Reducing Cache-Associated Context-Switch Performance Penalty Using Elastic Time Slicing

Nagakishore Jammula
Georgia Institute of Technology
nagakishore@gatech.edu

Moinuddin Qureshi
Georgia Institute of Technology
moin@ece.gatech.edu

Ada Gavrilovska
Georgia Institute of Technology
ada@cc.gatech.edu

Jongman Kim
Georgia Institute of Technology
jkim@ece.gatech.edu

Abstract

Virtualization enables a platform to have an increased number of logical processors by multiplexing the underlying resources across different virtual machines. The hardware resources are time-shared not only among different virtual machines (VMs), but also among different workloads of the same VM. An important source of performance degradation in such a scenario are the cache-warmup penalties that a workload experiences when it's scheduled, because the working set belonging to the workload gets displaced by other concurrently running workloads. We show that a VM that time-switches among four workloads can cause some of the workloads a slowdown of as much as 54%. However, such performance degradation depends on the workload behavior, with some workloads experiencing negligible degradation and some severe degradation.

We propose *Elastic Time Slicing* (ETS) to reduce the context-switch overhead for the most-affected workloads. We demonstrate that by taking the workload-specific context-switch overhead into consideration, the CPU scheduler can make better decisions to minimize the context-switch penalty for the most-affected workloads, thereby resulting in substantial performance improvements. ETS enhances performance without compromising on response time, thereby achieving dual benefits. To facilitate ETS, we develop a low-overhead hardware-based mechanism that dynamically estimates the sensitivity of a given workload to context switching. We evaluate the accuracy of the mechanism under various cache-management policies and show that it is very reliable. Context-switch–related warmup penalties increase as optimizations are applied to address traditional cache misses. For the first time, we assess the impact of advanced replacement policies and establish that it is significant.

1. Introduction

Virtualization enables sharing of hardware resources by multiple guest operating system (OS) instances. The resources are shared not only among different VMs, but also among different workloads of the same VM. To facilitate high utilization through consolidation, the system must support a large number of workloads. Some systems adopt coarse-grained division at the level of single cores, and others employ fine-grained division through time-sharing a core among workloads [1]. The latter phenomenon is referred to as *multitasked virtualization*. Factors such as cost, security, and system-management convenience lead to more workloads per system. The transition from dedicated workstations to virtualized desktop infrastructure environments is another trend in this direction.

In a virtualized environment with multiple workloads per VM, the time slice allocated to a VM is split equally among the constituent workloads [1]. As a result, each workload obtains a share of the time slice allotted to the VM, which is inversely proportional to the number of workloads. Such an aggressively multitasked environment serves as the basis for our work. Multitasked virtualization affects performance in two ways: (1) direct overhead incurred to switch among the workloads and (2) indirect overhead incurred due to the displacement of the system state. The second factor contributes significantly to the performance degradation and can be further viewed as composed of multiple components: lost register, translation look-aside buffer, branch predictor, and cache states. Among these components, the major overhead is due to the displaced state in the last-level cache (LLC) [1] and is the focus of this work. We designate the additional cache misses suffered due to a context-switch (CS) event as CS *misses*. The performance penalty associated with CS misses is severe in the case of multitasked virtualization, due to an additional degree of multitasking above and beyond the OS-level multitasking.

Modern computer systems feature large-LLC and long-latency main memory. When run on such systems, memory-intensive tasks cache a large volume of data in the LLC. We use the terms *workload*, *task*, and *application* as synonyms. After running a task of interest for the duration of its time-slice value, when the CPU scheduler context-switches to a different task or a set of different tasks, the cache lines belonging to the former are replaced by those of the latter. Depending on the memory-access behavior of the intervening tasks, when the task of interest gets a schedule on the processor again, it is likely to encounter a partially or completely cold cache. Depending on the memory-reuse behavior of the task of interest, its performance could be affected across the spectrum ranging from no or slight degradation to significant degradation. Some tasks experience only slight degradation because sometimes caches hold data irrelevant to future accesses [2].

We illustrate the variation in CS penalty across applications by using an example. Figure 1 shows the impact of CS events on the performance of two different applications. On a CS event, the cache warmup penalty is minimal for application (a) and significant for application (b). Whereas (a) is not sensitive to CS events, (b) is highly sensitive. Even though the complete cache state is lost in case of both applications (a) and (b) on a CS event, (a) suffers only minor performance degradation because its data reuse is low. (b) suffers significant performance degradation because its data reuse is high. In the following section, we use actual data to show that different tasks suffer from CS misses differently. For a task that suffers from CS misses significantly, a small time-slice value causes the task to experience CS events and CS misses more times than a large time-slice value. This phenomenon translates to an increase in the execution time of the task and a corresponding increase in the energy consumed across the entire system. The problem can be addressed by allocating fewer but longer time slices to the most-affected tasks (as illustrated in Figure 1(c)).

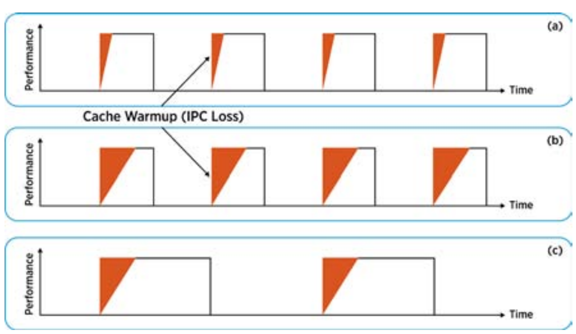

Figure 1. (a) When CS penalties are small, using short time slices does not cause any noticeable overhead. (b) For some workloads, short time slices can cause significant slowdown. (c) Only for such workloads is using longer and infrequent time slices desirable.

In this paper, we propose ETS to reduce the CS miss penalty. Whereas a uniform time-slicing (UTS) CPU-scheduling algorithm allocates time slices of equal duration to all tasks irrespective of their specific CS miss behavior, an ETS CPU-scheduling algorithm analyzes the CS miss behavior of the tasks and allocates fewer but longer time slices to those tasks that suffer significant performance degradation due to CS events. Performance penalty due to CS events can be naïvely addressed by allocating 10ms time slices to all tasks. 10ms is the default time-slice value allocated by the Linux OS. However, this solution suffers from high latency or response time between

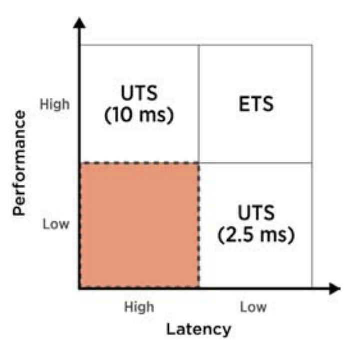

Figure 2. ETS provides the performance benefits of UTS with 10ms time slices as well as the latency benefits of UTS with 2.5ms time slices.

consecutive schedules, as depicted in Figure 2. In contrast, a UTS algorithm with 2.5ms time slices achieves low latency between consecutive schedules, but it suffers from low performance. 2.5ms is obtained by dividing 10ms equally among four tasks of a VM. Our ETS algorithm combines the best of both worlds and offers high performance (within 4% of UTS-10) as well as low latency (similar to UTS-2.5).

Enabling ETS requires dynamically estimating the extent to which a task suffers from CS misses. We develop a low-cost hardware-based Monte Carlo mechanism to estimate the cost of a CS event in terms of the number of CS misses suffered. The CS cost estimator works reliably under various cache-management policies because it is based on sampling of actual CS miss information. It facilitates incorporating the information about CS miss behavior into the design of a CPU-scheduling algorithm and exploiting the potential of such an enhanced CPU scheduler.

Most solutions that attempt to improve the cache hit rate by addressing the traditional cache misses (such as those due to capacity, conflict, coherence, and replacement) accentuate the problem of CS misses. These include: increasing the capacity of cache, employing compression in cache, prefetching lines into cache, improving the replacement algorithm, and so on. The number of CS misses tends to increase with increase in cache capacity (section 6.2) and improvement in replacement algorithm (section 6.1), thus worsening the problem. This paper shows that as systems optimize cache organization, addressing the problem associated with CS misses becomes more important, and a scheme like ETS becomes even more relevant.

2. Motivation

The locality properties of applications vary, and hence losing the cache state due to context switch can lead to variation in performance degradation for different applications. To demonstrate this, we conducted an experiment by reducing the allocated time-slice value. Figure 3 shows the variation in slowdown (measured in terms of cycles per instruction [CPI]) for SPEC CPU2006 benchmarks as the assigned time-slice value is reduced from 10ms. We flush the caches after each time slice to emulate a CS event. The rationale behind flushing the caches on a CS event will be described shortly. The parameters of the simulation infrastructure used to generate the results provided in Figure 3, and the basis for the choice of the parameters, are provided in section 4. Here, we capture the important aspects in order to enable comprehension of Figure 3. The results correspond to a LLC capacity of 2MB. We consider a processor running at a frequency of 4GHz. On such a processor, 10 million elapsed cycles correspond to an execution time of 2.5ms. The Y-axis represents the CPI corresponding to the execution of 500 million instructions. The labels 2.5ms and 5ms correspond to the cases when the VM comprises four and two workloads respectively, and a time-slice value of 10ms allocated to the VM is divided equally among the workloads. The CPI values for labels 2.5ms and 5ms are normalized with respect to the values corresponding to 10ms. A large value on the Y-axis corresponds to a higher CPI and, therefore, the smaller the Y-axis value the better.

We show the behavior for all 29 SPEC CPU2006 benchmarks in Figure 3 to make our case. The benchmarks are sorted in ascending order of the performance degradation incurred as the allocated time-slice value is reduced. Throughout this paper, we identify the benchmarks in figures using the first four letters of their names. For applications that appear on the left of the figure, the CPI varies very little as the duration of the time-slice value is reduced from 10ms to 2.5ms. However, the CPI varies significantly in the case of applications that appear on the right. For the remaining applications, the variation in CPI as the time-slice value is decreased is distributed across the spectrum. The maximum degradation for a 2.5ms time slice is observed in case of HMMER and is 54%. An analysis of the results reveals that different applications indeed suffer from CS events differently; some suffer mildly while others suffer severely. Further, the CS performance penalty varies over the duration of execution of an application (section 5.1). The variation in performance degradation can be addressed by adopting ETS. The key insight behind ETS is to allocate fewer but longer time slices to address the performance penalty incurred by the most-affected workloads. To facilitate ETS, a dynamic mechanism is essential for estimating the extent to which an application suffers from CS events. We now describe an assumption and justify the reason for making it before presenting the dynamic mechanism.

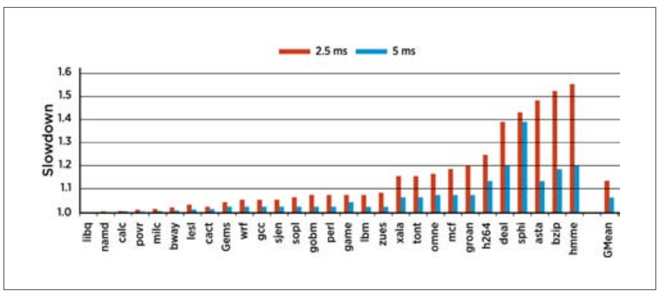

Figure 3. Variation in slowdown (measured in terms of CPI) as the time-slice value is reduced for SPEC CPU2006 benchmarks. The impact of CS events is significant on some workloads and imperceptible on others.

We assume that the data cached by an application in the LLC during the duration of its time slice is completely evicted by the data brought in by the intervening applications, before it is scheduled again. This assumption holds because of the aggressive multitasking employed by the virtualized systems described in section 1. We co-scheduled eight applications in a round-robin (RR) fashion, each for a time-slice duration of 2.5ms. This co-schedule is analogous to a scenario in which there are two VMs, each containing four workloads. The baseline time-slice value of 2.5ms is obtained by dividing 10ms equally among the workloads comprising a VM. The values we considered for the number of VMs and the number of workloads in this work are conservative. The actual numbers are even larger [1] [3], and our assumption is still valid under such conditions. We evaluated 30 different co-schedules, each made up of eight distinct applications, and observed the number of residual lines from one schedule of the application to its next schedule. Residual lines are those lines that remain in the cache from one schedule to the next. Over the total duration of execution, the number of residual lines for all applications is zero. Similar behavior was also observed in previous works [1] [3]. Even though the base time-slice value is small, when the execution of interleaved applications is considered, the total time is sufficient for an application's data in the LLC to be evicted completely. This phenomenon has two implications. First, invalidating the cache entries faithfully emulates a CS event. Second, there is no negative impact due to extending the time-slice value of an application on those applications whose time-slice value remains unchanged.

3. Framework for Estimating and Addressing the CS Performance Penalty

The motivational results presented in section 2 suggest that a dynamic mechanism is essential for estimating the penalty incurred due to a CS event. Such a mechanism can be used to characterize the impact of a CS event on an application. Now we describe the mechanism designed to estimate the cost of a CS event in terms of the number of CS misses incurred. The mechanism is capable of computing the cost of a CS event effectively while incurring a minimal overhead. Further, we present an augmentation to the baseline UTS RR CPU-scheduling algorithm in order to derive an ETS RR CPU-scheduling algorithm. The latter is capable of leveraging the calculated cost of CS events to mitigate the performance degradation.

3.1 Cost Estimation of a CS Event

The number of CS misses suffered by an application can be estimated in a simple but inefficient manner by making a copy of the tag directory (of the cache) on a CS event. When the application obtains a schedule again, the accesses that miss in the main tag directory but hit in the copy tag directory are tracked. The number of such accesses corresponds to the number of CS misses suffered. This simple scheme suffers from the following drawbacks. If there are multiple co-scheduled applications, we need a corresponding number of copy tag directories, which incur a significant area overhead. Multiple copy tag directories can be avoided by storing all but the one required (at any given time) in the memory. This approach requires maintaining space in the memory and logic to store and restore the copy tag directory to and from the memory respectively. Further, additional memory bandwidth is required to perform the store and restore operations. Now we propose a solution that overcomes these disadvantages. The solution is based on the following key ideas. CS miss count can be estimated by emulating a CS event. This requires only one copy tag directory (Figure 4a). The hardware overhead due to the copy tag directory can be reduced

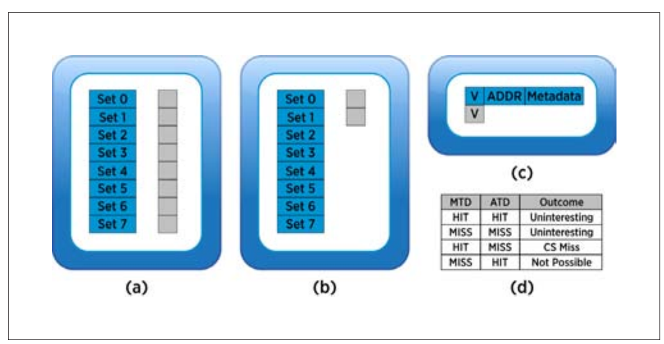

Figure 4. (a) The MTD (wide) and the full ATD (narrow) (b) The MTD (wide) and the ATD (narrow) with sample sets (c) An entry in the MTD (wide) and the ATD (narrow) (d) Status of accesses in the MTD and the ATD after a CS event.

by maintaining copy tags only for a fraction of the sets in the cache (Figure 4b). Further, the copy tag directory entry needs to contain only one bit of information (a valid bit), as opposed to a main tag directory entry that contains a valid bit, address bits, and other metadata (Figure 4c).

The working of our cost-estimation framework is modeled after that of a Monte Carlo (MC) method. MC methods rely on random sampling to determine an approximate answer to a question. In general, the answer determined using a MC method becomes more accurate as the number of samples considered increases. The proposed mechanism consists of an auxiliary tag directory (ATD) in addition to the regular main tag directory (MTD) in the cache. The ATD contains tags corresponding to a certain number of sets in the MTD. These sets in the MTD are referred to as *sample sets* (SS). We reason about the exact number of SS required in section 5. An entry in the ATD contains only one bit of information and can be either valid or invalid. It should be noted that a hit in the ATD is analogous to the line being valid and a miss to the line being invalid. At the start of execution, the state of SS in the MTD and the ATD is consistent, which means that lines in the MTD and the ATD are either both valid or both invalid.

To estimate the number of CS misses for an application, the entries in the ATD are invalidated. This process emulates a CS event. After the point of invalidation, corresponding to subsequent cache accesses, one of the following scenarios can arise (Figure 4d): access hits in both the MTD and the ATD, access misses in both the MTD and the ATD, or access hits in the MTD but misses in the ATD. The first two events are not of interest to us. The third event corresponds to a CS miss. A miss in the ATD and a hit in the MTD happen because the ATD experienced a cache-flush event, which is analogous to a CS event. The corresponding entry in the ATD is made valid on recording the CS miss. So, further accesses to the same cache line do not generate CS misses. The ATD entry corresponding to the second event is made valid as well. We use a counter (CS-MISS-CNT) to keep track of the CS misses. The counter value is read at the time when the application is being switched out. It indicates the number of CS misses suffered by SS. To estimate the total number of CS misses experienced by the application, the counter value is multiplied with the ratio of the total number of sets to the number of SS. This ratio is chosen to be a power of 2 so that the multiplication operation degenerates to a simple left-shift operation. After the invalidation point, an event corresponding to a miss in the MTD and a hit in the ATD does not happen by construction (Figure 4d). The set of hits in the ATD is always a proper subset of the set of hits in the MTD. We depict the steps for estimating the CS miss count in Algorithm 1, using pseudocode.

The mechanism proposed above aids in estimating the number of CS misses for a private cache. The trend in modern computer systems is to employ simultaneous multiple threading (SMT) and/or multiple cores to enhance performance while keeping power consumption in check. We refer to the hardware thread instances in the case of SMT and the cores in a multicore processor commonly as *sharers*. When the cache is shared by two or more sharers, the CS cost-estimation mechanism needs to be augmented as follows to support the cost estimation for each sharer. The modification

```
initialize() {
 invalidate entries in ATD; CS-MISS-CNT = 0; }

count_cs_misses() {
 if ((MTD.lookup == hit) &&
 (ATD.lookup == miss)) CS-MISS-CNT++; }

estimate_cs_penalty() {
 CS-MISS-CNT X (number-of-sets-in-cache/
 number-of-sample-sets); }
```

Algorithm 1. CS cost computation in terms of the number of CS misses

required is to replicate CS-MISS-CNT counter per sharer. Because lines belonging to each sharer are uniquely identified in the tag entry of the cache, the identifier can be used to match and update the corresponding counter. Note that one copy of the ATD is sufficient irrespective of the number of sharers.

3.2 Design of an ETS RR CPU-Scheduling Algorithm

Previously, we pointed out that fewer but longer time slices must be used for those applications that are severely impacted by CS events. By doing so, we can alleviate the negative impact of CS events on the performance of such applications. In this section, we describe the design of a CPU-scheduling algorithm that achieves this goal. Specifically, we augment the baseline UTS RR CPU-scheduling algorithm to derive an ETS RR CPU-scheduling algorithm. Recall that we described the distinction between the two in section 1. To keep the discussion precise, we choose the following values for parameters (same as the values used throughout this paper). The baseline UTS algorithm allocates a time-slice value of 2.5ms for all applications in a RR fashion. We consider a system with a LLC capacity of 2MB. The cache consists of a total of 32,768 lines, each of size 64 bytes. The ETS algorithm categorizes these lines into four groups, as shown in the Group column of Table 1. The groups are based on the number of CS misses. For an application belonging to a particular group, the algorithm extends the time slice to the value indicated in the Slice column. The size of the group is doubled from one group to the next, and the time-slice value is increased by 2.5ms. In this manifestation, we capped the maximum time-slice value at 10ms. In an actual system, we expect that this value will be set after taking the response-time constraints and other factors into consideration.

	GROUP	SLICE		GROUP	SLICE
(1)	≤1,500	2.5 ms	(2)	1,501 - 4,500	5.0 ms
(4)	>10,501	10 ms	(3)	4,501 - 10,500	7.5 ms

Table 1. An ETS round-robin CPU-scheduling algorithm implementation. CS miss count is used as index in order to determine the time slice value for the next schedule.

We measure the number of CS misses experienced by the application every time it obtains a schedule on the processor. The number of misses estimated using Algorithm 1 is used as index into Table I. The corresponding value of Slice is assigned as the time-slice value for the next schedule of the application. It must be noted that the discretization presented in Table 1 is realized in software and therefore can be customized to a target system. We developed the presented discretization by heuristically running the benchmarks and analyzing the results. A high-level overview of the working of the ETS framework is provided as a flow chart in Figure 5.

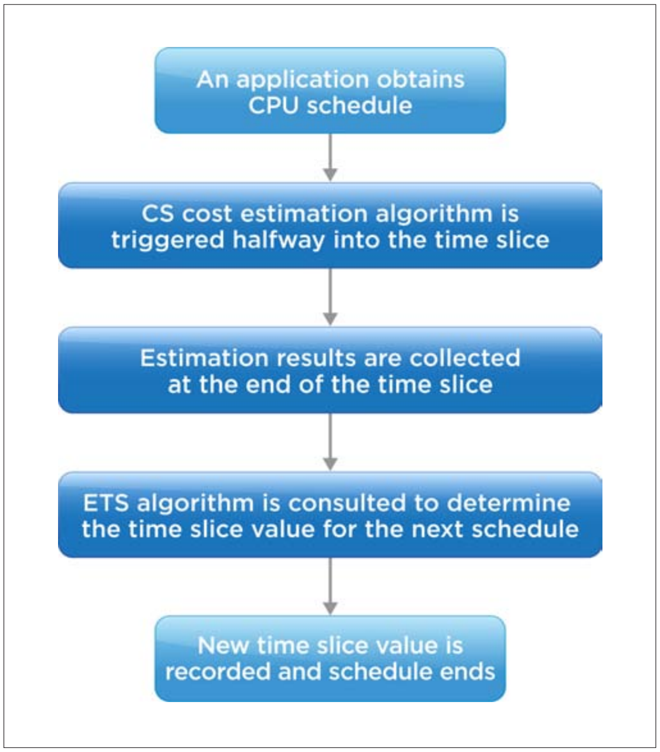

Figure 5. High-level overview of the ETS framework

It is not our objective to propose an alternative CPU-scheduling algorithm. There is a large body of work that investigated such algorithms. However, we argue that these algorithms must be supplemented to make them aware of the cost of the CS events. The design of the scheduling algorithm could incorporate CS miss information together with other currently used factors such as priority and interactivity. Here, we described how a UTS RR CPU-scheduling algorithm can be augmented to account for the CS penalty incurred.

4. Experimental Methodology

We use an in-house trace-driven simulator to conduct the experiments. The processor is modeled as an in-order core, and the simulator is capable of handling multiple cores. The memory hierarchy consists of three levels of caches: separate instruction and data caches at the first level, and unified caches at the middle and the last levels. A uniform value of 64 bytes is used for line size across the entire hierarchy. We use a baseline value of 2MB for the LLC capacity in our experiments. Our simulator can model the LLC as private to each core or as shared among multiple cores. In either case, multiple applications can be co-scheduled on each core. All cache levels implement the LRU replacement policy. The parameters of the simulated machine are shown in Table 2.

Processor	4GHz, Single Issue, In-Order
L1 I-cache	32KB, 2-way
L1 D-cache	32KB, 2-way
L2 cache	256KB, 4-way
LLC	2MB, 16-way, 24 cycles
Main memory	400 cycles

Table 2. Machine configuration

In the event of a context switch, the employed framework eliminates effects other than the loss of saved state in the LLC. We use all benchmarks (29 in number) from the SPEC CPU2006 suite to obtain a comprehensive set of results. Each benchmark is comprised of a representative set of 500 million instructions. For our experiments, we combined disparate benchmarks to generate 29 diverse workload mixes (co-schedules). When an LLC capacity other than 2MB is used, we keep the associativity constant and increase the number of sets. We apply the CS cost-estimation mechanism to the LLC in the system as the distance between the LLC, and the main memory is far in units of CPU cycles. Our baseline system employs the UTS RR CPU-scheduling algorithm and uses 2.5ms for the time-slice value. The UTS algorithm is representative of the mechanism in IBM PowerVM virtualization, in which a fixed scheduling period can be shared by up to 10 vCPUs through micropartitioning. The prominent parameters used in this work are modeled after those used in the most recent related paper [3]. These include the number of co-scheduled applications, the baseline CPU-scheduling algorithm and time-slice value, and the capacity of the LLC. Further, the LLC capacity of 2MB per core used in this work is representative of the LLC capacity in server class machines.

5. Results and Analysis

Hereafter, we use the word *cache* to refer to the LLC by default. We now attempt to answer the following questions by relying on experimental results: What is the performance improvement that can be obtained by adopting ETS? How accurately can we estimate the number of CS misses?

5.1 Advantage of Using the ETS CPU-Scheduling Algorithm

The results obtained by employing the ETS RR scheduling algorithm described in section 3.2 are shown in Figure 6 for all SPEC CPU2006 benchmarks. The results correspond to a cache capacity of 2MB. In each case, a total of eight applications are co-scheduled onto a single core. We study the impact of CS events on the application indicated by the X-axis label, which is the application of interest. We apply the ETS algorithm to it and modify its time-slice value. The time-slice value for the remaining applications is 2.5ms, which is the baseline time-slice value of the UTS RR CPU-scheduling algorithm. The evaluation metric used is the IPC corresponding to the execution of 500 million instructions. The values corresponding to the ETS algorithm are normalized with respect to the values for the UTS algorithm.

Figure 6(a) shows the improvement in IPC obtained by employing the ETS algorithm compared to that obtained using the UTS algorithm (2.5ms time slices). The applications are sorted in ascending order of the benefit derived from the ETS algorithm. Figure 6(b) shows the distribution of time slices allocated by the ETS algorithm. Some applications, such as libquantum and CalculiX, are minimally impacted by CS misses. The time-slice allocation distribution shows that the time slices allocated to these applications are predominantly of duration 2.5ms. In contrast, applications such as HMMER, bzip2, and astar are severely impacted by CS misses. The distribution shows that the time slices allocated to these applications are predominantly of duration 5ms, 7.5ms, and 10ms. The ETS algorithm allocates longer time slices on the basis of their utility to applications. The maximum improvement in IPC is obtained in case of HMMER and is as much as 54%. The remaining applications span the entire spectrum of performance improvement.

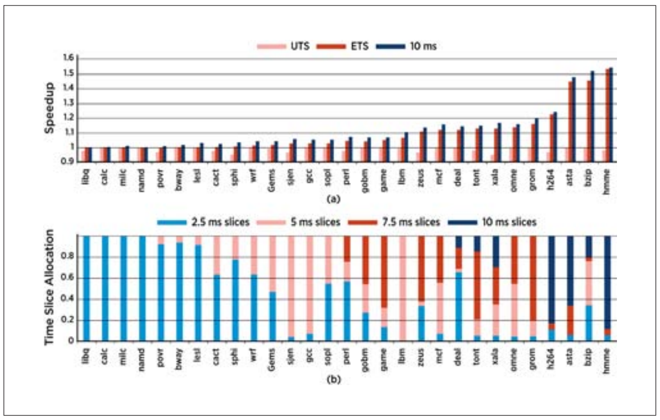

Figure 6. (a) IPC improvement by adopting the ETS RR CPU-scheduling algorithm (b) Distribution of allocated time slices

The diversity of the results shown in Figure 6 reinforces our hypothesis that we should track the number of CS misses dynamically and allocate longer time slices to those applications that suffer from CS misses significantly. Out of a total of 29 applications studied, the performance improvement due to the ETS algorithm is 5% or more in the case of 15 applications and more than 10% in the case of 11 applications. Figure 6(a) also shows the IPC results corresponding to the case when a constant value of 10ms is used for the time slice. The results are once again normalized with respect to those for the UTS algorithm (2.5ms time slices). The IPC results obtained using the ETS algorithm are within 4% of the results obtained using a constant value of 10ms for the time slice. In summary, the results provide substantial evidence in favor of the ETS algorithm to address the negative performance impact of CS events. It should be noted that the ETS algorithm is implemented in software. Therefore, it can be customized and optimized for a target system. However, we expect that the implementation will be kept simple to contain the direct overhead associated with a CS event. In our implementation, the additional cost is approximately 10 instructions.

The cumulative CPU time allocated by the ETS algorithm to all applications (including the application of interest) is equal to that allocated by the UTS algorithm. We demonstrate this using an example in Figure 7. For clarity of discussion, we consider the case in which there are a total of three co-scheduled applications. However, the reasoning also applies to co-schedules involving a different number of applications. In Figure 7, P3 is the application of interest. The time-slice allocation performed by the UTS algorithm is shown in the row labeled 2.5ms. The time-slice allocations made by the ETS algorithm for three different scenarios are shown in the other rows. Although the ETS algorithm allocates longer time slices, it allocates fewer such longer time slices. As the length of the allocated time slice increases, the number of allocations of the time slice decreases. Therefore, the ETS algorithm offers the same fairness guarantees as the UTS algorithm.

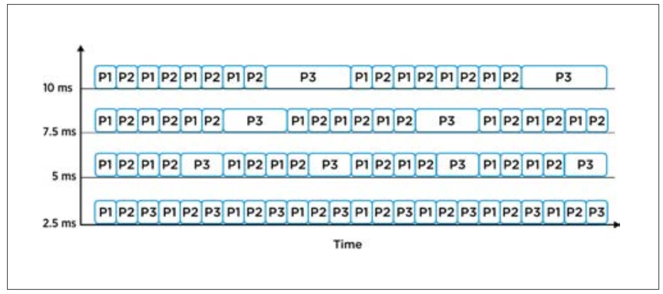

Figure 7. Example to illustrate the allocation of time slices by the ETS algorithm

Latency or response time—time elapsed between two consecutive schedules of an application—is another important aspect of a CPU-scheduling algorithm. In Figure 8, we provide quantitative information regarding the latency behavior of three algorithms: UTS algorithm with 10ms time slices, UTS algorithm with 2.5ms time slices, and ETS algorithm. For an application indicated by the X-axis label, the Y-axis value corresponds to the latency incurred by a co-scheduled application. The average latency for UTS-10, UTS-2.5, and ETS is 30ms (horizontal solid line), 7.5ms (horizontal dotted line), and 7.5ms (rectangles) respectively. Whereas the standard deviation in latency for UTS-10 and UTS-2.5 is 0, the value for each application in the case of ETS is shown in the form of error bars above the rectangles. The maximum value of the standard deviation is 3.7 and is observed in case of HMMER. From the data presented in Figure 8, it can be inferred that the latency behavior of the ETS algorithm is very similar to that of UTS-2.5. In addition, the performance behavior of the ETS algorithm is nearly identical to that of UTS-10 (Figure 6(a)). The ETS algorithm adapts to the dynamic behavior of the applications to achieve the best of both worlds.

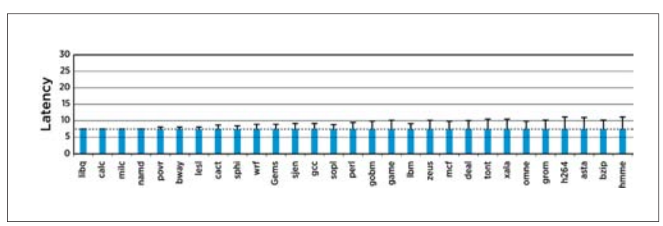

Figure 8. Latency behavior of UTS-10 (horizontal solid line), UTS-2.5 (horizontal dotted line), and ETS algorithms. Error bars indicate the standard deviation in latency for the ETS algorithm. The latency behavior of the ETS algorithm is very similar to that of UTS-2.5.

The CS performance penalty varies not only across applications but also over the duration of execution of an application. This is because applications go through phases of execution. We provide experimental results in Figure 9 to substantiate our claim. Figure 9 shows the time-slice transitions for all benchmarks over their total duration of execution. The label **Same** indicates the fraction of transitions from a time-slice value to the same time-slice value, and the label **Different** indicates the fraction of transitions to a different time-slice value. A **Different** transition happens when the number of CS misses changes considerably from a time slice to the next. Hence, a large value for the fraction of **Different** transitions is indicative of the change in the CS miss behavior over the duration of execution. The fraction of **Different** transitions is 10% or more for a total of 19 applications. A maximum value of 68% is recorded in case of xalancbmk. The results corroborate our hypothesis that the CS miss behavior indeed varies over the duration of execution of an application.

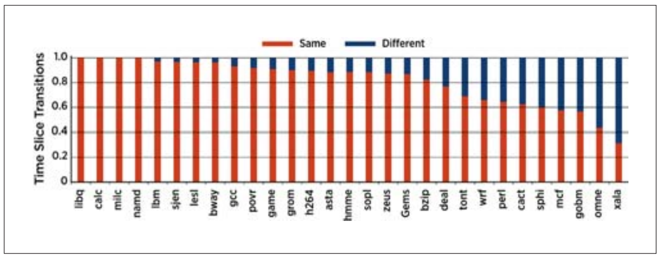

Figure 9. Variation in CS miss behavior over the duration of execution of applications. Same label indicates the fraction of transitions from a time-slice value to the same time-slice value, and Different indicates the fraction of transitions to a different time-slice value. A large value for the fraction of Different transitions indicates that the CS miss behavior changes over the duration of execution.

5.2 CS Cost-Estimation Accuracy

The ability to dynamically estimate the cost of a CS event is central to the operation of the ETS algorithm. We use the augmented CS cost-estimation mechanism described in section 3.1 to estimate the number of CS misses per sharer. The corresponding results are presented in Figure 10. Specifically, we provide the results when two cores share a 4MB cache. The sharers are identified uniquely through X-axis labels. The experiment used 256 SS, which correspond to 1/16 of the total number of sets. We evaluated various values for the number of SS and narrowed it down to 1/16 of the total number of sets. This choice achieves a good trade-off between area and accuracy. The estimation error is represented in percentage terms and indicates the separation between the value computed using the SS and the actual value. The average value of the estimated error across all sharers is 2.5% (excluding milc). The average error and the estimated error for most applications are both below 5%, indicating the usefulness of the proposed mechanism. We address the inaccuracy in estimation for milc in section 5.3.

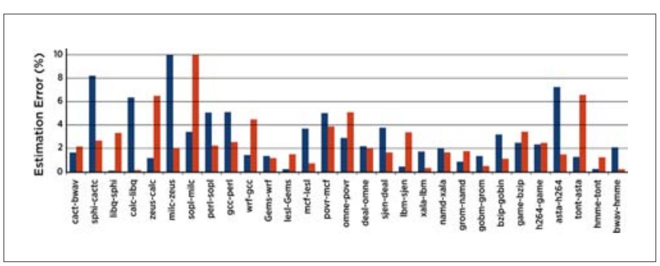

Figure 10. Percentage error in estimation of the number of CS misses when two applications share a 4 MB LLC

It is important to consider the mechanism that is employed to partition the cache among the sharers and how the mechanism affects CS miss count estimation. The results presented in this section are for the case when the ways are equally partitioned among the sharers and the LRU replacement policy is employed within each partition. We will now discuss the impact of more-advanced partitioning mechanisms on the accuracy of CS cost estimation. Global LRU replacement policy allows for dynamic sharing based on demand. However, it was previously shown that demand for cache does not always translate to benefit from cache [4]. Several proposals were made to improve the benefit derived from a shared cache: utility-based cache partitioning (UCP) [4], thread-aware dynamic insertion policy (TADIP) [5], and software-based shared-cache management techniques such as page coloring. The common goal of these works is to determine what is likely to be the optimum partition of the shared cache and enforce the applications to stay within the limit of the determined optimum partition. These methods allocate ways of sets or sets of cache among the applications. Such structured allocation lends itself well to the proposed CS cost-estimation mechanism, which works on the principle of uniform sampling. In summary, we anticipate that employing the proposed CS cost-estimation scheme in conjunction with advanced partitioning mechanisms will result in as accurate estimates as we obtained here.

We also evaluated the accuracy of the CS cost-estimation mechanism for a 2MB private cache and when four cores share an 8MB cache. The average value of the estimated error across all workloads is 2.7% and 2.5% respectively (excluding milc). The results corresponding to the private cache are shown in Figure 11. Estimating the number of CS misses to within a 10% value can provide very important information for a CPU-scheduling algorithm to factor the CS event cost. More concretely, we anticipate that the trend for the number of CS misses rather than the actual value will be used by the CPU scheduler. We discussed the performance improvement obtained using one manifestation of such a scheduler in section 5.1.

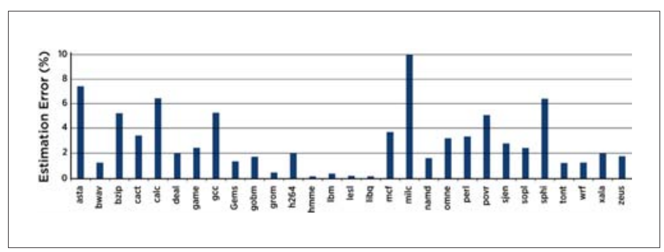

Figure 11. Percentage error in estimation of the number of CS misses for a 2MB private LLC

5.3 Addressing CS Cost-Estimation Inaccuracy

In section 5.2, we pointed out that the estimated value of CS misses is inaccurate (by 50%) for milc. We now propose a mechanism, which incurs minimal overhead, to determine when the CS miss count estimate is inaccurate. The ATD presented in section 4.1 is organized as two logical entities, with each one comprising half the original number of sample sets. We associate each logical entity with a separate CS-MISS-CNT counter. The hardware overhead of the enhanced estimator is this additional counter and a small amount of logic necessary to organize the logical entities. It should be noted that the physical organization of the ATD is still the same as it is for the original estimation mechanism described in section 3.1.

The estimation mechanism works as before, with the SS updating the respective counters for the number of CS misses. When the application is being switched out, the values of the two counters are used to determine if the estimated value is accurate. Specifically, we compute the ratio of the absolute difference of the two counter values and their sum. A large value of this ratio indicates that the estimate is inaccurate. Otherwise, the computed sum of the two counter values serves as the estimated value for the number of CS misses corresponding to the SS. This number is scaled to the total number of sets in the cache to get the final estimated value. The specified ratio is provided for all benchmarks in Figure 12. The value for milc is 0.30, while the corresponding value for all other benchmarks is at most 0.15. The computed ratio is a measure of divergence between the two counter values. When the estimate is inaccurate, the divergence is large. If this is the case, we ignore the estimate and keep the time-slice value intact.

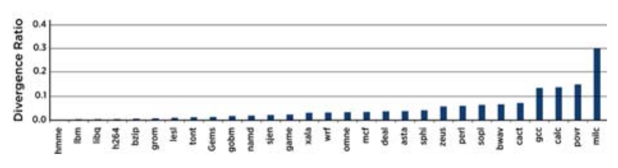

Figure 12. Divergence ratio of CS-MISS-CNT counter values, used to determine the accuracy of CS cost estimate. A large value of the ratio denotes that the CS cost estimate is inaccurate.

5.4 Hardware Overhead

In this section, we quantify the additional hardware necessary to support the ETS mechanism. The storage component of the overhead consists of the following: (1) tags for the sample sets in the ATD and (2) counters for tracking the number of CS misses. This component is computed in Table 3. We assume a physical address space of 40 bits and use the same baseline LLC capacity of 2MB used previously. The LLC is organized as a 16-way associative cache for a total of 2,048 sets. Note that we used 1/16 of total sets for the number of SS throughout this paper. Additional logic is required in order to invalidate the ATD entries (to emulate a CS event), detect a CS miss (a 2-ip logic gate), and increment the CS- MISS-CNT counter. The overhead due to the extra logic is negligible, similar to the storage overhead.

Size of an MTD entry (address + metadata)	4B
Size of an ATD entry (valid-bit)	1 bit
Number of ATD entries (16 per set × 128 sets)	2048
Overhead of ATD	256B
Size of counter *(log 2 {number − of − cache − lines − in − ATD})*	1.5B
Overhead due to 2 counters	3B
Area of baseline LLC (128kB tags + 2MB data)	2176kB
% increase in LLC area (260B/2176kB)	0.012%

Table 3. Storage overhead to support the ETS mechanism

Prior to developing the sampling-based CS cost-estimation framework, we attempted to make use of the change in the IPC of the task before and after a CS event to estimate the cost of the CS event. The advantage of this approach is that it does not require any additional hardware. It makes use of the hardware-performance counters built into modern microprocessors. However, this mechanism suffers from the following drawbacks. First, in several instances, the IPC value before and after a point where a CS event will be scheduled are different. This inherent change in the IPC over the duration of execution serves to increase or decrease the actual cost of a CS event. Second, the duration for which the IPC value must be tracked varies across instances. In our experiments, after a CS event, we noticed that the CS misses are interspersed between regular misses, and the distribution of CS misses within the regular misses varies. This spread of CS misses adds to the problem of inherent difference in the IPC value. We developed the sampling-based framework after realizing the limitations of the IPC-based mechanism.

6. Impact of Cache Optimizations on CS Misses

In Section 1, we mentioned that the mechanisms that attempt to improve the cache hit rate by addressing the traditional cache misses exacerbate the problem associated with CS misses. Such optimizations attempt to retain more data that will be useful in the future, and CS events result in the loss of this data. Now, we evaluate the impact of improving the replacement algorithm and increasing the capacity of the cache on the number of CS misses. We also assess the accuracy of the CS cost-estimation hardware when it is applied to advanced replacement algorithms. Furthermore, we provide the performance-improvement results obtained by using the ETS mechanism in conjunction with these algorithms. In our experiments, the first-level and the middle-level caches use the LRU replacement policy, and our replacement-policy studies are limited to the LLC.

6.1 Replacement Algorithm

Thus far, we have assumed that the cache implemented the LRU replacement policy. It performs poorly in the following two scenarios: when the size of the working set is larger than the capacity of the cache, and when references to nontemporal data (scans) cause a frequently referenced working set to be discarded. Solutions were proposed to address one or more of these shortcomings of the LRU algorithm: dynamic insertion policy (DIP), rereference interval prediction (RRIP), and signature-based hit prediction (SHiP). DIP chooses between two policies—bimodal insertion policy (BIP) and LRU policy—dynamically depending on which policy incurs fewer misses. The selection is made through dynamic set sampling and set dueling [6]. RRIP policy works by predicting the rereference interval of a cache line. The work proposes two policies: static RRIP (SRRIP) and dynamic RRIP (DRRIP). DRRIP again uses set dueling to identify which of SRRIP and bimodal RRIP (BRRIP) performs the best [7]. Finally, SHiP works by correlating the rereference behavior of a cache line with its signature [8]. We use the program counter (PC) value to derive the signature.

Now, we present the results to show how the estimation hardware performs for replacement policies other than LRU. Specifically, we present the results for DRRIP and SHiP algorithms. The results correspond to a private cache with a capacity of 2MB. In Figure 13(a), we show the percentage error in estimation of the number of CS misses. For each algorithm, we present the results generated using 128 SS. Using 128 SS, which correspond to a 1/16 fraction of the total number of sets, we obtain an accurate estimate for the number of CS misses. The average percentage error in estimation is 2.8% and 2.4% for DRRIP and SHiP algorithms (excluding milc) respectively. The estimation error for milc is 29% and 18% respectively. These results demonstrate that the estimation hardware, because it is based on sampling, lends itself very well to other replacement algorithms.

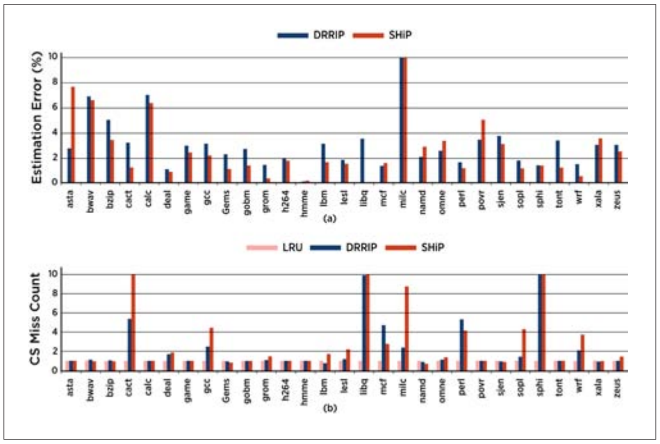

Figure 13. (a) Percentage error in estimation of the number of CS misses (b) Impact of advanced replacement algorithms on the number of CS misses

Next, we consider the impact of the replacement algorithm on the number of CS misses suffered by an application. In Figure 13(b), we show how the number of CS misses varies with the replacement algorithm. The values for DRRIP and SHiP are normalized with respect to the values for LRU. For several benchmarks, the number of CS misses increases pronouncedly for DRRIP and SHiP when compared to LRU. The maximum increase in the number of CS misses (by a factor of 20X) is observed in case of sphinx3 for both DRRIP and SHiP. The geometric mean across all benchmarks is 1.6 and 2 for DRRIP and SHiP respectively. These results substantiate our hypothesis that adopting advanced replacement algorithms accentuates the problem associated with CS misses.

The IPC results obtained by employing the ETS RR scheduling algorithm are shown in Figures 14(a) and 14(b) for DRRIP and SHiP policies respectively. The experimental methodology used is similar to that employed in section 5.1. The values corresponding to the ETS algorithm are normalized with respect to the values for the UTS algorithm (2.5ms time slices). The applications are sorted in ascending order of the benefit derived from the ETS algorithm. Figure 14 also shows the IPC results corresponding to a 10ms time-slice value normalized with respect to the results for the UTS algorithm. The IPC results obtained using the ETS algorithm are within 4% of the results obtained using a constant value of 10ms for the time slice. It can be inferred from these results that the ETS algorithm is equally applicable for advanced replacement policies. For both DRRIP and SHiP policies, the performance improvement due to the ETS algorithm is 10% or more in case of 11 applications.

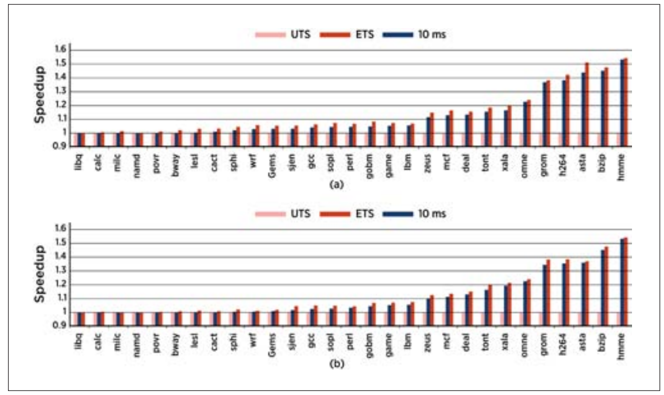

Figure 14. IPC improvement by adopting the ETS RR CPU-scheduling algorithm for (a) DRRIP and (b) SHiP replacement algorithms

6.2 Cache Size

We now consider the impact of another important optimization—increasing the capacity of the cache—on the number of CS misses. We present the results for all the benchmarks in Figure 15. We obtained the results for three different cache sizes: 1MB, 2MB, and 4MB. We provide the evaluation results for three replacement algorithms: LRU, DRRIP, and SHiP.

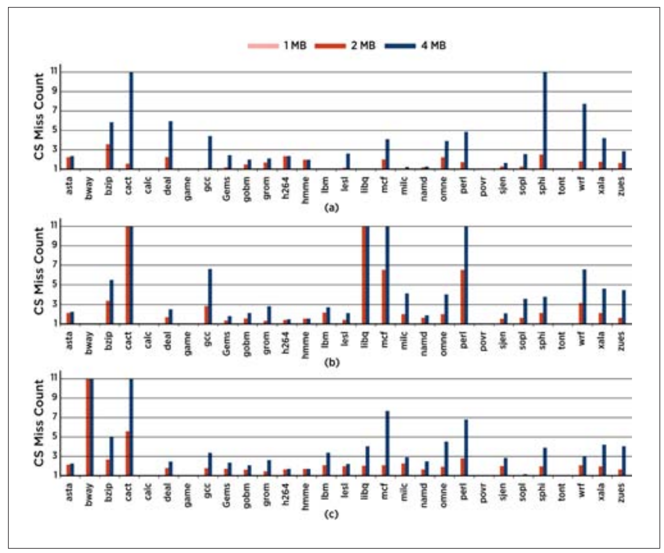

Figure 15. Impact of increasing the cache size on the number of CS misses for (a) LRU, (b) DRRIP, and (c) SHiP replacement algorithms

The number of CS misses for 2MB and 4MB cache sizes are normalized with respect to that for the capacity of 1MB. In general, for all benchmarks and replacement policies, the number of CS misses increases as the cache size increases. In several cases, the increase is by a factor of 5X or more from 1MB capacity to 4MB capacity. For cactusADM, with the DRRIP replacement policy, the number of CS misses increases by a factor of 101X from 1MB capacity to 4MB capacity. In summary, the results indicate that increasing the cache capacity has a significant impact on the number of CS misses experienced by the application. The performance penalty due to the increased number of CS misses will be commensurate with the magnitude of increase in the number.

In summary, cache optimizations accentuate the problem associated with CS misses. Therefore, in the presence of such optimizations, estimating the CS miss cost accurately and incorporating the estimate into CPU-scheduling algorithms become even more important.

7. Related Work

Studies related to context switching have received much attention from both the industry and the academia over a long period of time. We summarize, compare, and contrast the works that closely relate to the techniques proposed in this paper under four different categories.

7.1 Performance Impact of Context Switching

Many studies aimed at understanding the performance impact of CS events. Agarwal et al. [9] showed that multiprogramming activity significantly degrades cache performance, and that the impact grows with increase in cache size. Mogul et al. [10] estimated the performance reduction caused by a CS to be in the order of tens to hundreds of microseconds, depending on the cache parameters. Suh et al. [11] considered the performance impact of context switching on page faults. They proposed to mitigate the problem using job speculative prefetching. Chiou et al. [12] proposed that memory scheduling, potentially at all levels of the memory hierarchy, should drive CPU scheduling rather than the other way around as it is done in most systems.

Koka et al. [13] characterized CS misses and quantified their impact in the case of transactional workloads. They investigated the potential for intelligent process scheduling that minimizes cache misses across CS boundaries. Li et al. [14] concluded that the indirect CS overhead due to cache perturbation is more significant than the direct overhead. Tsafrir [15] and David et al. [16] calculated the indirect overhead due to a CS event for Intel and ARM platforms respectively. In summary, most of the works concluded that the indirect overhead, due to cache perturbation, associated with CS events is significant. It is this overhead that we attempt to address through our proposal.

7.2 Analytical Models

Several analytical models were proposed to explain the relationship between an application's temporal reuse behavior and its vulnerability to CS misses. Such models need to factor in all essential variables to have a sufficient resolution. Agarwal et al. [17] and Suh et al. [18] [19] proposed analytical models to estimate the overall cache miss rate, including fully associative cache, in order to obtain a continuous miss-rate curve, which is required as profiling information. This assumption does not hold true in case of the LLC. The model by Hwu et al. [20] was aimed at predicting the worst-case number of CS misses. Liu et al. [21] classified CS misses into two categories: replaced misses and reordered misses. Further, they developed an analytical model to reveal the relationship among cache design parameters, an application's temporal reuse pattern, and the number of CS misses the application suffers from. They applied the devised model to study the impact of prefetching and cache size on the number of CS misses.

Analytical models make certain assumptions to render the task of making the model tractable. For example, the model by Liu et al. [21] is designed under the assumption of LRU replacement policy. However, advanced replacement algorithms were proposed that perform better than the LRU algorithm. Also, analytical models are suitable for offline analysis, but the feasibility of their implementation in hardware while incurring a low area overhead is not considered. Our solution's approach is implemented using very low hardware overhead to work in a dynamic environment for any cache configuration, thereby addressing the previously pointed-out limitations of the analytical models.

7.3 Employing Prefetching to Cope with CS Misses

The performance degradation due to CS misses can be addressed through two different means: by (1) increasing the time slice value (2) prefetching the cache state just before or when the new schedule starts. The former method is a preventive measure and the latter is a cure. Prefetching was suggested to mitigate the cost of additional cache misses incurred because of a CS event. The general idea is to record the application's locality at the time when it gets swapped out. The locality is restored through prefetching the next time application gets CPU time. Previously proposed solutions that employ prefetching differ in how the locality is stored and restored. Cui et al. [22] employ global-history-list (GHL) prefetching. GHL maintains a complete list of cache lines, which is ordered by recency of use. Daly et al. [1] studied the impact of CS misses in highly partitioned virtualized systems. They proposed cache restoration as a solution to prefetch the working set and thereby warm the cache. GHL and cache restoration, although they differ in implementation details to some extent, perform similarly. GHL performs slightly better at the expense of more hardware and complexity. In the most recent related work [3], the authors proposed methods to reduce the bandwidth overhead of these prefetchers.

Brown et al. [23] proposed accelerating postmigration thread performance by predicting and prefetching the working set of the application. In the proposed solution, access behavior of a thread is captured and summarized into a compact form premigration. On the new core, the summary is used to prefetch appropriate data to create a warm state. Prefetching the data after a CS event serves to cure the cold-start problem. However, ETS works to minimize the number of cold starts for those applications for which it matters. The techniques presented in this paper can provide guidance as to when prefetching can be beneficial and when it is not likely to help. Zebchuk et al. [3] identified the inability of all cache-restoration prefetchers to dynamically adapt to the workload behavior as their main limitation. Our framework can be potentially used in conjunction with prefetching to address this key drawback. They can complement each other to achieve a synergistic effect.

7.4 Dynamic Set Sampling

Dynamic set sampling (DSS) was previously used to achieve multiple goals. The key intuition behind set sampling is that it is sufficient to monitor a relatively small fraction of the sets in the cache in order to understand the behavior of the entire cache. DSS was used in conjunction with set dueling to decide which of two or more policies performs the best at any given point. This technique was used to select the best-performing replacement policy: LIP versus BIP [6], MLP-aware versus traditional [24], and SRRIP versus BRRIP [7]. In a system with private LLCs, it was also used to determine if each cache should act as a spiller or a receiver [25]. In the context of a shared cache, it was used to determine whether each thread among a group of threads sharing the cache should implement LIP versus BIP policy [5]. Also, in the context of a shared cache, DSS was used independently (without set dueling) to partition the ways of the cache in the best possible manner by monitoring utility [4]. To our knowledge, this is the only instance in which DSS is used to estimate the absolute value of a parameter as we used it to estimate the number of CS misses.

8. Conclusion

In a system employing multitasking, an application suffers from cache misses due to CS events in addition to the typical cache misses. CS misses happen as a result of the displacement of the cache state, which is caused by other applications intervening between two consecutive schedules of an application of interest. CS misses are more of a problem in systems that support multitasked virtualization. Such systems experience severe cache pollution as a consequence of the additional degree of multitasking, above and beyond the regular OS-level multitasking. However, the extent to which an application suffers from CS misses varies from one to another, depending on the temporal reuse behavior. Whereas some applications suffer only mildly, others suffer severely. We made the following contributions through this paper:

- We demonstrated that applications suffer by varying degrees because of context switching. In response to this phenomenon, we proposed to estimate the penalty due to a CS event and use it to facilitate intelligent time slicing by employing ETS. The intuition behind ETS is to provide longer yet infrequent time slices to those applications that are affected severely, while keeping the time slices allocated to the unaffected applications intact.
- We developed a hardware-based dynamic CS cost-estimation mechanism that incurs low area overhead. We characterized the accuracy of estimation of the proposed mechanism for multiple configurations and showed that the mechanism is very reliable.
- We provided insights into how the CS cost estimate can be incorporated into the design of a CPU-scheduling algorithm. We validated the potential of ETS to reduce the negative impact of CS events on performance without sacrificing response-time behavior.
- Furthermore, we evaluated the impact of advanced replacement algorithms and increasing the cache size on CS misses and found that these optimizations aggravate the problem associated with CS misses.

The ETS algorithm developed in this paper allocates longer time slices on the basis of their utility to applications. For various cache-management policies, the speedup obtained using the ETS algorithm is within 4% of that realized using a constant value of 10ms for the time slice. We augmented the UTS RR CPU-scheduling algorithm in order to derive the ETS RR CPU-scheduling algorithm. However, the ETS algorithm is implemented in software and can be optimized for a target system. One possible direction for future research is to investigate how CS cost estimate can be incorporated into other CPU-scheduling algorithms while respecting their original objectives. The hardware overhead of the proposed CS cost-estimation mechanism is only 0.01% for a 2MB cache. We used the estimated cost of a CS event, in terms of the number of CS misses, to modify the time slice in an elastic manner. In the case of cache-restoration prefetchers, the estimated number of CS misses can provide guidance as to when prefetching can be beneficial and when it is not likely to help. The inability of all cache-restoration prefetchers to dynamically adapt to the workload behavior has been identified as their main limitation. Our CS cost-estimation framework can be potentially used in conjunction with them to address the specified key drawback.

Acknowledgments

This work was supported in part by the Ministry of Science, ICT & Future Planning, Korea, under the R&D program supervised by the Korea Communications Agency (KCA- 2013-11921-03001), VMware, and C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

1. D. Daly and H. W. Cain, “Cache restoration for highly partitioned virtualized systems,” in *International Conference on High Performance Computer Architecture (HPCA)*, 2012, pp. 1–10.
2. A.-C. Lai, C. Fide, and B. Falsafi, “Dead-block prediction & dead-block correlating prefetchers,” in *ISCA*, 2001, pp. 144–154.
3. J. Zebchuk, H. W. Cain, V. Srinivasan, and A. Moshovos, “Recap: a region-based cure for the common cold cache,” in *International Conference on High Performance Computer Architecture (HPCA)*, 2013, pp. 83–94.
4. M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in MICRO, 2006, pp. 423–432.
5. A. Jaleel *et al.*, “Adaptive insertion policies for managing shared caches,” in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 208–219.
6. M. K. Qureshi et al., “Adaptive insertion policies for high performance caching,” in *International Symposium on Computer Architecture (ISCA)*, 2007, pp. 381–391.
7. A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, “High performance cache replacement using re-reference interval prediction (RRIP),” in *International Symposium on Computer Architecture (ISCA)*, 2010, pp. 60–71.
8. C.-J. Wu *et al.*, “Ship: signature-based hit predictor for high performance caching,” in *MICRO*, 2011, pp. 430–441.
9. A. Agarwal, J. Hennessy, and M. Horowitz, “Cache performance of operating system and multiprogramming workloads,” *ACM Trans. Comput. Syst.*, vol. 6, no. 4, pp. 393–431, 1988.
10. J. C. Mogul and A. Borg, “The effect of context switches on cache performance,” in *International Conference on Architectural Support for Programming Languages and Operating systems (ASPLOS)*, 1991.

11. G. E. Suh, E. Peserico, S. Devadas, and L. Rudolph, "Job-speculative prefetching: Eliminating page faults from context switches in time-sharing systems," 2001.

12. D. Chiou *et al*., "Scheduler-based prefetching for multilevel memories," 2001.

13. P. Koka and M. H. Lipasti, "Opportunities for cache friendly process scheduling," 2005.

14. C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Workshop on Experimental Computer Science*, 2007.

15. D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *Workshop on Experimental Computer Science*, 2007.

16. F. M. David, J. C. Carlyle, and R. H. Campbell, "Context switch overheads for Linux on ARM platforms," in *Workshop on Experimental Computer Science*, 2007.

17. A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Trans. Comput. Syst*., vol. 7, no. 2, pp. 184–215, 1989.

18. G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning," in *International Conference on Supercomputing (ICS)*, 2001, pp. 1–12.

19. G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *International Conference on High Performance Computer Architecture (HPCA)*, 2002, pp. 117–128.

20. W.-m. Hwu and T. M. Conte, "The susceptibility of programs to context switching," *IEEE Transactions on Computers*, vol. 43, no. 9, pp. 994–1003, 1994.

21. F. Liu and Y. Solihin, "Understanding the behavior and implications of context switch misses," *ACM Trans. Archit. Code Optim*., vol. 7, no. 4, pp. 21:1–21:28, 2010.

22. H. Cui and S. Sair, "Extending data prefetching to cope with context switch misses," in *International Conference on Computer Design (ICCD)*, 2009, pp. 260–267.

23. J. A. Brown, L. Porter, and D. M. Tullsen, "Fast thread migration via cache working set prediction," in *International Conference on High Performance Computer Architecture (HPCA)*, 2011, pp. 193–204.

24. M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *International Symposium on Computer Architecture (ISCA)*, 2006, pp. 167–178.

25. M. K. Qureshi, "Adaptive spill-receive for robust high-performance caching in CMPs," in *International Conference on High Performance Computer Architecture (HPCA)*, 2009, pp. 45–54.