Characterizing and Modeling the Behavior of Context Switch Misses *

Fang Liu, Fei Guo, Yan Solihin
Dept. of Electrical and Computer Engineering
NC State University
{fliu3,fguo,solihin}@ece.ncsu.edu

Seongbeom Kim
VMWare Inc.
skim@vmware.com

Abdulaziz Eker
Scientific and Technological Research Council
Ankara, Turkey
aziz.eker@uzay.tubitak.gov.tr

ABSTRACT

One of the essential features in modern computer systems is context switching, which allows multiple threads of execution to time-share a limited number of processors. While very useful, context switching can introduce high performance overheads, with one of the primary reasons being the cache perturbation effect. Between the time a thread is switched out and when it resumes execution, parts of its working set in the cache may be perturbed by other interfering threads, leading to (context switch) cache misses to recover from the perturbation.

The goal of this paper is to understand how cache parameters and application behavior influence the number of context switch misses the application suffers from. We characterize a previously-unreported type of context switch misses that occur as the artifact of the interaction of cache replacement policy and an application's temporal reuse behavior. We characterize the behavior of these "reordered misses" for various applications, cache sizes, and the amount of cache perturbation. As a second contribution, we develop an analytical model that reveals the mathematical relationship between cache design parameters, an application's temporal reuse pattern, and the number of context switch misses the application suffers from. We validate the model against simulation studies and find that it is accurate in predicting the trends of context switch misses. The mathematical relationship provided by the model allows us to derive insights into precisely why some applications are more vulnerable to context switch misses than others. Through a case study, we also find that prefetching tends to aggravate the number of context switch misses.

Categories and Subject Descriptors

B.3.3[**Memory Structures**]:Performance Analysis and Design Aids-*Simulation*;
C.4[**Performance of Systems**]:Modeling techniques;

General Terms: Algorithms, Design, Experimentation, Performance

Keywords: Context switch misses, Stack distance profiling, Analytical model, Prefetching

*This work is partially supported by NSF Award CCF-0347425 and gifts from Intel. Abdulaziz Eker and Seongbeom Kim contributed to this work when they were MS and PhD students at NCSU.

PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.

1. INTRODUCTION

One of the essential features in modern computer systems is context switching. It allows a system to provide an illusion of many threads of execution running simultaneously on a limited number of processors by time-sharing the processors. While very useful, context switching introduces high overheads *directly* and *indirectly* [2, 5, 6, 14, 17, 21]. Direct context switch overheads include saving and restoring processor registers, flushing the processor pipeline, and executing the OS scheduler. Indirect context switch overheads include the perturbation of the cache and TLB states. When a process/thread is switched out, another process/ thread runs and brings its own working set to the cache. When the switched-out process/thread resumes execution, it has to refill the cache and TLB to restore the state lost due to the context switch. Prior research has shown that indirect context switch overheads, mainly the cache perturbation effect, are significantly larger than direct overheads [5, 6, 14, 21]. We refer to the extra cache misses that occur as a result of the cache perturbation by context switching as *context switch cache misses*.

The number of context switch misses suffered by a thread is determined by the *frequency* of context switches as well as the number of context switch misses that occur after each context switch. The factors that affect the frequency of context switches and how they affect it are relatively easy to pinpoint and straightforward to understand. For example, the frequency of context switches is proportional to the level of *processor sharing*, i.e. the number of threads/processes that time-share a single processor. Factors that tend to increase the degree of processor sharing include thread-level parallelism and virtualization. Factors that tend to decrease the degree of processor sharing include multicore design. The frequency of context switches is inversely proportional to the time quantum size allocated by the OS.

However, factors that affect the number of context switch misses after a single context switch are not well understood. For example, it seems reasonable to expect that the part of the working set of a thread displaced from the cache due to a context switch will need to be fully reloaded back into the cache by the thread through cache misses, which we refer to as *required reload misses*. One may expect that the number of context switch misses a thread suffers from to be closely related to the number of required reload misses. For the ease of discussion, we refer to this expectation as the *re-*

quired reload misses hypothesis. Unfortunately, the hypothesis is grossly inadequate at explaining the actual behavior of context switch misses. Figure 1 shows the average number of context switch misses that occur on a single context switch for eleven SPEC2006 benchmarks [18]. Each benchmark time-shares a single processor with *libquantum* using 5ms time quantum, for a period of 2 billion instructions. The context switch misses are collected on a full system simulation that runs an unmodified Linux operating system (OS), with the processor having a 1MB L2 cache [1]. The x-axis shows the applications sorted based on the L2 cache miss rate (shown in the parentheses) of each application when it runs alone on a dedicated processor.

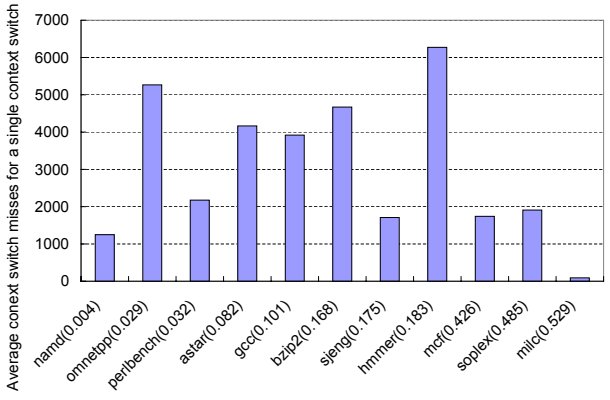

Figure 1: The average number of context switch misses for a single context switch suffered by SPEC2006 applications running on a processor with a 1MB L2 cache when they are so-scheduled with *libquantum*.

The required reload misses hypothesis cannot explain the variation of context switch misses across benchmarks, since the number of bytes in the working displaced by *libquantum* on a single context switch is roughly the same across all benchmarks. The reason for this is because the hypothesis ignores the fact that not all displaced bytes of the working set of a thread are going to be accessed again by the thread when it resumes execution. Hence, the temporal reuse pattern of an application affects its context switch misses, but the exact relationship between them is still unclear. For example, ignoring cold misses, the cache miss rate of an application represents how likely a block that was used in the past and has been replaced from the cache will be reused. However, the figure shows that there is no apparent correlation between an application's miss rate with how many context switch misses it suffers from. For example, despite having similar miss rates, *hmmer* suffers from more than three times the context switch misses that *sjeng* suffers from.

Another reason why the required reload misses hypothesis is inadequate is that it assumes that context switch misses only arise due to the displaced working set. However, cache perturbation actually affects context switch misses through another channel. Specifically, it causes the non-displaced part of the working set to become less "fresh" in the cache, as it is shifted in its recency (or stack) order to be closer to the *least recently used* (LRU) position in the cache. This recency reordering causes the reordered cache blocks to have an elevated chance to be replaced by the *thread itself* when it resumes execution, before the thread has a chance to reuse them. Because these blocks are replaced by the thread itself (rather than by interfering threads from other applications), these misses have not been correctly reported as context switch misses in prior studies [1, 2, 12, 20], causing the number of context switch misses to be systematically under-estimated. To give an illustration of the magnitude of the under-estimation, Figure 2 shows the fractions of two types of context switch misses for twelve SPEC CPU2006 applications. *Replaced misses* are context switch misses due to the part of working set that is displaced by the interfering thread, while *reordered misses* are ones due to the part of working set that is merely reordered in the cache. The evaluation setup is the same as in Figure 1, except that the result shown for each application is the average context switch misses when the application is co-scheduled with eleven other applications. The figure shows that reordered misses account for between 10% to 28% of the total context switch misses for a 1-MB cache and slightly higher for a 2-MB cache. In other words, not counting reordered misses as context switch misses may under-estimate the number of context switch misses rather significantly.

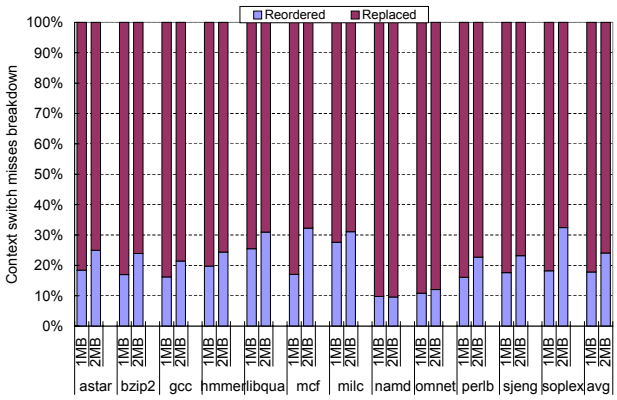

Figure 2: Two types of context switch misses suffered by SPEC2006 applications running on a processor with 1-MB L2 cache or 2-MB cache.

Clearly, a simple hypothesis such as the required reload misses hypothesis is inadequate in explaining what factors affect the number of context switch misses a thread suffers from and how they exactly affect it. Hence, the goal of this paper is to *understand how cache parameters, a thread's temporal reuse patterns, and the amount of cache perturbation influence the number of context switch misses the thread suffers from*. We hope that a better understanding of the nature of context switch misses can help researchers in coming up with techniques to reduce them. Our first contribution is characterizing context switch misses, especially focusing on the reordered misses: their magnitude, how they affect different applications, and how they are affected by cache sizes and the amount of cache perturbation. The main findings are that context switch misses can contribute to a significant increase in the total L2 cache misses, they tend to increase along with the cache size up until the cache size is large enough to hold the entire combined working sets, reordered misses tend to contribute to an increasing fraction of context switch misses as the cache size increases, and the maximum number of reordered misses occurs when cache perturbation displaces roughly a half of the total cache blocks.

Our second contribution is an analytical model-based investigation to establish how the temporal reuse pattern of a thread, the amount of cache perturbation, and the number of context switches that the thread suffers from are *mathematically related*. Compared to empirical studies, such

[1]Please refer to Section 3 for details on other processor and memory hierarchy parameters.

mathematical relationship allows us to gain clearer insights into the behavior of context switch misses. We validate our Markov-based model against the results obtained from full-system simulation of SPEC2006 applications. We find that the model is sufficiently accurate in predicting the trends of context switch misses. The model allows us to derive insights into precisely why some applications are more vulnerable to context switch misses than others. Then, as a case study, we apply the model to analyze how prefetching and context switch misses interact with each other in various applications. The investigation leads us to a previously-unreported observation that prefetching can aggravate the number of context switch misses an application suffers from. In addition, perversely, the more effective prefetching is for an application, the higher the number of context switch misses the application suffers from.

The rest of this paper is organized as follows: Section 2 reviews related work, Section 3 characterizes the context switch misses across different applications, Section 4 describes the analytical model in details, Section 5 analyzes the validation results, Section 6 discusses the case study on prefetching, and Section 7 concludes this paper.

2. RELATED WORK

There have been many empirical studies aiming at understanding the magnitude and behavior of context switching overheads [5, 6, 12, 13, 14, 17, 21]. Mogul and Borg [17] evaluated the impact of context switching on the cycles per instruction (CPI) and cache miss rates for various cache parameters and found that the overheads for an average context switch could be up to tens to hundreds of microseconds. Kwak et al. [13] evaluated multithreaded systems and showed that fast context switching could improve performance by exploiting data locality. Koka and Lipasti [12] characterized context switch misses of an application for various cache parameters and investigated how a potentially optimum scheduling policy could reduce them. Li et al. [14], and Fromm and Trehaft [6] measured both direct and indirect context switching overheads through simulation and concluded that indirect context switch overheads due to the cache perturbation effect are much more significant than direct overheads. Tsafrir [21] and David et al. [5] performed similar experiments on Intel Pentium and ARM platforms respectively and arrived at the same conclusion. Li et al. [14] also showed that the working set and data access patterns of an application could significantly affect the context switch overheads. Our study complements the findings from prior studies. While prior studies did not identify context switch misses as consisting of two types (replaced and reorder misses), our study identifies reordered misses as context switch misses and establishes the interaction between them and the LRU cache replacement policy, cache size, and the amount of cache perturbation.

In addition, previously there were no studies that could show the exact relationship between an application's temporal reuse pattern with its vulnerability from suffering context switch misses. Empirical studies, either simulation studies or measurement on real systems, cannot reveal the underlying mathematical relationships between the various factors that affect the context switch misses and the number of context switch misses an application suffers from. Analytical models can potentially fill that role but they need to capture all essential variables in order to have a sufficient resolution. Several analytical models have been proposed in the past [1, 7, 19, 20]. Thiebaut and Stone [20] modeled context switch misses by considering the cache size and the working set size of an application in a time-shared processor environment. However, their model ignores the impact of the temporal reuse patterns of the application.

Agarwal et al. [1] proposed an analytical cache model to predict the overall cache miss rate including context switch misses. Agarwal's model takes into account the temporal reuse patterns of applications, but it ignores the interaction of cache replacement policy and the behavior of context switch misses. The model only captures the replaced context switch misses and ignores the reordered context switch misses.

Suh et al. [7, 19] proposed an analytical model for estimating the total miss rate of an application that includes all types of context switch misses. Unfortunately, the model requires a *continuous miss rate curve* as profiling information, i.e. the miss rate of a thread for any given cache size, including non-integer values. Such profiling information is difficult to obtain since cache parameters (size, associativity, block size) have discrete rather than continuous values. While discrete data series can be interpolated into continuous one, the interpolation only produces accurate modeling if there are many data points. Hence, the model in [7, 19] assumes a small cache that is fully-associative, which is an unrealistic assumption for L2 or L3 caches, in which the overheads due to context switch misses are more significant than in a L1 cache.

The model proposed in the second part of our study models the relationship between temporal reuse patterns of applications, cache configurations, and the LRU cache replacement policy. It includes all types of context switch misses, relies on a realistic profiling information, and is appropriate for modeling large and set associative caches.

3. CHARACTERIZING CONTEXT SWITCH MISSES

In this section, we will present the first contribution of this paper on characterizing the behavior of reordered misses for various applications, cache sizes, and degrees of cache perturbation.

3.1 Types of Context Switch Misses

Figure 3 illustrates the causes and types of context switch misses. The figure shows a single cache set in a 4-way set associative cache. Figure 3(a) shows the cache state right before a process is context switched, containing blocks A, B, C, and D in use-recency order with A being the *most-recently used* (MRU) block while D being the *least-recently used* (LRU) block. When the process resumes execution after the context switch, the cache has been perturbed by another process which left one of its cache blocks at the MRU position (indicated by the "hole" in Figure 3(b)). The cache perturbation causes block D to be replaced from (or shifted out of) the cache. Suppose that now the application chronologically accesses block D, C, and then E. Without the cache perturbation, only the access to block E would result in a cache miss. With the cache perturbation, the access to D results in a cache miss (Figure 3(c) shows the resulting cache state). Since D was *replaced* by the perturbation, we refer to the miss as a *replaced context switch miss*. Next, when the process accesses block C, it also suffers from a cache miss (Figure 3(d) shows the resulting cache state). Note that block C was merely reordered in the LRU stack and was not replaced by the cache perturbation, so the fact that it causes a cache miss is the artifact of the LRU cache replacement policy. We refer to such a miss as a *reordered context switch miss*. Finally, an access to E results in a cache miss, but it is not a context switch miss since it occurs regardless of whether there was a context switch.

From the figure, we can observe that the number of context switch misses after a context switch is influenced by the probability that replaced or reordered blocks will be accessed

again when a thread resumes execution. Such probability is affected by the temporal reuse pattern of the application, and the specific causal relationship will be established in Section 4. In this section, we are interested in the composition of context switch misses: what fraction of them is due to reordered misses versus due to replaced misses. Clearly, if the context switch results in the entire cache state to be displaced from the cache, then all context switch misses will be replaced misses, since no block is reordered in the cache. Hence, reordered misses only occur when only a fraction of the cache is displaced during a context switch.

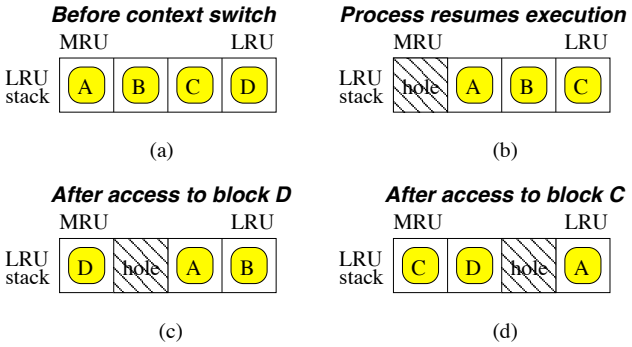

Figure 3: The content of a cache set right before a process is context switched (a), when it resumes execution (b), after an access to block D (c), and after an access to block C (d).

3.2 Characterization Methodology

We use a cycle-accurate processor simulator based on Simics [15], a full-system processor simulation infrastructure. We run an unmodified Fedora Core 4 distribution of the Linux Operating System on the processor. The processor has a scalar in-order issue pipeline with a 4GHz frequency. The memory hierarchy has two cache levels. The L1 instruction cache has a 16KB size, 2-way associativity, and 1-cycle access latency. The L1 data cache has a 32KB size, 4-way associativity, and 2-cycle access latency. The L2 cache is 8-way associative, has an 8-cycle access latency, and its cache size ranges from 512KB to 4MB. All caches have the block size of 64-byte, implement write-back policy and use the LRU replacement policy. To collect the number of context switch misses, we implement two L2 caches: one *real* L2 cache that is affected by the cache perturbation, and a hypothetical *isolated* L2 cache that is not affected by the cache perturbation. Each L1 cache miss is passed to both caches to determine whether it will hit or miss on each of the L2 caches. The difference in the numbers of cache misses between these two caches measures the number of L2 context switch misses.

We consider all seventeen C/C++ benchmarks from the SPEC2006 benchmark suite [18]. We omit benchmarks written in Fortran because of a limitation in our compiler infrastructure. Table 1 shows each benchmark's L2 cache miss rates (number of L2 cache misses divided by L2 cache accesses) and the L2 cache miss frequency (number of L2 cache misses per unit time) on a 512KB cache. Of the seventeen benchmarks, we choose twelve representative applications that have distinct temporal reuse patterns (including miss rates) and miss frequency. We compile the benchmarks using `gcc` compiler with `O1` optimization level into `x86` binaries. We use the *ref* input sets and simulate two billion instructions after skipping the initialization phase of each benchmark, which is identified through manual inspection of the source code.

Table 1: SPEC2006 benchmarks considered. The unit for L2 miss frequency is the average number of cache misses in a single cache set that occur in 50 milliseconds. The L2 cache size is 512KB.

Benchmarks	Miss Rate	Miss Frequency
astar	10.72%	181
bzip2	27.35%	224
gcc	14.15%	206
gobmk	3.23%	96
h264ref	0.66%	3
hmmer	33.38%	131
lbm	32.27%	465
libquantum	50.81%	269
mcf	46.64%	602
milc	56.26%	79
namd	0.48%	5
omnetpp	13.32%	94
perlbench	2.85%	98
povray	0.34%	5
sjeng	20.00%	206
soplex	54.76%	488
sphinx3	70.96%	515

3.3 Characterization Results

Figure 4 shows the breakdown of the L2 cache misses for twelve SPEC2006 benchmarks when each benchmark is co-scheduled with another interfering benchmark, for various cache sizes. Since there are eleven other benchmarks, there are eleven possible co-schedules. Hence, the number of L2 cache misses of each benchmark is taken as the average over eleven co-schedules the benchmarks can have, normalized to the case in which 512-KB cache is used. A time slice of 5ms is used for both benchmarks in a co-schedule. The Linux kernel version 2.6 assigns time quanta between 5 ms to 200 ms based on a process/thread priority level, so 5ms corresponds to the lowest priority level.

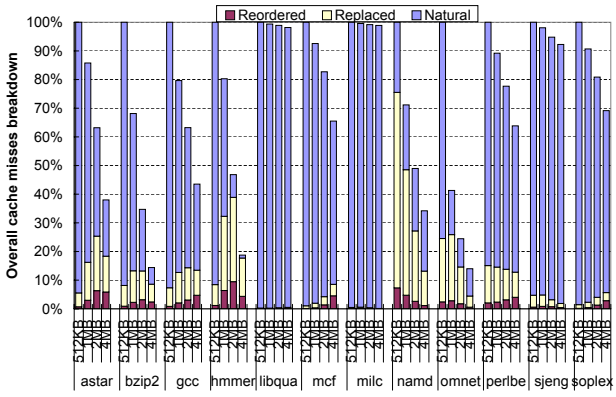

Figure 4: Breakdown of the types of L2 cache misses suffered by SPEC2006 applications with various cache sizes.

The figure shows that as expected, as the cache size increases, the "natural misses" (the sum of cold, capacity, and conflict misses) decrease. However, the number of context switch misses exhibits very different behavior. Sometimes, they decline (in namd, perlbench), increase (in mcf, soplex), or increase then decline (in astar, bzip2, gcc, hmmer, omnetpp and sjeng). The reason for this behavior is because when the cache is very small, there are not many cache blocks that can be displaced by a cache perturbation, so the number of context switch misses is low. As the cache be-

comes larger, a cache perturbation may displace more cache blocks, causing more context switch misses. However, when the cache becomes large enough to hold the total working sets of both benchmarks, cache perturbation only displaces very few blocks, so the number of context switch misses declines. Rather than being displaced, most blocks are now reordered, but reordered blocks do not incur context switch misses if there are no other misses to cause them to be displaced when the thread resumes execution. Hence, the absolute number of reordered misses also declines as the cache becomes large enough.

The figure also shows that context switch misses can represent a significant fraction of the total L2 cache misses especially on larger cache sizes, which implies that they need to be taken into account when designing a system. If an application uses a larger time quantum and uses it up all the time, then number of context switch misses will be small. However, applications with low priorities are granted small time quanta so they likely suffer significantly from context switch misses. In addition, some applications that are I/O intensive often block before they use up a time quantum. Finally, applications that time-share a processor with an I/O intensive application are also forced to context switch very frequently due to interrupts. For example, when the GNU compiler `gcc` and a secure copy application `scp` are co-scheduled, both suffer from a very frequent context switches and each of them runs for much less than 1ms before it is context switched out (versus 5ms we use in this study).

How does the composition of replaced and reordered misses change with different cache sizes? Figure 5 shows the number of reordered vs. replaced misses as a fraction of the total context switch misses. When the cache size is small, a cache perturbation is more likely to displace cache blocks than reorder them, so the fraction of reordered misses is small (roughly 15% on average for a 512KB L2 cache). When the cache size increases, a cache perturbation is more likely to reorder cache blocks than to replace them, so the fraction of reordered misses increases (to roughly 32% on a 4MB L2 cache). In some applications, reordered misses even account for roughly 50% of all context switch misses (libquantum, mcf, and soplex in 4MB cases). From the figure, we observe that reordered misses are a significant component of context switch misses, and counting only replaced misses as context switch misses as in prior studies [1, 2, 12, 20] can result in significantly and systematically under-estimating the total number of context switch misses.

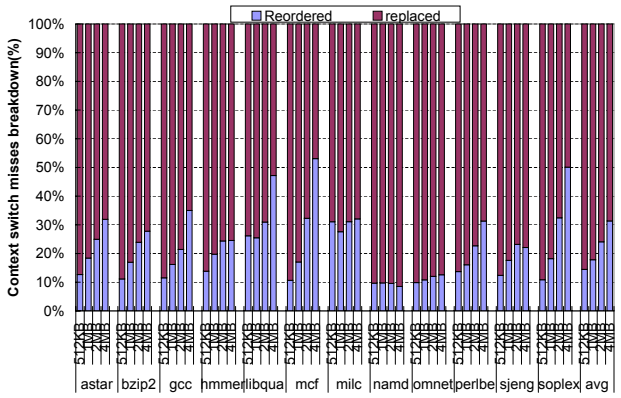

Figure 5: The composition of context switch misses for various cache sizes.

The number of context switch misses an application suffers from when it is co-scheduled with other applications are essentially determined by two factors: the amount of cache perturbation (how many blocks are reordered and replaced from the cache), and how likely the affected blocks will be needed again by the application. The amount of cache perturbation depends on how many interfering applications there are, how long they run, and how fast they bring in new cache blocks to cause the cache perturbation. Since it is difficult to produce all possible scenarios, in order to better understand the behavior of reordered misses, we abstract the amount of cache perturbation by a single metric which we call the *shift amount*. During a cache perturbation, when a new block is brought in, it is placed at the most recently used entry in a cache set, causing all other blocks in the set to be shifted in the recency order, with the LRU block displaced from the cache set. The shift amount refers to the number of new blocks brought into each cache set, which is also equal to the number of positions old blocks are shifted in their recency order. A larger shift amount indicates more cache perturbation from other applications. The maximum value of shift amount is by definition equal to the cache associativity.

With the amount of cache perturbation abstracted as the shift amount, we run each of the twelve SPEC2006 benchmark, and every 5ms, we pause the benchmark, perturb the L2 cache by the shift amount, and resume the benchmark execution. Figure 6 shows the percentage increase in the number of L2 cache misses due to context switching. From this figure, we can make a few observations. First, the number of context switch misses monotonically increases as the shift amount increases for all benchmarks. The reordered misses, however, reach their maximum when the shift amount is 4 (50% of the cache size) or 6 (75% of the cache size). The number of reordered misses is affected by the number of blocks reordered and how many recency order positions they are shifted. With a small shift amount, many blocks are reordered but they are only slightly shifted. As the shift amount grows, fewer blocks are reordered but they are shifted by more positions in the recency order, causing an increase in reordered misses. However, a very large shift amount implies that very few blocks are reordered, so the number of reordered misses declines.

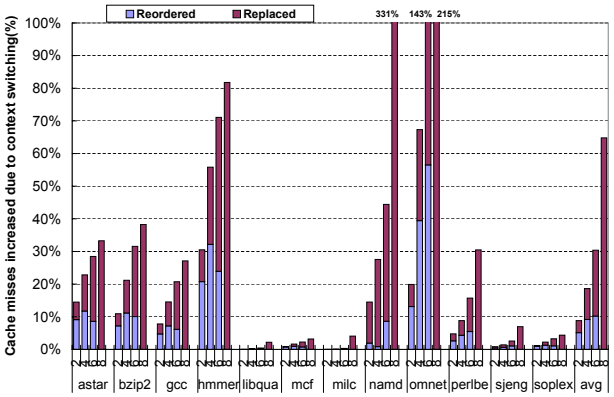

Figure 6: Increase in the L2 cache misses due to context switch misses for various shift amounts (2, 4, 6, and 8) on an 8-way associative 1-MB L2 cache. A shift amount of 8 means the entire cache state is replaced.

The analysis explains why the fraction of reordered misses increases with larger cache sizes in Figure 5. With the time quantum unchanged, larger cache sizes correspond to going from a very large shift amount to a smaller shift amount, causing an increase in the fraction of reordered misses.

Overall, from this characterization study, we have learned that the number of context switch misses tends to increase as the cache size increases but only up to a point in which it starts to decline. Context switch misses also correspond to an increasing fraction of total cache misses. The number of reordered misses tends to reach its peak when cache perturbation affects roughly a half of the cache size, while the fraction of reordered misses of the total context switch misses increases as the cache size increases. The implication of these observations to cache design is: designers must take into account context switch misses in addition to other types of misses, especially when the level of processor sharing is high (i.e. high cache perturbation), and when the cache size is relatively large.

Although simulation studies could help us understand major behavior trends of context switch misses, they are insufficient for understanding how exactly different applications suffer from different numbers of context switch misses. To gain such understanding, as a second contribution, in the next section we derive the mathematical relationship between an application's temporal reuse pattern and the behavior of context switch misses through an analytical Markov-based model.

4. THE ANALYTICAL CACHE MODEL

4.1 Assumptions and Input of the Model

Estimating how many context switch misses an application will likely suffer from requires us to determine the probability of an individual cache access incurring a context switch miss, and sum up such probabilities over a group of cache accesses. From the example in Figure 3, estimating the probability of a single cache access incurring a context switch miss requires us to take into account: (1) the amount of cache perturbation introduced by the context switch, (2) the temporal reuse pattern of the application, (3) the cache replacement policy, and (4) and the location of the holes introduced by the context switch at the time the access is made. The location of the holes matters because reuses to cache blocks that are in more-recent stack positions than the holes do not change the location of the holes, while those that are in less-recent stack positions than the holes shift the position of the holes.

The temporal reuse pattern of an application can be captured through a popular profiling technique called the *stack distance profiling* [3, 4, 9, 16], or sometimes also referred to as marginal gain counters [19]. Basically, the stack distance profiling records the distribution of accesses to different LRU-stack positions, either as global information for the entire cache, or as local information for each cache set. To collect the global information, for an A-way associative cache, $A+1$ counters are kept: $C_1, C_2, \ldots, C_A, C_{>A}$. The LRU-stack positions are enumerated, with the MRU position as the first stack position, and the LRU position as the A^{th} position. On each cache access (to any set), one of the counters is incremented. A cache access to a block at the i^{th} LRU-stack position increments C_i. If a cache access does not find the cache block in the LRU stack, it increments $C_{>A}$. After the access, the stack is updated, with the block recently accessed moved up to the first stack position, while other blocks between the first stack position and the old position of the accessed block are shifted down. Collected over a long period, the stack distance profile can be used to estimate the probability that a block in the i^{th} stack position to be reused in the next cache access as:

$$P_{reuse}(i) = \frac{C_i}{C_{>A} + \sum_{j=1}^{A} C_j} \quad (1)$$

To collect the local (per-set) profiling information, we can maintain one set of counters for each cache set. If cache accesses are uniformly distributed across all cache sets, the global stack distance profile is sufficient in capturing the overall temporal reuse pattern of the application. Due to its simplicity, we collect the global stack distance profile. A stack distance profile can be obtained statically through static analysis [4], or at run time through simulation or direct hardware implementation [19].

When a cache brings in a new block, it must find a block to replace to make room for the new block. Which block is selected for replacement is determined by the cache replacement policy. Applications often have regular *temporal reuse* behavior in which it tends to reuse more recently used data more than less recently used data. For this reason, most cache implementations today implement least-recently used (LRU) replacement policy or its approximations. Hence, in our model we assume LRU cache replacement policy.

Similar to the characterization in Section 3.3, we abstract the amount of cache perturbation as the *shift amount*, i.e. the number of stack positions a process' data blocks are shifted in the LRU stack, which is also equivalent to the number of holes introduced in each cache set. We refer to such number as the *shift amount*, denoted as δ. The range of values for δ is $\delta = 0 \ldots A$.

We further assume that the location of the holes in all cache sets to be contiguous, which rests on two smaller assumptions. First, consecutive context switches are sufficiently separated in time that holes from one context switch have been completely shifted out prior to the next instance of context switch. The assumption is mostly valid given that time quanta used by Operating Systems today are in the order of at least a few milliseconds. Second, processes that time-share a processor do not share data, which is mostly valid for independent and unrelated processes. Communicating processes may have a high degree of data sharing, but context switching is not necessarily harmful for them, so we do not seek to model them. Finally, the assumption of contiguous holes is not a fundamental restriction of our model. Its purpose is to simplify the model by reducing the state space. The assumption can be relaxed at the expense of additional model complexity.

4.2 Model Formulation

We will start by defining some basic terms that we will use throughout the discussion. The process of which the context switch misses we want to model is referred to as the *target process*. The target process' miss rate without cache perturbation from context switches is referred to as the *natural cache miss rate*, computed as the ratio of the number of non-context switch misses divided by the number of cache accesses. To simplify the discussion, we begin our model formulation for a *single cache set* and a *single instance of context switch*. The cache state necessary for our model is defined as follows.

DEFINITION 1. *For a single cache set in an A-way associative cache, a* **cache state** *is a tuple (c, h) where c is the capacity (or number of blocks) belonging to the target process, while h is the most-recently-used stack position of holes in the LRU stack.*

The range of values for c is $c = 0 \ldots A$ while for h is $h = 1 \ldots A+1$, where $h = A+1$ indicates that the holes have been completely shifted out of the cache set. An important invariant is $h \leq c+1$ because the farthest position holes can start is when all c cache blocks of the target process are lumped together starting at the MRU position. As an example, the state is $(3, 1)$ in Figure 3(b), $(3, 2)$ in Figure 3(c), and $(3, 3)$ in Figure 3(d).

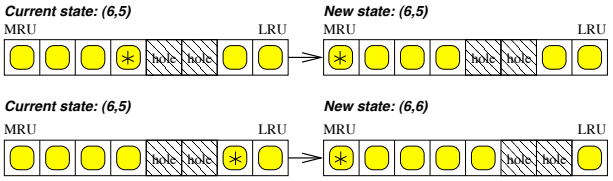

Figure 7: The state transition illustration for the case in which the right-most hole is *not* at the LRU position. Accessed blocks are marked by asterisks.

The first step in our approach of modeling context switch misses is that for a given state, we find out what events can occur and what the resulting new states can be. To do that, we distinguish two cases: one case in which the right-most hole is not at the LRU position, versus the other case in which the right-most hole is at the LRU position. The former case occurs when $c \geq h$ and is illustrated in Figure 7, while the latter occurs when $h = c + 1$ and is illustrated in Figure 8. Figure 7 shows a process having six cache blocks in an 8-way associative cache set, with two holes starting at the fifth stack position, corresponding to state $(6, 5)$. In the upper case, the process accesses the block to the left of the holes at the fourth stack position (marked by an asterisk), resulting in the block moving to the MRU stack position, but the holes remain where they are. The new state is the same as the current state. In the lower case, the access is to a block to the right of the holes at the seventh stack position, causing the block to move up to the MRU position and shift the holes down one stack position. The new state is now $(6, 6)$.

The second case in which the holes are at the end of the LRU stack is shown in Figure 8. The current state is $(6, 7)$. In the upper case, the process accesses the block to the left of the holes at the fourth stack position, resulting in the block moving to the MRU stack position, but the holes remain where they are. The new state is the same as the current state. In the lower case, the access is to a block not found in the LRU stack, causing the block to be brought into the MRU position and the holes shifted down one stack position and the right-most hole shifted out, so the new state is $(7, 8)$.

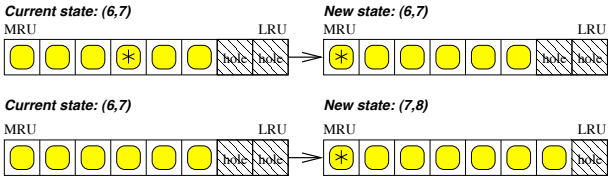

Figure 8: The state transition illustration for the case in which the right-most hole is at the LRU position. Accessed blocks are marked by asterisks.

These cases and their state transitions can be captured using Markov modeling. However, rather than starting with a current state and determines what next states it may transition into, we reverse the process and start with a current state and determine what previous states can possibly lead to the current state. In addition, we need to consider one more input variable N, which specifies how many accesses to a cache set a process will make when it resumes execution after a context switch. An iterator variable n will initially take the value of 0 and as the Markov model makes the state transitions on each access, n is incremented until it reaches N. Hence, we modify the definition of a state to include n.

DEFINITION 2. *For a single cache set in an A-way associative cache, a* **cache state** *is a tuple (c, h, n) where c is the capacity (or number of blocks) belonging to the target process, h is the most-recently-used stack position of holes in the LRU stack, and n is the number of accesses to the cache set that has occurred so far.*

The Markov model corresponding to the current state (c, h, n) is shown in Figure 9. One possibility to reach the current state is when the previous state $(c, h, n-1)$ and the n^{th} cache access uses a block that is to the left of the holes. In this case, the holes *stay* in their previous positions. The probability of such transition is thus the probability of accessing blocks in stack position $1, 2, \ldots, h-1$, i.e. $P_{stay}(h) \equiv \sum_{j=1}^{h-1} P_{reuse}(j)$, where $P_{reuse}(j)$ is obtained from Equation 1.

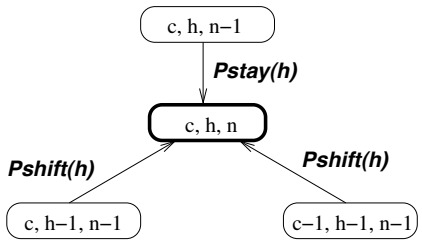

Figure 9: The Markov model for what previous states can lead to the current state (c, h, n).

Another possible previous state is $(c-1, h-1, n-1)$. When the rightmost hole is at the LRU stack position (i.e., $h = c+1$), an access that misses in the LRU stack will bring a block into the cache set and *shift* the holes by one position. This case only occurs when the application accesses a block that is not one of the $c-1$ blocks previously in the LRU stack, i.e. the access is to blocks in position $h-1, h, \ldots, \infty$. Hence, the probability of this state transition is $P_{shift}(h) \equiv P_{reuse}(> A) + \sum_{j=h-1}^{A} P_{reuse}(j)$. The final possible previous state is $(c, h-1, n-1)$. When the right-most hole is not at the LRU stack position (i.e., $h < c+1$), an access to a block that is to the right of the holes shifts the holes by one stack position but does not increase the number of blocks of the target process. Since blocks to the right of the holes have stack distances of $h-1, h, \ldots, \infty$, the probability of this state transition is the same as before: $P_{shift}(h) \equiv P_{reuse}(> A) + \sum_{j=h-1}^{A} P_{reuse}(j)$.

Now we would like to express the Markov model in Figure 9 with a mathematical expression. Let $P(c, h, n)$ denote the probability of the state (c, h, n) being reached in n accesses. We first note when the process resumes execution, it will find its cache blocks are already shifted by the shift amount δ, where δ is also the number of holes starting from the MRU position. Hence, the initial state is $(A-\delta, 1, 0)$. The probability to be in the initial state is 1, since it is a given state. Hence, the mathematical expression for the Markov model can be written as in Figure 10.

The first line in the expression states that the probability to reach the initial state when $n = 0$ is 1. The second line is directly taken from the Markov process in Figure 9. When $h = c+1$, all three previous states can possibly lead to the current state with each own transition probability. The third line is when $h < c+1$. In this case, the previous state cannot be $(c-1, h-1, n-1)$ since the LRU block is not a hole, so there are only two possible previous states: $(c, h-1, n-1)$ or $(c, h, n-1)$. Finally, the fourth line states that the probability value for all other cases is zero. With the

$$P(c,h,n) = \begin{cases} 1 & \textbf{if } c = A-\delta, h = 1, n = 0 \\ P(c,h,n-1) \times P_{stay}(h) + P(c,h-1,n-1) \times P_{shift}(h) + & \\ P(c-1,h-1,n-1) \times P_{shift}(h) & \textbf{if } c \geq A-\delta, h = c+1, n > 0 \\ P(c,h,n-1) \times P_{stay}(h) + P(c,h-1,n-1) \times P_{shift}(h) & \textbf{if } c \geq A-\delta, h < c+1, n > 0 \\ 0 & \textbf{otherwise} \end{cases}$$

Figure 10: Mathematical expression for the probability to reach a state (c, h, n).

expression, $P(c, h, n)$ can be computed recursively for any combination of (c, h, n) values. To determine the number of cache misses in a cache set that a target process likely suffers from, we note that each Markov state has its own probability of a cache miss. If the current state of (c, h, n) is already reached, a new cache access suffers a cache miss when the accessed block is not among the c blocks currently in the LRU stack. Hence, the conditional probability of miss, denoted as $P_{condmiss}(c)$, is the sum of probabilities of accessing blocks in the original stack positions $c+1, c+2, \ldots, \infty$:

$$P_{condmiss}(c) = P_{reuse}(> A) + \sum_{j=c+1}^{A} P_{reuse}(j) \qquad (2)$$

The contribution of such a miss to the overall cache misses is the product of the probability of reaching the current state and the probability that the current state causes a cache miss in the next access. Hence,

$$P_{miss}(c,h,n) = P(c,h,n) \times P_{condmiss}(c) \qquad (3)$$

To get the overall cache misses, all that is left is to sum up the probabilities of cache misses for all combination values of c, h, and n, leading to:

$$TotMisses(N) = \sum_{c=A-\delta}^{A} \sum_{h=1}^{A+1} \sum_{n=0}^{N-1} P_{miss}(c,h,n)$$

The next step in our modeling is to determine which of the total cache misses are context switch misses, and which are natural cache misses. We note that for N accesses, the number of natural cache misses is simply the product of the probability of each cache access incurring a cache miss (absence of holes) times the number of cache accesses, that is:

$$NatMisses(N) = N \times P_{reuse}(> A) \qquad (4)$$

Therefore, the estimated number context switch misses in a cache set over N accesses is:

$$CSMisses(N) = TotMisses(N) - NatMisses(N) \qquad (5)$$

Finally, the total estimated context switch misses can be obtained by summing up the estimated context switch misses of all cache sets and over all instances of context switching. For simplicity, in the rest of the paper we will assume that a constant shift amount is applied equally to all cache sets at each instance of context switching.

5. MODEL VALIDATION

5.1 Validation Methodology

To validate our model, we compare the number of context switch misses estimated by our model against that obtained from the simulator for SPEC2006 applications. The processor and memory hierarchy models are as described in Section 3, except that we use 512KB L2 cache size to reduce the set variation on the global stack distance profile. We chose fourteen applications that have relatively large miss frequency in 50ms time quanta in order to satisfy the assumption that holes are completely shifted out in one time quantum. Hence, we do not include *h264ref, namd* and *povray* in the validation.

We run one application at a time on the simulation model. Every time quantum, we perturb the "real" L2 cache by inserting δ holes to simulate the cache perturbation effect of context switching, while the "isolated" L2 cache is not perturbed. All L1 cache misses are passed to both L2 caches simultaneously. The time quantum is chosen to be 50ms, which corresponds to a relatively low priority level in Linux OS. After the perturbation, the application is resumed.

Another statistic that is collected over each time quantum is the average number of cache accesses per set, which is used as the input N to our model so that the model can produce an estimate of context switch misses for that time quantum. For validation, the number of context switch misses estimated by the model is compared against that collected by the simulation.

5.2 Validation Results

Figure 11 shows the total number of context switch misses over all instances of context switches as collected by the simulation (labeled as *sim*) and as estimated by our model (labeled as *model*), with the shift amount (δ) varied from 0 (no cache perturbation – zero context switch misses) to 8 (the entire cache contents are shifted out).

The figure shows that the predicted number of context switch misses presents remarkably similar trends (shapes of the curves) with the actual number of context switch misses. There is some divergence between them when the shift amount is large, but the similarity in the overall trend indicates that our model has captured most of the important variables that affect context switch misses. It also indicates that our model is accurate enough for identifying behavior trends of context switch misses, or for comparing which of two cache configurations yields fewer context switch misses.

We dig deeper into why the predicted number of context switch misses diverges from that collected from simulation when the shift amounts are large. For this purpose, we choose four benchmarks that exhibit the largest estimation errors from Figure 12: astar, mcf, milc, and perlbench.

First, we suspect that the stack distance profile that we collect for the entire cache may not be representative of the stack distance profile of individual cache sets. It is well-known that a relatively small number of sets have disproportionately high number of addresses that map on them (*hot sets*) while the majority of sets have fewer addresses that map on them (*cold sets*). As a result, hot sets have much higher miss rates than cold sets, and the total number of accesses required to shift holes out is much smaller in hot sets than in cold sets. Meanwhile, we use the global stack distance profile, and the number of accesses (i.e. N) fed into the model is the average over all sets. If cold sets outnumber hot sets significantly, N is slightly small for cold sets, but much too large for hot sets. As a result, the number of context switch misses is likely slightly under-estimated for cold sets, but significantly over-estimated for hot sets.

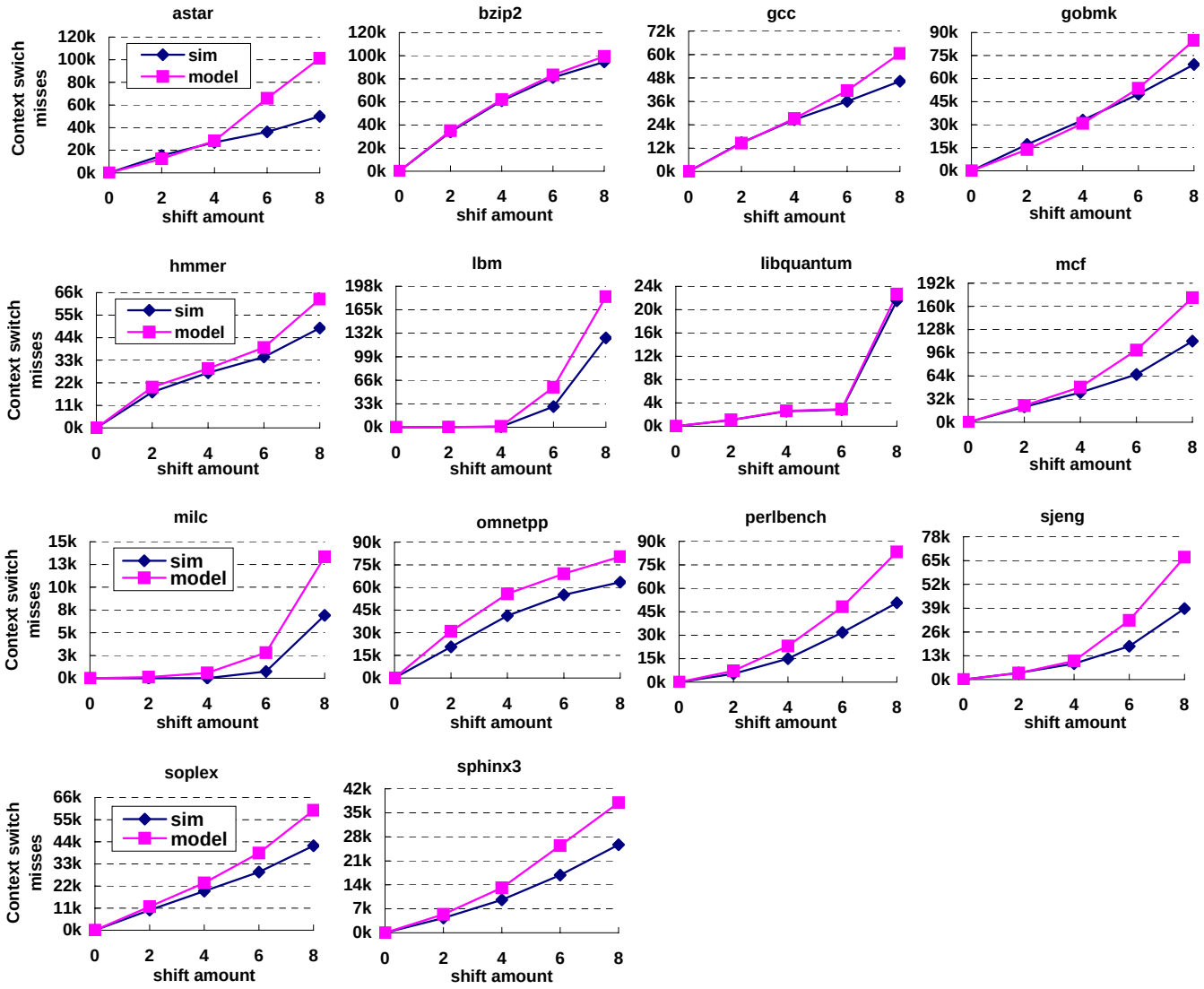

Figure 11: Total context switch misses collected by the simulation model versus estimated by our model with variable shift amounts.

To verify this guess, we choose one set of the cache, collect the local stack distance profile and the number of accesses required to shift all holes in the set out. They are input to the model to predict the number of context switch misses that occur on that set. Figure 12 shows predicted versus actual number of context switch misses for the entire cache (from Figure 11) in dashed lines, super-imposed with the predicted versus actual number of context switch misses for a single cache set in solid lines. The figure shows that the predicted and actual number of context switch misses are much closer on a single cache set than on the entire cache. This confirms that the estimation error on a large shift amount is mostly caused by the limitation of the input to the model (the global profiling information) rather than the inaccuracy of the model itself.

5.3 Why Applications Suffer from Context Switch Misses Differently

Figure 11 shows that the number of context switch misses increases differently for different benchmarks as the shift amount increases: they may increase slowly at first then rapidly later (such as in lbm), increase at a steady rate (such as in gobmk), or increase rapidly first then slowly later (such as omnetpp). This phenomena is related to why different applications suffer from context switch misses differently (recall the discussion on sjeng vs. hmmer in Section 1). The reason for this is the difference in the benchmarks' temporal reuse patterns.

Figure 13 shows the probability function of accesses to different stack positions for lbm and omnetpp. The figure shows that lbm only reuses cache blocks at the four most recent stack positions, so a shift amount of four replaces blocks that are not likely reused by lbm, resulting in slow increase of context switch misses. However, a shift amount of eight would replace blocks that are highly reused, so the context switch misses increase rapidly. In contrast, omnetpp highly reuses cache blocks at all stack distances, so even a small shift amount increases the context switch misses rapidly. As the shift amount increases from six to eight, the MRU and second MRU blocks are also replaced, so at the first glance it might seem that the context switch misses will still rise rapidly. However, note that as the shift amount grows, more blocks are replaced and fewer blocks are reordered. Hence,

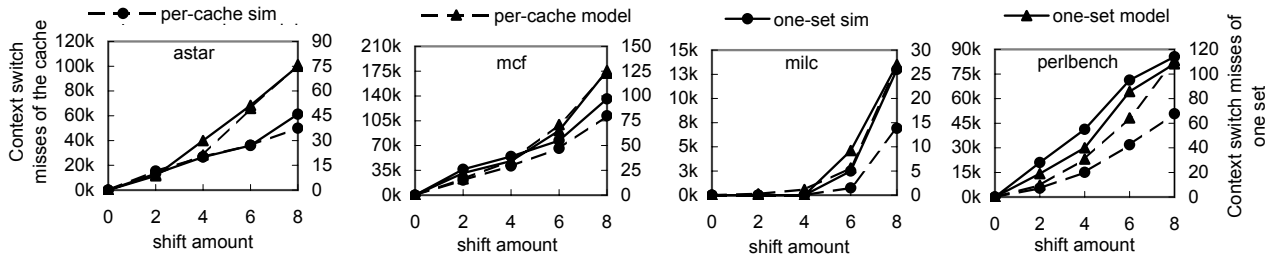

Figure 12: Comparing the prediction accuracy of our model using global (entire cache) profiling information versus local (one set) profiling information.

the number of reordered misses tapers off and becomes zero when the shift amount is eight, and this in effect significantly slows the growth of context switch misses on larger shift amounts. Overall, the figure points out that different applications react to different shift amounts differently in terms of how much they suffer from cache perturbation.

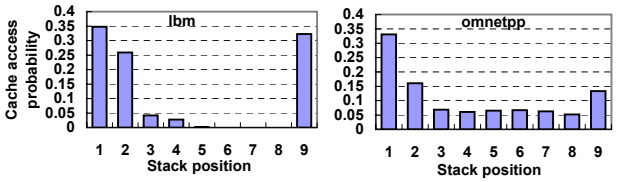

Figure 13: Stack distance probability function showing probability of accessing different stack positions.

6. CASE STUDY ON PREFETCHING

In this section, we apply the model to understand the impact of the use of prefetching on the number of context switch misses a thread suffers from. The reason why we choose prefetching is because it is commonly implemented in current processors, but studies in prefetching usually only consider natural misses (cold, capacity, and conflict) and ignore context switch misses. In the case study, we use a shift amount of eight and set the time quanta to be the time for the holes introduced by an instance of cache perturbation to be completely shifted out of the cache. Such time quanta range from 1ms to 30ms (average of 10ms) for different applications on a 512KB cache.

For the prefetching algorithm, we implement Jouppi's *stream buffers* [11], a popular prefetching algorithm whose variants are implemented in real systems [8, 10]. Stream buffers detect accesses to block addresses that form a sequential or stride pattern (called a *stream*), and prefetch the next few blocks in the stream in anticipation that the processor will continue accessing blocks from that stream. Stream buffers tend to have high *coverage* (a large fraction of cache misses can be prefetched) and high *accuracy* (most prefetched blocks are actually used by the processor), and is relatively simple to implement. We implement a stream buffer prefetcher with an unlimited number of streams, and up to four blocks can be prefetched for a single stream. The prefetched blocks are placed directly in the L2 cache.

Figure 14 shows the fractions of the original (natural or total) L2 cache misses that remains across different benchmarks after prefetching is applied, sorted in descending order based on the fraction of natural misses remaining. In other words, applications for which prefetching is more effective in eliminating natural cache misses are placed on the right. The original total misses are obtained by adding the number of natural misses with the predicted context switch misses, based on the original stack distance profile of each application. The remaining total misses are obtained by adding the number of natural misses with the predicted context switch misses, based on the stack distance profile of each application *when prefetching is applied*. Note that the divergence error we discuss in Section 5 plays little role here since it affects the total misses in both the original and prefetching cases. Also note that the model lumps together all additional fetches as context switch misses, although some of them may be due to additional prefetch requests rather than additional demand misses. We do not need to distinguish them if the focus is the total off-chip bandwidth utilization.

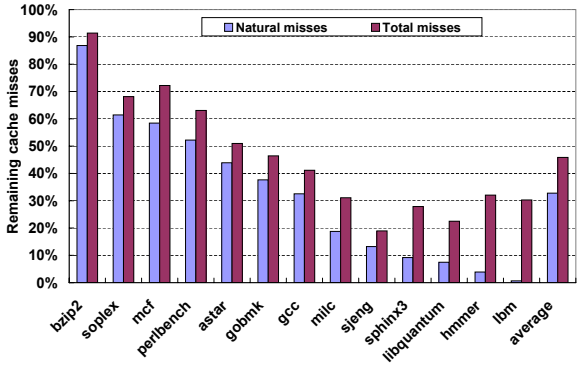

Figure 14: The fraction of the original natural and total L2 cache misses that remains after prefetching is applied.

As expected, the figure shows that stream buffer prefetching is highly effective. It removes two thirds of the natural cache misses on average. However, surprisingly in all benchmarks the fraction of remaining total cache misses is consistently higher than the fraction of remaining natural misses (46% vs. 33% on average). The effectiveness of prefetching can sometimes be largely over-stated if context switch misses are not included. In addition, the difference between the fractions of the remaining natural misses versus the remaining total misses is the highest for benchmarks on the right side, i.e. the benchmarks for which prefetching is highly effective in eliminating the natural cache misses.

To understand the reasons behind these observations, recall that prefetching brings blocks into the cache early, causing them to wait in the cache until they are used by the processor. However, while waiting, they gradually shift close to LRU positions whenever new blocks are brought into the cache. Consequently, some prefetched blocks are replaced before the processor has a chance to use them. When a processor starts with an "empty" cache (no cache blocks belong to itself) after cache perturbation, it will start demand-

fetching and prefetching blocks. All demand-fetched blocks are always useful but some of the prefetched blocks may be replaced before their first use, and have to be refetched later on. Such refetches waste bandwidth and are counted by our model as context switch misses.

For benchmarks that have a large difference between the fractions of the remaining natural misses versus the remaining total misses (milc, sjeng, sphinx3, libquantum, hmmer, and lbm), there is an additional explanation. The reason has to do with the *probability of prefetched blocks to be used* (denoted as PPU) versus the *probability of demand-fetched blocks to be reused* (denoted as PDFR). For applications in which prefetching is highly effective, their PPUs are necessarily high. For them, a large number of useful prefetched blocks are lost during cache perturbation, requiring them to be refetched, manifesting in future context switch misses. In contrast, applications in which prefetching is not effective have low PPUs, so perturbed prefetched blocks do not result in future context switch misses due to refetching. Thus, *perversely*, the more effective the prefetching is, the more context switch misses are incurred. We collect both PPUs and PDFRs and found that for the six applications for which prefetching is highly effective, PPU is indeed significantly larger than PDFR (79% vs. 56% on average). For the other seven applications, the opposite is true, i.e. PPU is smaller than PDFR (33% vs. 77% on average).

Why does prefetching aggravate context switch misses? It all comes down to the temporal reuse pattern of an application as represented by its stack distance profile. As we have discussed in Section 5.3, a flatter stack distance profile produces higher number of context switch misses. Because prefetching tends to make a stack distance profile flatter (through increasing the reuse distance), it invariably increases the number of context switch misses. To our best knowledge, the insights that prefetching aggravates context switch misses and that the more effective prefetching is the more context switch misses it incurs, have not been reported in literature.

7. CONCLUSIONS

We have characterized the behavior of context switch misses, especially focusing on the behavior of reordered misses. We have shown that reordered misses occur due to the interaction of cache replacement policy with the temporal reuse pattern of an application. The number of reordered misses tends to reach its peak when the cache perturbation affects roughly a half of the cache size, and the fraction of reordered misses of the total context switch misses increases as the cache size increases.

We have also presented an analytical model that reveals the mathematical relationship between cache parameters, the temporal reuse behavior of a thread, and the number of context switch misses the thread suffers from. We found that applications with a flatter stack distance profile are more vulnerable from suffering context switch misses than applications with a concentrated stack distance profile. We also showed that prefetching, because it makes stack distance profile flatter, aggravates context switch misses. In addition, perversely, the more effective the prefetching is for an application, the higher the number of context switch misses the application suffers from.

Many computer architecture studies focus solely on cold, capacity, and conflict misses. However, our findings suggest context switch misses are an important part of the picture and should be included in the studies of cache designs. Our model can serve as a useful tool for analyzing the behavior of context switch misses.

8. REFERENCES

[1] A. Agarwal and J. Hennessy and M. Horowitz. An Analytical Cache Model. *ACM Trans. on Computer Systems*, 7(2):184–215, 1989.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Trans. on Computer Systems*, 6(4):393–431, 1988.

[3] C. Cascaval and David A. Padua. Estimating Cache Misses and Locality Using Stack Distances. In *Proc. of 17th Intl. Conf. on Supercomputing*, pages 150–159, 2003.

[4] C. Cascaval, L. DeRose, D. A. Padua, and D. Reed. Compile-Time Based Performance Prediction. In *Proc. of the 12th Intl. Workshop on Languages and Compilers for Parallel Computing*, pages 365–379, 1999.

[5] F. M. David, J. Carlyle, and R. H. Campbell. Context Switch Overheads for Linux on ARM Platforms. In *Proc. of the 2007 Workshop on Experimental Computer Science*, 2007.

[6] R. Fromm and N. Treuhaft. Revisiting the Cache Interference Costs of Context Switches. *http://citeseer.ist.psu.edu/252861.html*, 1996.

[7] G. Edward Suh and S. Devadas and L. Rudolph. Analytical Cache Models with Applications to Cache Partitioning. In *Proc. of Intl. Conf. on Supercomputing*, pages 1–12, 2001.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, (1Q), 2001.

[9] F. Guo and Y. Solihin. An Analytical Model for Cache Replacement Policy Performance. In *Proc. of the ACM SIGMETRICS/Performance 2006 Joint Intl. Conf. on Measurement and Modeling of Computer System*, pages 228–239, 2006.

[10] IBM. *IBM Power4 System Architecture White Paper*, 2002.

[11] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, pages 364–373, 1990.

[12] P. Koka and M. H. Lipasti. Opportunities for Cache Friendly Process Scheduling. In *Workshop on Interaction Between Operating Systems and Computer Architecture*, 2005.

[13] H. Kwak, B. Lee, A. Hurson, S. Yoon, and W. Hahn. Effects of Multithreading on Cache Performance. *IEEE Trans. on Computers*, 48(2):176–184, 1999.

[14] C. Li, C. Ding, and K. Shen. Quantifying The Cost of Context Switch. In *Proc. of the 2007 Workshop on Experimental Computer Science*, 2007.

[15] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer Society*, 35(2):50–58, 2002.

[16] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[17] J. Mogul and A. Borg. The Effect of Context Switches on Cache Performance. In *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, 1991.

[18] Standard Performance Evaluation Corporation. Spec cpu2006 benchmarks. *http://www.spec.org*, 2006.

[19] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proc. of Intl. Symp. on High Performance Computer Architecture*, pages 117–126, 2002.

[20] D. Thiebaut and H. S. Stone. Footprints in the Cache. *ACM Trans. on Computer Systems*, 5(4):305–329, 1987.

[21] D. Tsafrir. The Context-Switching Overhead Inflicted by Handling Hardware Interrupts. In *Proc. of the 2007 Workshop on Experimental Computer Science*, 2007.