SET-DUELING-CONTROLLED ADAPTIVE INSERTION FOR HIGH-PERFORMANCE CACHING

THE COMMONLY USED LRU REPLACEMENT POLICY CAUSES THRASHING FOR MEMORY-INTENSIVE WORKLOADS. A SIMPLE MECHANISM THAT DYNAMICALLY CHANGES THE INSERTION POLICY USED BY LRU REPLACEMENT REDUCES CACHE MISSES BY 21 PERCENT AND REQUIRES A TOTAL STORAGE OVERHEAD OF LESS THAN 2 BYTES.

Moinuddin K. Qureshi
IBM Research

Aamer Jaleel
Intel

Yale N. Patt
University of Texas at Austin

Simon C. Steely Jr.
Joel Emer
Intel

One of the major limiters of computer system performance has been the access to main memory, which is typically two orders of magnitude slower than the processor. To bridge this gap, modern processors already devote more than half the on-chip transistors to the last-level cache (in our studies, the L2 cache). These designs typically use the least-recently-used (LRU) replacement policy, or its approximations, for managing all levels of the cache hierarchy. Because smaller levels of the cache hierarchy filter out temporal locality, the access stream of the last-level cache has very little temporal locality. As a result, the LRU policy causes a significant percentage of cache lines in the last-level cache to remain unused after cache insertion. We refer to cache lines that are not reused between insertion and eviction as *zero-reuse lines.* Figure 1 shows that for the baseline 1-Mbyte, 16-way, LRU-managed L2 cache, more than half the lines installed in the cache are never reused before being evicted. Thus, the LRU policy results in inefficient use of L2 cache space because most of the inserted lines occupy cache space without contributing to cache hits.

Zero-reuse lines occur because of two reasons: First, the line has no temporal locality, which means that the line is never re-referenced. It is not beneficial to insert such lines in the cache. Second, the line is re-referenced at a distance greater than the cache size, so the LRU policy evicts the line before it is reused. For example, if a workload frequently reuses a working set of 2 Mbytes, and the available cache size is 1 Mbyte, the LRU policy will evict all the inserted lines before they are reused, causing zero reuse for almost all the lines. Figure 2 shows misses per thousand instructions (MPKI) for two memory-intensive benchmarks (art and mcf) when the cache size is varied under the LRU policy. Art frequently uses a 1.3-Mbyte data structure, and mcf frequently uses a 3.5-Mbyte data structure. For the baseline 1-Mbyte cache, the LRU replacement policy causes thrashing for these workloads and the cache lines are evicted before being reused. Zero-reuse lines account for more than 90 percent of the installed cache lines for these two workloads, indicating inefficient use of cache space.

0272-1732/08/$20.00 © 2008 IEEE Published by the IEEE Computer Society

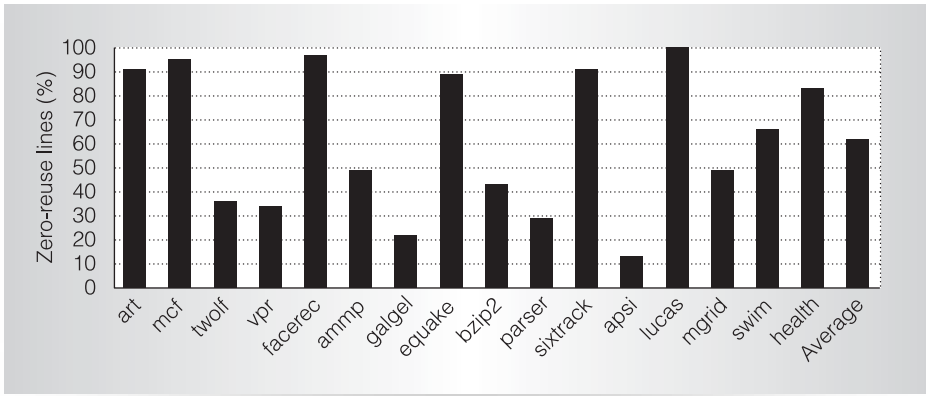

Figure 1. Zero-reuse lines for a 1-Mbyte, 16-way L2 cache.

For such memory-intensive workloads, cache performance can be improved significantly if the cache retains some fraction of the working set, so that at least that fraction can provide cache hits. Decades of research in cache replacement have produced alternatives to LRU, but most previous proposals either required too much hardware, incurred significant cache structure changes, or performed poorly for LRU-friendly workloads (see the "Related Work" sidebar for details). To improve the replacement policy without these drawbacks, we first separate the replacement policy into two parts: *victim selection* and *insertion*. The victim selection policy decides which line to evict to provide storage for an incoming line. The insertion policy decides where in the replacement list the incoming line is placed—traditional LRU policy always inserts the incoming line in the most recently used (MRU) position. In this article, we propose simple changes to the insertion policy that can significantly improve cache performance for memory-intensive workloads while requiring negligible hardware overhead.

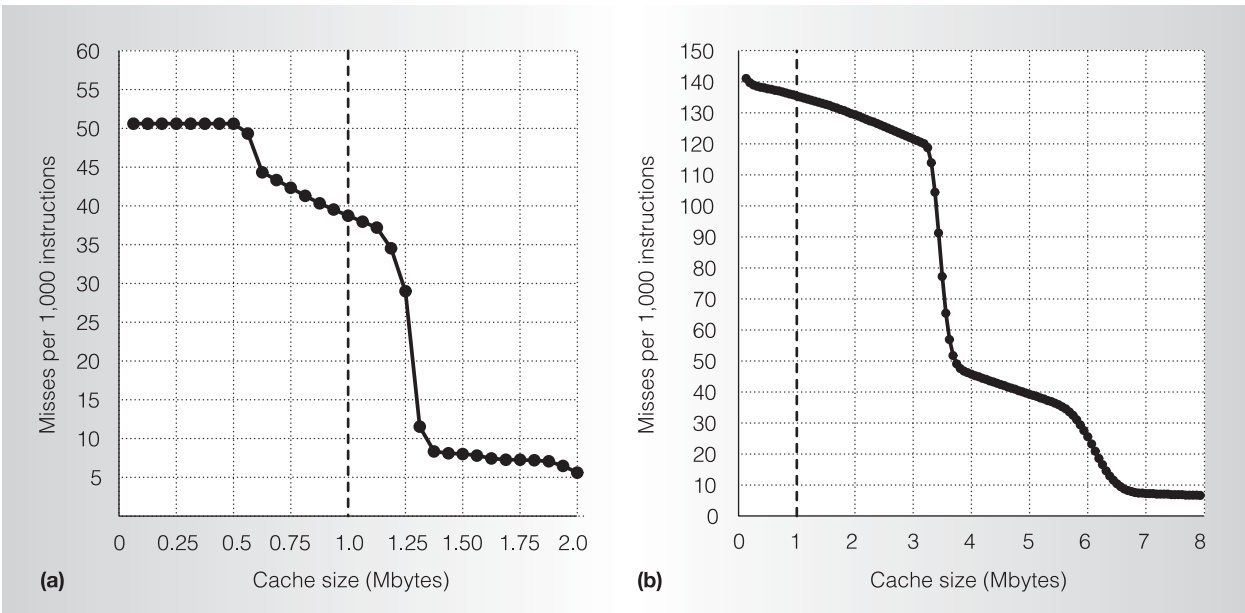

Figure 2. Misses per thousand instructions versus cache size for art (a) and mcf (b).

Static insertion policies

Our first proposal for improving LRU replacement is the *LRU insertion policy* (LIP), which places all incoming lines in the LRU position. The policy promotes lines from the LRU position to the MRU position only if they are used while in the LRU position. LIP prevents thrashing for workloads whose working set is greater than the cache size and obtains a near-optimal hit rate for workloads with a cyclic access pattern.

LIP can retain lines in the non-LRU position of the recency stack even if they cease to be re-referenced. Because LIP does not have an aging mechanism, it might not respond to changes in the given application's working set. Thus, we also propose the *bimodal insertion policy* (BIP), which is similar to LIP, except that it infrequently (with a low probability, ϵ) places some incoming lines in the MRU position. We implement BIP with one global 5-bit counter (BIPCTR) that is incremented on each cache miss. When BIPCTR is zero, the incoming line is inserted in the MRU position; otherwise it is inserted in the LRU position. BIP retains the thrashing protection of LIP but also adapts to changes in the working set, as we will show next.

Analysis with cyclic reference model

To analyze workloads that cause thrashing with the LRU policy, we use a theoretical model of cyclic references.[1] Let a_i denote the address of a cache line. Let $(a_1 \ldots a_T)$ denote a sequence of references a_1, a_2, …, a_T. A sequence that repeats N times is represented as $(a_1 \ldots a_T)^N$.

Let there be an access pattern in which $(a_1 \ldots a_T)^N$ is followed by $(b_1 \ldots b_T)^N$. We analyze this pattern's behavior for a fully associative cache that contains space for storing $K (K < T)$ lines. We assume that parameter ϵ in BIP is small and that both sequences in the access pattern repeat many times ($N \gg T$ and $N \gg K/\epsilon$). Table 1 compares the hit rates of LRU, OPT,[2] LIP, and BIP for this access pattern. OPT is an oracle-based replacement scheme that provides an upper-bound for the hit-rate by replacing the line that is accessed furthest in the future.

Related Work

Cache replacement studies have received much attention from both industry and academia. For workloads that cause thrashing with the LRU policy, both random-based and frequency-based replacement schemes have fewer misses than LRU. However, these schemes significantly increase misses for LRU-friendly workloads. Recent studies have investigated hybrid replacement schemes that dynamically select from two or more competing replacement policies. Examples of hybrid replacement schemes include sampling-based adaptive replacement (SBAR)[1] and adaptive cache (AC).[2] The problem with hybrid replacement is that it requires tracking separate replacement information for each of the competing policies. For example, if the two policies are LRU and LFU (least frequently used), each tag entry in the baseline cache must be appended with frequency counters (≥5 bits each), which must be updated on each access. Also, the hybrid schemes require extra structures for dynamic selection (2 Kbytes for SBAR and 34 Kbytes for AC) which consumes hardware and power.

Table A compares SBAR-based hybrid replacement schemes with the dynamic insertion policy (DIP) proposed in this article and the best-offline replacement (Belady's OPT)[3]. Each hybrid scheme has two component policies, LRU and one of the following replacement policies: *MRU-repl* replaces the MRU line; *NMRU-mid*[4] replaces a line randomly from the less recent half of the recency stack; *Rand* is random replacement; *RLRU-skew (RMRU-skew)*[4] is a skewed random policy that uses a linearly increasing (decreasing) replacement probability for recency positions ranging from MRU to LRU; and *LFU* is the least-frequently-used policy implemented with 5-bit saturating counters.[2] DIP outperforms the best-performing hybrid replacement while obviating the design changes, hardware and power overheads, and complexity of hybrid replacement. In fact, DIP bridges two-thirds of the gap between LRU and OPT for memory-intensive applications while requiring less than 2 bytes of extra storage.

Table A. Replacement policy comparison: Hybrids vs. DIP.

Replacement policy	MPKI reduction over LRU (%)	Hardware overhead (bytes)
Hybrids		
LRU + MRU-repl	8.8	2 K
LRU + NMRU-mid	5.1	2 K
LRU + Rand	8.9	2 K
LRU + RLRU-skew	6.6	2 K
LRU + RMRU-skew	11.3	2 K
LRU + LFU	14.7	12 K
DIP	21.3	2
Belady's OPT	32.2	N/A

References

1. M.K. Qureshi et al., ''A Case for MLP-Aware Cache Replacement,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 167-178.
2. R. Subramanian, Y. Smaragdakis, and G. Loh, ''Adaptive Caches: Effective Shaping of Cache Behavior to Workloads,'' *Proc. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 06), IEEE CS Press, 2006, pp. 385-396.
3. L.A. Belady, ''A Study of Replacement Algorithms for a Virtual-Storage Computer,'' *IBM Systems J.*, vol. 5, no. 2, 1966, pp. 78-101.
4. F. Guo and Y. Solihin, ''An Analytical Model for Cache Replacement Policy Performance,'' *Sigmetrics Performance Evaluation Rev., Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems* (Sigmetrics 06), vol. 34, no. 1, ACM Press, 2006, pp. 228-239.

Table 1. Hit rates for LRU, OPT, LIP, and BIP.

Policy	Hit rate for $(a_1 \ldots a_T)^N$	Hit rate for $(b_1 \ldots b_T)^N$
LRU	0	0
OPT	$(K-1)/(T-1)$	$(K-1)/(T-1)$
LIP	$(K-1)/T$	0
BIP	$\approx(K-1)/T$	$\approx(K-1)/T$

Because the cache size is less than T, LRU causes thrashing and results in zero hits for both sequences. The optimal policy (OPT) retains $(K-1)$ lines out of the T lines of the cyclic reference to provide a hit rate of $(K-1)/(T-1)$ for both sequences.[1] LIP's hit rate is similar to OPT's for the first sequence. However, LIP never allows any element of the second sequence to enter the cache's non-LRU position, so it achieves zero hits for the second sequence.

In each iteration, BIP inserts approximately $\epsilon(T-K)$ lines in the MRU position, which means a hit rate of $(K-1-\epsilon[T-K])/T$. Because the value of ϵ is small, BIP obtains a hit rate of approximately $(K-1)/T$, which is similar to LIP's hit rate for the first sequence. However, BIP probabilistically allows the lines of any sequence to enter the MRU position. Therefore, when the sequence changes from the first to the second, all the lines in the cache belong to the second sequence after K/ϵ misses. For large N, the transition time from the first sequence to the second sequence is small, and the hit rate of BIP is approximately equal to $(K-1)/T$. Thus, for small values of ϵ, BIP can respond to changes in the working set and retain LIP's thrashing protection.

Results

We evaluate the proposed insertion policies on a 1-Mbyte, 16-way L2 cache. The L1 caches are 16-Kbyte, 2-way. All caches in the baseline use LRU replacement and have a 64-byte line size. Figure 3 shows the reduction in L2 MPKI achieved by the two insertion policies, LIP and BIP, over the baseline LRU replacement policy. For BIP, we show results when every 32nd miss is inserted in the MRU position ($\epsilon = 1/32$). (We present sensitivity results for BIP elsewhere.[3])

The thrashing protection of LIP and BIP reduces MPKI by 10 percent or more for 9 of the 16 benchmarks. For benchmarks equake, parser, bzip2, and swim, however, both LIP and BIP increase the MPKI considerably. This occurs because these workloads have an LRU-friendly access

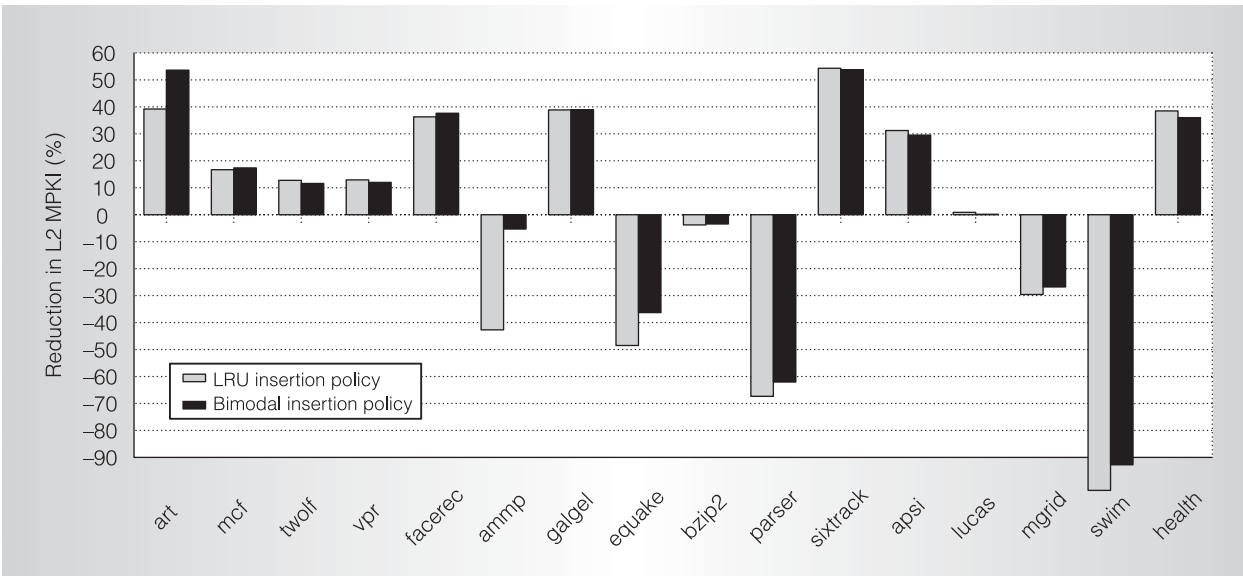

Figure 3. Comparison of static insertion policies.

pattern. Additionally, the MPKI increase can occur when the knee of the MPKI curve is less than the cache size and there is no significant benefit from increasing the cache size. For example, swim has two working sets: one 512 Kbytes and the other more than 64 Mbytes. The LRU policy is very close to OPT in such cases, so changing the insertion policy increases the MPKI considerably. For the insertion policy to be useful for a wide variety of workloads, we need a mechanism that can dynamically select between the traditional LRU policy and BIP, depending on which incurs fewer misses.

Dynamic insertion policy

We propose the *dynamic insertion policy* (DIP), which dynamically estimates the number of misses incurred by the two competing insertion policies and selects the one that incurs the fewest misses. A straightforward method of implementing DIP is to implement both LRU and BIP in two extra tag directories (data lines are not necessary for estimating the misses incurred by an insertion policy) and keep track of the number of misses incurred by the two policies. The cache's main tag directory can then use the policy that incurs the fewest misses. However, this implementation incurs prohibitive area and power overheads. For the proposed mechanism to be useful, it must incur very low overhead and minimal changes to the existing cache design. Thus, we next describe a cost-effective runtime mechanism to implement DIP.

Implementing DIP via set dueling

Since L2 caches typically have a large number (thousands) of sets, the extra tag directories can be avoided by sampling a few sets of the cache to estimate the performance of competing policies. We propose a mechanism called *set dueling*, which predicts the best policy by monitoring the two policies on a few dedicated sets. Figure 4 shows DIP implemented using set dueling for a cache that has 16 sets. Sets 0, 5, 10, and 15 always use the LRU policy, whereas, sets 3, 6, 9, and 12 always use BIP. A saturating counter (PSEL) tracks which of the two policies incurs fewer misses and selects that policy for the remaining follower sets. (In another publication, we derive analytical bounds and show that as few as 32 to 64 dedicated sets are sufficient for set dueling to select the best policy.[3])

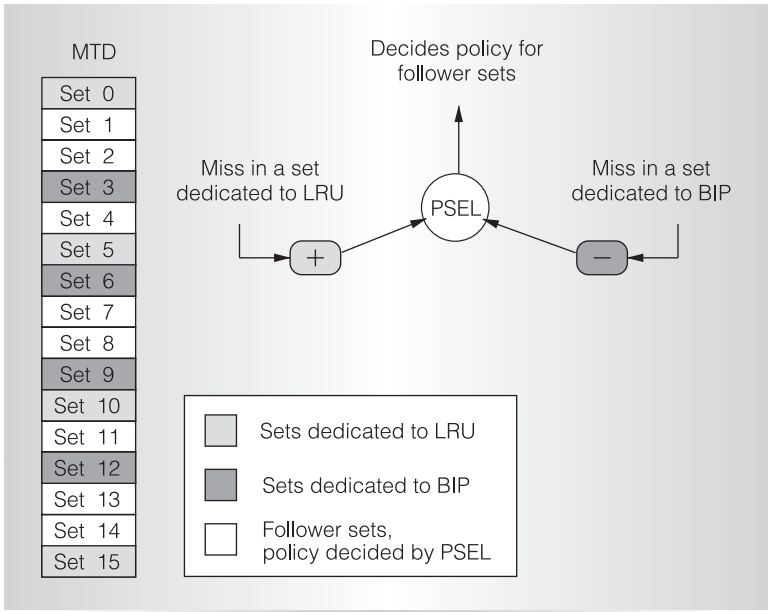

Figure 4. Implementing DIP using set dueling.

Results

Figure 5 shows the reduction in MPKI achieved by BIP, DIP-Global (two extra tag directories, 64 Kbytes each), and DIP-SD (set dueling with 32 dedicated sets). The arithmetic mean bar presents the reduction in arithmetic mean MPKI over all 16 benchmarks. DIP retains the MPKI reduction of BIP but eliminates the significant MPKI degradation of BIP on benchmarks equake, parser, mgrid, and swim. DIP-SD obtains an MPKI reduction similar to DIP-Global while avoiding the huge hardware overhead. On average, DIP reduces MPKI by 21 percent compared to LRU. DIP improves the instructions per cycle (IPC) of a 4-wide issue machine with 270-cycle memory latency by 9.3 percent.

Adaptation of DIP to runtime behavior

DIP can adapt to different applications as well as different phases of the same application. DIP uses the PSEL counter to select component policies. For a 10-bit

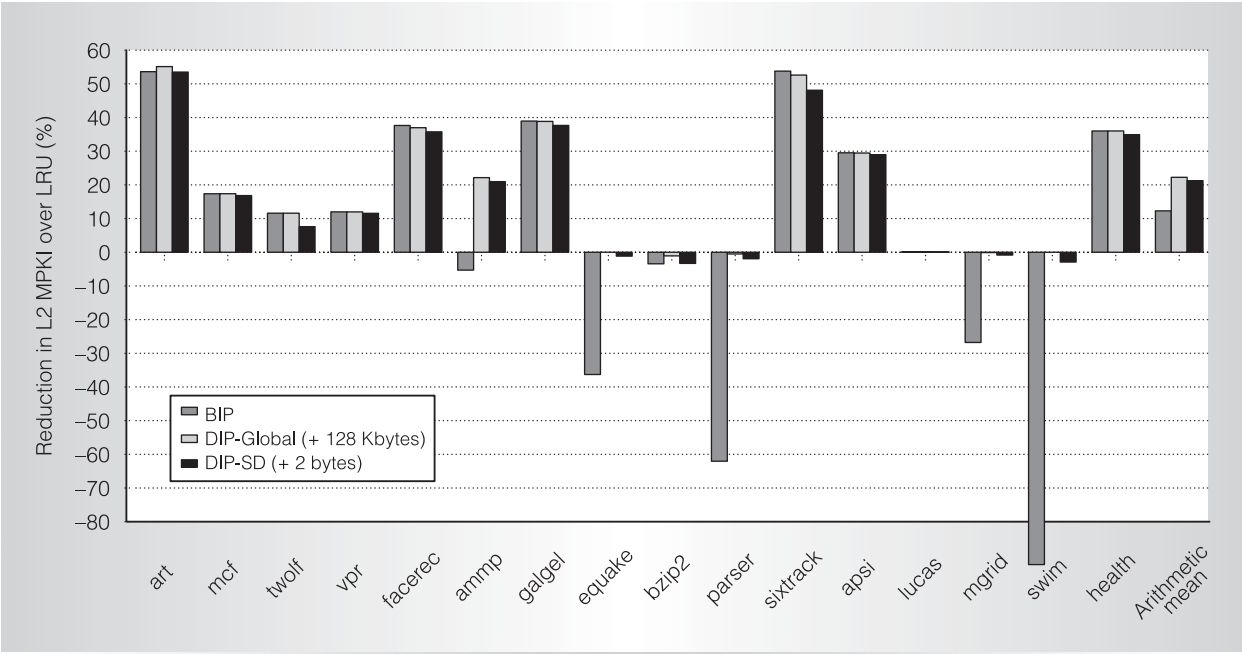

Figure 5. Comparison of static and dynamic insertion policies.

PSEL counter, a value of 512 or more indicates that DIP uses BIP; otherwise DIP uses LRU. Figure 6 shows the value of the 10-bit PSEL counter over the course of execution for the benchmarks mcf, health, swim, and ammp. We sample the PSEL counter's value once every million instructions.

For mcf, the DIP mechanism almost always uses BIP. For the health benchmark, the working set during the initial part of program execution fits in the baseline cache, and either policy works well.[3] However, as the data set increases during program execution, it exceeds the baseline cache's size, and LRU causes thrashing. Because BIP would have fewer misses than LRU, the PSEL value nears positive saturation, and DIP selects BIP. For the LRU-friendly swim benchmark, the PSEL value is almost always near negative saturation, so DIP selects LRU. Ammp has two phases of execution: In the first phase, LRU is better, and in the second. BIP is better. Because DIP selects the policy best suited to each phase, it has better MPKI than either of the component policies alone.

Implementation overhead

The proposed insertion policies (LIP, BIP, and DIP) require negligible hardware overhead and design changes. LIP inserts all incoming lines in the LRU position, which can easily be implemented by not performing the update to the MRU position that occurs on cache insertion. BIP is similar to LIP, except that it infrequently inserts an incoming line in the MRU position. To control the rate of MRU insertion in BIP, the 5-bit counter BIPCTR is incremented on every cache miss. BIP inserts the incoming line in the MRU position only if the BIPCTR is zero. Thus, BIP incurs a storage overhead of 5 bits. DIP requires storage for the 10-bit saturating counter PSEL.

Figure 7 shows the design changes incurred in implementing DIP. The implementation requires a total storage overhead of 15 bits (5-bit BIPCTR + 10-bit PSEL) and negligible logic overhead. A particularly attractive aspect of DIP is that it does not require extra bits in the tag-store entry, thus avoiding changes to the existing structure of the cache. The absence of extra structures also means that DIP does not incur power and complexity overheads. Because DIP adds no logic to the cache access path, the cache's access time remains unaffected. Furthermore, because the proposed inser-

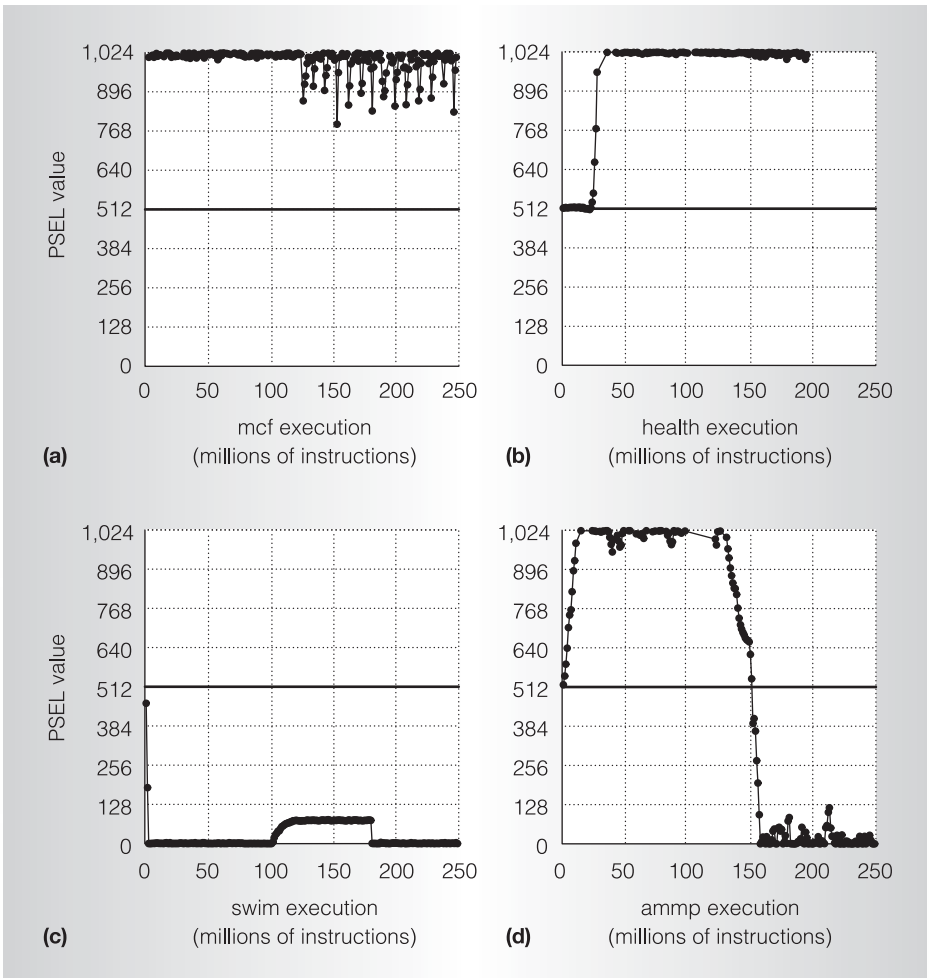

Figure 6. PSEL value during benchmark execution.

tion policies do not rely on true LRU, they are amenable to the LRU approximations widely used in current on-chip caches.

Simple changes to the insertion policy used by LRU replacement can significantly improve cache performance of memory-intensive workloads. The LRU insertion policy (LIP) protects the cache against thrashing and yields close to optimal hit rates for applications with a cyclic reference pattern. The bimodal insertion policy (BIP) enhances LIP by allowing for aging and adapting to changes in an application's working set. Finally, the dynamic insertion policy (DIP) dynamically chooses between BIP and traditional LRU replacement depending on which policy incurs fewest misses for the given workload. We also

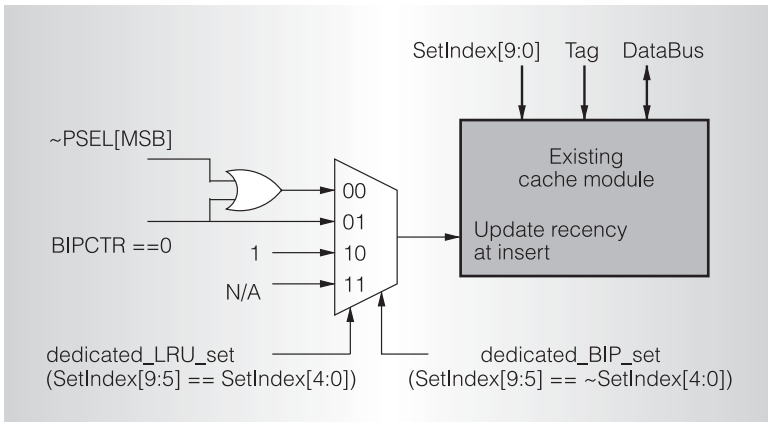

Figure 7. Hardware changes for implementing DIP.

proposed set dueling as a means to implement cost-effective dynamic selection between competing policies. This article evaluated the insertion policies for private caches. However, the proposed insertion policies can easily be extended to shared caches. Set dueling can be used as a basic building block for several other cache optimizations including dynamically tuned prefetchers and adaptive cache compression. MICRO

Acknowledgments

We thank Archna Monga, Andy Glew, William Hasenplaugh, Veynu Narasiman, and Aater Suleman for their comments and feedback. This research was conducted while the first author was a graduate student at the University of Texas at Austin. The studies at UT were supported by gifts from IBM, Intel, and the Cockrell Foundation.

References

1. J.E. Smith and J.R. Goodman, ''A Study of Instruction Cache Organizations and Replacement Policies,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 10), ACM Press, 1983, pp. 132-137.
2. L.A. Belady, ''A Study of Replacement Algorithms for a Virtual-Storage Computer,'' *IBM Systems J.*, vol. 5, no. 2, 1966, pp. 78-101.
3. M.K. Qureshi et al., ''Adaptive Insertion Policies for High-Performance Caching,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 34), ACM Press, 2007, pp. 167-178.

Moinuddin K. Qureshi is a research scientist at IBM Research. His research interests include computer architecture, especially the design of high-performance and scalable memory systems. Qureshi has a PhD in electrical and computer engineering from the University of Texas at Austin. He was a recipient of the IBM PhD fellowship for academic years 2003 through 2006. He is a member of the IEEE and the ACM.

Aamer Jaleel is a hardware engineer at Intel, working in the VSSAD group. His research interests include cache/memory system design, parallel architectures, computer architecture, microarchitecture, performance modeling, and workload characterization. Jaleel has a PhD in electrical engineering from the University of Maryland, College Park. He is a member of the IEEE.

Yale N. Patt is the Ernest Cockrell, Jr., Centennial Chair in Engineering at the University of Texas at Austin. His research interests include harnessing the expected benefits of future process technology to create more effective microarchitectures for future microprocessors. He is coauthor of *Introduction to Computing Systems: From Bits and Gates to C and Beyond* (McGraw-Hill, 2nd edition, 2004). His honors include the 1996 IEEE/ACM Eckert-Mauchly Award and the 2000 ACM Karl V. Karlstrom Award. He is a Fellow of both the IEEE and the ACM.

Simon C. Steely Jr. is a senior principal engineer at Intel, working in the VSSAD group. His technical interests include cache/memory hierarchies, cache coherency, and computer architecture. Steely has a BE (computer science option) from the University of New Mexico. He is a member of the IEEE and the ACM.

Joel Emer is an Intel Fellow and director of microarchitecture research at Intel, where he leads the VSSAD group. Recently, he has also been teaching part-time at MIT. His research interests include performance-modeling frameworks, parallel and multithreaded processors, cache organization, processor pipeline organization, and processor reliability. Emer has a PhD in electrical engineering from the University of Illinois. He is an ACM Fellow and an IEEE Fellow.

Direct questions and comments about this article to Moinuddin Qureshi, IBM Research, PO Box 218, Yorktown Heights, NY 10598; mkquresh@us.ibm.com.