Characterization and Dynamic Mitigation of Intra-Application Cache Interference

Carole-Jean Wu and Margaret Martonosi
Depts. of Electrical Engineering and Computer Science
Princeton University
{carolewu, mrm}@princeton.edu

***Abstract*—Given the emerging dominance of CMPs, an important research problem concerns application memory performance in the face of deep memory hierarchies, where one or more caches are shared by several cores. In current systems, many factors can cause interference in the shared last-level cache (LLC). While predicting an application's memory performance is difficult enough in an idealized setup, it becomes even more complicated in real-machine environments in which interference can stem from operating system memory accesses, and even from an application's own prefetch requests and page table walks caused by TLB misses.**

This paper characterizes the degree by which intra-application interference factors such as page table walks and hardware prefetching influence performance. Using hardware performance counters on an Intel platform, we first characterize real-system LLC interference and show that application data memory references represent much less than half of the LLC misses, with hardware prefetching and page table walks causing considerable LLC interference.

Based on these characterizations, we propose dynamic management methods to reduce intra-application interference. First, we evaluate a dynamic OS-reference-aware cache insertion policy that reduces interference and improves user IPCs by as much as 19% (5% on average). Second, to mitigate prefetch-induced LLC interference, we propose, implement, and evaluate an automatic prefetch manager that uses Intel PEBS capabilities to dynamically estimate prefetch-induced interference and accordingly adjust the aggressiveness of hardware prefetchers as programs run. Overall, our characterizations are important in highlighting the challenges of intra-application interference, and our hardware and software proposals offer significant solutions for addressing them.

1 Introduction

It is common today to run multiple applications, such as web server, video-streaming, graphic-intensive, scientific, and data mining workloads, on chip-multiprocessor (CMP) systems, with multiple cores sharing the last level on-chip cache (LLC). This can cause performance degradation and performance unpredictability. Although inter-application interference in shared caches has been well-studied [4, 9, 11, 21, 24], the effects of *intra*-application interference also play a significant role and have received significantly less attention. Some past work has characterized some OS interference effects [1, 23] , but these studies are not very recent so they do not include effects seen in today's deep memory hierarchies accessed simultaneously by multiple cores. Furthermore, there is also a need to characterize intra-application interference in the face of modern, aggressive hardware prefetchers.

Our work provides a detailed real-system characterization of how user references in today's workloads are interfered with by operating system and prefetching references executing on the application's own behalf. For example, Figure 1 plots a breakdown of LLC miss types, collected using detailed hardware performance monitoring counters on a running Intel Nehalem (x86) system. (Section 2 gives methodology details.) Among all LLC misses, surprisingly few come from application memory data references. This is because of interference from operating system memory accesses, page table walks caused by Translation Lookaside Buffer (TLB) misses, speculative memory accesses, and even from an application's own prefetch requests.

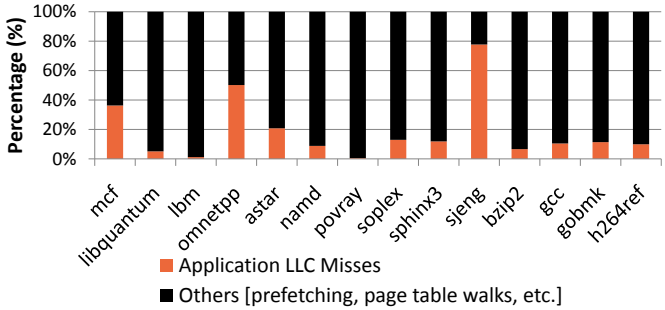

Fig. 1. LLC miss breakdown: User vs. System. The lower (orange) portion each bar represents misses caused by application memory instructions and the upper part (black) represents misses caused by other sources, e.g. prefetching, page table walks due to TLB lookups, etc. [Data collected on Intel Core i7].

Having established that intra-application interference from operating system references and prefetching requests is a severe problem in today's systems, this paper also proposes and evaluates dynamic management techniques to address these concerns. In particular, our first proposal uses a different cache insertion policy to reduce user-system interference within a single application. This method exploits the different reuse trends between user and system behavior to implement different cache insertion policies for each class of references. While simple, it offers user performance improvements by as much as 19% and by an average of 5%. This corresponds to 10% aggregate performance improvement and 3% on average.

In addition, we explore techniques for reducing prefetch-induced cache interference by dynamically modulating how aggressively prefetchers operate. Our dynamic prefetch manager, built upon Intel's Precise Event Based Sampling (PEBS) capability, periodically assesses the degree of LLC interference caused by prefetch requests. Based on its estimation,

TABLE I
ARCHITECTURAL PARAMETERS OF THE REAL SYSTEM (INTEL NEHALEM).

Operating System	Ubuntu 9.10 Desktop
Target Platform	Intel Core i7 CPU 950
L1 I-TLB	128-entry, 4-way
L1 D-TLB	64-entry (small pg.), 32-entry (large pg.), 4-way
L2 TLBs	512-entry (small-pages only), 4-way, Private
L1 I Cache	32KB, 4-way, Private, 3 cycles
L1 D Cache	32KB, 8-way, Private, 4 cycles
L2 Cache	256KB, 8-way, Private, 11 cycles
L3 Cache (LLC)	1MB per-logical-core, 16-way, Shared, 30-40 cycles

TABLE II
ARCHITECTURAL PARAMETERS OF THE BASELINE SIMULATED PLATFORM (SIMICS/GEMS).

Operating System	Sun Solaris 10
Target Platform	UltraSPARC III Cu
L1 I-TLB	16-entry, fully assoc. (locked/unlocked pages); 128-entry, 2-way (unlocked pages)
L1 D-TLB	16-entry, fully assoc. (locked/unlocked pages); 512-entry or 1024-entry 2-way assoc. (unlocked pages)
L1 I/D Caches	64KB, 4-way, Private, 3 cycles
L2 Cache (LLC)	1MB per-core, 16-way, Shared, 35 cycles

the dynamic manager tunes the aggressiveness of hardware prefetching by setting the corresponding machine state registers (MSRs). Our approach eliminates prefetch requests predicted to be low-payoff.

With this dynamic prefetch control, intra-application cache interference caused by aggressive hardware prefetching is reduced by 25% compared to the system default (when all prefetchers are on). Most importantly, the proposed dynamic prefetch control improves the application performance the most among the available prefetch options because of its ability to adapt to program phase changes at runtime. As a result, the application (i.e. `lbm`) miss counts are further reduced by 20% compared to the best of all prefetch options explored.

Overall, this paper's mix of real-system characterizations with detailed evaluations of hardware and system proposals can help guide future work in operating systems and architecture regarding the importance and challenges of intra-application cache interference.

The remainder of this paper is structured as follows. In Section 2, we describe the real-system and simulation infrastructure for the target platform and experimental setup. Then in Section 3, we present our characterizations of both system and prefetch-induced intra-application LLC interference. This is followed by Section 4 where we discuss the proposed cache insertion policies and their effect on performance. Section 5 presents the design of our dynamic prefetch management technique and evaluates it on a real running system. This is followed by Section 6 where we discuss related work. Finally, Section 7 offers conclusions.

2 Methodology

This section introduces our experimental methodology for both real-system measurements and full-system simulation.

2.1 Real-System Measurement Infrastructure

We use an Intel Nehalem-based Core i7 CPU 950 (Bloomfield) platform for the real-system characterization. This platform includes 4 physical processor cores. Each core has private L1 and L2 caches, and all cores share the L3 cache. Each of the cores also has a private L1 I-TLB, a private L1 D-TLB, and a private L2 unified TLB. The detailed architectural parameters of the target platform are described in Table I.

In real systems, hardware performance monitoring counters (PMCs) allow us to measure misses and prefetches accurately. (To dig deeper into the causes of observed interference, we use simulation as later described.) We use the `perfmon2` [16] performance monitoring interface to access hardware PMCs on the target platform. We count the number of LLC misses which are application data read memory references with the hardware event `MEM_LD_RETIRED:LLC_MISS`. We count the aggregate number of LLC misses with `UNC_LLC_MISS`. Then we use `TLB_MISSES:WALK_COMPLETED` for the number of page table walks caused by TLB misses, `L1D_PREFETCH:MISS` for the number of L1D prefetch misses, and `L2_RQSTS:PREFETCH_MISS` for the number of L2 prefetch misses. More detailed information about the performance monitoring events can be found in the Intel Software Developer's Manual [7].

2.2 Full-System Simulation

In order to understand the sources of LLC misses in more detail, we use a full-system processor simulator based on Simics/GEMS [12, 19]. The simulated processor models Sun's UltraSPARC III Cu processor family (Table II) with two memory management unit (MMU) configurations. One configuration has a total of 512 translations, organized as 2 256-entry L1 D-TLBs. This configuration represents today's high-performance desktops, with a TLB size/organization comparable to the Nehalem platform described in Section 2.1. We also evaluate a larger TLB organization that has a total of 1024 entries, similar to the Sun Fire 3800 (one of the largest TLB size/organizations to date). Both MMU architectures have separate 16-entry fully-set associative L1 I and D-TLBs used by the operating system for locked pages.

Because page table walks caused by TLB miss handling are important to our characterization, we instrument the Simics source code to track instruction and memory references incurred for each TLB miss handling. Our method for capturing this information is described in Section 3.2.

2.3 Benchmarks and Input Sets

We perform our evaluation for intra-application LLC interference using sequential applications in the SPEC CPU2006 benchmark suite [20]. All applications are run with `ref` input data sets. For real-system measurements, applications are run until completion. Since full-system simulations are orders of magnitude slower than real-system ones, we run the same set of SPEC CPU2006 applications but skip the first 100M instructions and then collect the results for windows of 1B instructions.

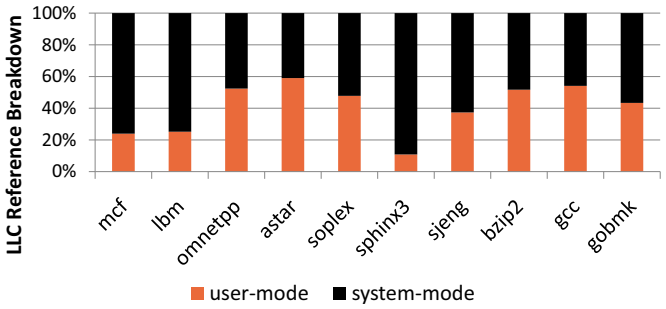

Fig. 2. LLC user- and system-mode reference breakdown. On average, more than 50% of LLC references are system-mode. [Data collected with Simics/GEMS full-system simulation].

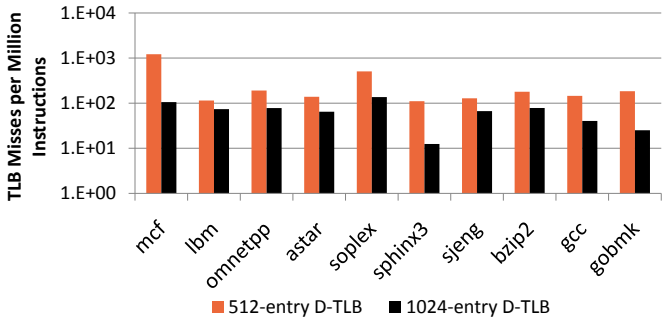

Fig. 3. TLB misses per million instructions on SPARC. [Real-system measurement performed on an Intel-Nehalem-based platform running Linux shows similar behavior].

3 Characterizing References to the Last-Level Cache

A key contribution of our paper is in providing detailed characterizations of CMP memory hierarchy behavior for current workloads and operating systems, with a particular focus on intra-application interference stemming either from OS references or from prefetch requests. We first use real-system measurements to give an overview of the behavior and sources of cache misses. While real-system measurements are useful for seeing the full real-world story, it can be difficult to measure all the necessary details from the limited performance counters available. Our research therefore provides further characterizations based on full-system simulation.

3.1 Overview Characterizations

3.1.1 Real-System Characterization on Intel Nehalem: With hardware PMCs available in most of today's systems, we can estimate the degree of intra-application LLC interference caused by hardware prefetching with real-system measurement. We measure the number of user application LLC read misses (PMC event: `MEM_LD_RETIRED:LLC_MISS`) and compare it with the aggregate number of LLC read misses (`UNC_LLC_MISS:READ`). The difference between `MEM_LD_RETIRED:LLC_MISS` and `UNC_LLC_MISS` represents the number of LLC read misses from speculative accesses, prefetch requests, or operating system memory references.

We then collect the statistics for the number of prefetch requests issued by hardware prefetchers on an Intel Nehalem platform. Based on the aggressiveness of each hardware prefetcher, the number of application LLC misses, and the aggregate LLC miss counts, we estimate the degree of LLC interference from an application's own prefetch requests in real systems.

3.1.2 Characterization on Simics/GEMS: We use full-system simulation to further investigate the sources of LLC interference within an application. Similar to our real-system characterization results, Figure 2 shows that, on average, less than 50% of LLC references are from user applications. All applications experience significant LLC interference from system-mode references.

Section 3.2 characterizes such intra-application LLC interference caused by system references, which include page table walks from TLB misses, application-initiated system calls, and other operating system memory references. Because page table walk references incurred by TLB miss handling constitute a significant portion of system-mode memory references, we delve deeper into TLB miss handling in general and also specific to the SPARC and x86 (Intel Nehalem) architectures which we use as the full-system simulation and real-system measurement platforms in this study.

3.2 Digging Deeper: Characterizing System References

Since every memory reference requires virtual-to-physical address translation, TLBs cache frequently accessed page table entries to accelerate these translations. When a hit occurs in the TLB, the virtual to physical address translation can be used directly. For a TLB miss, page table walks resolve the address translation. Processors vary regarding whether the page table walk is controlled in hardware or software, but in either case references causing LLC interference can occur. This section first describes the software approach used for SPARC architectures.

3.2.1 TLB Miss Handling for SPARC Architectures: When a TLB miss occurs, the operating system is invoked. It first checks the translation storage buffer (TSB), a software cache that stores a few virtual to physical address pairs. If there is a hit in the TSB, the TLB miss is serviced by the TSB entry. If not, the operating system walks the page table to resolve the address translation. During the software page table walks caused by TLB misses, many additional instruction and data references are incurred, which stress the memory hierarchy.

3.2.2 Identifying Page Table Walk Memory References in Simics: A particular challenge lies in discerning which system references correspond to page table walks since all system references (including page table walks, application-initiated system calls, and other operating system memory references) arrive at the LLC as context-0 references. We instrument the TLB miss handler in the Simics MMUs to track instruction and memory references for page table walks. We instrument the MMUs right after the software interrupt is invoked and before the TLB miss is serviced. However, during this period of time, page faults can occur and the operating system could switch its context to handle other running processes. To accurately count references related to TLB miss handling only, we compare the

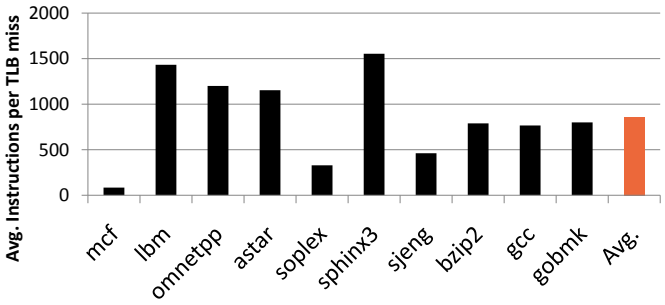

Fig. 4. Number of instruction references incurred per TLB miss handling. On average, more than 800 instruction references are incurred to handle each TLB miss (which might incur page faults). These additional references contribute a significant portion of the LLC system references and can cause intra-application LLC interference.

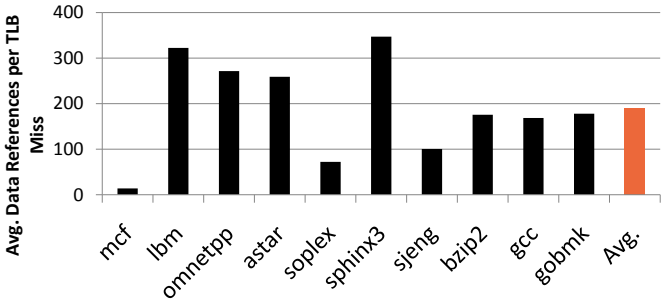

Fig. 5. Number of memory references incurred per TLB miss handling. On average, more than 200 memory references are incurred to handle each TLB miss (which might incur page faults). These additional references contribute a significant portion of the LLC system references and can cause intra-application LLC interference.

context IDs to only include instruction and memory references related to page table walks and page faults.

Figure 3 shows the number of TLB misses per million instructions for SPEC CPU2006 applications. In a 512-entry TLB, more than 100 TLB misses occur for every million application instructions. `mcf`, a large memory footprint application, has the highest TLB miss ratio among all applications. For the larger 1024-entry TLB organization, misses occur less frequently.

Figures 4 and 5 illustrate that, on average, handling each TLB miss incurs about 800 instruction and 200 data memory references. For some applications, such as `mcf`, `soplex`, and `sjeng`, the number of page table walk instruction and data references is relatively less. This is because more of the application's TLB misses hit in the TSB, so a complete page table walk is not required.

Overall, the page table walk instruction and memory references constitute a significant portion of all system-mode references. Figure 6 illustrates that, on average, 80% of system-mode references are caused by TLB miss handling. In addition to page table walks caused by TLB misses, the remaining 20% of the system references in Figure 6 can come from application-initiated system calls, operating system kernel process, etc. Since this represents a less significant portion among all system-mode references, we do not further separate these references into different categories.

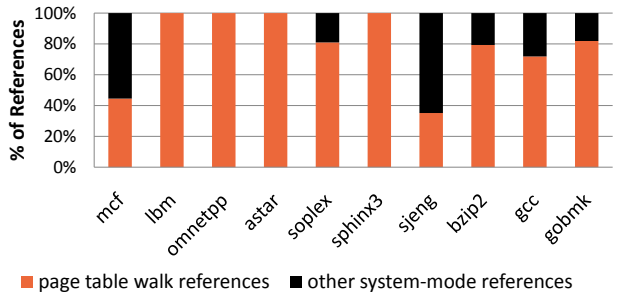

Fig. 6. System-mode reference breakdown. On average, 80% of system-mode references are page table walks caused by TLB misses.

3.2.3 TLB Miss Handling for x86 Architecture: Since our real-system characterization result is based on x86 architectures, we briefly discuss TLB miss handling in x86, which involves hardware page table walks, but nonetheless, page table walk memory references contribute to intra-application LLC interference. There are typically four levels of page table in 64-bit systems. When a TLB miss occurs, the CPU walks the four-level page table, so each TLB miss resolution requires roughly four memory references and potentially incurs four cache misses. In addition, the operating system has to ensure consistency between TLBs and page tables. As a result, when TLB shootdown or page faults occur, the operating system is responsible for invalidating corresponding TLB entries and updating the CR3 register. This case leads to many additional cache misses of a magnitude similar to SPARC page fault handling.

3.3 Memory Reuse Characteristics Analysis for User- and System-Mode References

In order to devise effective cache management policies to mitigate intra-application LLC interference between user and system memory references, this section studies the distinct reuse characteristics for these user and system references. When considered on their own, user application memory references often exhibit good reuse characteristics with excellent data temporal locality. Data brought into the LLC are reused several times before being evicted. However, this good reuse behavior is degraded when user and system references share the LLC. When user and system references are viewed together, many of user cache lines become never reused (zero-reused) before getting evicted from the LLC.

To illustrate this, Figure 7 examines the reuse frequencies of the user references when they use the LLC exclusively (User Only) and when they share the LLC with system references (Baseline). The graph shows that when user and system references share the LLC in the baseline case, 76% of user cache lines are never reused before being evicted from the LLC. However, when user references have exclusive access to the LLC, the number of zero-reused user cache lines drops to 36%. This is because many user cache lines that could be reused are instead evicted early due to intra-application LLC interference from system references.

Even worse, a significant portion of system references interfering with user references themselves exhibit bad reuse

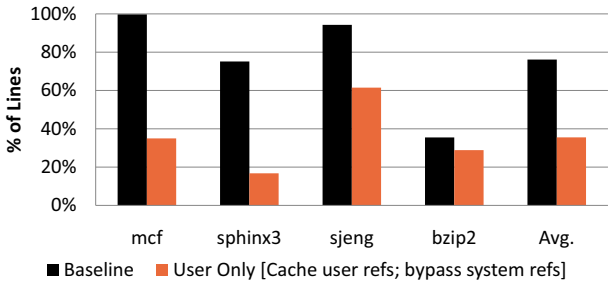

Fig. 7. Memory reuse characteristics for user references: Percentage of lines that are zero-reused. The good reuse patterns of user references are destroyed by LLC interference from system references.

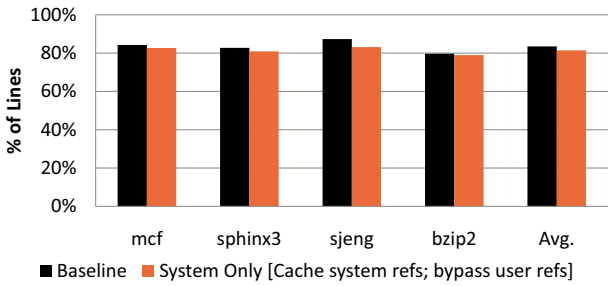

Fig. 8. Memory reuse characteristics for system references: Percentage of lines that are zero-reused.

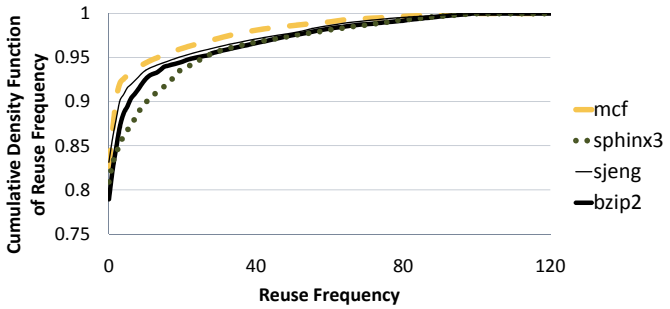

Fig. 9. Reuse characteristics of system references. While 82% of system references are zero-reused, the remaining 18% system references exhibit good reuse behavior. 5% of system cache lines are reused more than 20 times.

characteristics. Figure 8 shows that, when system references share the LLC with user references, 84% of system cache lines are zero-reused. However, even when system references have exclusive access to the LLC (System Only), 82% of system cache lines are still never reused. This means that the LLC which would be more useful to user-mode references is instead being devoted to system-mode references, some of which have little likelihood of ever being reused.

Given that many system references are zero-reused, it is tempting to simply bypass all system references and completely eliminate LLC interference between user and system references. However, examining the system references shows that the remaining system references are reused frequently. Figure 9 shows the reuse frequencies of system cache lines. The x-axis represents reuse frequencies and the y-axis represents the cumulative density function of reuse frequencies. While 82% of system references have zero reuse frequency, 5% of system references are reused heavily and bypassing them will degrade performance significantly.

3.4 Characterizing Hardware Prefetching

In addition to system references, prefetch requests represent a significant portion of LLC references. While often helpful, the benefits of aggressive prefetching hinge on its accuracy. In the past few decades,there have been numerous prior works [5, 6, 10, 13, 15, 22, 26] which focus on improving prefetching mechanisms by increasing prefetcher accuracy and/or reducing cache pollution. However, there has not been any research work in characterizing real-system prefetch-induced LLC interference within applications.

In order to understand intra-application LLC interference caused by hardware prefetching, we first give background on Intel Nehalem-specific hardware prefetching. Then we present our real-system characterization results for intra-application LLC interference caused by the mid-level cache prefetchers.

3.5 Hardware Prefetchers on Intel Nehalem

There are four distinct hardware prefetchers on Intel Nehalem platforms: data cache unit (DCU) IP-prefetcher, DCU streamer-prefetcher, mid-level cache (MLC) spatial-prefetcher, and MLC streamer-prefetcher.

The per-core DCU IP-prefetchers looks for sequential load history to determine whether to prefetch the data to the L1 caches; the DCU streamer prefetchers detect multiple reads to a single cache line in a certain period of time and choose to load the following cache lines to the L1 data caches. We leave these two DCU prefetchers always on because Intel does not allow any external control of them.

On the other hand, we can control the MLC spatial-prefetchers and streamer-prefetchers by setting the corresponding machine state register (MSR) bits. The MLC spatial-prefetchers detect requests on two successive cache lines and are triggered on if the adjacent cache lines are accessed. The MLC streamer-prefetchers work similarly to the DCU streamer-prefetchers, which predict the immediate future access patterns based on the current cache line readings.

3.6 Intra-Application Interference Caused by Hardware Prefetching

We first show the aggressiveness of the MLC prefetchers for all SPEC CPU2006 applications. This gives an estimate for the amount of intra-application LLC interference caused by Nehalem's hardware prefetchers.

Figure 10 illustrates the degree to which the LLC's workload stems from L2 prefetch misses. As many as 85% of the LLC requests are due to the MLC prefetchers. Since many of these will bring useful data from the LLC to the L2 cache, they do not always lead to harmful interference. Nonetheless, it is clear that the LLC's workload is heavily influenced by MLC prefetcher behavior, and useless prefetch requests can cause significant LLC interference.

Next, we demonstrate the impact of turning on and off MLC prefetchers. If turning off prefetchers helps reduce application

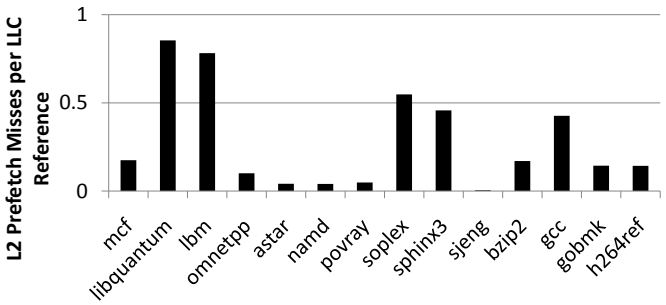

Fig. 10. Behavior of Nehalem's Mid-Level Cache (MLC) prefetchers. The bars represent the fraction of LLC references that are due to MLC prefetches (as opposed to demand requests due to misses in L1 or L2 caches).

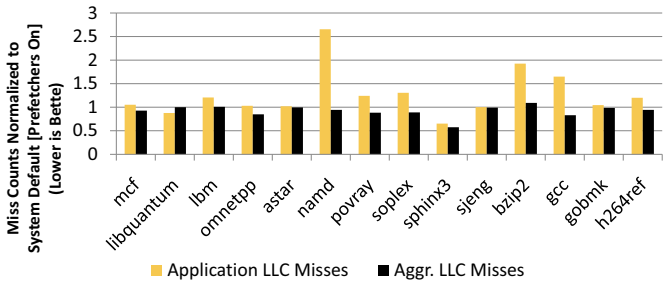

Fig. 11. Intra-core LLC interference caused by MLC prefetch requests. This graph compares miss counts (application-only or all) with prefetchers turned off, normalized to those with prefetchers turned on.

LLC miss counts, intra-application LLC interference caused by prefetch requests is considered significant. Otherwise, prefetching can effectively fetch useful data in advance.

Figure 11 shows the number of application LLC misses and the number of aggregate LLC misses (including both application and prefetch requests missed in the LLC) with MLC prefetchers turned off. The left bars represent the number of application LLC misses normalized to the baseline, for which MLC prefetchers are on. The right bars show aggregate LLC miss counts, also normalized to the case of prefetchers turned on. For SPEC CPU2006 applications, such as `lbm`, `namd`, `povray`, `soplex`, `bzip2`, `gcc`, and `h264ref`, turning off prefetchers significantly increases application LLC miss counts. This means that the MLC prefetchers can effectively fetch useful data in advance.

For applications, such as `mcf` and `gobmk`, turning off MLC prefetchers only slightly increases the number of application LLC misses. This means the MLC prefetchers are less effective but still help improve application LLC misses. For applications, such as `omnetpp`, `astar`, and `sjeng`, the MLC prefetchers are not very effective and, therefore, can be turned off with no performance impact.

Lastly for applications such as `libquantum` and `sphinx3`, we recommend turning off MLC prefetchers. As illustrated in Figure 11, the number of application LLC misses is reduced slightly for `libquantum`. More significantly, for `sphinx3`, turning off the MLC prefetchers reduces both the number of application LLC misses (left bar) and the aggregate number of LLC misses (right bar) significantly. This means the accuracy of the MLC prefetchers is very low for `sphinx3` which leads to more severe intra-core LLC interference.

4 Reducing OS-induced Cache Interference

4.1 OS-Aware Dynamic Cache Insertion

Our goal is to eliminate harmful intra-application LLC interference caused by zero-reused system cache lines (82% among all system references) while retaining cache lines with high reuse frequencies as discussed in Section 3.3. Thus, in this section, we present our design for a dynamic cache management scheme that exploits the distinct memory reuse characteristics of user- and system-mode references. Our dynamic management scheme is based on a previously proposed technique, called dynamic insertion policy (DIP) [17].

Our work explores a family of policies for varying cache insertion positions based on whether the reference comes from user or system modes. In LLCs, cache set associativities are often as high as 16 or 32. This means that there are many options for where to insert new data into the cache, rather than the baseline scheme which always inserts cache lines to the most-recently used (MRU) position. Prior work has proposed inserting cache lines in either MRU or LRU positions to avoid memory-intensive applications thrashing caches for the entire workload. Our work is distinct in exploiting such approaches while distinguishing user- and system-mode references. We investigate three cache insertion policies varying system reference insertion positions: **SYS-LRU**, **SYS-MID**, and **SYS-DYN**. While other options are possible, our policies focus on inserting user references into the MRU position and varying insertion positions for system references.

- **SYS-LRU**: This policy inserts user references in the MRU position as in a typical cache, but it inserts system references into the LRU position instead. System references are detected as context ID = 0.
- **SYS-MID**: As in **SYS-LRU**, user references are inserted in the MRU position. System references are, however, inserted in the middle of the MRU and LRU positions. For our 16-way set-associative LLC, this is position 8.
- **SYS-DYN**: This policy uses Set Dueling [9] described in Section 4.1.1 to periodically compare user and system miss counts for the two insertion policies, **SYS-LRU** and **SYS-MID**. It then dynamically chooses the insertion policy which results in fewer cache misses.

SYS-DYN is designed to adapt to changing user and system reuse characteristics at runtime. In our setup, **SYS-DYN** compares user and system miss counts for each insertion policy at every 1000 LLC misses. Furthermore, in order to obtain more good reuse system references, **SYS-DYN** periodically (every 64 system LLC misses) inserts system references in the MRU position to allow system references to utilize the LLC. This technique is similar to the bimodal insertion policy presented in [17].

The **SYS-DYN** algorithm is described in Algorithm 1. In the algorithm, we use a coefficient, **c**, which can be used to control the ratio of user and system miss counts. In our experiments, we

set **c** = 1. This means **SYS-DYN** starts using **SYS-LRU** until it detects that the system LLC miss count increase has surpassed the user LLC miss count reduction. Then **SYS-DYN** switches to use **SYS-MID**. For **c** greater than 1, system miss counts are weighted more, so **SYS-DYN** uses **SYS-MID** more frequently, and vice versa for **c** smaller than 1.

Algorithm 1 Algorithm for **SYS-DYN**.

$\Delta user\_miss\_counts$ = user miss counts of **SYS-LRU** - user miss counts of **SYS-MID**

$\Delta system\_miss\_counts$ = system miss counts of **SYS-LRU** - system miss counts of **SYS-MID**

while $(user\_miss\_counts + system\_miss\_counts)\ \%\ 1000 == 0$ **do**
  if $\Delta user\_miss\_counts + c * \Delta system\_miss\_counts > 0$ **then**
    SYS-DYN selects **SYS-MID**
  else
    SYS-DYN selects **SYS-LRU**
  end if
end while

4.1.1 Hardware Implementation: This section presents the implementation of **SYS-DYN** with modest hardware requirements. We use the set dueling mechanism to dynamically collect user and system LLC miss counts for **SYS-LRU** and **SYS-MID** policies. In the baseline system, we dedicate 32 out of the 1024 cache sets to each of the two insertion policies: **SYS-LRU** and **SYS-MID**. This means set 1, set 33, ..., and every 32nd set in the 1024-set LLC are dedicated to use **SYS-LRU**. Similarly, set 2, set 34, ..., and every 32nd set are dedicated to use **SYS-MID**.

We require four additional cache miss counters to count user and system LLC misses in each policy. As described in Algorithm 1, it then uses the policy resulting in fewer LLC misses for the remaining 960 cache sets.

Figure 12 illustrates the hardware implementation of **SYS-DYN**. The lower 5 bits of data address index field determine the insertion position of the incoming system cache lines. Periodically, **SYS-DYN** compares the miss counters and updates the insertion policy for the remaining cache sets.

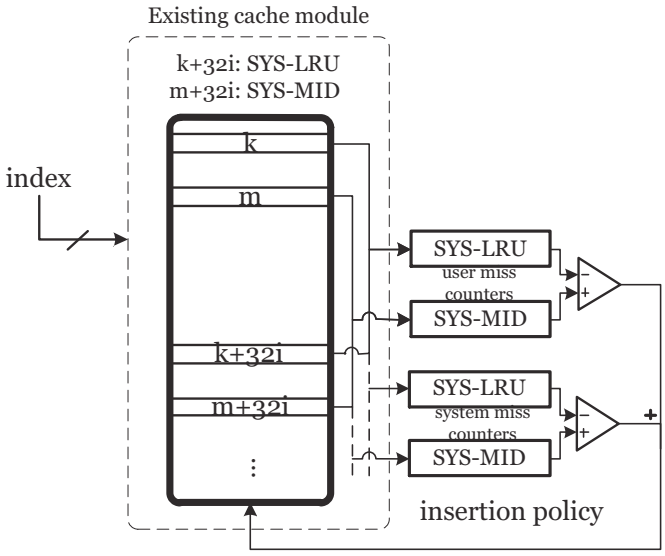

Fig. 12. Hardware implementation for **SYS-DYN**. In our setup, k equals 1 and m equals 2.

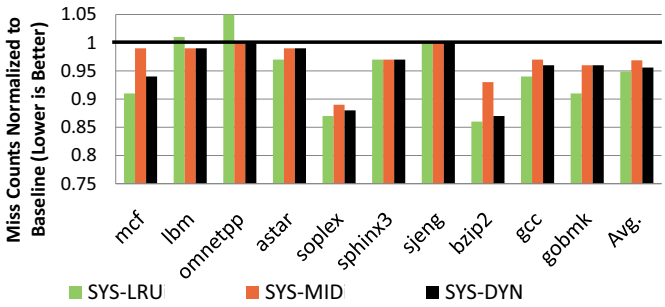

Fig. 13. User LLC miss count reduction for **SYS-LRU**, **SYS-MID**, and **SYS-DYN**. The user miss counts are normalized to a baseline policy of normal cache operation, where all references are inserted in the MRU position.

4.2 Performance Results

Figures 13–15 show the results for **SYS-LRU**, **SYS-MID**, and **SYS-DYN**. In particular, Figure 13 shows user LLC miss reduction for the three policies. When considering user LLC miss counts only, **SYS-LRU** (the leftmost bars in Figure 13) reduces user LLC miss counts more than the other two policies. User miss counts are decreased by as much as 14% for `bzip2` and by an average of 6%. This is because **SYS-LRU** aggressively inserts system references in the LRU position. As a result, it can eliminate LLC interference caused by system references most effectively.

The LLC miss reduction with **SYS-LRU** corresponds to as much as 22% user IPC improvement, with an average of 5%. Furthermore, Figure 14 shows that the aggregate IPC (including both user and system memory references) is improved by as much as 12% for `soplex`. While **SYS-LRU** is effective at reducing user LLC miss counts, its impact on aggregate behavior is more varied. For example, for `lbm`, `omnetpp`, `sjeng`, and `gobmk`, **SYS-LRU** degrades aggregate IPCs. This is because although **SYS-LRU** improves user IPCs, system performance degradation is more than user performance improvement. As a result, the overall IPCs are degraded. **SYS-MID** alleviates this problem.

Figure 13 shows that, although **SYS-MID** does not reduce LLC miss counts as much as **SYS-LRU**, it consistently improves user LLC miss counts for *all* applications. In addition, the middle bars in Figure 15 shows user IPCs are improved by 4% on average. We also see that aggregate IPCs are improved by as much as 10% for `soplex` and by an average of 3.5%. This is because inserting incoming system references in the middle of the LRU stack using **SYS-MID** helps retain more of the 18% system references which exhibit good reuse behavior while still reducing LLC interference caused by low reused system references.

Because each of **SYS-LRU** and **SYS-MID** has its own advantage in reducing LLC interference between user and system references, we examine a hybrid policy, **SYS-DYN**. Indeed, **SYS-DYN** is the most effective one among the three insertion policies. **SYS-DYN** reduces user LLC miss counts and helps improve both user and aggregate IPC performance. The rightmost bars in Figure 13 show that **SYS-DYN**, although it does not

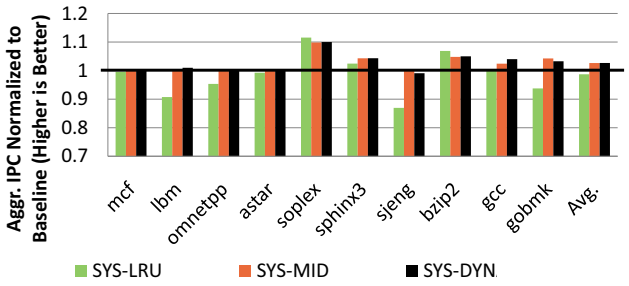

Fig. 14. IPC performance improvement for **SYS-LRU**, **SYS-MID**, and **SYS-DYN**. The aggregate IPCs are normalized to a baseline policy, where all references are inserted in the MRU position.

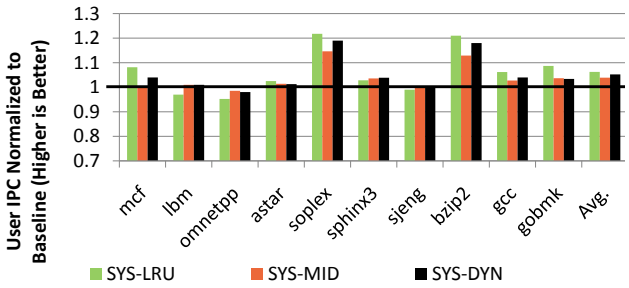

Fig. 15. IPC performance improvement for user references for **SYS-LRU**, **SYS-MID**, and **SYS-DYN**. The user IPCs are normalized to a baseline policy, where all references are inserted in the MRU position.

reduce user LLC miss counts as much as **SYS-LRU**, can help reduce more user miss counts than **SYS-MID**. For all SPEC CPU2006 applications, **SYS-DYN** reduces user LLC misses by as much as 13%. Furthermore, it improves user IPCs by as much as 19% and by an average of 5%.

More importantly, **SYS-DYN** can consistently improve the aggregate IPC performance for all applications. For some applications, such as `lbm` and `gcc`, **SYS-DYN** works the best among the three insertion policies. This is because **SYS-DYN** takes advantage of **SYS-LRU**, which can reduce user LLC miss counts more aggressively. But when it detects the increase in system miss counts, **SYS-DYN** dynamically switches to **SYS-MID** instead. As a result, the number of system LLC misses is improved. Overall, Figure 14 illustrates that **SYS-DYN** improves aggregate performance by as much as 10% and 3% on average for all applications.

5 Mitigating Prefetch-Induced Intra-Application Cache Interference

In addition to system-mode references, hardware prefetching contributes to intra-application LLC interference as well. Our real-system characterization in Section 3.6 hints that Nehalem's MLC prefetchers may issue prefetch requests too aggressively for some applications. This results in serious LLC interference with user memory references within the same application. To eliminate such prefetch-induced cache interference caused by hardware prefetching, Section 3.6 shows that turning off the MLC prefetchers statically for applications, such as `libquantum` and `sphinx3`, helps improve application LLC miss counts.

5.1 Dynamic Prefetch Management

To further demonstrate an automatic hardware prefetching control, we propose, implement, and evaluate a dynamic management technique to adjust Nehalem's hardware prefetchers based on the degree of prefetch-induced LLC interference observed at runtime.

For example, when an application experiences aggressive intra-application interference caused by prefetch requests, our management scheme turns off the prefetchers to eliminate such prefetch-induced interference dynamically. When the comparative samples indicate that prefetching should be turned on, the dynamic manager turns back on the prefetchers.

5.1.1 Implementation: Other prior methods have been proposed to reduce prefetch-induced cache interference, i.e. [13, 15, 26]. However, most of these designs require hardware modification. In contrast, this work implements an effective, dynamic prefetch manager *in a real system* solely by using counter-sampling techniques and available software-accessible registers that control the hardware prefetchers.

In particular, we take advantage of Nehalem's Precise Event Based Sampling (PEBS) capability to collect samples for application LLC miss counts periodically. For example, if the sampling period is set to `N`, an interrupt will be generated at every `Nth` application LLC miss, to dump records for the particular sample. We instrument the `perfmon2` interface to take a time stamp, when the sample is collected, by reading the time stamp counter with the `RDTSC` instruction. Furthermore, based on the decision of our dynamic prefetch control algorithm, we set the appropriate software accessible machine state registers (MSRs) to adjust the hardware prefetchers.

Next, we describe our simple but effective dynamic prefetch control algorithm. There are two major phases in our dynamic prefetch manager: a dynamic profiling phase and a program run phase. In the dynamic profiling phase, we turn on the MLC prefetchers and measure the time taken to complete **s** samples. We then repeat the measurement with the MLC prefetchers turned off. If it takes more time to complete **s** samples when prefetchers are turned off, this means application LLC misses occur less frequently in time. So we leave all prefetchers turned off for the program run phase. Otherwise, we turn back on the prefetchers. The pseudo-code for our dynamic prefetch controller is described in Algorithm 2. The sampling period used in the algorithm is 10,000. This means, at every 10,000 application LLC misses, a sample is recorded. We set **s** to 50 and **period** to be 10,000. This means the duration of the dynamic profiling phase is 100 samples and the duration of the program run phase is 9,900 samples. Our empirical data show that using 100 samples (1% of the program period) for the dynamic profiling phase can sufficiently determine the correct prefetch control option.

Since our dynamic prefetch control is built upon `perfmon2`, it inherits the performance monitoring interface overhead. To quantify this overhead, we measure the time spent in PEBS collection. For the sampling frequency used in our experiments, one sample collected for every 10,000 application LLC misses, the overhead is negligible. For all experiment runs, it is consistently less than 0.5% making our dynamic prefetch control

Algorithm 2 Algorithm for dynamic prefetch control. sample_N is the sample number.

```
if sample_N % period == 0 then
    // dynamic profiling phase: prefetchers on
    prev_time = curr_time;
    Turn on all hardware prefetchers;
else if sample_N % period == s then
    // dynamic profiling phase: prefetchers off
    Time_on = curr_time - prev_time;
    prev_time = curr_time;
    Turn off all hardware prefetchers;
else if sample_N % period == 2*s then
    // program run phase
    Time_off = curr_time - prev_time;
    if Time_on > Time_off then
        Turn on all hardware prefetchers;
    else
        Turn off all hardware prefetchers;
    end if
end if
```

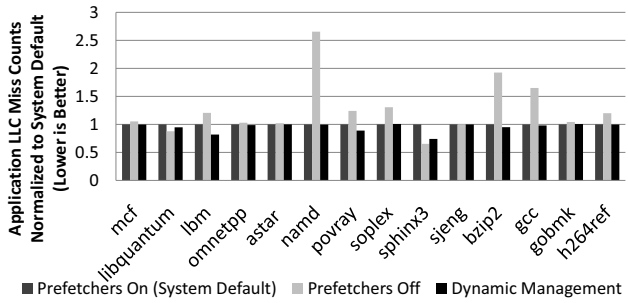

Fig. 16. Application LLC miss reduction for the three prefetch control options: Prefetchers On (System Default), Prefetchers Off, and Dynamic Management.

a lightweight and practical implementation in real systems.

5.2 Performance Results

The proposed dynamic prefetch controller helps reduce application miss rates by as much as 25%. Figure 16 illustrates the application LLC miss counts normalized to the system default, in which all prefetchers are turned on. The leftmost bars represent the system default option (always 1), the middle bars represent the prefetchers-off option, and the rightmost bars represent the dynamic prefetch control.

For most applications (`mcf`, `omnetpp`, `astar`, `namd`, `soplex`, `sjeng`, `bzip2`, `gcc`, `gobmk`, and `h264ref`) the dynamic prefetch manager can correctly choose the prefetch option that results in fewer application LLC miss counts. For `libquantum` and `sphinx3`, which experience aggressive prefetch-induced cache interference, the dynamic prefetch manager can reduce interference and, in turn, reduce application miss rates by 5% and 25% respectively.

Interestingly, the dynamic prefetch technique improves the application memory performance the most for `lbm` and `povray`. `lbm`'s application LLC miss reduction in the dynamic prefetch controller surpasses the miss reduction in both the System Default and Prefetchers Off options. This is because the dynamic prefetch manager observes changes in the program phase and adjusts the aggressiveness of hardware prefetchers accordingly. Figure 17 illustrates the prefetch control decisions over `lbm`'s program execution. By adjusting the aggressiveness of Nehalem's prefetchers based on the dynamic sampling phase, our

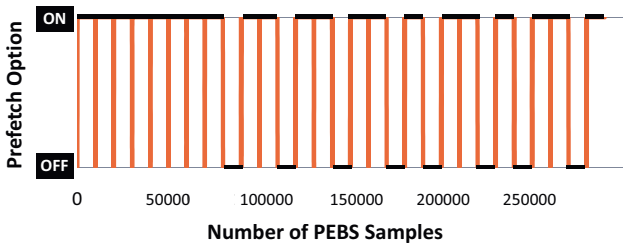

Fig. 17. Prefetch options chosen by the automatic prefetch manager.

method can determine the best prefetch option as programs run.

5.3 Summary

To summarize, we provide the first real-system characterization for intra-application interference caused by hardware prefetching and summarize the effects for a range of different applications. Using existing PMCs on today's platforms, we can characterize the degree to which prefetch requests interfere with user memory references. We find that hardware prefetching can harm performance for applications, such as `libquantum` and `sphinx3`, because of prefetch-induced intra-application LLC interference. In this case, turning off hardware prefetching improves application memory performance.

To mitigate prefetch-induced cache interference automatically, we implement a runtime prefetch manager that dynamically estimates the degree of LLC interference caused by prefetch requests and adjusts the aggressiveness of hardware prefetchers during program execution. We demonstrate that the proposed dynamic prefetch control can further improve application performance due to its ability to adapt to program phase changes.

6 Related Work

6.1 Shared LLC Capacity Management

LLCs are among the most critical shared resources in CMP systems. To achieve performance isolation, quality of service, and better system throughput for the LLC, numerous research efforts have attacked cache capacity management [4, 9, 11, 14, 17, 18, 21, 24, 25].

These efforts, however, do not distinguish user references from system references and likewise do not distinguish prefetch requests from other memory references. Our work demonstrates that intra-application LLC interference from page table and prefetch activity is significant and should not be neglected in the LLC capacity management. To our knowledge, we are the first to identify and analyze intra-application LLC interference caused by system memory references and prefetch requests for both real systems and full-system simulation platforms. Furthermore, we show such intra-application interference can be reduced significantly via dynamic management.

6.2 OS/User LLC Reference Characterization

While indeed OS caching effects have been studied in the past [1, 23], our paper characterizes the influence of operating systems in the context of modern CMPs and modern workloads, with state-of-the-art hardware prefetchers. Furthermore, while these prior studies recognized OS interference, they did not propose interference mitigation techniques. In contrast, our work

details the factors causing intra-application cache interference and proposes a dynamic cache management solution to the problem.

6.3 TLB Miss Handling

Jacob and Mudge [8] compared several memory management units and showed that page table walk references can conflict with user program and data in the shared memory hierarchies. This observation is consistent with what we have presented in this paper and stresses the importance of our work. Bhargava et al. [3] proposed two-dimensional (2D) page table walks to accelerate address translations in the virtualized environment. Although the 2D page table walks accelerate TLB miss handling, they can introduce more system references and stress the shared memory hierarchies even more. To reduce the overhead caused by page table walks, Barr et al. [2] proposed novel page table structures that can improve the number of memory references caused by page table walks. Both approaches [2, 3] focus on improving the page table organizations to accelerate TLB miss handling. This paper, however, offers an orthogonal solution. Our proposed dynamic cache insertion policies can be incorporated into [2, 3] to further eliminate LLC interference caused by low (or even zero) reused page table walk references.

7 Conclusion

Overall, our work has quantified the significant degree of intra-application interference in current workloads and systems. Our measurement methodology used real-system measurements based on Intel Nehalem hardware performance counters where possible, and followed up with Simics/GEMS full-system simulation to get more details beyond what performance counters offer. For each of the major sources of LLC interference, we offer strategies for reducing them dynamically. Our proposed insertion policy improves user IPCs by as much as 19%. Our proposed prefetch manager eliminates as many as 25% of LLC misses compared to the system default. These techniques can form an important part of an arsenal to improve user application performance and make it more predictable in the face of system complexities that make LLC interference challenging to predict and manage.

8 Acknowledgments

We thank Stephane Eranian and Jason Mars for their support on prefetching control and `perfmon2` usage in Nehalem. We also thank Yu-Yuan Chen, Aamer Jaleel, Joel Emer, and the anonymous reviewers for their useful feedback and insights related to this work. This material is based upon work supported by the National Science Foundation under Grant No. CNS-0627650 and CNS-07205661. The authors also acknowledge the support of the Gigascale Systems Research Center, one of six centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 1988.

[2] T. Barr, A. Cox, and S. Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th International Symposium on Computer Architecture*, 2010.

[3] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

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

[5] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th International Symposium on Microarchitecture*, 2006.

[6] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th International Conference on Supercomputing*, 2004.

[7] Intel 64 and IA32 Architecture Software Developer's Manuals. http://www.intel.com/products/processor/manuals/.

[8] B. Jacob and T. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[9] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[10] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

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

[12] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's GEMS simulator toolset. *SIGARCH Computer Architecture News*, 2005.

[13] K. Nesbit, A. Dhodapkar, and J. Smith. AC/DC: an adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, 2004.

[14] K. Nesbit, J. Laudon, and J. Smith. Virtual private caches. In *Proceedings of the 33rd International Symposium on Computer Architecture*, 2007.

[15] J. Peir, S. Lai, S. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proceedings of the 16th international Conference on Supercomputing*, 2002.

[16] Perfmon2: Hardware-based Performance Monitoring Interface for Linux. http://perfmon2.sourceforge.net/.

[17] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, 2007.

[18] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, 2006.

[19] Simics Simulation Platform. http://www.virtutech.com/.

[20] SPEC Benchmark Suite. http://www.spec.org/cpu2006/.

[21] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[22] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[23] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[24] Y. Xie and G. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 35th International Symposium on Computer Architecture*, 2009.

[25] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[26] X. Zhuang and H. Lee. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers*, 2007.