Insertion Policy Selection Using Decision Tree Analysis

Samira Khan[*], Daniel A. Jiménez[*†]

[*]*Department of Computer Science*
University of Texas at San Antonio

[†]*Barcelona Supercomputing Center*

skhan@cs.utsa.edu, djimenez@acm.org

Abstract—**The last-level cache (LLC) mitigates the impact of long memory access latencies in today's microarchitectures. The insertion policy in the LLC has a significant impact on cache efficiency. A fixed insertion policy can allow useless blocks to remain in the cache longer than necessary, resulting in inefficiency. We introduce insertion policy selection using Decision Tree Analysis (DTA). The technique requires minimal hardware modification over the least-recently-used (LRU) replacement policy. This policy uses the fact that the LLC filters temporal locality. Many of the lines brought to the cache are never accessed again. Even if they are reaccessed they do not experience bursts, but rather they are reused when they are near to the LRU position in the LRU stack. We use decision tree analysis of multi-set-dueling to choose the optimal insertion position in the LRU stack. Inserting in this position, zero reuse lines minimize their dead time while the non-zero reuse lines remain in the cache long enough to be reused and avoid a miss. For a 1MB 16 way set-associative last level cache in a single core processor, our policy uses only 2,069 additional bits over the LRU replacement policy. On average it reduces misses by 5.16% and achieves 7.19% IPC improvement over LRU.**

I. Introduction

A cache insertion policy dictates where a new block will be placed in a cache set. The insertion policy commonly used with the least-recently-used (LRU) replacement policy is to insert the new block into the top of the LRU stack, designated the most-recently-used (MRU) position. However, this is not necessarily the best place for all workloads. For example, if the block will not be used again, it will have to make its way down the LRU stack before it can be evicted. Thus, it makes sense to adaptively choose an insertion policy based on observed program behavior.

We introduce insertion policy selection using Decision Tree Analysis (DTA). Our policy requires little change in the least-recently-used (LRU) replacement policy hardware. For a single core 1MB last-level cache (LLC), this scheme requires only 2,069 additional bits over LRU replacement. We use LRU eviction for choosing the victim block. However, we insert incoming blocks at a specific position in the LRU stack learned by decision tree analysis from multi-set-dueling.

The LRU replacement policy inserts an incoming block in the MRU position. Because of temporal locality this block might be accessed again while it moves from the MRU position towards the LRU position. However, since the access stream is filtered by L1 and L2 caches, the LLC might not see this temporal locality. This is why LRU insertion has been proposed [1] for the last level cache. In that work, the cache learns whether it is best to insert into the LRU or MRU position. The two candidate insertion policies each have a *leader set* that always used that particular policy. The leader sets engage in *set-dueling* where the cache keeps track of which set is responsible for fewer misses. The rest of the cache sets are *follower sets* that use whichever of the policies has yielded the best performance.

However, that policy causes misses for blocks that were evicted but otherwise would have been accessed in some position nearer to the LRU position. Our insertion policy selects the appropriate insertion position where the workload can reduce dead time of zero reuse blocks, i.e., blocks that are never used again. It also retains the hits of non-zero reuse blocks by keeping a block long enough so that it is not evicted before its second access. We use decision tree analysis of multi-set-dueling to determine the optimal insertion position dynamically. Instead of having one leader set for each insertion position, our multi-set-dueling uses an adaptive insertion policy in the leader sets. Leader sets dynamically choose the insertion position based on the decision taken in the previous level of the decision tree. Thus, one leader set can implement many insertion policies which makes the number of policies that can be used in multi-set-dueling scalable.

II. Insertion Policy Selection Using Decision Tree Analysis

A. Motivation

The motivation behind this work is the filtered temporal locality in the last level cache. Due to hits in the L1 and L2 caches, the access stream in the LLC does not have much temporal locality. A large portion of the blocks brought to the cache are never accessed again.

Even if these blocks are reused they do not experience bursts and are accessed when they are nearer to the LRU position. Fig 1 shows the percentage of blocks brought to the LLC that are never accessed again. Fig 2 shows that only a small percentage of the hits occur when the blocks are near the MRU position. Most of the hits occur while the blocks move toward the end of the LRU stack. Without using any storage-intensive algorithm to accurately identify the zero reuse blocks, we can eliminate these blocks just by inserting them in the LRU position [1]. However, this will also evict blocks that are reused when they travel down the LRU stack. There is an optimal position in the LRU stack where inserting the blocks, zero reuse blocks will be evicted earlier while non-zero reuse blocks will remain in the cache avoiding a miss. We propose to use decision tree analysis to determine this optimal insertion position. This analysis is based on multiple set dueling [2]. However, we propose to use adaptive insertion policy for the leader sets. This reduces the number of sets in each leader set group. It also minimizes the negative effect of leader sets that implement losing insertion policies.
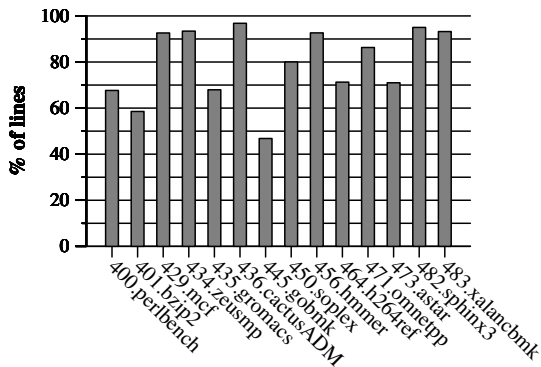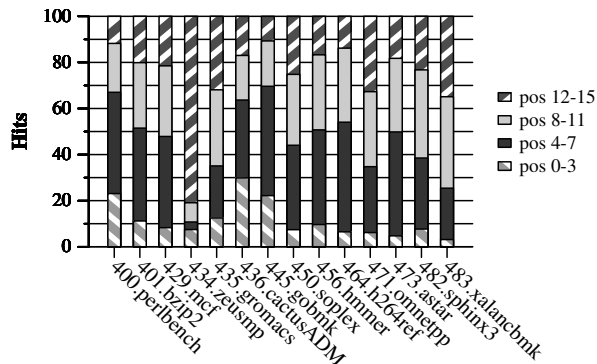

Fig. 1.   Percentage of zero reuse blocks

Fig. 2.   Hit Position, 0 is MRU and 15 is the LRU position

B. Decision Tree Analysis

Our scheme considers five different insertion positions in the LRU stack. It divides the LRU stack into four equal segments. The default placement is MRU. DIP [1] considers LRU as an insertion position. We consider the middle position of the LRU stack and other two equidistant positions from the middle position. These two positions are named *near LRU position* and *near MRU position*. Figure 3 shows these five positions in the LRU stack. It also shows how the appropriate insertion position is selected using the decision tree. The insertion position is chosen after a few rounds of competition as illustrated in Figure 3.

C. Adaptive policy in Leader Sets for Multi Set Dueling

Multi-set-dueling was proposed for multi-threaded workloads [3]. Each application has its own counter and it decides to insert in either LRU position or MRU position depending on that counter value. Multi-core multi-policy set-dueling was subsequently proposed [2]. In each core there are leader sets for each of the competing policies grouped into two. In the first round two policies in one group duel with each other. The winner policy of the first round are deployed in the partial follower sets (ϕ sets). The second level winner is then determined from the duel of these ϕ sets. Thus, the policy selection becomes a tournament where at each round half of the policies are eliminated. In the final round there are only two policies left and the winner policy is followed by all the other follower sets.

The problem with this approach is number of leader sets goes up with the number of policies being considered for multi set dueling. When many policies are dueling in a tournament manner, even if we can choose the best performing policy for the rest of the follower sets, all but one leader set continue using the wrong policy, potentially hurting performance significantly when the number of leader set increases. Another problem is the presence of partial follower sets. These sets are redundant as there are leader sets already present in the cache using that specific winner policy.

We have used the idea of multi set dueling in a single-core context. However, the problems of this scheme are solved by using leader set that can dynamically select specific insertion policy. We also remove the partial follower sets. Figure 4 shows the difference in two schemes. The first group of leader set is defined according to previous work [2]. First round is between policy p_a, p_b and p_c, p_d and p_e, p_f and p_h, p_g. The winner is deployed in partial follower sets ϕ_{ab}, ϕ_{cd}, ϕ_{ef} and ϕ_{gh}. These sets duel in pairs and the tournament goes to semi-final and final round (not shown in the figure).

We show our leader set with adaptive policy in the second group of the leader sets. Here we have only three kinds of leader sets. The first two leader sets implement policy p_a and p_b. The last set implements p_α. Depending on which set is winning, we can dynamically choose among the policies p_c, p_d, p_e, p_f, p_h and p_g. In the next section we describe how we use this idea in our insertion position selection.
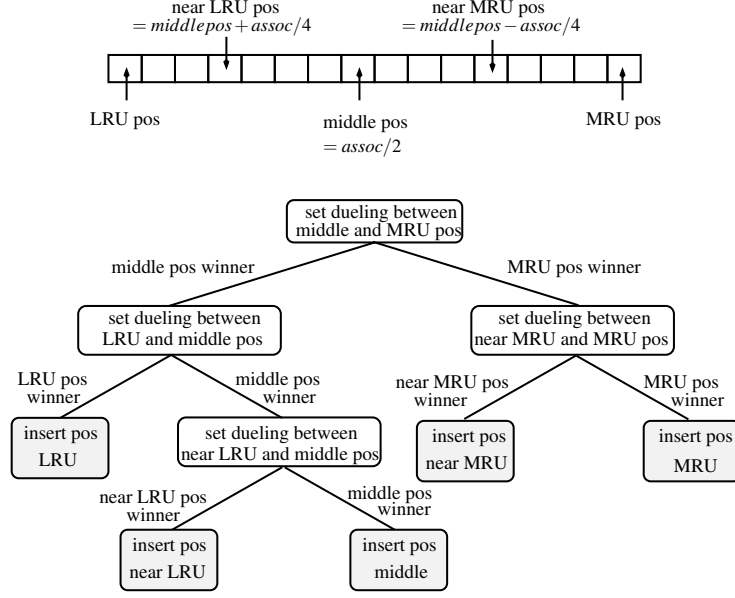
near LRU pos
$= middlepos + assoc/4$

near MRU pos
$= middlepos - assoc/4$

LRU pos

middle pos
$= assoc/2$

MRU pos

set dueling between middle and MRU pos

middle pos winner

MRU pos winner

set dueling between LRU and middle pos

set dueling between near MRU and MRU pos

LRU pos winner

middle pos winner

near MRU pos winner

MRU pos winner

insert pos LRU

set dueling between near LRU and middle pos

insert pos near MRU

insert pos MRU

near LRU pos winner

middle pos winner

insert pos near LRU

insert pos middle

Fig. 3.   Decision Tree Analysis

p_a -1 $psel_{ab}$
p_b $+1$
ϕ_{ab} -1
p_c -1 $psel_{cd}$
p_d $+1$ $psel_1$
ϕ_{cd}
p_e -1 $psel_{ef}$ $+1$
p_f $+1$
ϕ_{ef} -1
p_g -1 $psel_{gh}$
p_h $+1$ $psel_2$
ϕ_{gh} $+1$

leader sets in current multi set dueling scheme

all sets in LLC

p_a -1 -1, if p_a wins
$psel_1$
p_b $+1$ $psel_2$
p_α $+1$, if p_b wins
$+1$

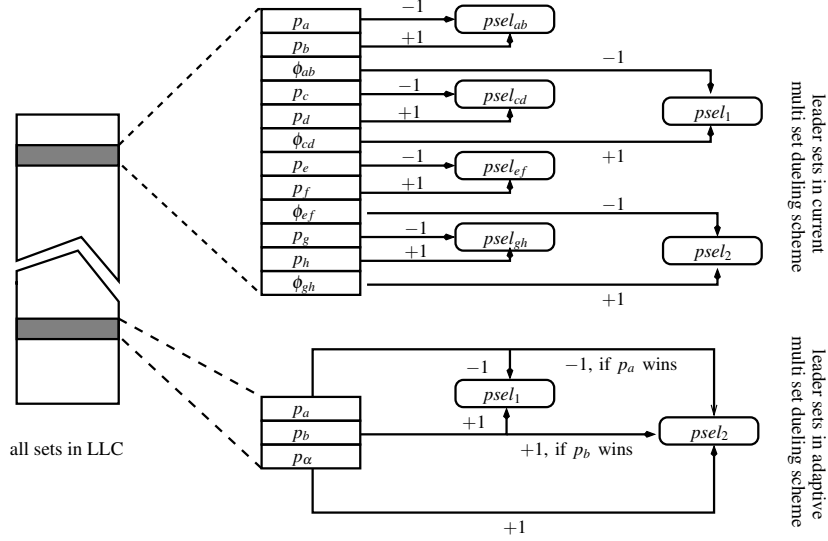leader sets in adaptive multi set dueling scheme

Fig. 4.   Reduction in Leader sets with adaptive policy

D. Insertion Policy Selection

According to previous work [2] we should have five leader sets for five insertion positions and two partial follower sets for 1st round winner. Instead we use only three leader sets. The first round duel is between the MRU position and middle position. Counter $psel1$ determines the winner in this round. If MRU position is the winner, the last leader set inserts in the near MRU position. The counter $psel2$ is responsible for the second level winner. But if middle position was the winner in the first round, last leader set inserts in the LRU position.

Thus, the second level duel takes place between middle position and LRU position. If middle position is still the winner, the last leader set starts inserting in near LRU position. We use a one bit counter s to keep track of the policy used in this set so that follower sets know which policy to use. Figure 5 shows how follower sets decide which policy is winning.

E. Storage Requirement

We have four kind of sets in our scheme; leader set inserting at MRU position and middle position, adaptive leader set and follower set. This requires extra 2 bits per
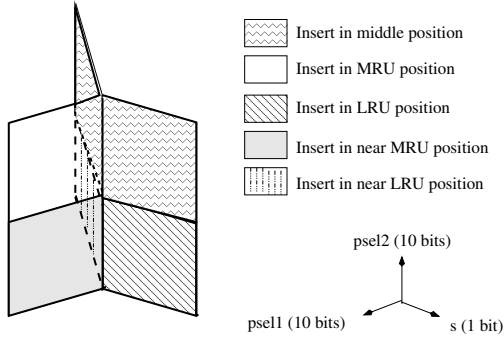
3

Fig. 5. Selecting insertion policy

TABLE I
STORAGE FOR 1MB 16 WAY CACHE

Parameter	Storage
set type	2 bits * 1024 sets
two counters (psel)	20 bits
one counter (s)	1 bit
Total	2069 bits

set. Then we need two counters (*psel*1 and *psel*2) and one extra bit for *s* to keep track of policies in adaptive leader set. Table I shows the space requirement for a 1*MB* 16-way last level cache.

III. SIMULATION METHODOLOGY

We have simulated our scheme with CMP$im [4]. It simulates a simple multiple issue out-of-order core. Table II shows the configuration of the simulated machine. We have used SPEC CPU 2006 benchmarks. We have chosen 14 memory intensive benchmarks that achieve more than 1% instructions-per-cycle (IPC)

TABLE II
CONFIGURATION

Parameter	Configuration
Issue width	4
Reorder Buffer	128 entry
Branch Prediction	perfect
L1 I-Cache	32KB, 4 way LRU,
	64B blocks, 1 cycle hit latency
L1 D-Cache	32KB, 8 way LRU,
	64B blocks, 1 cycle hit latency
L2 Cache	256 MB, 8 way LRU,
	64B blocks, 10 cycle hit latency
L3 Cache	1 MB, 16 way,
	64B blocks, 30 cycle hit latency
Main Memory	200 cycle

TABLE III
INSERTION POSITION SELECTED

Benchmark	Position	Benchmark	Position
400.perlbench	middle	450.soplex	middle
401.bzip2	middle	456.hmmer	nearLRU
429.mcf	nearLRU	464.h264ref	MRU
434.zeusmp	MRU	471.omnetpp	middle
435.gromacs	middle	473.astar	MRU
436.cactusADM	middle	482.sphinx3	LRU
445.gobmk	nearLRU	483.xalancbmk	LRU

speedup when the LLC is increased from 1MB to 2MB. We have fast forwarded the benchmarks for 40 billion instructions and then run the simulation for 100 million instructions.

IV. RESULT

In this section we show the results of our scheme. Table III shows the insertion position selected by our decision tree analysis for each of the benchmark. Fig
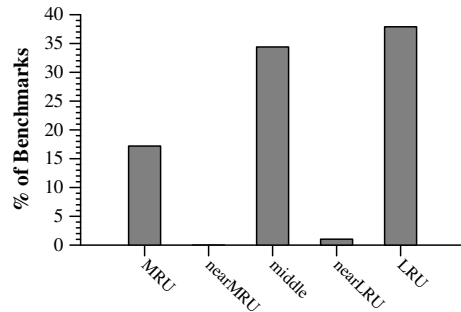

Fig. 6. Percentage of benchmarks at each insertion position

6 shows the percentage of benchmarks choosing each insertion position. Among 29 SPEC CPU2006 benchmarks, DTA chooses MRU insertion position in 5 benchmarks, middle position in 10 benchmarks, near LRU position in 3 benchmarks and LRU position in 11 benchmarks.

Fig 7 shows the percentage reduction in misses-per-1000-instructions (MPKI) of our scheme and DIP over LRU replacement policy. On average our policy reduces MPKI by 5.16% over LRU. On average DIP reduces MPKI by 1.8%. Harmonic mean IPC for DTA is 0.850399 which is 7.19% IPC improvement over LRU and 4.12% IPC improvement over DIP. Fig 8 shows the speedup of the memory intensive benchmarks. The geometric mean speedup of our scheme over LRU is 3.57% where DIP achieves only 1.3% speedup over LRU. DTA performs poorly in 450.soplex, 471.omentpp and 473.astar. The reason is the workloads do not show uniform access across the sets. 16 dedicated sets are
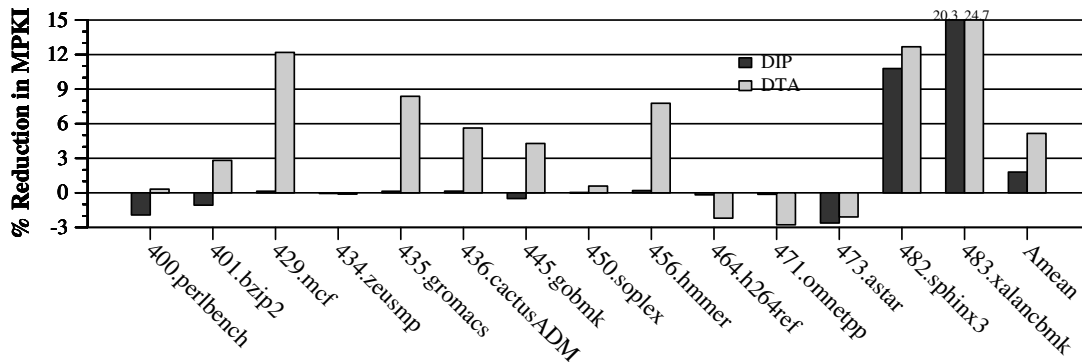
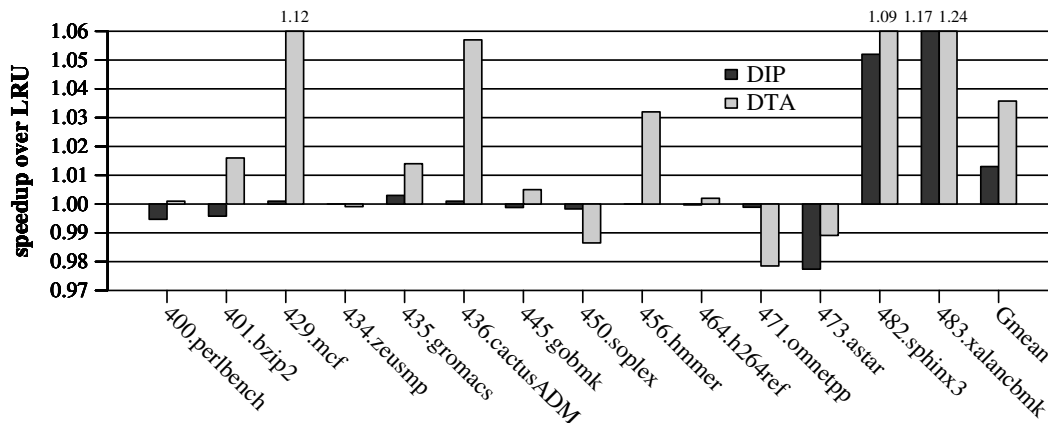Fig. 7. MPKI reduction compared to LRU

Fig. 8. Speedup over LRU

not enough for these benchmarks. For example if we increase the number of dedicated sets from 16 to 32, DTA chooses the MRU position for 450.soplex and its performance becomes similar to baseline.

It should be noted that our scheme can detect phase changes in the workloads. It has two static leader sets that always use the same specific insertion positions. Only the adaptive leader set chooses the insertion position dynamically. The static sets detect the phase change and adaptive set chooses the insertion position accordingly. Our benchmarks ran only 100 million instructions and do not experience any phase change.

Fig 9 shows speedup of our scheme over LRU for all SPEC CPU 2006 benchmarks. It achieves 1.7% IPC improvement over the baseline. We can see that DTA does not significantly slow down any of the non memory intensive benchmarks.

V. COMPARISON WITH A DEAD BLOCK PREDICTOR REPLACEMENT

In this section we compare our result with Counting Based Dead Block Replacement (CDBR) [5]. A dead block predictor can accurately identify zero reuse lines and replace them instead of the LRU block. However,

such a predictor requires a significant hardware budget. The counting based predictor needs to keep track of program counter (PC), access count, past access count and the confidence of the prediction for each cache line [5]. It also uses a 256×256 entry predictor table where each entry stores the number of access and the confidence. For a 1MB cache it uses 74KB extra storage where our scheme needs only $2,069$ bits. That is, our technique requires far less than 1% of the storage of a dead block predictor.

Figure 10 shows the speedup of DTA over counting based dead block replacement. On average we achieve 1.45% speedup over CDBR. The reason our scheme performs better on average is, although the dead block predictor learns zero reuse lines, these lines do not come back to the cache frequently. So even if the dead block predictor cannot identify those lines, by contrast, our scheme inserts them in near LRU position and gets rid of them quickly.

VI. RELATED WORK

Dynamic Insertion Policy (DIP) was proposed in by Qureshi *et al.* [1]. This work also proposed set-dueling. An adaptive insertion policy has also been proposed
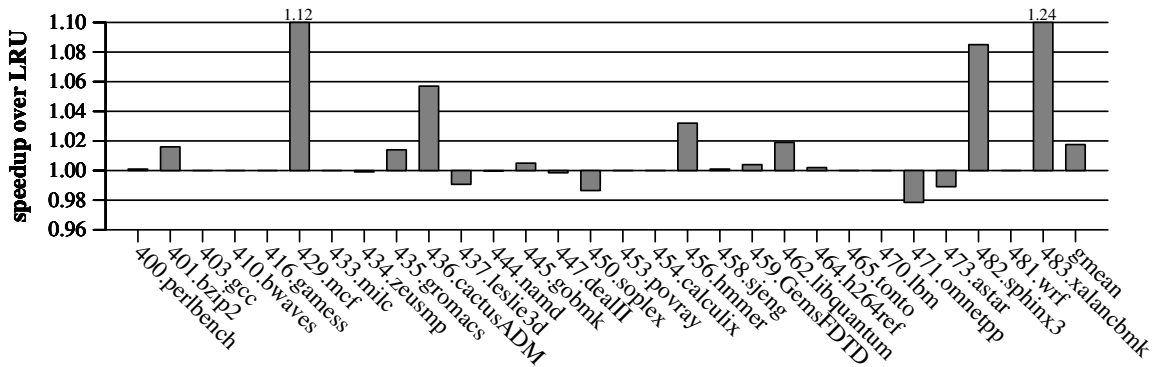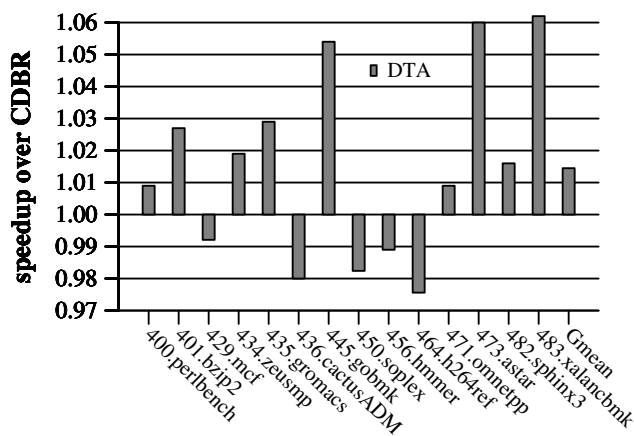
Fig. 9. Speedup over LRU replacement policy

Fig. 10. Speedup of over Counting Based Dead Replacement

REFERENCES

[1] M. K.Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr, , and J. Emer, "Adaptive insertion policies for high-performance caching," in *In the International Symposium on Computer Architecture (ISCA)*, June 2007.

[2] G. H. Loh, "Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy," in *International Symposium on Microarchitecture (MICRO)*, December 2009.

[3] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. S. Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *In the International Conference on Parallel Architectures and Compiler Techniques (PACT)*, October 2008.

[4] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A pin-based on-the-fly multi-core cache simulator," in *Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, Beijing, China, June 2008, pp. 28–36.

[5] M. Kharbutli and Y. Solihin, "Counter-based cache replacement algorithms," in *International Conference on Computer Design*, San Jose, USA, October 2005, pp. 61–68.

[6] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *International Symposium on Computer Architecture (ISCA)*, June 2009.

[7] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, November 2008.

[8] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction and dead-block correlating prefetchers," in *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, June 2001.

[9] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *International Conference on Computer Design*, 2007, pp. 245–250.

[10] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proceedings of the 27th annual international symposium on Computer architecture*, 2000, pp. 107–116.

[11] M. Chaudhuri, "Pseudo-lifo: the foundation of a new family of replacement policies for last-level caches," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2009, pp. 401–412.

for multi-threaded workloads [3]. Depending on the characteristic of the workloads, one thread may insert at the LRU position while some other thread may insert in the MRU position of the shared cache. Multi-set-dueling and different insertion positions for multithreaded workloads has been proposed by [6], [2]. Other replacement policies include dead block predictors [5], [7], [8]. Some works use reuse count for improving replacement policy [9], [10]. Mainak [11] proposes to manage cache set as a fill stack as opposed to the traditional access recency stack.

VII. CONCLUSION

The selection of insertion policy with decision tree analysis of multi-set dueling is a simple efficient technique that can be implemented in hardware with minimal change and minimal additional hardware cost. Nevertheless, this technique captures the distinct behavior of last level cache. Our scalable multi-set dueling ensures that we can use only a few leader sets but still can choose the best policy from a pool of options.