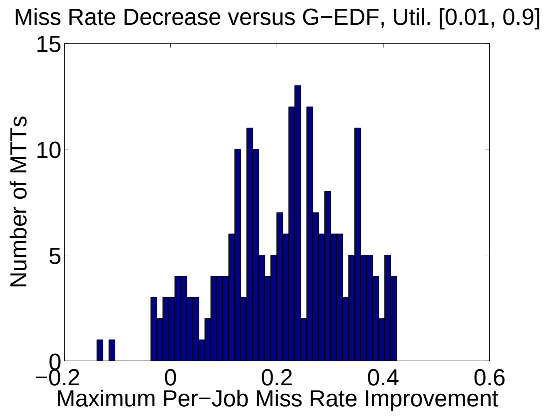
(a)



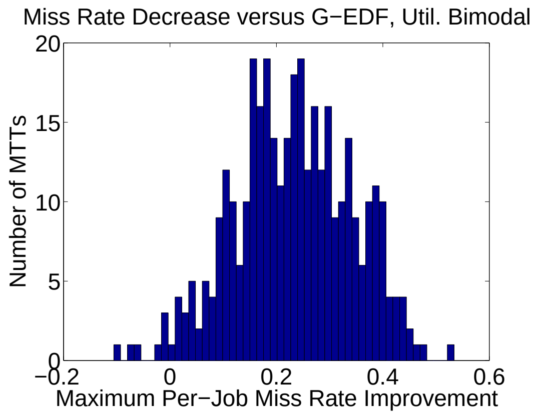
(b)



(c)



(d)



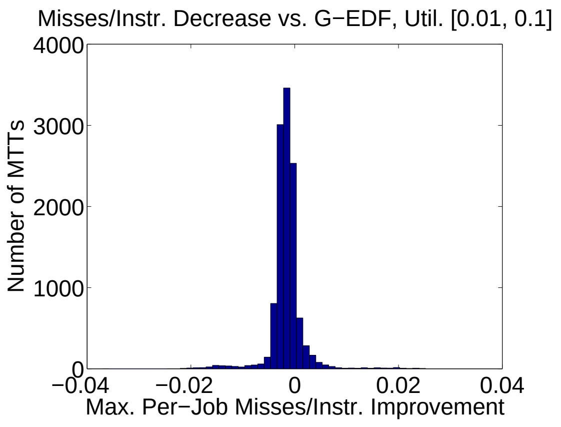
(e)

Util. Dist.	**Min.**	**Avg.**	**Max.**
[0.01, 0.1]	-0.192	0.213	0.503
[0.1, 0.4]	-0.029	0.257	0.470
[0.5, 0.9]	-0.062	0.219	0.405
[0.01, 0.9]	-0.139	0.212	0.425
Bimodal	-0.106	0.232	0.534
Overall	**-0.192**	**0.223**	**0.534**

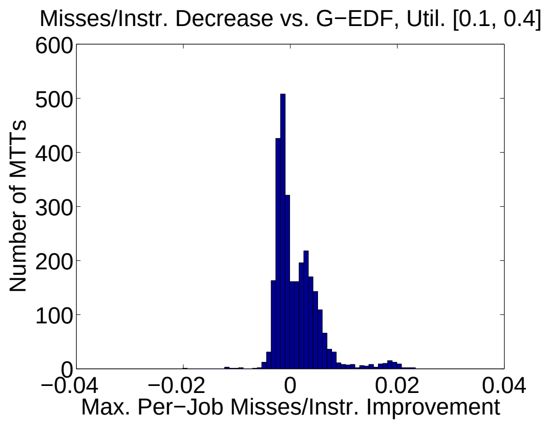
Util. Dist.	$x < -0.01$	$\lvert x\rvert \leq 0.01$	$x > 0.01$
[0.01, 0.1]	3.63%	1.11%	95.26%
[0.1, 0.4]	1.51%	0.30%	98.19%
[0.5, 0.9]	8.73%	1.59%	89.68%
[0.01, 0.9]	4.00%	2.22%	93.78%
Bimodal	1.39%	1.11%	97.50%
Overall	**3.29%**	**1.13%**	**95.58%**

(f)

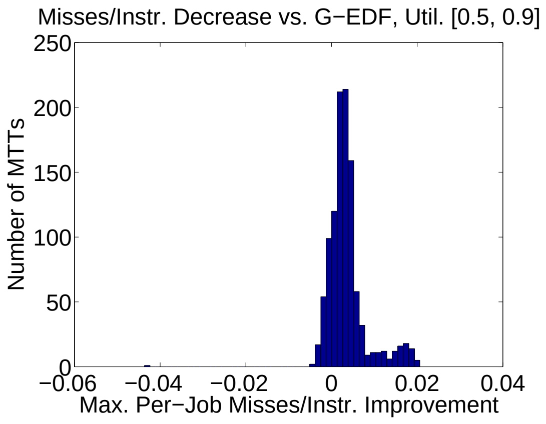
Figure 6.12: The decrease in the maximum (worst-case) per-job shared cache miss rate over each utilization distribution, as compared with GEDF, for machine B (sequential reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.



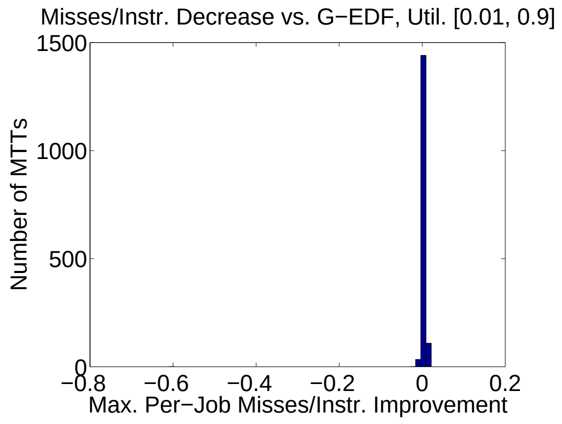
(a)



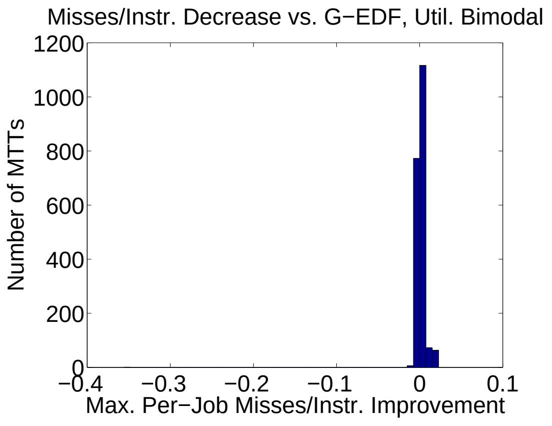
(b)



(c)



(d)



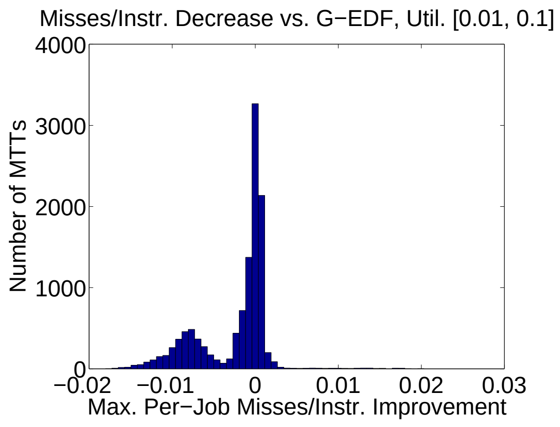
(e)

Util. Dist.	Min.	Avg.	Max.
[0.01, 0.1]	-0.0370	-0.0016	0.0252
[0.1, 0.4]	-0.0201	0.0012	0.0234
[0.5, 0.9]	-0.0436	0.0036	0.0206
[0.01, 0.9]	-0.6096	0.0021	0.0221
Bimodal	-0.3557	0.0017	0.0228
Overall	**-0.6096**	**-0.0002**	**0.0252**

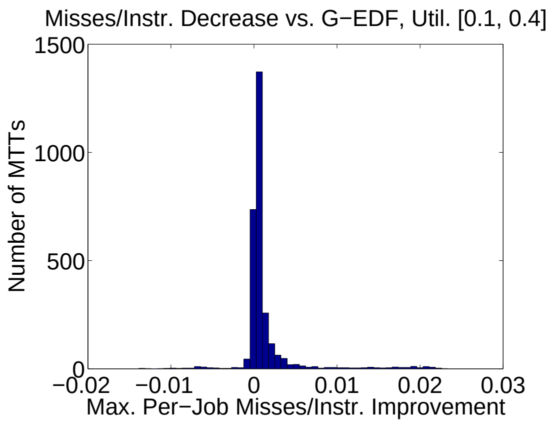
Util. Dist.	$x < -0.01$	$\|x\| \leq 0.01$	$x > 0.01$
[0.01, 0.1]	1.90%	97.24%	0.86%
[0.1, 0.4]	0.17%	96.22%	3.61%
[0.5, 0.9]	0.09%	91.04%	8.87%
[0.01, 0.9]	0.19%	93.51%	6.31%
Bimodal	0.20%	94.73%	5.07%
Overall	**1.22%**	**96.16%**	**2.61%**

(f)

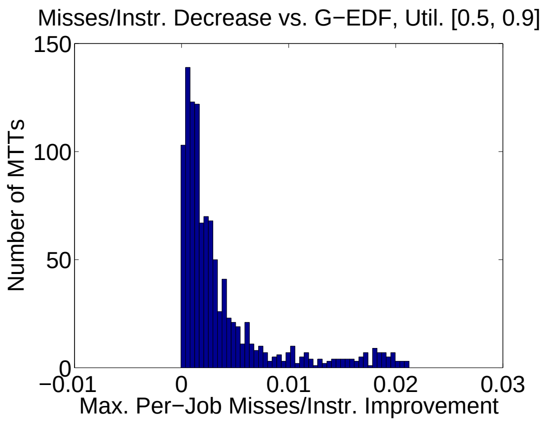
Figure 6.13: The decrease in the maximum (worst-case) per-job shared cache miss rate over each utilization distribution, as compared with GEDF, for machine C (random reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.



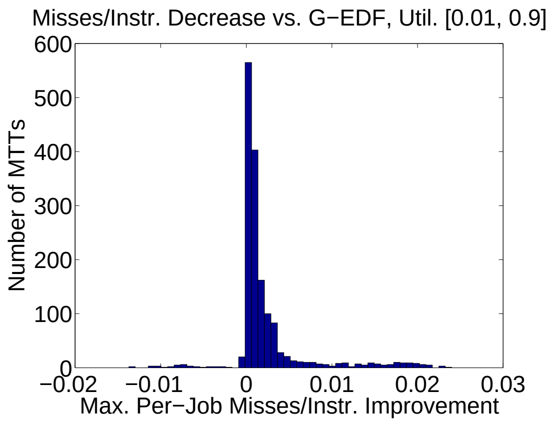
(a)



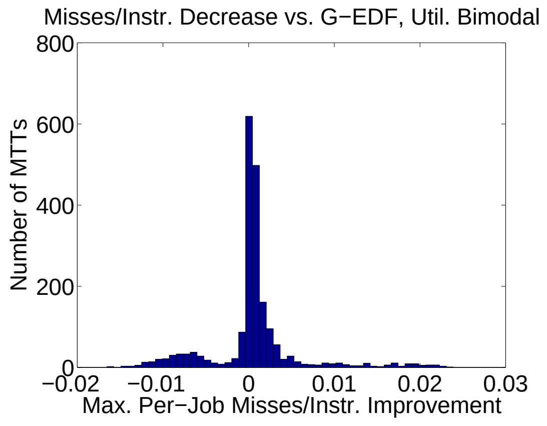
(b)



(c)



(d)



(e)

Util. Dist.	**Min.**	**Avg.**	**Max.**
[0.01, 0.1]	-0.0180	-0.0024	0.0203
[0.1, 0.4]	-0.0139	0.0012	0.0233
[0.5, 0.9]	-0.0001	0.0038	0.0212
[0.01, 0.9]	-0.0137	0.0022	0.0240
Bimodal	-0.0165	0.0005	0.0239
Overall	**-0.0180**	**-0.0008**	**0.0240**

Util. Dist.	$x < -0.01$	$\|x\| \leq 0.01$	$x > 0.01$
[0.01, 0.1]	6.60%	92.85%	0.55%
[0.1, 0.4]	0.24%	96.29%	3.47%
[0.5, 0.9]	0.00%	88.72%	11.28%
[0.01, 0.9]	0.50%	92.50%	7.00%
Bimodal	3.00%	91.98%	5.02%
Overall	**4.40%**	**93.01%**	**2.59%**

(f)

Figure 6.14: The decrease in the maximum (worst-case) per-job shared cache miss rate over each utilization distribution, as compared with GEDF, for machine C (sequential reference pattern). A positive value indicates a performance increase over GEDF, whereas a negative value indicates a decrease. Insets (a) through (e) present histograms for each distribution, while inset (f) presents statistics for each histogram.

on worst-case per-job cache miss rates for the *majority* of MTTs in *every* distribution. This result has one important implication: under our scheduler, worst-case per-job cache miss rates are generally lower, which can directly result in a decrease in worst-case execution times. As a result, on machine A, our scheduler has a much greater potential than GEDF to efficiently utilize the system.

Machine B. On machine B, the use of our cache-aware scheduler resulted in reductions in cache miss rates for the vast majority of MTTs, and *almost never* resulted in a significant miss rate increases, when considering worst-case per-job cache miss rates. Further, the observed miss rate reductions were quite high, especially with respect to the other machines: an average reduction of 40.9% in the worst-case per-job cache miss rate for each MTT under the random reference pattern (as seen in Figure 6.11(f)), and an average reduction of 22.3% in the worst-case per-job cache miss rate for each MTT under the sequential reference pattern (as seen in Figure 6.12(f)), over all utilization distributions. Unlike the average-case results, the distributions were not bimodal. One reason that miss rate reductions were higher under random reference patterns might be that such patterns are more likely to bring their entire footprint into the cache earlier, creating a greater potential for our scheduler to reduce cache miss rates by avoiding thrashing. To see this, recall that if a memory location is brought into the cache as the result of a cache miss, then the entire cache line containing the location of interest must be brought into the cache as well. This means that a sequential reference pattern will not bring its entire footprint into the cache until it has referenced nearly every location in its footprint, since it references those locations sequentially. On the other hand, a random reference pattern need only reference a single location in any cache line to bring the entire line into the cache; as a result, a random pattern is likely to reference at least one location within each cache line considerably earlier, resulting in the footprint being brought into the cache earlier. Overall, regardless of the reference pattern, and for the same reasons discussed for machine A, our scheduler has a much greater potential than GEDF to efficiently utilize machine B.

Machine A

Algorithm	Average	Maximum
GEDF	0.00	18.76
CA-SCHED	12.26	1037.69

Machine B

Algorithm	Average	Maximum
GEDF	0.00	46.29
CA-SCHED	28.48	979.16

Machine C

Algorithm	Average	Maximum
GEDF	0.00	93.23
CA-SCHED	47.89	3343.91

Table 6.9: Deadline tardiness results for each machine (in ms).

Machine C. As in the average case, changes in misses per instruction were least significant in machine C as compared to the other machines—see Figures 6.13 and 6.14. However, when we look at only the results for machine C, reductions in misses per instruction were considerably more significant than in the average case. While it is still the case that no significant reductions in misses per instruction were observed for the majority of MTTs, more substantial gains were observed for the MTTs when a benefit was seen, as compared to the average case.

6.2.2.3 Deadline Tardiness

Table 6.9 shows the average and maximum observed deadline tardiness of jobs executing under both GEDF and our scheduler (denoted CA-SCHED). Our scheduler resulted in higher average tardiness than GEDF, but the difference is not overly substantial, being well under the smallest period of any MTT for machines A and B, and slightly higher than the smallest period on machine C (recall that the minimum period was different on machine A than on machines B and C). Worst-case tardiness is much higher under our scheduler, but is artificially inflated by our large task periods, especially on machine C, which was heavily loaded relative to the other machines, since it had 32 logical CPUs. Nevertheless, as stated earlier in Section 6.1.2, a combination of early releasing and buffering can be employed to "hide" tardiness from an end user, as discussed in Chapter 4. In such a case, these values are reasonable—a buffer of several seconds could be typical for many types of (multimedia) applications.

6.2.2.4 Scheduling Overhead

Finally, we used *Feather-Trace* [14] to measure scheduling-related overheads under both GEDF 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. On machine A, we collected over 37 million overhead measurements (over 1 GB of raw data) during these experiments, and on machine B, we collected nearly 39 million overhead measurements (also over 1 GB of raw data). On machine C, we collected significantly more data due to the large number of logical CPUs and the large sizes of the task sets: over 434 million overhead measurements and over 14 GB of raw data.

Table 6.10 presents average- and worst-case overheads. (These overheads were defined earlier in Chapter 2.) Note that, in repeated measurements of some overhead, a small number of samples may be outliers. As overheads for both GEDF and our cache-aware scheduler must be determined experimentally using a real operating system running on real hardware, these outliers may be due to a variety of factors that are beyond our control. Such factors include warm-up effects in the instrumentation code, the many sources of unpredictability within Linux (such as interrupt handlers and priority inversions within the kernel), and the lack of determinism on the hardware platforms on which Linux typically runs (and on the machines that we used in our experiments). The last point is especially a concern, regardless of the operating system, on multiprocessor platforms. Indeed, research on timing analysis has not matured to the point of being able to analyze complex interactions among tasks due to atomic operations, bus locking, and bus and cache contention. In light of the fact that such outliers may exist, the top 1% of measurements were discarded. The same approach has been used previously, with similar justification, in both [15] and [16].[5]

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 GEDF, where the times at which jobs are released, and scheduling decisions are

[5]We should note that, despite these observations, there are now many advocates of using Linux to support applications that require some notion of real-time execution, under which "soft real-time" is usually interpreted to mean that deadline tardiness on average remains bounded, even if some tasks occasionally misbehave due to effects beyond our control. This notion of soft real-time is exactly that described in Chapter 2.

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 GEDF on *any* of the three machines, implying that overheads will not offset reductions in cache miss rates.

6.2.3 Multimedia Application Performance

We next present the results of an experiment conducted on machine A to compare our cache-aware scheduler to GEDF 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$^{\mathrm{RT}}$, determined the frame rate.

Multiple copies of the same segment of high-quality video (taken from [13]) 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 of the amount of execution time that an application needed to process a frame of this video, we decided to assign an execution cost of 14 ms to each application (though we note that execution costs vary widely per frame). (Our observations came from running the application multiple times, each time specifying a different execution cost.) We also observed that the maximum WSS of the application was roughly 3 MB per frame. Therefore, co-scheduling three of more MTTs has the potential to thrash the 8192K shared cache on machine A. Each application was represented as a single-threaded MTT.

In our experiments, we ran one, six, and twelve *mplayer* applications under both GEDF and our scheduler. Table 6.11 shows the measured worst-case job execution times for each case. (Note that, in the case of twelve applications, the system is overloaded since $(14/41) * 12 \approx 4.098 > 4$, although execution times are often lower in practice, which allows all twelve applications to be supported. Additionally, and also in the case of twelve applications, pairs of *mplayer* applications were forced to share the same video copy due to RAM disk space constraints.) When one video is scheduled in isolation, the execution time under both schedulers

Machine A

Algorithm	Tick AVG	Sched. AVG	Context Sw. AVG	Release AVG
GEDF	1.166	3.358	0.953	4.592
CA-SCHED	2.762	2.052	0.972	—

Algorithm	Tick WC	Sched. WC	Context Sw. WC	Release WC
GEDF	1.847	11.478	1.532	8.922
CA-SCHED	14.445	6.278	1.507	—

Machine B

Algorithm	Tick AVG	Sched. AVG	Context Sw. AVG	Release AVG
GEDF	4.179	7.011	2.125	7.410
CA-SCHED	2.803	1.761	1.052	—

Algorithm	Tick WC	Sched. WC	Context Sw. WC	Release WC
GEDF	7.602	34.005	3.483	13.869
CA-SCHED	9.153	6.183	1.938	—

Machine C

Algorithm	Tick AVG	Sched. AVG	Context Sw. AVG	Release AVG
GEDF	4.283	50.001	4.661	48.413
CA-SCHED	62.858	8.633	3.783	—

Algorithm	Tick WC	Sched. WC	Context Sw. WC	Release WC
GEDF	8.890	370.023	18.647	214.613
CA-SCHED	361.983	31.890	7.103	—

Table 6.10: Average and worst-case kernel overheads for each machine, in μs.

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

Table 6.11: Worst-case execution times for *mplayer* applications (in ms).

is about 14 ms. Under GEDF, 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 for each application is roughly 6-7% less than GEDF in the case of six and twelve applications. As a result, since the system is able to support twelve video applications under GEDF, the same system should be capable of supporting an additional *mplayer* application under our cache-aware scheduler. If a large number of these systems were used within a cluster of systems that support a multimedia server workload, then the ability to support an additional application on each machine could be quite significant.

Interestingly, the memory reference patterns of these *mplayer* applications are arguably more sequential than random (since the memory where a video frame is stored is often scanned in predictable manner, such as strides across memory, instead of randomly), yet we still see a significant performance improvement when using our scheduler on machine A. 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 on that machine.

6.3 Conclusion

In this chapter, we presented the results of experiments wherein the performance of our cache-aware scheduler was evaluated. First, we presented a set of experiments, conducted within SESC, that showed that, when the correct heuristic is selected, system performance can improve substantially as compared to GEDF. These experiments also allowed us to identify a "best-performing" heuristic, which was implemented within LITMUS$^{\mathrm{RT}}$ along with our cache profiler. Our second set of experiments then evaluated this LITMUS$^{\mathrm{RT}}$ implementation in terms if its capability to generate accurate WSS estimates, and its performance as compared to GEDF. We found that the accuracy of the profiler depended heavily on the performance monitoring features that were provided by hardware; however, when sufficient hardware support existed, our profiler was typically quite accurate. We also found that, when our cache-aware scheduler was used on platforms where the profiler was accurate, reductions in cache miss rates were often significant. Further, the overheads of our scheduler were roughly equivalent to those in GEDF. The "true" cost for achieving these reductions in cache miss rates was an increase in deadline tardiness; however, with a combination of early-releasing and buffering, this tardiness potentially could be hidden from an end user. Finally, we conducted experiments with a simulated multimedia workload, and found that execution costs were lower as compared to GEDF when our scheduler was used; these lower costs could allow a larger workload to be supported with the same hardware.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this dissertation, we extended research on multiprocessor real-time systems to support multicore platforms, specifically by designing a cache-aware real-time scheduler that reduces miss rates and avoids thrashing for shared caches on such platforms while ensuring real-time constraints. Prior work in the area of cache-aware scheduling for multicore platforms did not address support for real-time workloads, and prior work in the area of real-time scheduling did not address shared caches on multicore platforms. Further, the cache-aware real-time scheduling methods that we created earlier, described in Chapter 3, were only partial solutions and had not been evaluated within the context of a real operating system.

As multicore platforms are quickly becoming ubiquitous within many domains, creating scheduling policies to address bottlenecks that are common in these platforms, such as shared caches, is necessary to ensure acceptable system performance and permit a state-of-the-art real-time scheduler to remain relevant. Effectively addressing a bottleneck such as the shared cache can enable a larger real-time workload to be supported on a multicore platform, or may allow hardware-related costs (related to processor components, energy, or chip area) to be reduced.

7.1 Summary of Results

In Chapter 1, we presented the following thesis statement, which was to be supported by this dissertation.

Multiprocessor real-time scheduling algorithms can more efficiently utilize multicore platforms when scheduling techniques are used that reduce shared cache miss *rates. Such techniques can result in decreased execution times for real-time tasks, thereby allowing a larger real-time workload to be supported using the same hardware, or enabling costs (related to hardware, energy, or chip area) to be reduced.*

In support of this thesis statement, we have presented a cache-aware scheduler, consisting of both a set of cache-aware scheduling policies, embodied in a single heuristic, and a cache profiler that allows the cache behavior of each MTT to be quantified at runtime. The choice of exactly which heuristic to implement was supported by experimentation with roughly one hundred combinations of policy choices within an architecture simulator; however, all heuristics operate similarly in that they attempt to influence the co-scheduling of MTTs through job promotions so that cache miss rates are reduced. Namely, tasks within the same MTTs are co-scheduled to facilitate cache reuse, and the co-scheduling of tasks from different MTTs is avoided when it would cause cache thrashing.

Our cache profiler quantifies the cache behavior of each MTT as a per-job WSS; these WSSs are used by the heuristic to make scheduling decisions. Per-job WSS estimates are obtained during execution using performance counters. By profiling MTTs during execution, the need to profile MTTs offline is eliminated, which might make profiling less of an inconvenience for certain types of workloads. In experiments, we found that our profiler is quite accurate when sufficient hardware support is provided; otherwise, accuracy varied considerably depending on the nature of the limited support that was available.

As both a "proof of concept" that our cache-aware scheduler was viable in practice, and to allow the empirical evaluation presented in Chapter 6 to be conducted, we implemented our scheduler within Linux. In experiments, our cache-aware scheduler often outperformed GEDF under a wide variety of real-time workloads and machine architectures. This is best demonstrated in the experiments that were conducted on machines A and B in Chapter 6. Additionally, since overheads were found to be comparable to GEDF, we would not expect the scheduling-related overheads of our scheduler to offset any reductions in cache miss rates when compared with other scheduling approaches. We also described how deadline tardiness, while higher under our method, can often be managed through a combination of early-releasing jobs and buffering results. Lastly, in an effort to directly support the second half of the thesis statement of this dissertation, we conducted experiments involving a multimedia server workload, and showed that the use of our cache-aware scheduler allowed execution costs to be reduced, thereby allowing the size of the workload to be increased. Overall, the presented evaluation suggests that eliminating thrashing and reducing miss rates in shared caches should be first-class concerns when designing real-time scheduling algorithms for multicore platforms with shared caches.

Finally, we consider the introduction of the MTT abstraction in this dissertation to be a contribution in itself, as it allows a notion of concurrency to be incorporated into the periodic and sporadic task models, which traditionally handle only sequentially-executing tasks. If the chip manufacturing industry continues to adhere to the multicore paradigm (which is likely, given current projections), then increasing per-chip core counts will be the primary way in which the potential processing power of chips will be increased. In this case, exploiting the available parallelism within these chips will be essential to achieving performance gains from the use of new processor technology. As end users continue to demand applications with greater features and sophistication, or of higher quality, such as improved video quality in multimedia applications, there will be increasing pressure to address issues related to concurrency at some level within virtually all areas of computer science.

7.2 Other Contributions

We now describe two other contributions by the author during his tenure at The University of North Carolina at Chapel Hill that are not directly related to the research described in this dissertation, and were not discussed in prior chapters of this dissertation.

7.2.1 Real-Time Scheduling on Asymmetric Platforms

In [20], we devised an approach for supporting soft real-time periodic tasks in Linux running on an asymmetric multicore platform. In such a platform, multiple processing cores are placed on one chip or several chips, and all processing cores are capable of executing the same instruction set, but at potentially different performance levels. Such a platform is often more flexible than symmetric platforms when considering the types of workloads that can be supported, since acceptable performance can be achieved for both highly-parallel and highly-sequential applications, instead of for only one of these two classes of applications, when a reasonable mix of core counts and speeds are selected. In this work, we first identified deficiencies in Linux in supporting periodic real-time tasks, highlighting when these deficiencies were particularly problematic on an asymmetric platform, and then overcame each deficiency by making a small number of modifications to Linux. In experiments, we found that we were able to effectively support periodic real-time workloads in our modified version of Linux.

In the next portion of this work, we investigated ways in which performance could be improved for *non-real-time* tasks when they are executing alongside real-time tasks on an asymmetric platform. In many environments in which Linux runs, the performance of non-real-time tasks is at least as important as ensuring real-time constraints. In our approach, real-time tasks are assigned to the slowest available cores that can support their execution requirements—doing so leaves faster cores available for non-real-time tasks, for which response times are typically a greater concern (assuming that the timing constraints of real-time tasks are met). Additionally, a high-priority server is placed on each core to explicitly reserve a share of the core for non-real-time execution. These servers allow response times to be reduced, especially when tasks need short bursts of processing capacity, such as might be the case for an interactive application. We found that the addition of a small server to each core often had a significant impact on response times, while having little impact on the size of the real-time workload that could be supported.

7.2.2 The LinSched Linux Scheduler Simulator

LinSched [21] is a user-space program that hosts the Linux scheduler, and is intended to assist with the early stages of scheduler development. It was created in response to the steep learning curve experienced by those new to Linux kernel development—a spike in new developers is likely due to a recent resurgence of interest in the Linux scheduler. LinSched has recently been made publicly available at `http://www.cs.unc.edu/~jmc/linsched`.

7.3 Feedback to Chip Designers

To enable the performance improvements demonstrated in this dissertation 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 in Chapter 6, the effectiveness of such counters has declined in moving from the Intel Xeon chip in machine B (and its Core 2 counterpart) to the new Core i7 chip in machine A. Further, support for tracking misses and references in the shared cache is seriously lacking in the Sun UltraSPARC T1 processor (machine C)—this support is only marginally better in its successor, the UltraSPARC T2. We view these design choices 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.) In doing so, performance counters would become primary features of the chip, rather than secondary features that are supported less rigidly. As a result of becoming primary features of the chip, operating systems designers might take these features more seriously, and begin incorporating them into their designs as was done in this dissertation, hopefully to the benefit of end users.

7.4 Future Work

There are several ways in which the work described in this dissertation could be extended, as we discuss next.

Incorporating blocking into the scheduler. There are two relatively common scenarios where jobs may be forced to block. First, blocking can occur due to the synchronization required for jobs to safely access critical sections. Second, jobs can block due to precedence constraints, such as those that would be required to schedule jobs in a pipelined manner—Liu and Anderson [45] have recently explored issues related to the pipelined execution of real-time tasks, including the utilization loss resulting from the need to wait due to precedence constraints. If jobs block when our cache-aware scheduler is used, then it may cause assumptions related to how jobs are promoted and scheduled to be violated. For instance, when an urgent job blocks, it can prevent other jobs from being promoted, which could effectively disable cache-aware scheduling until the urgent job completes. We could make an exception to this rule for blocking jobs, and allow another job to be promoted, but this could reduce the effectiveness of the scheduler if a large number of jobs become promoted, and would greatly complicate a LITMUS$^{\text{RT}}$ implementation. As such, we need to develop methods that will allow job blocking to occur within our scheduler without having a significant impact on its effectiveness.

Dynamic real-time workloads. In the environment in which our scheduler was implemented, dynamic real-time workloads are likely to be common. Therefore, we must ensure that our scheduler can handle such workloads effectively. There are at least three types of dynamic behavior that we would like to consider: new task arrivals, exiting tasks, and changes in MTT WSSs over time. Under the cache-aware scheduler that we implemented within LITMUS$^{\text{RT}}$, adding new tasks at runtime, either as stand-alone tasks or to an existing MTT, is possible as long as doing so does not result in an over-utilized system; however, it will present some challenges for the profiler. For a stand-alone task, we can obtain WSS estimates using the same bootstrapping process that was used for all other tasks, described in Chapter 5. For a task that is part of an MTT, the task will initially be assigned the same WSS as all other tasks in that MTT, which could be problematic, as the addition of a new task to that MTT could result in an increased WSS for that MTT. Similarly, when a task exits the system and is no longer part of the real-time workload, and that task was part of an MTT that still contains at least one real-time task, then the WSS of that MTT may have decreased (although in this case, the higher WSS estimate is safe to use). Ultimately, the issue in these cases is that the WSS of an MTT changed over time, whereas we assumed in Chapter 5 that MTT WSSs do not change. A conservative solution to this problem would be to require an MTT to undergo the bootstrapping process again whenever the MTT gains or loses a task; in this case, the existing WSS can be used until it is discarded during bootstrapping (as it is probably a better estimate than zero). The effectiveness of such an approach depends on how frequently MTTs gain and lose tasks. In fact, such a method could be employed for MTTs where WSS changes over time even when the task count does not change—an MTT would be required to undergo bootstrapping whenever sufficient *divergence* between two consecutive measurements is detected (the opposite of the convergence test described in Chapter 5). Again, this may not be particularly effective if changes occur so frequently that the MTT is always bootstrapping.

Analytical cache thrashing guarantees. While our work is analytically "grounded" with respect to ensuring soft real-time guarantees, there is not yet any analytical work that would allow *cache thrashing guarantees* to be made under our scheduler. For example, it might be useful to have a test that takes as input a task set and a cache size, and outputs "yes" if the task set would cause cache thrashing when scheduled with our cache-aware scheduler (where thrashing is assumed to occur in the same scenarios where it was assumed to occur in Chapters 4 and 5), and "no" otherwise. A "no" answer would allow lower worst-case execution times to be used, since such an answer guarantees that cache thrashing will be avoided with our scheduler, allowing for effective use of the cache. If similar tests were provided for other algorithms, then it would allow us to analytically compare different algorithms in terms of expected cache miss rates and potential for cache thrashing, and incorporate any differences into an overall comparison of scheduling algorithms.

Handling frequent job priority changes. We want to allow run queues to handle frequent changes in job priority more efficiently; that is, in a way that results in lower scheduling overheads. Doing so might make a greater set of the policies from Chapter 4 feasible in practice—in particular, the lost-cause policies that had a substantial impact on cache miss rates, as we saw in Chapter 6. As a first step in this direction, we could focus on identifying and supporting scenarios where the manner in which job priorities can change is still somewhat restricted, but in such cases, designing an efficient implementation is easier. For instance, we could first identify those scenarios that could be supported in a straightforward manner through simple run-queue-related pointer manipulations. Work to support frequent job priority changes might also enable results such as those in [44], which showed that bounded deadline tardiness can be ensured for global scheduling approaches even when jobs can undergo frequently priority changes (within a bounded range), to have a greater practical impact.

Preventing caching. Tasks should be prevented from caching data that they do not reuse. Otherwise, tasks with a large memory footprint and little reuse, such as those associated with data streaming applications, can pollute or thrash a shared cache, causing data belonging to other tasks to be evicted before any reuse can occur. Additionally, polluting tasks receive little benefit themselves from caching data. Some hardware architectures provide a way to prevent certain memory pages from being cached (*e.g.*, recent Intel architectures [39]). For such an architecture, we would like to automatically identify pages that are not being reused, and prevent them from being cached. This might be accomplished by counting the number of times that a page is referenced, and allowing data from that page to be cached only when a certain reference count for the page is reached. By preventing data from being cached when it is not reused, the cache footprint of a task would more closely approximate its "real" working set, which may cause the WSS estimates provided to the cache-aware scheduling heuristics to decrease and result in an increased ability to avoid cache thrashing.

Multiple shared caches. We want to provide better support within our scheduler for handling multiple shared caches, using the clustered approach that was described briefly in Chapter 2. A clustered approach would require either an offline bin-packing of tasks onto clusters, or an online admission control protocol, where a new task is placed onto an available cluster, and the utilization of each cluster is tracked.

Buffering implementation. We have stated that soft real-time scheduling approaches are sufficient for hard real-time systems if early-releasing and buffering can be employed. Developing a system where such buffering is implemented correctly and in a low-overhead manner is an interesting and substantial area of future work, for two reasons. First, it is necessary to account for the impact of buffered data on the cache, either by including the buffered data in the WSS of each MTT, or by preventing buffered data from being cached. Second, it is necessary to ensure that buffered data can be quickly supplied to a job upon its "official" release (note that data will be more quickly supplied if it is cached, which further complicates the first issue). Otherwise, it might be necessary to divide jobs into two portions: one that can execute prior to its release time, and one that must execute after its official release time. Dividing jobs makes the use of early-releasing and buffering to hide tardiness considerably more difficult, and additional analytical results will be required to show that performing such a division will still allow all deadlines to be met.

Other hard real-time support. Considering hard real-time support again, we would like to determine if a variant of our scheduler could be appropriate for systems with hard real-time requirements, even when buffering and early-releasing cannot be employed. In this case, the first step would be to design a hard real-time schedulability test for our cache-aware scheduler. Such a schedulability test might be derived from a test concerning analytical tardiness bounds, by determining whether the tardiness bound is zero for all tasks. Unfortunately, analytical tardiness bounds are often very conservative, and most such bounds include in the bound the execution cost of the task for which a tardiness bound is being computed, which means that the bound cannot be zero by definition.

Safety-critical real-time tasks. Thus far, we have assumed that the profiler that is used in our cache-aware scheduler could be used in hard real-time systems without modification. However, in a *safety-critical* hard real-time system where worst-case execution costs must be carefully determined, the WSS estimates that are computed by our profiler will not be sufficient. This is primarily because we assume that conflict misses are negligible and therefore can be ignored, due to the high set associativity of the low-level shared caches that are of interest in this dissertation. While this is generally true in high set associativity caches, where the associativity is often large enough to mask conflict misses, there are cases where a job may still experience a significant number of conflict misses in a shared cache even when the cache has a high set associativity. For example, in a system where a 16-way set associative cache is shared by four cores, and all four cores are utilized, the "effective" set associativity for a job that is running in that system might be four ways or less, depending on the pressure that is placed on the cache by other running jobs. When the number of ways is reduced, conflict misses again have the potential to become a significant proportion of the cache misses that are experienced by a job. The violation of the assumption that conflict misses are negligible has a two-fold impact on our cache-aware scheduler. First, conflict misses will inflate the miss counts recorded by the performance counters, thereby resulting in inflated WSS estimates. Second, when these WSS estimates are used in our cache-aware scheduler, thrashing may sometimes occur due to conflict misses, even when jobs are scheduled such that the sum of the WSSs of all scheduled jobs does not exceed the shared cache size (which only ensures that *capacity* misses are avoided). Thus, methods that better account for conflict misses need to be incorporated into our scheduler when safety-critical tasks are supported. It might be easier to develop such methods if cache partitioning is employed (when supported by hardware), as doing so may allow conflict misses due to other co-scheduled jobs to be avoided (but not conflict misses that would occur in isolation), thus providing additional isolation so that conflict misses can be better accounted for during timing analysis.

Task migrations and private caches. In the work presented in this dissertation, we have been primarily concerned with shared cache miss rates and thrashing; private caches have mostly been ignored. During a task migration, overhead will be incurred due to a loss of private cache affinity, which must be incorporated into schedulability analysis by inflating execution costs, as we discussed in Chapter 2. While such execution cost inflation is typically sufficient for handling the impact of task migrations in the soft real-time systems considered in this dissertation, it may not be sufficient for safety-critical systems, since worst-case migration overheads may be too high for execution cost inflation to be a viable technique when considering schedulability. Sarkar *et al.* [59] recently considered the issue of private caches during task migrations. In experiments, they observed task migration overheads that were as high as 56.6% of the execution cost of a task. In such scenarios, the use of execution cost inflation to account for migration overheads would require significant system under-utilization in the best case. Motivated by this, Sarkar *et al.* proposed a hardware-based mechanism that reduces task migration overheads by proactively pushing data from the source core to the target core during a migration before that data is requested by the task when it resumes execution on the target core. Thus, migration overheads due to a loss of private cache affinity are reduced by anticipating the data needs of the task after migration and moving the data to the target core. The presence of such a mechanism can reduce worst-case migration overheads, thus allowing lower worst-case execution costs to be used. If hardware support like that presented by Sarkar *et al.* is not available, then more precise timing analysis will be required to account for the effects of private caches during task migrations.

Support for non-real-time tasks. The cache-aware real-time scheduler described in this dissertation may have some applicability to non-real-time workloads, if the tasks of which these workloads are comprised can be explicitly allocated a share of the processing resources of the system, or can otherwise be represented as recurrent tasks. In this case, more sophisticated cache profiling may be required, as non-real-time tasks are likely to violate the assumptions made by the profiler in Chapter 5, particularly the assumption that such tasks perform roughly the same operations every job on similarly-sized sets of data, and therefore per-job WSSs will remain relatively stable. If the profiler is modified so that it is better able to handle MTTs where WSSs change over time, as discussed earlier in this section, then a more sophisticated profiler may not be necessary; otherwise, we could devise methods to identify recurrent behavior within non-real-time tasks, particularly as related to memory reference patterns, and take that behavior into account when profiling non-real-time tasks so as to increase the accuracy of our WSS estimates.

Comparison to other scheduling approaches. Finally, we would like to compare our cache-aware scheduler to other approaches besides GEDF. In particular, it would be interesting to compare it to NP-GEDF scheduling, as NP-GEDF is comparable to GEDF in terms of most overheads and deadline tardiness, but executes jobs non-preemptively, which may reduce overheads that are related to a loss of cache affinity. Another interesting policy for comparison would be *first-in-first-out* scheduling, otherwise known as FIFO. Under FIFO scheduling, jobs execute non-preemptively in the absence of tardiness, overheads are likely to be lower than in EDF policies, and tardiness is higher than in EDF policies, but probably lower than in our cache-aware scheduler.

BIBLIOGRAPHY

[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] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

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

[4] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 179–190. IEEE, 2006.

[5] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43. IEEE, 2000.

[6] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

[7] Azul Systems. Azul compute appliances. http://www.azulsystems.com/products/compute_appliance.htm, 2008.

[8] 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.

[9] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[10] A. Batat and D. Feitelson. Gang scheduling with memory considerations. *Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium*, pages 109–114. IEEE, 2000.

[11] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 20–27. IEEE, 2004.

[12] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Proceedings of the SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 169–180. ACM, 2005.

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

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

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

[16] 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? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE, 2008.

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

[18] J. Calandrino and J. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 194–204. IEEE, 2009.

[19] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–256. IEEE, 2007.

[20] J. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 101–110. IEEE, 2007.

[21] J. Calandrino, D. Baumberger, T. Li, J. Young, and S. Hahn. LinSched: The Linux scheduler simulator. In *Proceedings of the ISCA 21st International Conference on Parallel and Distributed Computing and Communications Systems*, pages 171–176. ISCA, 2008.

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

[23] C. Cascaval, L. DeRose, D. Padua, and D. Reed. Compile-time based performance prediction. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 365–379. IEEE, 1999.

[24] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[25] H. Chen, K. Li, and B. Wei. Memory performance optimizations for real-time software HDTV decoding. *Journal of VLSI Signal Processing*, 41(2):193–207, 2005.

[26] P. Denning. Thrashing: Its causes and prevention. In *Proceedings of the AFIPS 1968 Fall Joint Computer Conference*, pages 915–922. ACM, 1968.

[27] P. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[28] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2006.

[29] U. Devi and J. Anderson. Improved conditions for bounded tardiness under EPDF Pfair multiprocessor scheduling. *Journal of Computer and System Sciences*, to appear.

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

[31] Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. RTAI - the RealTime Application Interface for Linux from DIAPM. http://www.rtai.org, 2007.

[32] 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.

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

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

[35] M. Garey and D. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.

[36] M. Hamadoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, 1995.

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

[38] Intel Corporation. Intel digital home software vision guide 2007. http://isdlibrary.intel-dispatch.com/isd/42/SSPR_DigHomeGuide_2007.pdf, 2006.

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

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

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

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

[43] 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.

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

[45] C. Liu and J. Anderson. Supporting pipelines in soft real-time multiprocessor systems. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 269–278. IEEE, 2009.

[46] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[47] C. Lu, J. Stankovic, S. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1-2):85–126, 2002.

[48] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[49] A. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-Time Environments*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1983.

[50] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. ACM, 1996.

[51] Open Source Automation Development Lab. OSADL Project: Realtime Linux. http://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html.

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

[53] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the International Conference on Computer-Aided Design*, pages 67–72. IEEE, 2006.

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

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

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

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

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

[59] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *Proceedings of the SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 80–89. ACM, 2009.

[60] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 66–76. ACM, 2002.

[61] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198. ACM, 2002.

[62] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*, pages 51–59. IEEE, 2003.

[63] J. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3):237–253, 2004.

[64] J. Stankovic and K. Ramamritham. The Spring kernel: A new paradigm for real-time systems. *IEEE Computer*, 8(3):62–72, 1991.

[65] J. Stohr, A. Bulow, and G. Farber. Using state of the art multiprocessor systems as real-time systems—the RECOMS software architecture. *Work-in-progress proceedings of the 16th Euromicro Conference on Real-Time Systems*. IEEE, 2004.

[66] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 117–128. IEEE, 2002.

[67] Sun Microsystems. UltraSPARC T1 processor supplement to UltraSPARC architecture 2005. http://www.sun.com/processors/documentation.html, 2006.

[68] The Open Group. The Realtime Extension. http://www.unix.org/version2/whatsnew/realtime.html, 1998.

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

[70] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *Multimedia Systems*, 4(4):197–208, 1996.

[71] V. Yodaiken and M. Barabanov. A real-time Linux. *Proceedings of the Linux Applications Development and Deployment Conference*, pages 1–9. USENIX, 1997.