MadCache: A PC-aware Cache Insertion Policy

Mitchell Hayenga, Andrew Nere, Mikko Lipasti

MadCache: A PC-aware Cache Insertion Policy

Mitchell Hayenga
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Email: hayenga@ece.wisc.edu

Andrew Nere
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Email: nere@wisc.edu

Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Email: mikko@engr.wisc.edu

***Abstract*—While the field of computer architecture is always looking for novel research directions to bring improved performance and efficiency, it is often simple improvements to more mature topics that have the most substantial impact. Cache replacement policy is one such research area, where innovations are highly sought after because of their direct improvement on performance. Furthermore, as chip-multiprocessors have become the dominant chip design, new cache replacement schemes should seek to improve the performance of workloads for both single-threaded and shared cache multithreaded systems.**

In this paper we propose MadCache, a cache insertion policy that uses memory access history based on the Program Counter (PC) to determine the appropriate policy for the L3 cache. A PC-based history table stores information regarding cache accesses and determines whether the L3 should default to the LRU replacement policy for workloads that exhibit good locality or bypass for streaming memory accesses. Furthermore, this PC-based history table allows individual PCs to override this default policy if its history indicates a behavior significantly different from the typical trend of the workload. We show that MadCache is able to improve IPC by 2.5% over LRU for a single-threaded 1MB 16-way L3 cache. Finally, we extend MadCache to a four thread, 4MB shared L3 cache and demonstrate a 6% improvement in throughput and 4.5% speedup over LRU averaged across the mixed benchmarks we tested.

I. Introduction

The past decade has seen a substantial shift in computer architecture research, as focus has changed from powerful single out-of-order cores to chip-multiprocessors (CMPs) like Sun Niagara and energy efficient in-order cores like the Intel Atom. While such major renovations to computer architectures have certainly enhanced performance and efficiency, it is often simple improvements to more mature research topics that have the most substantial impact. Cache replacement policy could be considered such a mature topic, though it is also one of the most important due to its direct effect on system performance. Multiple cache levels as well as the complexity of cache sharing CMPs provide challenging opportunities to be exploited by innovative cache replacement policies.

The LRU replacement policy has remained the standard for on-chip caches for many years. However, recent studies have shown that there is still reason to develop new and interesting policies. The work of [4] proposed a dynamically changing cache insertion policy with modest hardware overhead. Their proposed Dynamic Insertion Policy (DIP) monitors cache accesses to determine whether incoming cache lines would follow the traditional LRU policy or a newly proposed Bimodal Insertion Policy (BIP). BIP inserts the majority of incoming cache lines to the LRU position anticipating streaming activity and inserts a few lines to the MRU position as a form of thrashing protection. The DIP policy has also been extended to CMPs with shared last-level caches, which are a common design for current multiprocessor systems such as the Sun Niagara and Intel Nehalem. The authors of [2] created Thread-Aware Dynamic Insertion Policy (TADIP), which extended DIP to compensate for multiple threads competing for a shared last-level cache. Between the trend of increasing the number of cache levels, growing cache sizes, changing workloads, and unpredictable cache behavior for CMP systems, there is clearly room for investigating alternative replacement policies.

In this paper we propose MadCache, a single-threaded and multithreaded adaptive insertion policy that uses Program Counter (PC) behavior history to determine the appropriate insertion policy for a last level cache. This PC tracking information determines whether the line will follow the LRU replacement policy or simply bypass without polluting the L3 cache. Like TADIP, MadCache also considers the competitive nature of a multithreaded system with a shared cache by tracking separate PC behavior depending on the current insertion policies of the other threads. In this paper, we demonstrate how MadCache is able to take advantage of PC information and show a significant improvement over LRU replacement.

II. MadCache Insertion Policy

MadCache is an adaptive cache insertion policy which uses a PC's memory access history and behavior to determine the best insertion policy. Previous work on PC-based prediction techniques exploit the fact that the majority of programs will exhibit some form of repetitive behavior. These have been used for memory prefetching [6], cache management [3], and operating system buffer caching [1]. The underlying assumption of MadCache is if a particular PC in a program exhibits streaming behavior, it will likely continue to do so, and should be prevented from flushing out other useful entries in the cache. However, if a PC exhibits high amounts of locality with its accesses, LRU is probably the best policy for guaranteeing that useful entries will be available in the cache.

MadCache uses several structures to adapt the insertion policy of the L3 cache. The first of these structures is a subset of the entire cache, which we refer to as tracker sets. Second, a lookup table known as the PC-Predictor is used to track the

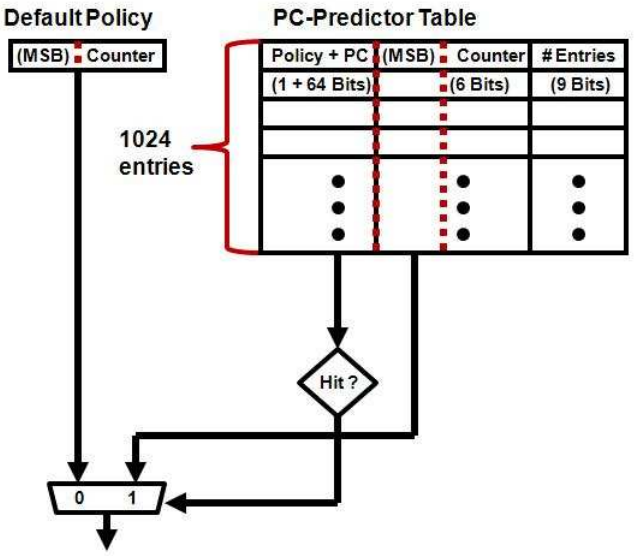

Fig. 1. The PC-Predictor table maintains the history of cache activity based on the PC for a subset of the L3. This history determines the default policy (LRU or bypass) of the cache. The PC-Predictor table has the power to override the default policy.

history of PCs that have accessed any of these tracker sets. Finally, set dueling among the tracker sets will determine the current default policy of the L3 cache, either LRU or simply bypassing to the L2 cache. This set dueling is implemented as a 10-bit counter, where the most significant bit (MSB) is used as the threshold to indicate if the cache is in bypassing mode. The overall structure of MadCache is depicted in Figure 1.

A. The PC-Predictor Table

The PC-Predictor is simply a lookup table based on the PCs that access the tracker sets, each of which updates a counter that indicates whether this PC exhibits streaming memory accesses or good memory locality. Since a PC may exhibit different behavior depending on what the cache's current default policy is, separate histories are tracked and both the current default policy and the PC are used to index into the table. The PC-Predictor uses a 6-bit counter where the MSB is used as a threshold to indicate if the PC entry will bypass. When this threshold bit is set, all future cache misses by this particular PC entry will be treated as streaming and simply bypass the L3 cache. This table also monitors the total number of tracker set cache entries that are associated with this PC.

B. Tracker Sets

Figure 2 illustrates MadCache's tracker sets, a sample subset of the L3 cache used to approximate cache behavior. This sampling was based on Dynamic Set Sampling proposed in [5]. The PCs that access these tracker sets will be added to and update the PC-Predictor table. Additionaly, the index to the table is stored along with the cacheline and a reuse bit in the tracker set, so future reuse or non-reuse can be accordingly reward or penalize the tracked PC entry. As mentioned, set dueling among these tracker sets will determine the current default policy of LRU or bypassing. Finally, on any cache

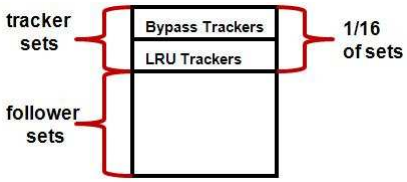

Fig. 2. A subsection of the L3 is tracked for behavior.

access, if the PC is currently in the PC-Predictor table, it can override the default policy. The underlying motivation for this override is that a PC's history will likely contain a better prediction for its behavior, even if it is contrary to the default insertion policy decided by the rest of the tracker set.

While a PC may exhibit streaming behavior for one portion of the program, it is possible it changes its behavior later on. For such a case, thrashing protection must be enabled to allow a PC to get out of bypass mode. MadCache extends BIP from [4] for thrash protection for such cases. Bypassing Bimodal Insertion Policy (BBIP) bypasses the L3 cache the majority of the time and is inserted into the MRU position the rest of the time.

Initially, the L3 cache and PC-Predictor table will be empty. After an initial L3 miss for a particular cache line in the tracker set, the invoking PC will be added to the PC-Predictor table. This PC will be entered in a free location, indicated by a PC-Predictor table entry with no cache entry associations. Initially, the counter associated with this PC defaults to indicate it is barely below the bypassing threshold, and the entry will be added to the L3 cache using the LRU insertion policy. If this particular cache block is accessed again, MadCache deems that the LRU insertion policy is appropriate for this particular PC and the PC-Predictor counter will be decremented so future misses by this PC will also be inserted into the L3 cache. However, if this entry is never touched again and is evicted from the L3 cache,the counter is incremented to indicate that future misses by this PC will be treated as streaming and bypass the L3 cache. If there are currently no free spaces in the PC-Predictor table after an access to a tracker set, the cache simply follows the overall default policy of the L3. Entries from the PC-Predictor will be evicted implicitly when the number of cache blocks referenced by a particular PC decrements to zero, creating a free space in the table. Finally, we only credit the PC that initially brought a line into the cache, and accesses to an L3 cache line by other PCs have no affect on MadCache.

C. Follower Sets

The majority of the L3 cache is simply follower sets. Cache accesses to the follower sets simply adhere to the default cache policy as determined by the tracker sets. It should be noted that if the default policy is bypassing, BBIP is again used for thrash protection. However, follower set cache accesses can also take advantage of the history information. While the majority of an application may exhibit streaming behavior, one particular PC may exhibit good locality and be allowed to insert into the L3. Conversely, for high locality workloads, a sparse number of

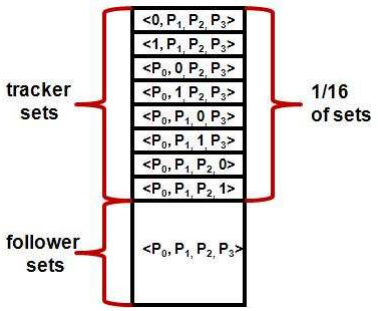

Fig. 3. Each thread must track behavior based on its own default policies.

PCs may be streaming. While the follower sets do not update the PC-Predictor table, PC matches between the follower set and the PC-Predictor allow the cache line to also override the default insertion policy.

D. Multithreaded MadCache

MadCache is also an adaptive insertion policy for multi-threaded systems with a shared cache. The basic framework and structures of multithreaded MadCache are the same. However, like the work proposed in [2], MadCache is designed to alter its insertion decisions based on the current insertion policies of the other threads sharing the cache. For the multithreaded case, each thread will maintain its own default insertion policy. To accommodate this extra information, the tracker sets need to be subdivided among the threads for both the LRU and bypass policies (Figure 3). To reflect this change in the tracker sets, the PC-Predictor table indices need to be accessed not only with the current PC, but also the current thread ID (TID) and the current policy of each of the threads in the system. Figure 4 shows the extensions to the structural components of multithreaded MadCache.

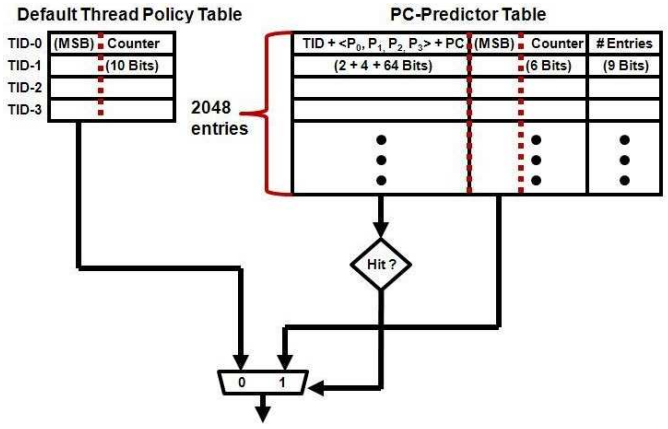

Fig. 4. The basic components of multithreaded MadCache are the same. However, since the TID and default policies of the other threads may affect cache behavior, this information must be appended to the PC-Predictor index.

E. Hardware Overhead

To implement single-threaded MadCache, the PC-Predictor uses a total of 80Kbits and tracking LRU for the cache requires 64Kbits. The default policy counter requires only 10 bits. Finally, each of the 1024 caches line requires 10 bits for indexing into the PC-Predictor and 1 bit to indicate reuse. As a result, MadCache requires less than 159Kbits of storage.

Multithreaded MadCache increases the number of PC-Predictor entries as well as the number of bits required for indexing, using a total of 170Kbits. Each of the 4096 cache lines again requires 1 bit to indicate reuse as well as 11 bits for indexing to the PC-Predictor table. Finally, another 40 bits are needed so each thread will track its own default policy and 256Kbits are required for tracking LRU. The total memory overhead required for multithreaded MadCache is less than 475Kbits. The PC-Predictor table for both the single and multithreaded configurations were sized by available storage budgets, though MadCache performed comparably in some preliminary tests with smaller configuration sizes.

III. Experimental Methodology

A. Configuration

For the experiments in this paper we used a simulation framework based on the CMP$im simulator. Table I shows the basic configuration of the processor and memory hierarchy of the system. We allocated a 1MB L3 cache for comparing single-threaded performance and 4MB for a four threaded shared last-level cache system. In our experiments, we compared using MadCache as the L3 cache policy against true LRU, DIP, and RAND.

TABLE I
System configuration

Processor	8-stage, 4-wide pipeline
Instruction window size	128 entries
Branch Predictor	Perfect
L1 inst. cache	32KB, 64B linesize, 4-way SA, LRU, 1 cycle hit
L1 data cache	32KB, 64B linesize, 8-way SA, LRU, 1 cycle hit
L2 cache	256KB, 64B linesize, 8-way SA, LRU, 10 cycle hit
L3 cache (1 thread)	1MB, 64B linesize, 30 cycle hit
L3 cache (4 threads)	4MB, 64B linesize, 30 cycle hit
Main memory	200 cycles

B. Benchmarks

The experiments of this paper were conducted using a selection of 15 benchmarks from SPEC CPU2006 compiled using the GCC compiler. For the multithreaded workloads, we used 15 workload mixes each created from a selection of four different SPEC CPU2006 benchmarks. Table II lists the workload mixes for the multithreaded experiments. All simulations were run for 200 million cycles.

IV. Results

Figure 5 shows the results of our single threaded implementation of MadCache normalized to true LRU performance. We also compared performance to a purely random cache replacement policy and implemented DIP as well. We see that gcc is the only benchmark where MadCache performs worse than LRU, and many other benchmarks exhibit a substantial

TABLE II
WORKLOAD MIXES

Mix0	xalancbmk, mcf, milc, gcc
Mix1	xalancbmk, mcf, hmmer, soplex
Mix2	bzip, zeusmp, lbm, hmmer
Mix3	soplex, xalancbmk, h264ref, astar
Mix4	libquantum, milc, soplex, bzip
Mix5	soplex, xalancbmk, soplex, bzip
Mix6	zeusmp, h264ref, gcc, soplex
Mix7	astar, GemsFDTD, hmmer, gcc
Mix8	zeusmp, xalancbmk, xalancbmk, soplex
Mix9	mcf, xalancbmk, astar, perlbench
Mix10	libquantum, soplex, perlbench, hmmer
Mix11	soplex, milc, libquantum, zeusmp
Mix12	lbm, xalancbmk, soplex, milc
Mix13	gamess, perlbench, h264ref, hmmer
Mix14	zeusmp, libquantum, soplex, milc

improvement. Looking at the geometric mean across all benchmarks tested, MadCache shows a 2.5% improvement in IPC, also showing a slight improvement over DIP as well.

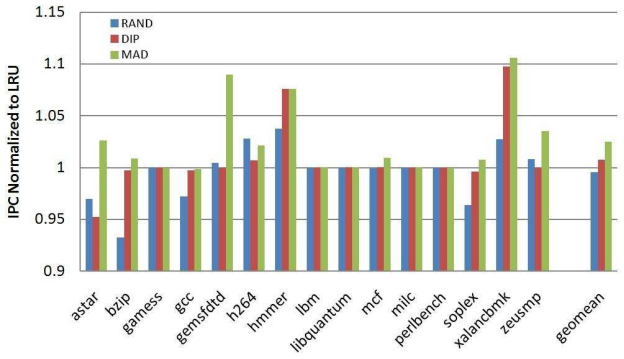

Fig. 5. IPC improvement of single-threaded MadCache normalized to LRU.

Figure 6 shows the throughput improvement for the multithreaded implementation of MadCache. For the mixes we tested, MadCache out performed both LRU and DIP. Again, we see a significant improvement across most of the other benchmarks, and MadCache exhibits a 6% average improvement in throughput over LRUfor the mixes tested. Figure 7 shows the weighted speedup of MadCache normalized to LRU. Again, MadCache performs faster than both LRU and DIP, resulting in a 4.5% speedup over LRU.

V. CONCLUSIONS AND FUTURE WORK

Considering changing workloads, growing cache sizes and structures, and the complex behavior of shared caches, challenging opportunities exist in the realm of cache replacement policies. Acknowledging the success of using PC-based program information for other architecture applications, we have proposed MadCache as a PC-aware adaptive cache insertion policy. With a limited amount of hardware overhead, we are able to estimate the L3 cache behavior and choose the correct policy at both the level of the thread and the PC granularity. Single-threaded MadCache showed a 2.5% improvement in IPC over LRU and multithreaded MadCache showed a 4.5% speedup with a 6% improvement in throughput over LRU.

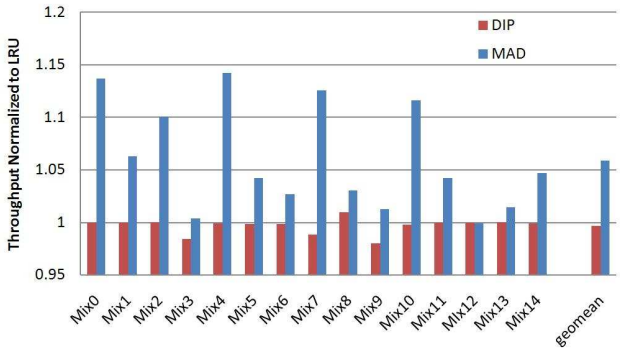

Fig. 6. Comparison of multithreaded MadCache for throughput.

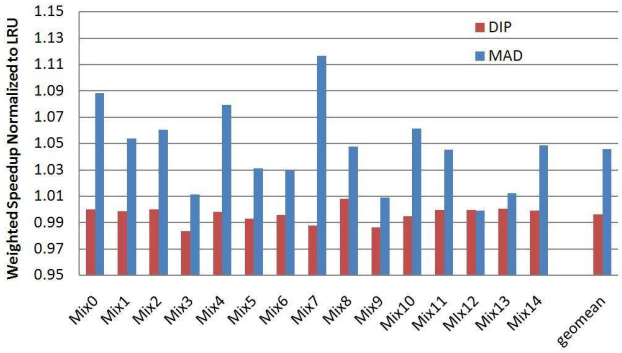

Fig. 7. Comparison of multithreaded MadCache for weighted speedup.

Although MadCache was able to beat DIP for the benchmarks tested, in the future we would like to compare against another thread-aware policy like TADIP. We also would like to reduce the hardware overhead of MadCache by storing only partial tags in the PC-Predictor table, though given time constraints we did not have time to adequately test such an implementation. Finally, we would like to extend MadCache to change tracker sets during operation to better capture the behavior of the cache.

REFERENCES

[1] C. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-based pattern classification in buffer caching. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 27–27, Berkeley, CA, USA, 2004. USENIX Association.

[2] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 208–219, New York, NY, USA, 2008. ACM.

[3] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 139–148, New York, NY, USA, 2000. ACM.

[4] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 381–391, New York, NY, USA, 2007. ACM.

[5] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.

[6] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 42–53, New York, NY, USA, 2000. ACM.