MRU-Tour: A Concept to be Applied in Last-Level Cache Replacement

Alejandro Valero*,1, Julio Sahuquillo*,1, Salvador Petit*,1, Pedro López*,1, José Duato*,1

* Department of Computer Engineering (DISCA), Universitat Politècnica de València, Camí de Vera s/n, 46022, Valencia, Spain.

ABSTRACT

This paper presents the concept of MRU-Tour of a block as the number of times that a block occupies the MRU position during its live time. Based on the fact that most of the blocks experience only a single MRU-Tour, this work proposes the MRU-Tour-based algorithm aimed at exploiting this block behavior to improve performance. Experimental results show that, on average, the MPKI reduction is up to 19% with respect to LRU, whereas hardware complexity is also reduced.

KEYWORDS: Last-level cache; MRU-Tour; replacement algorithm.

1 Introduction

Last-level caches implement the LRU algorithm to exploit temporal locality, but its performance is quite far from Belady’s optimal algorithm [Bel66] as the number of ways increases. One of the main reasons because of LRU does not reach good performance in last-level caches is that this policy forces a block to descend until the bottom of the stack before eviction. Nevertheless, most of the blocks that leave the MRU position are not referenced again before eviction. On the other hand, the implementation of a strict LRU algorithm is expensive in terms of area and power.

This work pursues to select candidate blocks to be victimized before reaching the bottom of the stack. To this end, this work defines the number of MRU-Tours (MRUTs) of a block as the number of times that a block enters in the MRU position during its live time. Based on the fact that most of the blocks exhibit a single MRUT, this work presents the MRUT-based algorithm aimed at exploiting this block behavior to improve performance. Variants of this policy exploiting both MRUT behavior and recency of information will be also proposed.

1E-mail: alvabre@gap.upv.es, jsahuqu, spetit, plopez, jduato}@disca.upv.es.
Experimental results show that, on average, the baseline MRUT algorithm reduces the MPKI by 15% compared to LRU. This percentage is up to 19% when considering recency of information just for a few blocks. In addition, complexity is largely reduced with respect to the LRU policy which keeps the stack order of all the blocks.

## 2 Motivation and Proposed Approach

To explore the potential benefits of the MRUT concept on the replacement policy, we quantified the number of MRUTs and split the accessed blocks depending on whether they exhibit a single or multiple MRUTs at the time they were evicted. Figure 1 depicts the results for the SPEC2000 benchmark suite and a 128B/line 1MB-16way L2 cache under the LRU policy. It can be seen that blocks having a single MRUT dominate those having multiple MRUTs, especially in those applications having a high number of replacements.

![Figure 1: Number of replacements split into single and multiple MRUTs.](image)

Instead of tracking the total number of MRU-Tours, the devised algorithms only need to be aware if each line has had one or multiple MRUTs, so reducing the hardware complexity. Figure 2 shows the baseline MRUT algorithm. This policy only requires two status bits per cache line (MRU-bit and MRUT-bit). The first one is used to identify the MRU block and the latter to know if the block has had a single or multiple MRUTs. The policy works as follows. When a block is fetched into the cache, its associated MRUT-bit is reset to indicate that its first MRUT has started. When the block leaves the MRU position for the first time, it can potentially be replaced. Then, if the same block is referenced again, it returns to the MRU position and the MRUT-bit is set to one to indicate that the block has had multiple MRUTs. The algorithm aims to avoid that blocks with a single MRUT stay in cache. To make hardware simple, it randomly selects the victim among those blocks with a single MRUT except the MRU block. If there is no block with its MRUT-bit cleared, all the blocks (except the MRU one) are candidates for eviction and the victim is randomly selected. Please refer to [VSP+11] for further details.

Variants of the baseline MRUT policy maintaining recency of information for a few blocks have been also studied. In this way, the MRUT-x family of algorithms store the order of the last x referenced blocks, and will not be considered as candidates for eviction. Notice that
Algorithm: Baseline MRUT

Cache hit in block x:
   if (x is not in the MRU position)
      set the MRU-bit of x to 1

Cache miss:
   a) select the block to be replaced
      if (there are candidates with MRU-bit=0)
         randomly among candidates (except the MRU)
      else
         randomly among all blocks (except the MRU)
   b) set the MRU-bit of the incoming block to 0

End algorithm

Figure 2: Baseline MRUT algorithm implementation.

complexity is largely reduced with respect to the LRU algorithm which keeps all the blocks arranged in a stack.

3 Experimental Results

The algorithms studied in this work have been modeled on an extended version of the SimpleScalar framework simulator. The memory hierarchy consists of a 64B/line 16KB-2-way L1 instruction cache, a 64B/line 16KB-2way L1 data cache, and a 128B/line 1MB-16way L2 unified cache. Experiments have been performed for the Alpha ISA and running the SPEC2000 benchmarks with the ref input set. We skipped 1B instructions before collecting statistics, and then simulate 500M instructions. Applications having an MPKI less than one or a percentage of compulsory misses higher than 75% have been skipped for this study.

Figure 3 plots the MPKI results of LRU, a recently proposed algorithm referred to as DC-Bubble [ZX09], and MRUT-x policies varying x from 1 to 4. It can be seen that, on average, the baseline MRUT algorithm (MRUT-1) reduces MPKI by 4% and 15% compared to both DC-Bubble and LRU algorithms, respectively. MRUT-1 performs much better than DC-Bubble and LRU in applications showing a high MPKI, whereas it performs a bit worse in applications showing a low MPKI. This is due to the MRUT-1 policy selects the victim block regardless the recency of information. On the other hand, MRUT-3 provides the best results on average, meaning that, in general, the stack order beyond three positions is not important. This policy achieves an MPKI reduction up to 4%, 8%, and 19% compared to MRUT-1, DC-Bubble and LRU, respectively.

4 Conclusions

The LRU algorithm does not perform well in last-level caches with a high number of ways. One of the main reasons is that LRU forces a block to descend until the bottom of the stack before eviction. Nevertheless, most of the blocks that leave the MRU position are not accessed again. Based on this observation, in this work has been defined the number of MRU-Tours of a block as the number of times that the block enters in the MRU location during its
live time, and the MRU-Tour-based algorithm has been proposed to improve performance. Experimental results showed that, on average, the simplest algorithm of the family reduces the MPKI up to 15% with respect to LRU. In addition, maintaining the order of just the last three accessed blocks and preventing them from being replaced enhances the performance of the baseline MRU-Tour algorithm. In this case, the MPKI is reduced, on average, by 19% with respect to LRU. Finally, algorithm complexity is largely reduced since the baseline MRU-Tour policy just requires a pair of status bits per cache block regardless the number of ways.

Acknowledgments

This work was supported by the Spanish MICINN, Consolider Programme and Plan E funds, as well as European Commission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04-01.

References

