Parallel Computing 49 (2015) 13–27

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

# Improving last level shared cache performance through mobile insertion policies (MIP)

Pablo Abad*, Pablo Prieto, Valentin Puente, Jose-Angel Gregorio

*Computer Architecture Group, University of Cantabria, Santander 39005, Spain*

ARTICLE INFO

*Article history:*
Received 27 January 2015
Revised 28 April 2015
Accepted 8 June 2015
Available online 14 June 2015

*Keywords:*
Multiprocessor
Cache hierarchy
Last level cache
Replacement policy

ABSTRACT

For those cache hierarchy levels where program locality is not as evident as in L1, LRU replacement does not seem to be the optimal solution to determine which blocks will be requested soon. The literature is prolific on alternative reuse-distance estimations at last on-chip cache level, proving the difficulty of achieving an optimal hit rate. One of the key aspects for performance is knowing inter and intra application reuse-distance variability. Many solutions already do this, but most of them rely on a simple choice among a few alternative policies. The experiments performed to motivate the proposal confirm application variability, but also show that the behavior of applications is much more than bimodal. This means that there is a performance gap that current hybrid policies are not able to cover. In this paper we propose a mobile insertion position replacement policy (MIP), which combines well known LRU ordering and promotion policies with a completely adaptive insertion mechanism. The dynamic behavior of insertion is able to capture hit-rate variability in a more accurate way. Making use of set dueling and dynamic set sampling for prediction, our mechanism continuously estimates the insertion position that maximizes the cache hit rate. The hardware overhead compared to a LRU replacement algorithm is merely three 3-bit saturating counters per LLC bank. Our experiments show that for a wide range of applications, MIP is able to improve the hit rate of LRU by 30% on average. MIP outperforms current state-of-the-art replacement policies with a similar implementation cost by 10% on average and in single-thread or multi-thread workloads by 20%.



## 1. Introduction

The performance of current processor architectures is constrained by the memory hierarchy. Originally, Memory Wall [1] was an exclusively latency problem (a processor can only compute as fast as operands become available), caused by the speed divergence of processor and memory. Technology evolution has allowed a much higher transistor budget, making multiprocessors on chip a reality nowadays and turning Memory Wall into a bandwidth problem too [2]. Additionally, future technology trends (higher levels of integration and/or non-volatile memories) could extend the Memory Wall problem to even more aspects, such as power consumption or endurance. All these issues make on-chip cache memory a critical component in maintaining the same performance improvement quotas. Cache hierarchy is able to soften many of the Memory Wall issues, accelerating processor access to data and filtering off-chip requests. The relevance of this part of the system becomes much more evident if we

* Corresponding author. Tel.: +34 942206769; fax: +34 942206771.
*E-mail address:* abadp@unican.es, pablo@atc.unican.es (P. Abad).

http://dx.doi.org/10.1016/j.parco.2015.06.002
0167-8191/

observe its evolution from the first microprocessors with simple one-level on-chip caching to the current sophisticated multi-level hierarchies.

The ultimate objective of the cache hierarchy is to keep the data that will be needed in a near future close to the processor. Unfortunately, due to the incompatibility between speed and capacity, in most cases application working sets are larger than storage at close-to-processor cache levels. The replacement algorithm implements the arbitration process required to decide which memory blocks are evicted when new data arrives at a full cache. Ideally, to ensure the highest performance cache should retain those blocks that are likely to be accessed within a short interval. Defining block *reuse distance* as the time interval between two consecutive accesses to its content, the optimal choice for replacement is to evict the block with largest reuse distance (that will be accessed further in the future), as stated in Belady's algorithm [3]. Optimal, but not realizable in real time, this algorithm is only useful to measure the effectiveness of other proposals.

The unattainable objective of any replacement policy is to mimic Belady's algorithm, making use of available information. In general, temporal locality inherent to most applications is the property exploited to anticipate future reuse distance with current information. Blocks accessed recently are more likely to be accessed again in a near future. Temporal distribution of block references determines both *recency* and *reuse frequency*. Recency is defined as the time since the last reference to a block, while reuse frequency is calculated as the number of consecutive references in a time interval between consecutive references. These two parameters are employed by most algorithms to estimate reuse distance. Most replacement algorithms are implemented as a priority queue where elements are inserted, ordered and evicted according to this estimation. This is the usual structure for a replacement algorithm:

- **Priority queue**: In a set-associative cache, every block is assigned a priority value, which defines its position in the queue. Usually, the block with lower priority value is evicted.
- **Insertion policy**: Determines the initial priority value assigned to a block when it enters in the priority queue for the first time.
- **Promotion policy**: On a cache hit, the priority of the block is modified, usually promoting it to positions with higher priority.

Basic replacement algorithms are defined fixing each of these three components. The most common algorithm, widely employed for L1 caches, is LRU (or approximations). This static approach is valid when data locality of applications is high and stable, reuse distance being efficiently estimated by the mix of recency (insertion) and frequency (promotion) provided by the LRU algorithm. However, those hierarchy levels where locality is more volatile present a much more variable behavior in terms of reuse distance [4]. To further increase the complexity of the problem, usually these levels are shared by different threads, often with interfering behavior. There is an ample diversity of options trying to predict the reuse distance in such complex scenarios. Simple static policies balancing recency and frequency [5], hybrid policies that conjugate multiple basic ones [6] or reuse-distance prediction [7] are a few examples of the large number of proposals in this sense.

In this paper we propose yet another approach, which to the best of our knowledge has not been explored before. We implement an adaptive insertion policy, extending the insertion of new blocks to any position of the priority queue. With the help of a few dedicated cache sets we explore how hit rate evolves as insertion position is moved away from the highest priority[1] position of a classic LRU policy to the opposite end of the priority queue. Applied at last level cache, where application behavior is highly dissimilar, MIP (mobile insertion policy) is able to dynamically adjust recency weight to a value providing higher hit rate. The hardware implementation of the MIP algorithm has minimal overhead and complexity. Three 3-bit counters per LLC bank are the only additional storage required compared to LRU as well as simple operations outside the critical path to calculate optimal insertion position. We compare our proposal with classic LRU and a state-of-the-art algorithm with similar overhead and complexity: Dynamic re-reference interval prediction policy [4]. Our evaluation shows that MIP outperforms LRU and DRRIP for a wide variety of workloads, both multi-programmed and multithreaded. Summarizing, these are the main contributions of this work:

- We show that there is a high variability in access patterns, which is hardly covered by static replacement policies. We also show that when different applications interact, even current dynamic policies have not enough flexibility.
- We present a low-overhead, practical implementation of a fully dynamic replacement policy based on mobile insertion position, which rapidly adapts to sudden changes in access patterns during application runtime.
- We compare MIP with two alternative proposals using a wide range of workloads and show that MIP provides 30% more replacement policy performance than LRU and 10% compared to DRRIP.
- We show that, in contrast with DDRIP, cache hit-rate is improved 20% when the workload is a multithreaded or single-thread application.

The rest of the paper is organized as follows. Section 2 includes a detailed discussion of the experiments used to demonstrate the motivation for our work. Section 3 provides a brief description of related work, focusing only on proposals with a similar target to ours. Section 4 describes in detail how the mobile insertion policy works, as well as some design alternatives. Section 5 summarizes the main simulation framework aspects and the applications employed for evaluation. Section 6 presents an extensive set of results and finally Section 7 highlights the main conclusions of the work.

[1] It is important to distinguish between replacement algorithm and insertion policy. In LRU replacement algorithm, insertion policy is MRU and vice versa.

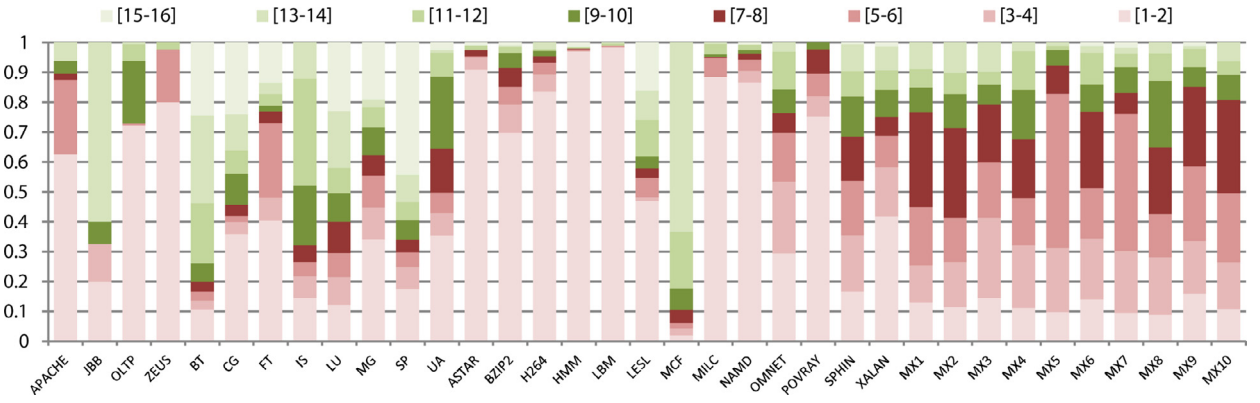


**Fig. 1.** Distribution of optimal insertion point throughout application runtime.

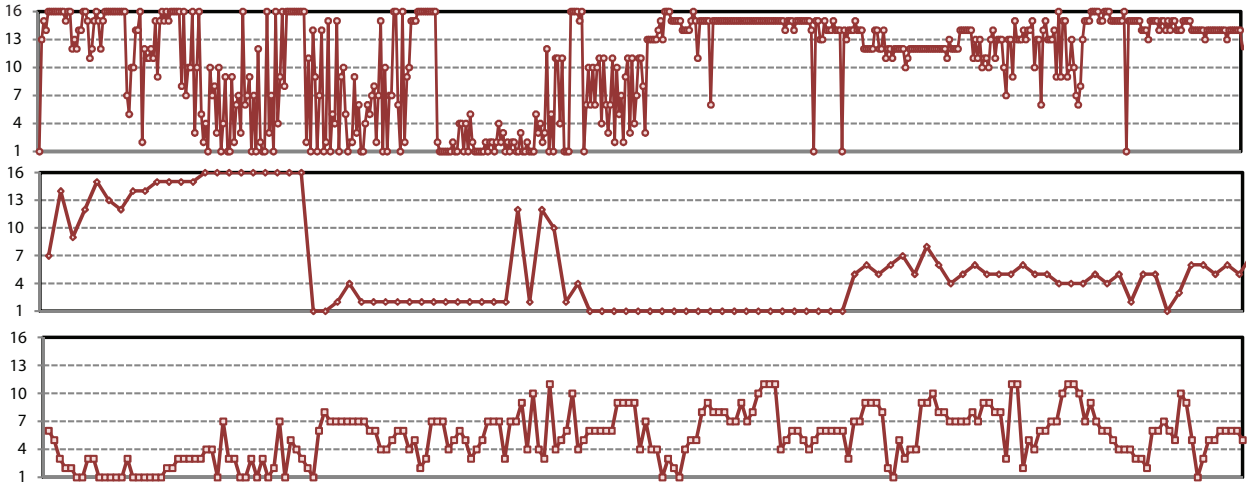


**Fig. 2.** Optimal insertion position throughout execution time for: CG (above), FT (middle) and MIX17 (below).

## 2. Motivation

In order to estimate the hit-rate variability according to the insertion point selected for the replacement policy, we conducted an extensive experiment analyzing the near-optimal insertion position at runtime in a 4-core CMP (see Section 5 for the configuration details). Last level cache (LLC) sets are divided into 16 different types (the same as cache ways). Each set type implements a different insertion policy, ranging from highest to lowest priority positions in priority queue depending on the four least significant bits of the address itself. Sets with the 4 least significant bits at 0 ($lsb_4 = 0 \times 0$) insert new blocks with the highest priority, $lsb_4 = 0 \times 1$ in the second position and so on until sets with $lsb_4 = 0 \times F$, which insert new blocks with the lowest priority. Both promotion and priority queue have the same configuration for every set type, equivalent to the one used in LRU replacement policy. With a frequency of 100,000 Processor clock cycles, the hit rate of each set type is calculated, and the insertion position providing the highest value is profiled as optimal. The set of applications employed for the experiment is described in detail in Section 5 and Figs. 1 and 2 show the results obtained.

In Fig. 1, the *Y* axis represents the fraction of runtime when the maximum hit rate has been found at each insertion position. Insertion positions have been grouped to augment graph legibility. Label {1–2} corresponds to sets inserting in positions with the highest priority (MRU positions correspond to the classic LRU replacement policy) while label {15–16} corresponds to sets with the lowest priority at insertion. The results obtained in this preliminary experiment show that static replacement policies are not able to capture the real behavior of applications at last level cache. Optimal insertion position is far from static, finding dissimilar results between applications. Additionally, it should be noted that for a wide range of applications, maximum hit rates concentrate on intermediate insertion positions, which could reduce the effectiveness of some hybrid proposals that dynamically change between scan resistant (MRU) and trash resistant (LRU) replacement policies.

Fig. 2 provides more detailed insight about how optimal insertion position evolves. In these graphs we represent this value throughout execution time for three representative applications of the common behavior. Phases with different access patterns can be clearly identified for multithreaded applications (CG and FT). Phases with high reuse are identified when optimal insertion

position is close to 1, while when thrashing prevails the optimal insertion point is 16. As can be appreciated, best insertion point at priority queue borders is unlikely. When workload is generated with multiple independent applications, such as MIX17 (see Section 5), the presence of "pure" access patterns becomes highly improbable. Even in multithreaded applications such simple access patterns are not frequently found in LLC because different data types present different behaviors (instructions vs data, load vs store, shared blocks vs private blocks, etc.). Therefore, we might expect that the optimal insertion point is neither at the LRU nor at the MRU positions.

The previous observations indicate that to extract the maximum performance benefit from replacement, the selected policy should dynamically adapt to access patterns. The presence of steady reuse patterns over time seems not to be the common case. Therefore, dynamic replacement policies that simply choose between schemes designed for specific patterns might impair some performance. Figs. 1 and 2 suggest that there is room for further improvement with more flexible cache management policies. For example, if somehow we are able to estimate optimal insertion position during runtime, LLC hit rate might be improved. The main challenge is to predict future behavior with the available information and a constrained hardware overhead.

## 3. Related work

The amount of work in this area is profuse. This section only provides a brief survey of basic bibliography on replacement policies and focuses on state-of-the-art proposals directly related to ours (i.e., with similar cost constraints). Aspects such as dead block detection [8,9], block bypassing [10] prefetching interaction [11] or optimizations targeting shared caches [12,13] can be considered as orthogonal to this work.

Belady's algorithm [3] represents a clear reference of optimal behavior for replacement policies. For this reason, the final objective of any proposal is to replicate this policy with realistic information (i.e., no need for knowledge about the future). Exhibited past program locality determines cache access pattern, being the key element in predicting the reuse distance of a cache block in the future. LRU replacement policies [14,5] exploit this, trying to keep the most recently accessed block in the set. Therefore, the first access to the block at insertion time is the most recently accessed block. Consequently, at insertion, a cache block should be placed in the MRU position. Basic LRU replacement policies work well when perceived locality is significant, which makes them the most suitable choice for cache levels close to the processor.

Some applications present special access patterns which are not recency-friendly. Some scanning patterns where the running application has low locality benefit more from MRU replacement policies (and indirectly LRU insertion point), which tend to maintain the oldest blocks in the cache. To deal with this dual behavior of applications, the authors of [15] propose a hybrid replacement policy able to dynamically change between LRU and MRU insertion policies.

In the absence of recency, reuse frequency is a good metric to infer future behavior of a block. Frequently reused blocks are more likely to be accessed sooner than rarely reused ones. LFU algorithms only take into account frequency for their decisions, making them more suitable than LRU in the absence of locality. However, L1 cache is not always able to completely filter temporal locality, which limits the potential benefits of replacement algorithms based only on frequency [5]. Many authors propose hybrid solutions that combine both LRU and LFU policies in different ways [5,14,16–18]. Work in [14] employs one bit to identify blocks with temporal locality (reused) and non-reused blocks are selected first for eviction. LRFU policy [5] implements two priority queues to mix LRU and LFU with different weights. SRRIP [17] implements a replacement algorithm which makes use of reuse frequency for decisions, but still takes into account recency considering that new blocks have a long reuse interval but not the longest. Additionally, works like SBAR [19] or AC [6] propose alternative combinations to the conventional LRU/LFU.

When recency and frequency cannot be captured properly, many authors propose trying to directly estimate reuse distance through alternative mechanisms [20–22]. The work in [20] estimates distance according to the PC that loaded that block. The distance disparity between reads and writes is employed in [21] to dynamically divide cache capacity between loads and stores. The work in [22] assumes that conflict miss detection can also improve replacement policy results.

Some recent works have started exploring the benefits of considering multiple insertion points. In [23], the LRU stack is divided into two partitions, applying a different replacement policy on each side. Different partition sizes are dynamically analyzed through shadow tags. For each partition size, two insertion positions are evaluated, MRU and "partition border". Despite allowing multiple insertion points, this feature is only analyzed after partition size is chosen, meaning the final decision is still bimodal. In [24], the authors propose moving from the bimodal insertion policy of DIP to a multi-level decision tree able to choose between five different insertion positions. Despite relying also on the dynamic selection of the optimal insertion point, the differences with our proposal are clear. They assume a single-core, while our proposal is focused on multi-core chips, where the problem of mutually interfering workloads is much tougher. Their mechanism is not able to detect phase changes after 100 million instructions, which is not the case with workloads like the ones assumed in our work. Our proposal allows insertion at any position, not only a subset. As shown in Figs. 1 and 2, the optimal insertion point is sometimes different to the five positions proposed. Despite claiming their method is adaptive, results seem to show static behavior, which makes the proposal a bit confusing.

Many of the policies described in this section are orthogonal and could be implemented to work together. The disparity in implementation overhead also makes it difficult to make a fair comparison. Some of them drastically change the organization of the cache. For these reasons, we have decided to reduce our counterparts to only two, a well-known baseline mechanism and a state-of-the-art proposal with a similar structure and implementation overhead. The two counterparts selected for evaluation are LRU and DRRIP [17] policies. As LRU is the baseline counterpart of most proposed mechanisms (as well as RRIP), comparison of MIP with other proposals could be indirectly performed through these two counterparts (if similar methodology and workloads are employed).

## 4. Mobile insertion policy (MIP)

This section describes the mechanism that can determine the most suitable insertion position throughout application runtime under constrained cost conditions. Similarly to the set dueling concept [15], for this purpose we measure hit rate in a few dedicated sets, where different insertion positions are employed. The insertion position of the rest of sets is periodically updated according to these estimations. Hit rate estimation is performed through an $N$-bit saturating counter (form 0 to $MAX_{HR}$) which is updated on every access to the set. The counter is decremented or incremented according to the access result (hit or miss). After a fixed number of bank accesses (Update Interval), hit rates are evaluated to implement insertion policy.

The sets in each cache bank are divided into groups of size $N$ ($N$ being a power of 2). Each group is made up of two set types, evaluation and conventional. The insertion position of evaluation sets is selected with the aim of finding the position providing the optimal hit rate. Conventional sets select their insertion position according to the estimations performed by evaluation sets.

A straightforward way to perform hit rate estimation is to dedicate one evaluation set to each insertion position available. This means that in an M-way cache, $M$ evaluation sets would be required. The replacement policy of each evaluation set would employ a different insertion position, ranging from 1 to $M$ in the priority queue. Saturation counters of every evaluation set would periodically be compared and the insertion position of conventional sets would be chosen according to the values obtained. This is a way to find the near-optimal insertion position providing maximum hit rate, but it has a severe drawback: too many evaluation sets. The usually high associativity of the LLC could drastically reduce the number of conventional sets, artificially increasing cache misses despite making accurate replacement decisions. For this reason, we analyze a more cost-effective way of estimating the best insertion position, minimizing the number of evaluation sets in each group.

### 4.1. Optimal hit-rate exploration with two evaluation sets

We explore a particularization of similar strategies already employed for different purposes [25,26], reducing the number of evaluation sets per group in an $M$-way cache from $M$ to 2. One set will have an insertion policy at a fixed point, acting as a *reference* set and another set will move its insertion point looking for maximum hit rate, acting as an *explorer* set. Therefore, in each group we define three different kinds of sets:

- **Reference**: In these sets a reference policy, in this case classic LRU, is implemented. The insertion point is fixed over the whole execution time. In the case of a LRU policy, an insertion point for a new block always corresponds to the position with highest priority (MRU). The purpose of this estimation is to have a fixed baseline as a beacon. This fallback point guarantees at least LRU performance in most cases.
- **Explorer**: The role of the second set of the group is to start the exploration of different insertion alternatives to the reference one. Insertions in these sets are initialized at the MRU + 1 position and can move through all the positions in the priority list according to the hit rates obtained. The purpose of this estimation is to know how the cache might perform if we move the insertion point one click toward the LRU insertion point.
- **Conventional**: The other sets in the group are those where the insertion position is considered to be optimal for the next update interval. This position is selected according to its own hit rate and the values obtained by the reference and explorer sets. The initial insertion position for conventional sets is MRU (as in reference). Conventional sets always maintain the same insertion position relative to explorer sets.

Each set type has its own counter, named $HR_{REF}$, $HR_{EXP}$ and $HR_{CON}$, which are updated with each reference. At each *update interval*, these three values are compared and the maximum determines the insertion position of the conventional sets, $I_{pos}$, for the next interval. If the hit rate of conventional sets ($HR_{CON}$) is greater than or equal to $HR_{REF}$ and $HR_{EXP}$, the insertion positions are maintained (conventional sets have found the maximum). If $HR_{EXP}$ is greater than $HR_{REF}$ and $HR_{CON}$ conventional and explorer sets move away one position from the reference. Finally, when $HR_{REF}$ is the largest, both conventional and explorer sets move their insertion point one position closer to reference. It should be noted that conventional and explorer sets always have adjacent insertion positions. Formally, if MRU is position 1 and LRU is position $M$:

$$I_{\text{pos}} = \begin{cases} \text{I}_{\text{pos}} \text{ if } (\text{HR}_{\text{CON}} \geq \text{HR}_{\text{REF}} \,\&\, \text{HR}_{\text{CON}} \geq \text{HR}_{\text{EXP}}) \\ \text{I}_{\text{pos}}+1 \text{ if } (\text{HR}_{\text{EXP}} \geq \text{HR}_{\text{CON}} \,\&\text{HR}_{\text{EXP}} > \text{HR}_{\text{REF}}) \\ \text{I}_{\text{pos}} - 1 \qquad \text{otherwise} \end{cases} \tag{1}$$

Fig. 3 provides a simple example of the insertion policy operation for a 4-way cache. Each of the three colored columns represents a different kind of set (from the same cache bank), according to the previous classification presented in this section. The left column represents a reference set, with a replacement policy equivalent to LRU. The central and right columns represent conventional and explorer sets respectively. Insertion position of conventional sets is initialized to the value of the reference policy ($I_{pos} = 1$) and the insertion position of explorer sets is placed one position from conventional i.e. to $I_{pos} + 1$. Hit rate of each set is evaluated every four references. After each reference interval, the insertion position of both conventional and explorer sets is updated according to the hit rate values obtained in the previous interval and Eq. (1). For the sake of simplicity, we consider the same access pattern in each set, which represents a mix of highly reused and "dead" blocks with the following reference order: [*X1, X2, X3, X4, X5, X6, X7, X8, X1, X2, X2, X9, X10, X11, X1, X2*]. In Fig. 3 blocks are labeled according to their set (*X1* in explorer set is labeled as *E1*).

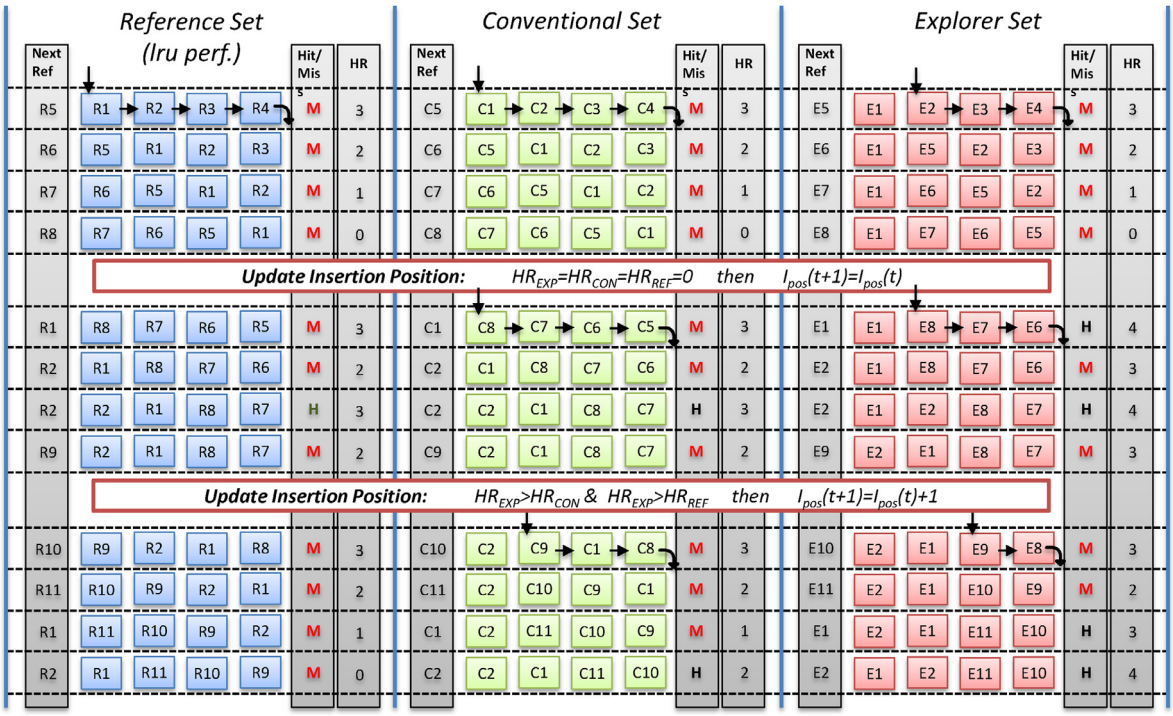


**Fig. 3.** Behavior of MIP (and LRU) for a mixed access pattern.

During the first reference interval, both the reference and conventional sets offer no protection for MRU blocks, and new insertions move R1 and R2 blocks towards eviction. In contrast, the explorer set evaluates hit rate behavior when LRU blocks are only slightly protected. During the first interval all cache accesses are references to new blocks, which return the same hit rate for every set, leaving the same insertion positions for the second interval. Thanks to MRU protection, the explorer set is able to obtain better hit rate during the second interval (E1 was not evicted after the insertion of four new blocks). For the next interval, the replacement algorithm decides to increase MRU protection in conventional sets, guided by explorer results. As a consequence, the explorer insertion position is also moved away from the reference one. Finally, the third interval shows that for this particular access pattern the modification of insertion position provides a better hit rate, both for the conventional and explorer sets. This means that insertion position will be modified again for the next interval (not shown).

### *4.2. Hardware overhead*

We assume exact LRU as baseline policy to build MIP. PseudoLRU implementations limit the flexibility of our proposal, because the number of different insertion positions is reduced. In a 16-way cache, total ordering requires more bits to be implemented than pseudoLRU or RRIP policies, but the performance benefits presented in Section 6 seem to clearly compensate for the additional storage required. While tree-based pseudoLRU requires a storage overhead of 15 bits per cache line, RRIP increases this overhead to 32 bits and our proposal uses 64 bits. As can be seen, compared with state-of-the-art cache line sizes (1 kB in our case), the storage overhead of MIP is 0.78%.

Aside from the hardware required to maintain the LRU priority queue, the only area overhead is devoted to implementing the three saturating counters per bank, which is several orders of magnitude below bank capacity. Hit rate updating makes use of a demultiplexor and the two least significant bits of the address to determine which counter should be updated. Finally, the arbitration process only requires the comparison of estimated hit rates and an adder to update position.

Note that in contrast to other (bimodal) dynamic insertion policies we use three-reference points and not just two. This simple extension provides a continuously variable insertion point. In this way we will try to capture and take advantage of the observations explained in Section 2 in order to maximize cache hit-rate.

## 5. Experimental methodology

We use GEMS [27] as the main tool for our evaluation, therefore full system activity will be modeled. We provide both hit-rate and final performance impact (execution time). Current processors, even in the embedded domain, are out-of-order with speculative execution. Therefore, the feedback between cache management policies and processor should be taken into account. Using traces or non-MLP systems might have a non-negligible effect on final results.

**Table 1**
Summary of system configuration.

| | | |
|---|---|---|
| Core arch. | Functional units | 4 × I-ALU/4 × FP-ALU/4 × D-MEM |
| | Instruction window size/issue width | 128, 4-way |
| | Frequency/processor count | 3 GHz, 4 |
| Private caches | (L1) Size/associativity/block size/access time | 32 kB I/32 kB D, 8-way, 64 B, 1 cycle |
| | (L2) Size/associativity/block size/access time/type | 256 kB unified, 8-way, 64 B, 4 cycles, exclusive with L1 |
| Shared L3 | Size/associativity/block size/type | 8 MB, 16-way, 64B, inclusive |
| | NUCA mapping | Static, interleaved by LSB |
| | Coherence prot., consistency mod. | MOESI directory (in-cache), TSO |
| | Data slice size/access time | 2 MB/6 cycles |
| Mem. | Capacity/access time/BW | 4 GB/350 cycles/32 GB/s |
| NoC | Topology/link latency/link width | 2 × 2 mesh/1 cycle/16 B |
| | Router latency/flow control/routing | 1 cycle/Wormhole/DOR |

**Table 2**
Single-core applications.

| | | | | |
|---|---|---|---|---|
| Spec | 401.bzip2 | 437.leslie3d | 464.h264ref | 482.sphinx3 |
| | 403.gcc | 444.namd | 470.lbm | 483.xalancbmk |
| | 429.mcf | 450.soplex | 471.omnetpp | |
| | 433.milc | 456.hmmer | 473.astar | |

**Table 3**
Multi-threaded applications.

| | | | | |
|---|---|---|---|---|
| Server | OLTP | TPC-C like, IBM DB2 DBMS. 800 Mb database, 4000 warehouses (3 districts, 30 customers, 10 items per warehouse) | | |
| | Apache | Dynamic Spec-Web like, Apache. 20,000 file repository (500 Mb), zero think time clients | | |
| | JBB | Spec-JBB like, Oracle JRE. 24 warehouses, 500 Mb data size | | |
| | Zeus | Static Spec-Web alike, Zeus | | |
| NPB | Conjugate gradient (CG) | Size A | LU Gauss–Seidel (LU) | Size A |
| | Fourier transform (FT) | Size W | Scalar penta-diagonal (SP) | Size A |
| | Integer sort (IS) | Size A | Unstruct. adaptive mesh (UA) | Size A |
| | Block tri-diagonal (BT) | Size A | Multi-grid (MG) | Size W |

Our base system tries to mimic the main characteristics of the Intel Haswell architecture [28]. We model a 4-core CMP where out-of-order processors have a 4-way superscalar pipeline with 4 ways and a 128 entry ROB. The first two levels of the memory hierarchy are private to each processor. L1 instruction and data caches are 8-way associative, with 32 kB size. L2 maintains the same associativity but is 256 kB. Finally, LLC configuration corresponds to a shared SNUCA [29] with 4 banks and a total capacity of 8 MB. L2 is exclusive with L1 (i.e., acts as a victim cache of L1) and LLC is inclusive with private caches [28]. All caches in the hierarchy use 64Byte blocks. We use an in-cache MOESI directory coherency protocol. Main memory access is simulated through a fixed delay of 250 cycles, assuming a 32 GB off-chip bandwidth. Additionally, in order to provide a wide use scenario, we have evaluated the proposal in a single-core system. In this case we reduce the LLC capacity of the baseline system to 2 MB.

More than 70 diverse workloads, running on top of the Solaris 10 OS, have been considered for evaluation. We include both single-core configurations and multi-core ones. For single-core evaluation we use 14 workloads from the SPEC2006 benchmark suite. We include both memory intensive and non-memory intensive applications. Multi-core applications include both multi-programmed and multi-threaded applications (both scientific and commercial servers). There are eight numerical applications from the NAS parallel benchmarks suite (Openmp implementation version 3.2 [30]). The server benchmarks correspond to the whole Wisconsin Commercial Workload suite [31]. The multi-programmed workloads are made up of 46 different mixes of the SPEC CPU2006 suite, representing variable cache sensitivities and memory behaviors. The lists in Tables 1–4 provide a brief summary of the whole application set employed for our experiments.

For each workload evaluated, multiple runs are employed to fulfill strict 95% confidence intervals. Benchmarks are fast-forwarded to the point of interest, during which page tables, TLBs, and caches are warmed up. In iteration-based applications, corresponding to the NPB suite, the warm checkpoint is taken in the middle of the main loop execution and simulations perform a fixed number of loop iterations after warm checkpoint. Transactional workloads are warmed up by running hundreds of thousands of transactions, and accurately simulated for a fixed number of additional transactions. SPEC workloads are fast-forwarded to the point of interest and simulate 5 billion instructions.

## 6. Results and analysis

### *6.1. Setting MIP configuration parameters*

Prior to MIP performance evaluation, we must analyze and understand the influence that the different configuration alternatives could have on results. The two basic aspects for MIP replacement are hit-rate calculation and update interval. Hit rate precision is proportional to the number of bits devoted to the saturation counters. The more bits in the counter, the more

**Table 4**
Multi-programmed applications.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Spec | MX1 | astar/bzip/lbm/mcf | MX18 | gcc/lbm/mcf/sphin | MX35 | libq/milc/hmmer/gcc |
| | MX2 | astar/bzip/milc/mcf | MX19 | gcc/libq/milc/sphin | MX36 | mcf/milc/lbm/lbm |
| | MX3 | astar/gcc/hmmer/lbm | MX20 | gcc/milc/libq/hmmer | MX37 | mcf/milc/lbm/libq |
| | MX4 | astar/gcc/hmmer/mcf | MX21 | hmmer/astar/bzip/bzip | MX38 | milc/lbm/hmmer/gcc |
| | MX5 | astar/lbm/milc/sphin | MX22 | hmmer/astar/omne/milc | MX39 | milc/mcf/gcc/hmmer |
| | MX6 | astar/libq/bzip/omne | MX23 | hmmer/bzip/xala/sphin | MX40 | milc/mcf/libq/hmmer |
| | MX7 | astar/libq/xala/sphin | MX24 | hmmer/gcc/astar/omne | MX41 | omnet/libq/bzip/sphin |
| | MX8 | astar/mcf/hmmer/gcc | MX25 | hmmer/gcc/bzip/libq | MX42 | omnet/mcf/hmmr/gcc |
| | MX9 | astar/omnet/lbm/milc | MX26 | hmmer/gcc/lbm/lbm | MX43 | omnet/mcf/lbm/gcc |
| | MX10 | bzip/libq/mcf/sphin | MX27 | hmmer/gcc/mcf/mcf | MX44 | omnet/milc/mcf/sphin |
| | MX11 | bzip/omnet/lbm/mcf | MX28 | hmmer/hmmer/gcc/gcc | MX45 | omnet/omnt/lbm/lbm |
| | MX12 | bzip/omnet/xala/hmmer | MX29 | hmmer/libq/milc/gcc | MX46 | omnet/xala/milc/lbm |
| | MX13 | bzip/sphin/libq/xala | MX30 | hmmer/milc/omne/lbm | | |
| | MX14 | bzip/xala/omnet/sphin | MX31 | lbm/milc/libq/libq | | |
| | MX15 | gcc/bzip/lbm/milc | MX32 | leslie/lbm/libq/mcf | | |
| | MX16 | gcc/hmmer/lbm/lbm | MX33 | libq/lbm/mcf/xala | | |
| | MX17 | gcc/hmmer/milc/lbm | MX34 | libq/mcf/milc/sphin | | |

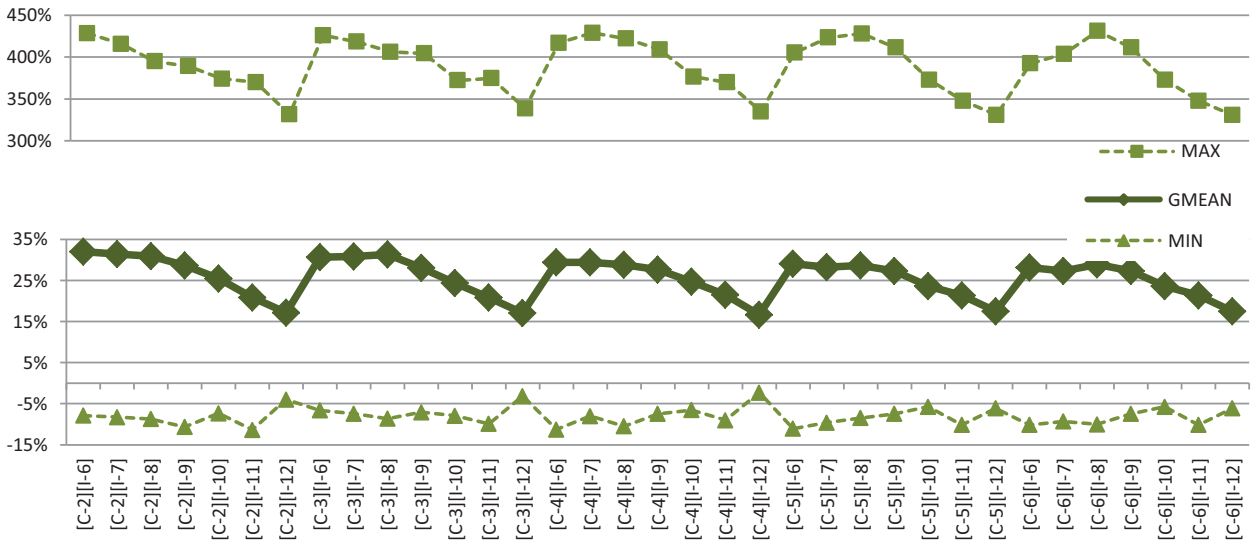


**Fig. 4.** Hit rate estimation for different configuration values. Results normalized to LRU.

accurate the estimation of hit rates. Unfortunately, precision is not free. Larger counters need more updates to reach saturation. In these cases where hit rate tendency changes suddenly, it takes more cache accesses to reflect this fact in the estimated hit rate value (larger distance between 0 and $MAX_{HR}$). Accesses to LLC are not profuse and if this parameter is not selected carefully insertion decisions could be taken too late. Similarly, update interval could also be relevant for performance. If it is set too long, abrupt changes in optimal insertion position could be ignored by the insertion policy.

We have performed a design-space exploration for these two parameters, evaluating the hit-rate evolution for the whole set of applications. All the simulations have been performed with cache groups of 32 sets, long enough to minimize the possible disturbance caused by the reference and explorer sets. The hit rate improvements obtained for all the configuration options analyzed are shown in Fig. 4. Values have been normalized to LRU results and only average; maximum and minimum values are displayed (i.e., workload with best and worst behavior). Each *x*-axis position represents a different size for update interval (I) and saturation counter (C). Size is expressed in bits, I-6 meaning that 64 accesses to LLC are performed in each update interval. C-2 represents a two-bit counter, ranging in value from 0 to 3.

The improvement over LRU is consistent across the set of configuration points, average values being between 15% and 35%. It should be noted that the application providing the worst results is very close to LRU, while the best results improve hit rate by more than five times in any case. Fig. 4 shows that the highest average hit rate is obtained when minimal precision is employed to calculate hit rate. With only two or three bits per saturation counter, we are able to extract the maximum benefit. In contrast, as precision grows, hit rate estimation slowly decreases. If too many bits are employed for hit-rate encoding, the movement from $MAX_{HR}$ to 0 requires more accesses and the slower explorer and conventional set movement slightly affects performance. Concerning update interval, the behavior of the MIP replacement policy remains nearly constant for a wide range of update values. Only when the update interval is extremely long (more than 1024) does performance degradation become significant.

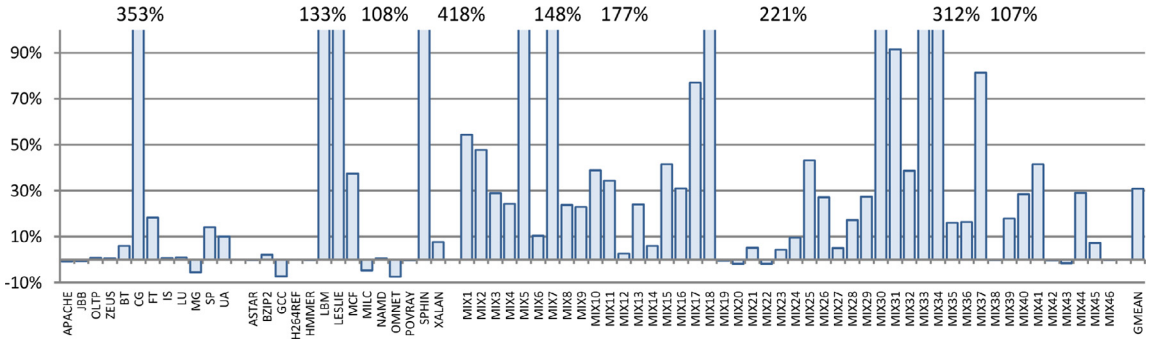


**Fig. 5.** Hit rate improvement over LRU results.

The results in Fig. 4 suggest that MIP benefits are mostly independent of configuration point. Saturation counter size has negligible effect on performance, providing maximum benefit with minimal overhead. Degradation of performance for large counters is below 5%. In the case of update interval, in the range from 64 to 512 accesses per update, performance remains nearly constant, also providing a minimal storage overhead in this case. For the rest of evaluation sections, we have chosen 3-bit saturation counters and an update interval of 256 accesses. This means a storage overhead of only 17 bits and some simple additional logic to choose and shift the insertion point.

### 6.2. MIP performance

Fig. 5 shows the behavior of the MIP proposal when compared to LRU replacement. The *y*-axis shows hit rate improvement for LLC, while the *x*-axis shows applications. Compared to LRU, MIP is able to significantly improve Cache performance. Most applications benefit from dynamic insertion and hit rate degradation is only observed in isolated cases (and always below 10%). On average, MIP is able to increase cache performance by 35% compared to LRU. Analyzing results group-by-group, we observe that the lowest performance benefit corresponds to the multi-threaded group. Recency-friendly patterns predominate in numerical applications, making LRU the best option in most cases. Minimal hit rate degradation can be observed in a few applications, such as *jbb*. In those cases where LRU is the optimal policy during the whole execution time, MIP suffers a small performance degradation caused by explorer sets, which never reach the MRU insertion position. However, some numerical applications have special access patterns without reuse (unlike MRU). This is the case of CG, where the MIP policy is able to move the insertion point to the last position in the priority queue, improving hit rate by more than 400%. The single-core group has bimodal behavior. Applications with a small working-set, such as *astar*, *hmmer* or *omnetpp*, extract no benefit from the replacement algorithm. The number of accesses to LLC in these cases is low, and the estimation of optimal insertion position becomes extremely difficult due to the low number of accesses to evaluation sets. In contrast, other applications such as *lbm* or *leslie3d* seem to be able to quickly find the optimal insertion position despite infrequent references. Again, MIP replacement is able to outperform LRU in these cases. Finally, SPEC 2006 mixes seem to extract the maximum benefit from MIP policy. For this group, the temporal and per-core variability of access patterns increases irregularity, improving the utility of dynamic insertion. Those cases where no performance improvement is observed correspond to mixes where no (or few) applications have a large enough working set to stress LLC.

Fig. 6 represents full-system performance (execution time) improvement normalized to LRU values. As can be seen, results are consistent with observed hit-rate improvements. On average, MIP improves execution time by nearly 5%. It should be noted that these results are highly dependent on the workload working set. Thus, applications such as *lbm* or *leslie3d* extract marginal performance benefit from replacement policy, even when doubling LLC hit rate as seen in Fig. 5. In contrast, other applications

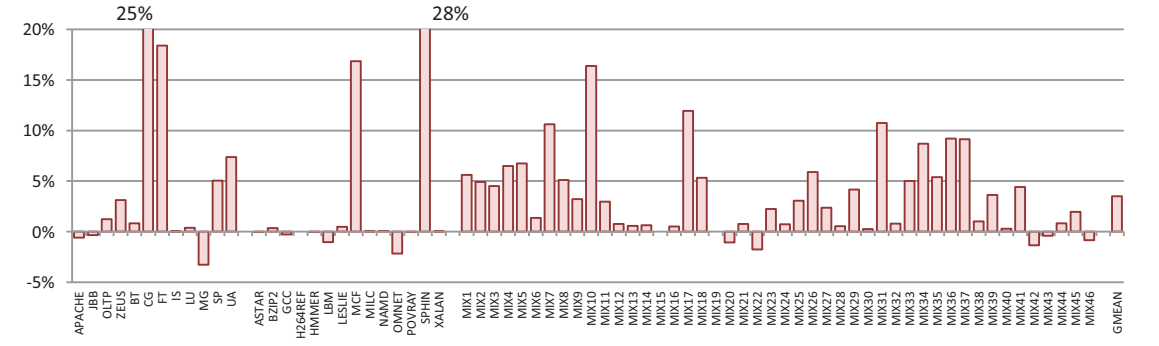


**Fig. 6.** Full system performance improvement over LRU.

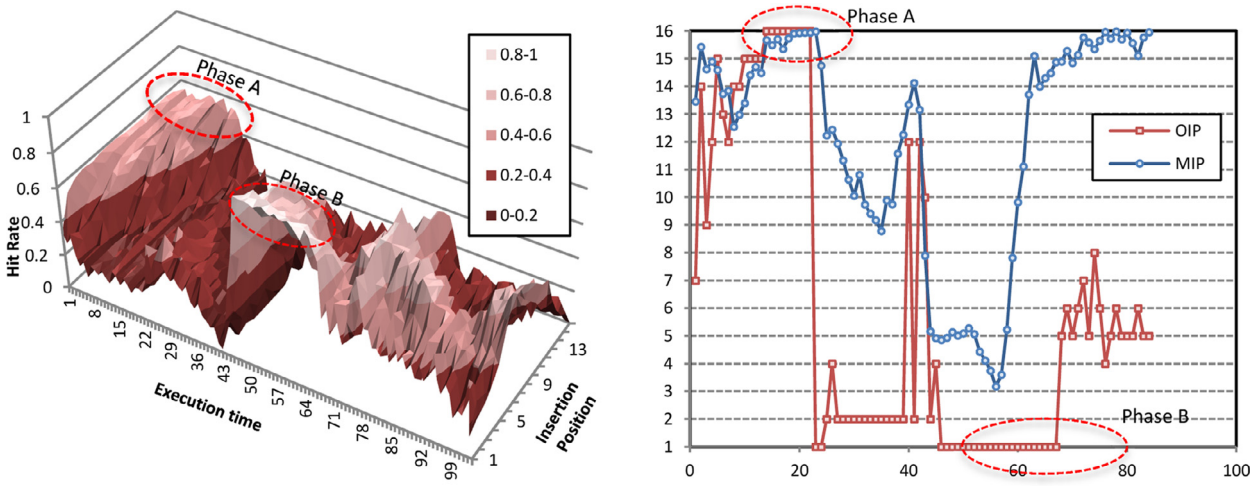


**Fig. 7.** *FT* workload. Optimal insertion position (left). MIP distance to optimal position (right).

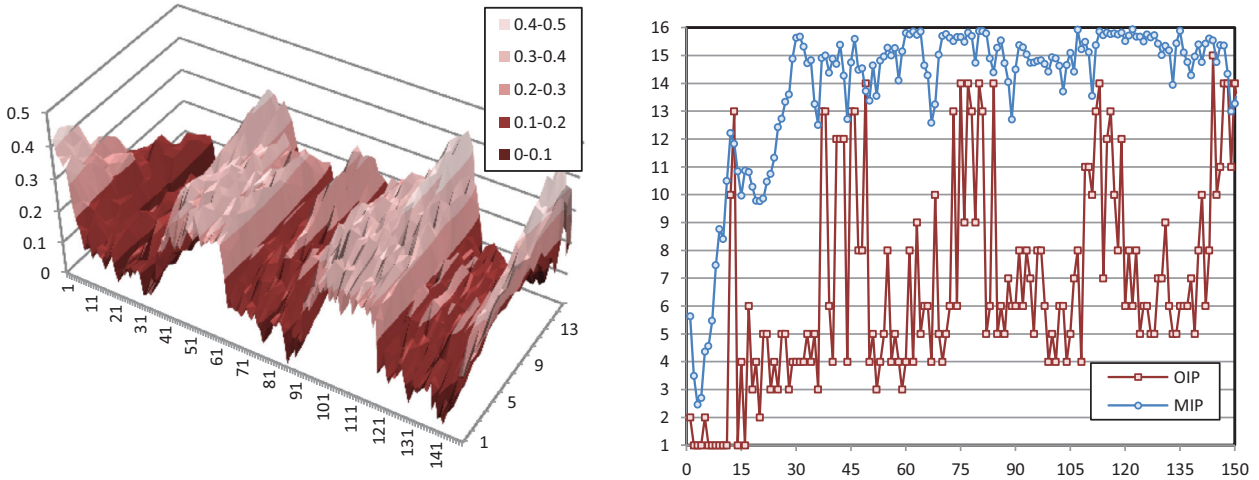


**Fig. 8.** *MIX17* workload. Optimal insertion position (left). MIP distance to optimal position (right).

such as *FT* or *CG* are able to transform a great part of hit rate improvement into performance benefits. In the case of CG, the utilization of MIP as the replacement algorithm leads to a 25% reduction in execution time.

### 6.3. MIP distance to optimal insertion point

In order to better understand the behavior of the MIP policy, this section describes how close the optimal insertion point curve is followed over time. We have selected two workloads with different distances between MIP and OIP (optimal insertion policy) insertion position. For each case analyzed, Figs. 7(left) and 8(left) are a three-dimensional representation of the hit-rate obtained for each insertion position over execution time. The *Z*-axis measures hit-rate, the *Y*-axis represents the different insertion and the *X*-axis the millions of cycles simulated. The optimal insertion position for each time interval is represented as a 2D curve in Figs. 7(right) and 8(right), comparing its shape with the one obtained by MIP replacement policy for the same workload.

Fig. 7 corresponds to the results obtained by FT workload, where MIP policy is able to improve hit rate over 15% compared to LRU. As can be seen, this application presents execution phases where the optimal insertion point is far from the MRU position, which degrades LRU performance in favor of MIP. For this application we observe in Fig. 7 how MIP provides a curve with a similar shape to the curve described by OIP, demonstrating the correct behavior of our proposal. In some execution phases distance between optimal and MIP increases. This phenomenon is more clearly detected in Fig. 8 in which, while optimal insertion position rapidly moves between distant values, MIP insertion position remains nearly constant for a distant-to-LRU insertion position. Although we might think that the MIP algorithm is not working correctly in these cases, Fig. 8(left) provides additional information that helps with understanding this phenomenon. In this case, the 3D representation shows that the difference between the hit rate at optimal position and adjacent ones is minimal. This shows a kind of flattened shape where many insertion

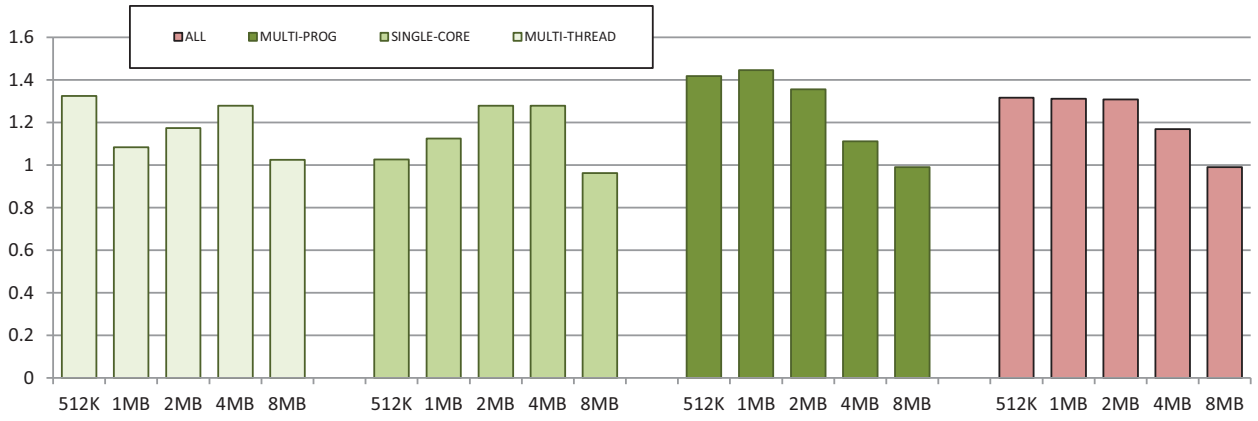


**Fig. 9.** MIP sensitivity to cache size per core.

positions share a similar hit rate value. In these cases, MIP policy tends to place insertion position at the further-from-LRU border of the flattened shape. In this way, despite being far (in distance) from the optimal insertion point, hit-rate is still very close to its optimal value.

### 6.4. MIP sensitivity to LLC size

Fig. 9 represents MIP performance for the three workload types with different LLC sizes, ranging from 512 kB to 8 MB. In the case of single-core applications, these values correspond to total LLC. In contrast, for the rest of the applications, size values are per-core parameters, total LLC size being calculated as size multiplied by the number of cores. Associativity and block size is maintained for all cache sizes evaluated. All the results provided show hit-rate improvement normalized to LRU values. Results show that MIP consistently outperforms LRU while application working sets are larger than LLC size.

A different trend can be observed between multi-programmed applications and the other groups. When LLC is too small, inclusiveness reduces the effective capacity. In the case of single-core applications, this reduction significantly affects LLC performance, making the replacement policy less relevant. A similar behavior is observed for multi-threaded applications, where the replacement algorithm is not effective when room for shared data is scarce. In contrast, multi-programmed applications do not suffer from smaller LLC capacities, because storage is shared among applications with different working sets in which applications with larger working sets can make use of LLC that is not utilized by those with smaller working sets. The 512 kB value for multi-threaded applications seems to be inconsistent with the rest of the results. For the LU application the hit rate of LRU is extremely degraded for such a small LLC size, which increases the MIP value to a 10× better hit rate in this case. If LU is excluded, the geometric mean value for 512 kB is similar to that obtained for 1 MB.

When the total amount of LLC storage is increased to 32 MB, the working set of most applications fits with LLC. The number of cache misses is low and consequently the relevance of the replacement algorithm is minimal.

### 6.5. Comparing MIP to other policies

Our last experiment compares the results obtained for MIP to a state-of-the-art proposal with similar characteristics. We have chosen DRRIP [17] as a counterpart because both replacement policies share a similar structure (improvement over a baseline policy) and hardware overhead (one/three saturating counters per cache bank). DRRIP has demonstrated to perform better than a wide range of counterparts, such as LRU, NRU, DIP, or hybrid NRU/LFU [17]. For this reason we limit our evaluation to a single counterpart. DRRIP has been configured according to its optimal values as described in [17]. Results for this evaluation are shown in Figs. 10 and 11. Fig. 10 represents the hit-rate improvement of the two counterparts using LRU as normalization point. Results have been grouped according to the workload class. More detailed results are provided in Fig. 11, where the *s*-curve for the hit-rate of MIP and DRRIP is shown. The *X*-axis represents all 72 workloads, while the *y*-axis represents hit-rate relative to LRU.

As can be seen, MIP consistently outperforms DRRIP for any of the application groups analyzed. The dynamic behavior of MIP allows it to rapidly adapt to application changes, regardless of the access pattern. In contrast, DRRIP can degrade performance when reuse frequency varies significantly over the application runtime. If reuse disappears suddenly for cached data, DRRIP is unable to handle such an amount of thrashing. The replacement policy is not fast enough to eliminate these dead blocks, because only infrequent insertions move dead blocks to eviction positions. In this scenario, MIP rapidly observes that the optimal hit rate is obtained when current cache content is evicted, moving the insertion point to MRU positions. When multiple applications with different access patterns are mixed, these fast changes in reuse are rarely found. In these cases, both DRRIP and MIP are able to clearly outperform LRU, obtaining a 20% and 30% hit-rate improvement respectively. However, isolated applications are in some cases much more sensitive to this thrashing effect. In these cases DRRIP results are highly application dependent. The left-side values observed in Fig. 11 for DRRIP show that in some cases LRU performance can be degraded by more than 30%. For

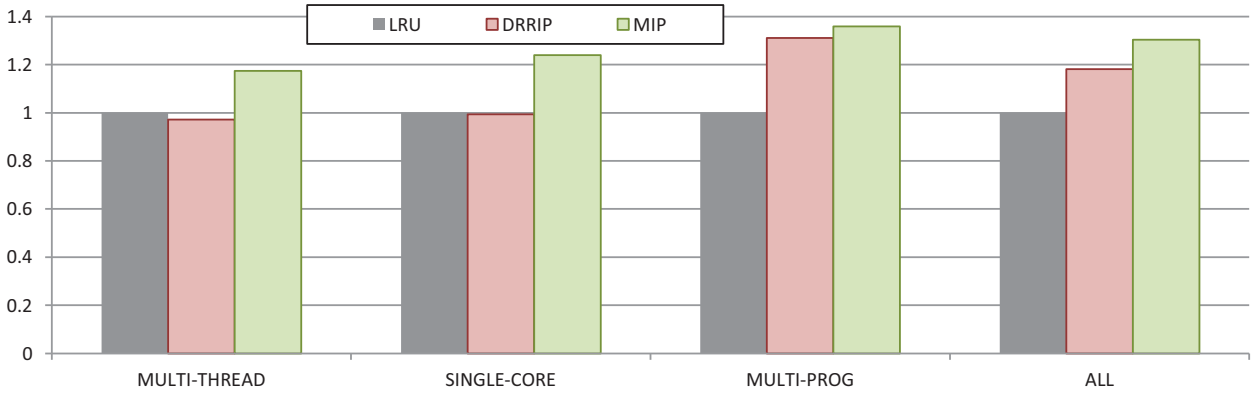


**Fig. 10.** LRU normalized hit rate per application group, gmean values.

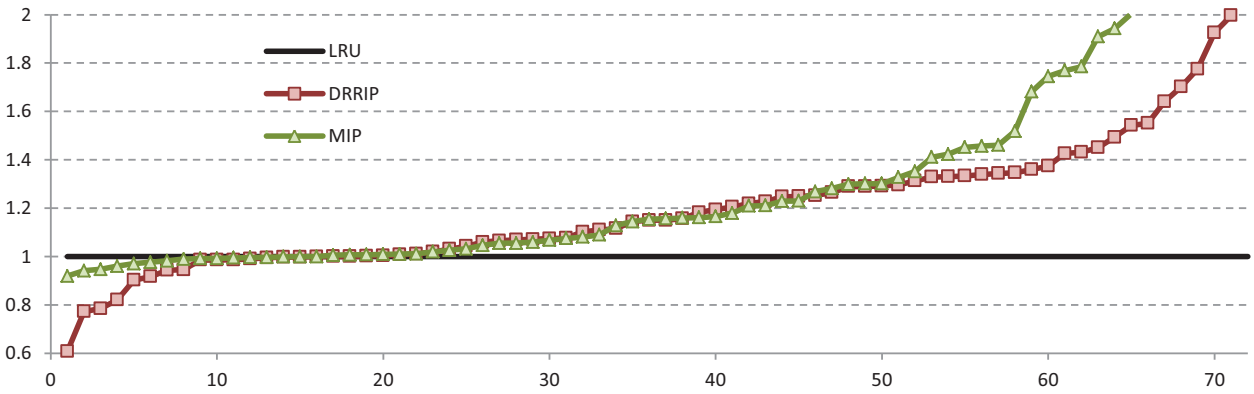


**Fig. 11.** LRU normalized performance, applications ordered from worst to best results.

both multi-threaded and single-core applications MIP results are much more consistent, with left-side values very close to LRU and significant improvements in some applications. This leads to a sustained 20% improvement of MIP over DRRIP in both groups.

Fig. 11 summarizes the potential advantages of MIP over DRRIP. Having LRU as reference policy, in a worst-case scenario the fallback reference point guarantees close-to-LRU performance. When reuse frequency correlates with reuse distance DRRIP and MIP behave similarly, DRRIP being better on some occasions. Finally, when reuse distance is highly variable and locality better estimates reuse distance, MIP obtains its optimal results. From Fig. 11 we observe that 10 out of 72 applications at least double hit-rate results obtained by LRU, while only one application reaches this value in the case of DRRIP.

A noteworthy result is that in contrast to DRRIP, for multithreaded or single-thread applications, LLC hit-rate benefits from the proposal. Since general purpose CMPs include such workloads, this scenario of usage is relevant. Therefore DRRIP seems to be useful when there are multiple independent threads accessing the LLC. If these threads exhibit dissimilar reuse characteristics, the mechanism seems to work better than LRU. When the threads have the same behavior, such as in the case of multithreaded workloads, DDRIP seems to impair LLC performance slightly. Our proposal is capable of going beyond, and improving cache performance in a broader usage scenario.

### *6.6. Exploring additional benefits in alternative MIP configurations*

The search for maximum hit rate with the current MIP configuration favors the displacement of insertion position to distant positions from the highest priority insertion point (LRU replacement). In those cases where reference sets have a poor hit rate the only useful information comes from the explorer sets, but there is no information at all about what is happening on the "left side" of the conventional sets (positions from MRU + 1 to current insertion position).

With this unbalance, insertion position tends to favor those values "far" from the reference, which is not bad a priori, but could leave many insertion positions unexplored for a long period of time. Additionally, in the presence of a single explorer we could experience the appearance of local or "false" hit-rate maximums, as depicted in Fig. 12. In an attempt to solve this problem we explored the possibility of adding a second explorer to the mechanism, with an insertion position between conventional and reference. In this way one of the explorers will be aware of what happens at those insertion positions that are unexplored when moving too far from the reference.

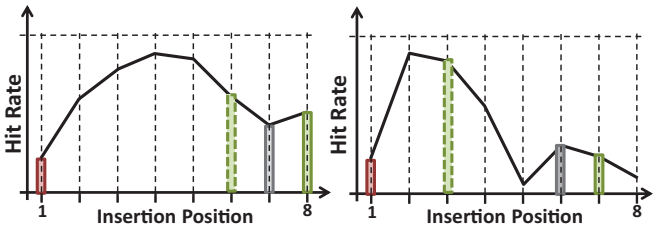


**Fig. 12.** Two hit-rate distributions where FM cannot find the maximum value but FMM can.

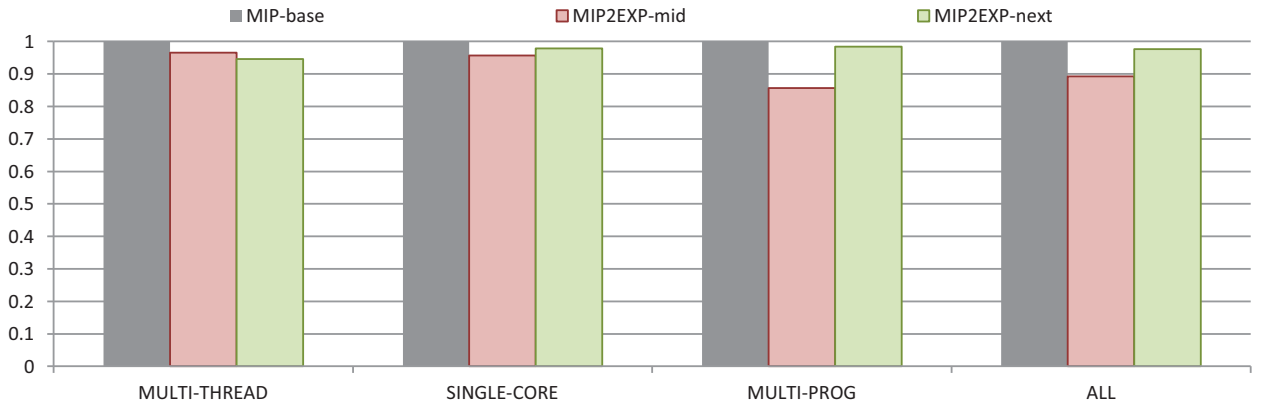


**Fig. 13.** Base MIP normalized hit rate per application group, gmean values.

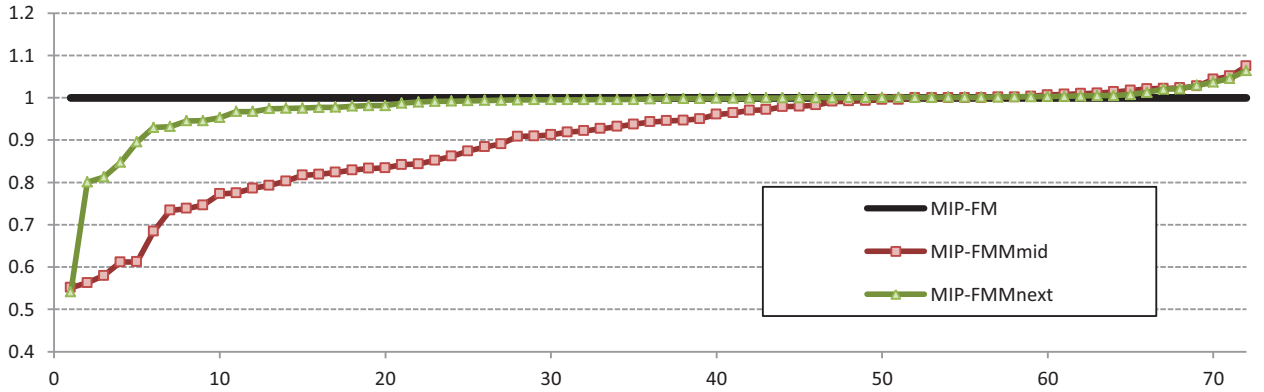


**Fig. 14.** Base MIP Normalized performance, applications ordered from worst to best results.

There are multiple ways to select the insertion position of this second explorer. For this analysis we have opted for two different approaches. The first one, named $MIP_{2EXP}$-next, always places the insertion position of the second explorer next to the insertion position of conventional sets. This means that the explorer insertion positions are always $I_{pos} + 1$ and $I_{pos} - 1$ (for right and left explorers respectively). With this configuration we can take the correct decision for situations such as the ones shown in Fig. 12(left), where the utilization of only one explorer could lead to a suboptimal insertion point. However, with both the explorer and conventional sets having a close insertion point, we could be confined to a local maximum, as shown in Fig. 12(right). In an attempt to reduce the possible impact of these situations, an alternative proposal is to place the insertion point for the second explorer at a distance half way between the reference and conventional insertion points. With this configuration, named $MIP_{2EXP}$-mid, the number of conventional sets in each group is reduced by one, which could have some impact on global hit rate if groups are small. The update of insertion position is performed in the same way as before, replacing only $HR_{REF}$ by $HR_{LEFT} = MAX(HR_{REF}, HR_{EXP\text{-}left})$.

We evaluated these two MIP alternatives with the results shown in Figs. 13 and 14. Fig. 13 shows the hit-rate improvement of the two alternatives when compared to baseline MIP (denoted here as MIP-FM), while Fig. 14 represents the *s*-curve for the same counterparts. Focusing on results of both graphs, the single-explorer configuration (MIP-FM) seems to be the best approach. Despite having more information to search for a global maximum hit rate, algorithms with two explorers do not improve base results. The reason behind this is the shape of Hit-Rate curve. The second explorer was included to identify false maximums when

conventional and explorer sets were far from LRU insertion position (reference set). However, when this false maximum is close to MRU, the additional explorer has the opposite effect. Insertion position can be held back by the new explorer, in a similar way but in an opposite direction as explained in Fig. 12 for the first explorer. This effect is much more significant when the insertion position of the new explorer sets is not close to the conventional sets' insertion position. In this situation, global maximums far from MRU could be missed, losing a great opportunity to improve hit rate. Looking carefully at the results obtained we can infer that the situations where a false maximum is close to LRU are more frequent than those next to MRU. For this reason baseline MIP is more effective than the two-explorer based approaches.

## 7. Summary

The work devoted to finding the optimal way to estimate reuse distance of cache blocks is profuse. Through our initial experiment we have confirmed that applications show non-bimodal reuse behavior over runtime. This fact limits the potential benefits of replacement policies that assume a bimodal reuse, either statically or dynamically. In this paper we propose cache replacement using mobile insertion policies (MIP). MIP dynamically adjusts the insertion position of new blocks looking for the maximum hit rate. Making use of a set dueling mechanism, we have evaluated alternative ways to find the optimal insertion point. We have found that the optimal results are obtained when a single explorer is employed. We show that MIP outperforms LRU hit rate by an average of 30%. We also show that MIP improves LRU performance by an average of 5% on a 4-Core CMP with a 16-way 8 MB shared last level cache. Finally, we compared our proposal to DRRIP, demonstrating 10% better performance on average and improvement of single-thread workloads and multithreaded applications by 20% on average.

## Acknowledgments

The authors would like to thank Jose Angel Herrero for his valuable assistance with the computing environment HPC cluster Calderon within the datacenter 3Mares. Also, we greatly appreciate all the comments made by reviewers, which helped to improve the quality of the paper. This work has been supported by the MICCIN (Spain) under contract TIN2010-18159 and the HiPEAC European Network of Excellence.

## References

[1] W.A. Wulf, S.A. McKee, Hitting the memory wall: implications of the obvious, Comput. Archit. News 23 (1995) 20–24.
[2] B.M. Rogers, A. Krishna, G.B. Bell, K. Vu, X. Jiang, Y. Solihin, Scaling the bandwidth wall: challenges in and avenues for CMP scaling, ACM SIGARCH Comput. Archit. News 37 (3) (2009) 371–382.
[3] L. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Syst. J. (1966).
[4] A. Jaleel, K.B. Theobald, S.C. Steely, J. Emer, High performance cache replacement using re-reference interval prediction (RRIP), in: Proceedings of the 37th Annual International Symposium on Computer Architecture – ISCA '10, 2010, p. 60.
[5] D. Lee, J. Choi, J.H. Kim, S.H. Noh, S.L. Min, Y. Cho, C.S. Kim, LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies, IEEE Trans. Comput. 50 (12) (2001) 1352–1361.
[6] R. Subramanian, Y. Smaragdakis, G. Loh, Adaptive caches: effective shaping of cache behavior to workloads, in: 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), 2006, pp. 385–396.
[7] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, A.V. Veidenbaum, Improving cache management policies using dynamic reuse distances, in: 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012, pp. 389–400.
[8] A.-C. Lai, C. Fide, B. Falsafi, Dead-block prediction & dead-block correlating prefetchers, ACM SIGARCH Comput. Archit. News 29 (2) (2001) 144–154.
[9] H. Liu, M. Ferdman, J. Huh, D. Burger, Cache bursts: a new approach for eliminating dead blocks and increasing cache efficiency, in: 2008 41st IEEE/ACM International Symposium on Microarchitecture, 2008, pp. 222–233.
[10] M. Kharbutli, Y. Solihin, Counter-based cache replacement and bypassing algorithms, IEEE Trans. Comput. 57 (4) (2008) 433–447.
[11] C.-J. Wu, A. Jaleel, M. Martonosi, S.C. Steely, J. Emer, PACMan, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture – MICRO-44 '11, 2011, p. 442.
[12] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, J. Emer, Adaptive insertion policies for managing shared caches, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques – PACT '08, 2008, p. 208.
[13] Y. Xie, G.H. Loh, PIPP, ACM SIGARCH Comput. Archit. News 37 (3) (2009) 174.
[14] W.A. Wong, J.-L. Baer, Modified LRU policies for improving second-level cache behavior, in: Proceedings of the Sixth International Symposium on High-Performance Computer Architecture – HPCA-6 (Cat. No. PR00550), 2000, pp. 49–60.
[15] M.K. Qureshi, A. Jaleel, Y.N. Patt, S.C. Steely, J. Emer, Adaptive insertion policies for high performance caching, ACM SIGARCH Comput. Archit. News 35 (2) (2007) 381.
[16] N. Megiddo, D.S. Modha, ARC: a self-tuning, low overhead replacement cache, in: FAST'03 Proceedings of the 2nd USENIX Conference on File and Storage Technologies, vol. 3, 2003, pp. 115–130.
[17] A. Jaleel, K.B. Theobald, S.C. Steely, J. Emer, High performance cache replacement using re-reference interval prediction (RRIP), ACM SIGARCH Comput. Archit. News 38 (3) (2010) 60.
[18] J.T. Robinson, M.V. Devarakonda, Data cache management using frequency-based replacement, ACM SIGMETRICS Perform. Eval. Rev. 18 (1) (1990) 134–142.
[19] M.K. Qureshi, D.N. Lynch, O. Mutlu, Y.N. Patt, A case for MLP-aware cache replacement, in: 33rd International Symposium on Computer Architecture (ISCA'06), vol. 34, no. 2, 2006, pp. 167–178.
[20] G. Keramidas, P. Petoumenos, S. Kaxiras, Cache replacement based on reuse-distance prediction, in: 2007 IEEE International Conference on Computer Design, ICCD 2007, 2007, pp. 245–250.
[21] S. Khan, A. Alameldeen, C. Wilkerson, Improving Cache Performance by Exploiting Read-Write Disparity, *istc-cc.cmu.edu*.
[22] J.D. Collins, D.M. Tullsen, Hardware identification of cache conflict misses, Nov. 1999, pp. 126–135.
[23] V.V. Fedorov, S. Qiu, A.L.N. Reddy, P.V. Gratz, ARI, ACM Trans. Archit. Code Optim. 10 (4) (2013) 1–19.
[24] S. Khan, D.A. Jimenez, Insertion policy selection using decision tree analysis, in: Proceedings of IEEE International Conference on Computer Design, 2010, pp. 106–111.
[25] J. Merino, V. Puente, J.A. Gregorio, ESP-NUCA: a low-cost adaptive non-uniform cache architecture, in: 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA), 2010.
[26] W. Hasenplaugh, P.S. Ahuja, A. Jaleel, S. Steely Jr., J. Emer, The gradient-based cache partitioning algorithm, ACM Trans. Archit. Code Optim. 8 (4) (2012) 1–21.

[27] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, D.A. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, ACM SIGARCH Comput. Archit. News 33 (4) (2005) 92.
[28] N. Kurd, P. Mosalikanti, M. Neidengard, J. Douglas, R. Kumar, Next generation Intel Core micro-architecture (Nehalem) clocking, IEEE J. Solid-State Circuits 44 (4) (2009).
[29] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, S.W. Keckler, A NUCA substrate for flexible CMP cache sharing, in: Proceedings of the 19th Annual International Conference on Supercomputers – ICS '05, 2005, p. 31.
[30] H. Jin, M. Frumkin, J. Yan, The OpenMP implementation of NAS parallel benchmarks and its performance, in: Natl. Aeronaut. Sp. Adm. (NASA), Tech. Rep. NAS-99-011, Moffett Field, USA, October, 1999.
[31] A.R. Alameldeen, M.M.K. Martin, C.J. Mauer, K.E. Moore, M. Xu, M.D. Hill, D.A. Wood, D.J. Sorin, Simulating a $2 M commercial server on a $2 K PC, Computer (Long. Beach. Calif). 36 (2) (2003) 50–57.