# Insertion Policy Selection Using Decision Tree Analysis

Samira Khan, Daniel A. Jimenez

# Insertion Policy Selection Using Decision Tree Analysis

Samira Khan, Daniel A. Jiménez
Department of Computer Science
The University of Texas at San Antonio
{skhan, dj}@cs.utsa.edu

## Abstract

The last-level cache (LLC) mitigates the impact of long memory access latencies in today's microarchitectures. The insertion policy in the LLC can have a significant impact on cache efficiency. However, 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 entry uses only 2069 bits over the LRU replacement policy.

## 1 Introduction

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.However, this 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.

## 2 Insertion Policy Selection Using Decision Tree Analysis

### 2.1 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 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

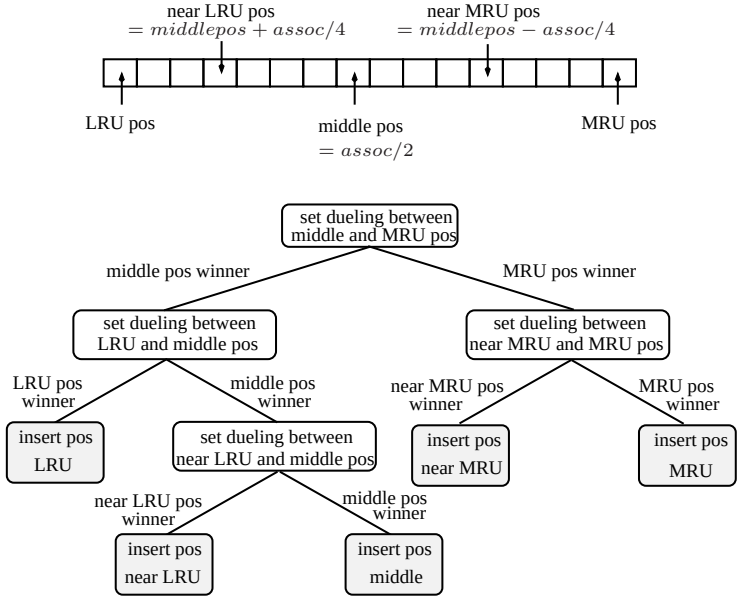


Figure 2: Decision Tree Analysis

a miss. We propose to use decision tree analysis to determine this optimal insertion position. This analysis is based on multiple set dueling [3]. 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.

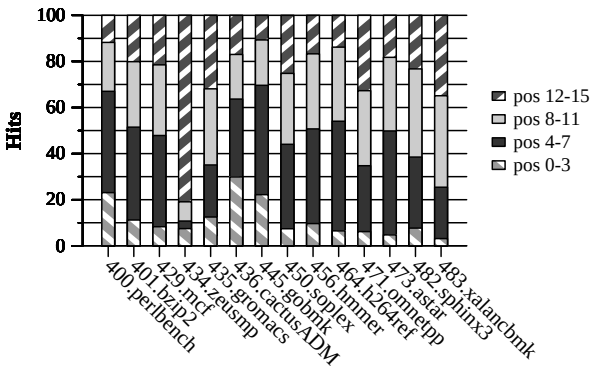


Figure 1: Hit Position (0 is MRU and 15 is LRU)

## 2.2 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 2 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 2.

## 2.3 Insertion Policy Selection

Multi-set-dueling was proposed for multi-threaded workloads [2]. 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 [3]. 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 ($\phi$ sets). The second level winner is then determined from the duel of these $\phi$ 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

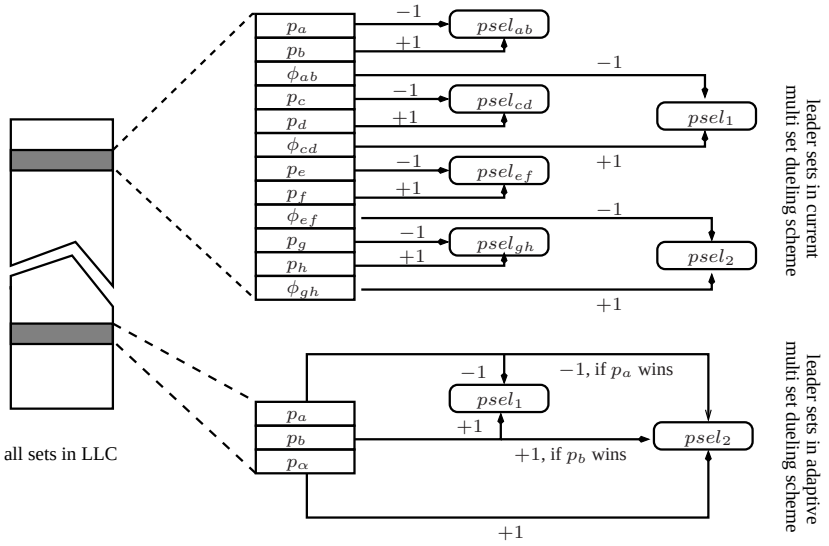


Figure 3: Reduction in Leader sets with adaptive policy

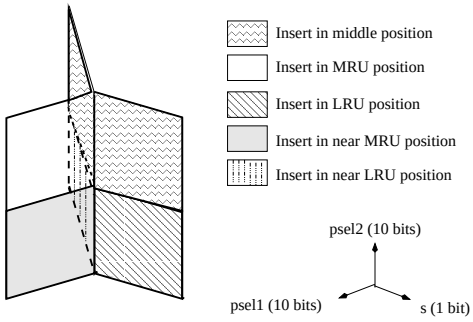


Figure 4: Selecting insertion policy

| Parameter | Storage |
|---|---|
| set type per set | 2 bits |
| two counters (psel) | 20 bits |
| one counter (s) | 1 bit |
| Total | 2069 bits |

Table 1: Extra storage for 1MB 16 way cache

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. But the problems of this scheme is solved by using leader set that can dynamically select specific insertion policy. We also remove the partial follower sets. Figure 3 shows the difference in two schemes. The first group of leader set is defined according to previous work [3]. 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 $\phi_{ab}$, $\phi_{cd}$, $\phi_{ef}$ and $\phi_{gh}$. 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 $p_\alpha$. 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.

According to previous work [3] 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. So 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 4 shows how follower sets decide which policy is winning.

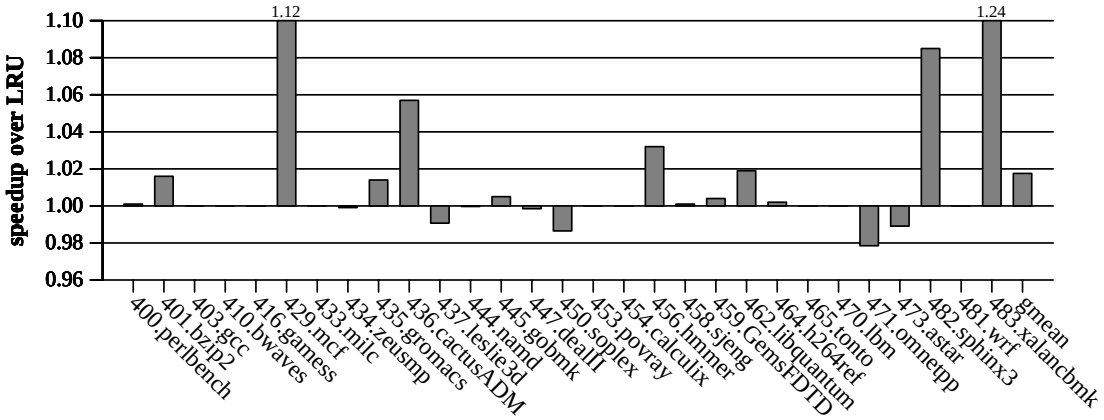


Figure 5: Speedup over LRU replacement policy

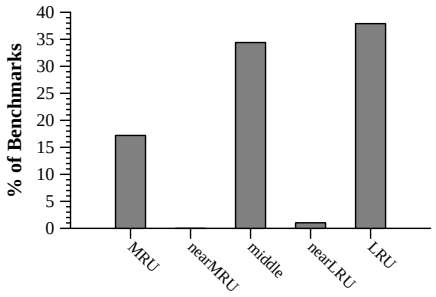


Figure 6: Benchmarks at each insertion position

### 2.4 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 set. Then we need two counters ($psel1$ and $psel2$) and one extra bit for $s$ to keep track of policies in adaptive leader set. Table 2.3 shows the space requirement for a $1MB$ 16-way last level cache.

## 3 Result

Figure 2.4 shows the speedup of our policy over baseline LRU. It achieves 1.7% IPC improvement over the baseline. Fig 6 shows the percentage of benchmarks choosing each insertion position when using our insertion policy selection through decision tree analysis.

## 4 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 for multi-threaded workloads [2]. 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 [4, 3].

## 5 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.

## References

[1] Moinuddin K.Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr and and Joel Emer. Adaptive Insertion Policies for High-Performance Caching. In the International Symposium on Computer Architecture (ISCA), 2007

[2] Aamer Jaleel, William Hasenplaugh, Moinuddin K. Qureshi, Julien Sebot, Simon Stelly Jr. and Joel Emer. Adaptive Insertion Policies for Managing Shared Caches. In the International Conference on Parallel Architectures and Compiler Techniques (PACT) 2008

[3] Gabriel H. Loh. Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy. In the International Symposium on Microarchitecture (MICRO), 2009

[4] Yuejian Xie, Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In the International Symposium on Computer Architecture (ISCA), 2009