Combining Recency of Information with Selective Random and a Victim Cache in Last-Level Caches

ALEJANDRO VALERO, JULIO SAHUQUILLO, SALVADOR PETIT, PEDRO LÓPEZ,
and JOSÉ DUATO, Universitat Politècnica de València

Memory latency has become an important performance bottleneck in current microprocessors. This problem aggravates as the number of cores sharing the same memory controller increases. To palliate this problem, a common solution is to implement cache hierarchies with large or huge Last-Level Cache (LLC) organizations.

LLC memories are implemented with a high number of ways (e.g., 16) to reduce conflict misses. Typically, caches have implemented the LRU algorithm to exploit temporal locality, but its performance goes away from the optimal as the number of ways increases. In addition, the implementation of a strict LRU algorithm is costly in terms of area and power.

This article focuses on a family of low-cost replacement strategies, whose implementation scales with the number of ways while maintaining the performance. The proposed strategies track the accessing order for just a few blocks, which cannot be replaced. The victim is randomly selected among those blocks exhibiting poor locality. Although, in general, the random policy helps improving the performance, in some applications the scheme fails with respect to the LRU policy leading to performance degradation. This drawback can be overcome by the addition of a small victim cache of the large LLC.

Experimental results show that, using the best version of the family without victim cache, MPKI reduction falls in between 10% and 11% compared to a set of the most representative state-of-the-art algorithms, whereas the reduction grows up to 22% with respect to LRU. The proposal with victim cache achieves speedup improvements, on average, by 4% compared to LRU. In addition, it reduces dynamic energy, on average, up to 8%. Finally, compared to the studied algorithms, hardware complexity is largely reduced by the baseline algorithm of the family.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles—Cache memories

General Terms: Design, Performance

Additional Key Words and Phrases: Last-Level Cache, MRU-Tour, replacement algorithm

ACM Reference Format:

Valero, A., Sahuquillo, J., Petit, S., López, P., and Duato, J. 2012. Combining recency of information with selective random and a victim cache in last-level caches. *ACM Trans. Architect. Code Optim.* 9, 3, Article 16 (September 2012), 20 pages.

DOI = 10.1145/2355585.2355589 <http://doi.acm.org/10.1145/2355585.2355589>

1. INTRODUCTION

Computer architects have implemented cache memories [Smith 1982] since late 1960s to mitigate the huge gap between processor and main memory speed. This problem was originally solved by using a single cache, but as the memory gap continued growing,

---

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.

Author's addresses: A. Valero, J. Sahuquillo, S. Petit, P. López, J. Duato, Department of Computer Engineering (DISCA), Universitat Politècnica de València, Valencia, Spain; email: alvabre@gap.upv.es, jsahuqui@disca.upv.es, plopez@disca.upv.es, jduato@disca.upv.es.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1544-3566/2012/09-ART16 \$15.00

DOI 10.1145/2355585.2355589 <http://doi.acm.org/10.1145/2355585.2355589>

several cache levels were necessary for performance. The first level (L1 cache) is the closest to the processor and it is designed for speed, while the second or the third level (if any) is referred to as Last-Level Cache (LLC) and it is designed to hide as much as possible the long miss penalty of accessing to main memory, which involves several hundreds of processor cycles in current microprocessors.

The system performance strongly depends on the cache hierarchy performance. Thus, many research has been done to improve the cache performance. Techniques like load-bypassing, way-prediction, or prefetching, have been widely investigated and implemented in many commercial products. Although these techniques have been successfully implemented in typical monolithic processors, the pressure on the memory controller is much higher in multicore and manycore systems than in monolithic processors. Therefore, the performance of the cache hierarchy in general, and the performance of the LLC in particular, is a major design concern in current microprocessors.

LLCs are designed as very large structures in order to keep as much information as possible so reducing capacity misses, whose sizes range from several hundreds of KB up to several tens of MB [Stackhouse et al. 2009; Kalla et al. 2010]. Moreover, this storage capacity is expected to grow as transistor features continue shrinking in future technology generations. In addition, in order to keep low the number of conflict misses, current LLCs implement a large number of ways (e.g., 16 ways).

Typically, caches exploit temporal locality by implementing the Least Recently Used (LRU) replacement algorithm. This algorithm acts as a stack that places the Most Recently Used (MRU) block on the top of the stack and the LRU block on the bottom, which is the evicted block when space is required. Although this algorithm works well in L1 caches with a low number of ways; with high associativities, like 8- and 16-ways that are encountered in current LLCs, strict LRU is too expensive to implement. Therefore, approximations to LRU are the norm in commercial processors but their performance start to deviate from the strict LRU [Baer 2010].

On the other hand, the performance of the LRU is quite far from the optimal replacement strategy referred to as Belady's algorithm [Belady 1966]. There are several reasons that explain why the LRU algorithm does not reach good performance in LLCs with large stacks. First, it suffers from thrashing effects in those workloads whose working set is greater than the available cache size, resulting in cyclic accessed blocks that will pass through the stack without being used again. Second, it forces a block to descend down to the bottom of the stack before eviction. The latter issue can lead to severe impact on performance since most of the blocks that are brought into the LLC are not referenced again before eviction (as experimental results will show). Recent research work has focused on how to improve this second shortcoming by predicting when a block can be evicted while it is still *walking* the LRU stack to the bottom [Kharbutli and Solihin 2008; Lin and Reinhardt 2002; Liu et al. 2008; Chaudhuri 2009]. Other works handle replacements by arranging blocks in a queue [Zhang and Xue 2009], and victim blocks are selected either from the top or from the bottom of the queue.

Experimental results show that most of the blocks are not referenced again once they leave the MRU position and, although some blocks return to this position, they do that very few times. An interesting observation is that the probability for a block to be referenced again does not always depend on the location it occupies in the stack, therefore, there is no need to maintain the order of the last n referenced blocks in n -way set-associative caches, which can become prohibitive for large associativities in commercial processors.

To palliate this drawback, in this article is proposed a low-cost replacement scheme that uses recency of information (which keeps the order of a few referenced blocks) while applying for the remaining blocks a selective random strategy to select the victim

Fig. 1. Replacement algorithms schemes.

block. In this way, hardware complexity can be largely reduced. Figure 1(a) shows a block diagram of the strict LRU with the required information (grey color), whereas Figure 1(b) depicts the diagram of the proposed scheme.

To define the selective random strategy, this article uses the concept of *MRU-Tour* (MRUT). The number of MRUTs of a block is defined as the number of times that a block enters in the MRU location during its live time. Based on the fact that most of the blocks exhibit a single MRUT, this article proposes the MRUT algorithm. Blocks showing only one MRUT are considered as candidates for replacement, and one of such blocks is randomly evicted when space is required. This way largely reduces hardware complexity over existing solutions while even improving the performance of the LRU algorithm. Variations of this policy maintaining recency of information for a few blocks are also studied leading to a new family of replacement strategies. Although the proposal, in general, performs better than the LRU algorithm, the accuracy of selective random strategy can hurt the performance in specific workloads. This drawback can be straightforwardly solved by simply adding a small victim cache to the LLC memory.

The proposed baseline MRUT algorithm improves performance over LRU and recently proposed algorithms. The MRUT-3-adaptive algorithm, which is the best algorithm of the family, reduces the Misses Per Kilo Instruction (MPKI) up to 22% with respect to LRU, whereas MPKI reduction falls in between 10% and 11% compared to a set of the most representative approaches. In addition, this version reduces dynamic energy by 3% with respect to LRU. Regarding the addition of a small victim cache, the proposal can achieve speedups up to 30% in some benchmarks and, on average, by 4% and 2% with respect to LRU and LRU with victim cache. In this case, dynamic energy savings are up to 8% compared to LRU. Finally, the baseline MRUT policy largely reduces the required number of control bits compared to both LRU and other existing approaches.

The remainder of this article is organized as follows. Section 2 presents the related work. Section 3 analyzes the main reason why the LRU algorithm does not reach good performance for some applications. Section 4 discusses the concept of MRUT, and presents the proposal. Section 5 analyzes the different MRUT patterns that a block can experience and introduces the victim cache. Section 6 presents and analyzes the performance of the proposal compared to other existing approaches and using the victim cache. Section 7 analyzes the hardware complexity of the studied replacement algorithms. Finally, Section 8 presents some concluding remarks.

2. RELATED WORK

Although it has been proved that strict LRU performance is far from the optimal replacement algorithm, especially in LLCs with high associativities, its performance is still better than the implemented in commercial processors. A major drawback of the LRU policy is that its implementation is costly in terms of power and hardware complexity. Because of both reasons, substantial research has been conducted on devising low power strategies aimed at improving its performance. This is the case of pseudo-LRU strategies [Wong and Baer 2000], which have also been implemented in recent commercial processors.

Reuse information has been exploited to improve the cache performance, especially in L1 caches. These approaches store information of previously cached blocks to estimate the length of the current *life* of a block. The NTS [Tyson et al. 1995] and MAT [Johnson et al. 1999] approaches are two examples of these schemes. The former marks a block as cacheable based on its reuse information, while the latter classifies the blocks as temporal and not temporal based on their reuse information during its past live time. Rivers et al. [1998] propose to exploit reuse information based on the effective address of the referenced data and on the program counter of the load instruction.

Lin and Reinhardt [2002] propose a hardware approach that predicts when to evict a block before it reaches the bottom of the LRU stack. The first approach is referred to as *sequence-based* prediction. This approach records and predicts the sequence of memory events leading up to the *last touch* of a block. The second one is the *time-based* approach, which tracks a line's timing to predict when a line's last touch will likely occur.

The counter-based L2 cache replacement [Kharbutli and Solihin 2008] predicts when to evict a block before it reaches the bottom of the stack. Two approaches were presented: the Access Interval Predictor (AIP) and the Live-time Predictor (LvP). The former bases its predictions by using counters to keep track of the number of accesses to the same set during a given access interval of a cache line. If the counter reaches a threshold value (learned from the previous behavior of the line), the associated line can be selected for replacement. The latter differs from the first one in that it counts the number of accesses to each line instead of to the same set.

The *cache burst-based* prediction [Liu et al. 2008] is applied in L1 caches and predicts the number of bursts that a block will exhibit during its live time in L1. Authors argue that such a predictor does not work at L2, since most cache accesses are filtered by L1 and are not seen by L2. As a result of this observation, they propose another kind of predictor (i.e., reference counter-based) for L2. Contrary to this work, we claim that the MRU-Tour concept can contribute to performance in L2 since we found that most blocks have a single MRU-Tour in L2. In addition, it can be done with simple hardware and without help of any assistant predictor. Finally, the proposed MRUT policy make use of recency of information for just a few blocks to enhance the performance.

The *instruction-based reuse-distance* prediction [Petoumenos et al. 2009] attempts to relate the reuse distance of a cache line to memory-access instructions in LLCs. These instructions usually access lines that exhibit predictable reuse behavior due to program locality. The predictor uses a history table indexed by the PC of the instruction, and each entry contains a reuse-distance value and a confidence value for the prediction. To update the predictor at run-time, the approach employs two sampler structures. The replacement policy selects the line to be evicted from those having a long reuse distance prediction (they will be referenced again far into the future) or those that were unaccessed in the cache.

Other state-of-the-art approaches improve the performance of LRU by using modified LRU [Dybdahl et al. 2007; Jiang and Zhang 2002; Wong and Baer 2000] or a pseudo-LIFO stack [Chaudhuri 2009].

In the context of victim caches, Scavenger [Basu et al. 2007] consists of an LLC architecture that divides the cache organization in two exclusive parts: a traditional LLC and a victim file. The latter part aims to retain the blocks that most frequently missed the LLC structure. On a miss in the LLC structure but a hit in the victim file, the block is transferred to the LLC structure. On a miss in both LLC and victim file, a Bloom Filter is used to keep track of the frequency of misses to the target block address. The replaced block from the LLC is allocated in the victim file depending on its frequency value and the lowest frequency value of the blocks residing in the victim. A pipelined priority heap is used in the victim file to maintain the priority values of all the blocks.

The Bubble scheme [Zhang and Xue 2009], unlike the LRU scheme, uses a queue instead of a stack that works as follows. An incoming block is allocated at the bottom of the queue, which is the location with the lowest access frequency. Anytime a block hits again, it is promoted one-position upwards the queue. In this way, the blocks closer to the top of the queue evince a higher access frequency than those closer to the bottom. When there is a lack of space, the block to be evicted is selected either from the bottom or the top of the queue, depending on whether the previous access to that set resulted in a cache miss or a cache hit, respectively. This work also presents a *divide-and-conquer* technique referred to as DC-Bubble, which divides the blocks in each cache set into independent groups, so that each group has its own replacement logic. In this scheme, when a block is fetched, the target group within the set is randomly selected. These schemes exploiting both recency and frequency of information and adapting to changes in the working set have been modeled for comparison purposes in this article.

In Qureshi et al. [2007], three adaptive insertion policies based on LRU are proposed. The first one, referred to as *LRU Insertion Policy (LIP)*, inserts all incoming blocks in the LRU position, and then they are promoted to the MRU position if they are referenced again. This behavior prevents LIP from the effect of cache thrashing. The second one, namely *Bimodal Insertion Policy (BIP)*, differs from LIP in that every x cache misses, the incoming block is inserted in the MRU position. This policy adapts to changes in the working set and, like LIP, provides thrashing protection. Finally, the third policy, referred to as *Dynamic Insertion Policy (DIP)*, dynamically combines LRU and BIP using the *Set Dueling* strategy. DIP uses a small fraction of the cache sets to measure the performance of each policy, and applies to the remaining sets the policy that achieves the best performance. All these policies have been also evaluated in this work.

Jaleel et al. [2010] propose a family of algorithms based on *Re-Reference Interval Prediction (RRIP)* of cache blocks to deal with cache thrashing and bursts of accesses to non-temporal data. The simplest version, called *Static RRIP (SRRIP)* policy, uses a saturating counter per cache block to predict if the block will be rereferenced sooner or later in the future. On a cache insertion, the *distance* prediction of the incoming block is set to be in the *distant* future (i.e., the counter is set to its maximum value), while subsequent accesses to this block reduce its distance prediction to be in the *near* future (i.e., decrease the counter by one each time the block is accessed). The victim block is selected among those blocks predicted to be accessed in the distant future. If there are not candidates, all the counters are increased by one until one of them saturates. An enhancement of SRRIP is the *Bimodal RRIP (BRRIP)* policy. It differs from SRRIP in that every x cache misses the counter of the incoming block is set to its maximum value minus one. Finally, both SRRIP and BRRIP policies were used together using

Table I. Probability for a Block to be Referenced Again Using the LRU Algorithm

Stack locat.	Benchmarks												
	swi.	mgr.	app.	vpr	gal.	art	mcf	fac.	ammp	luc.	bzi.	two.	apsi
1	38.2	72.6	92.6	49.4	85.7	9.8	53.4	6.7	4.9	44.8	38.9	31.1	37.5
2	61.4	3.5	4.9	5.4	7.2	0	16.3	0.2	2.4	15.0	13.7	12.8	9.1
3	0.3	0.7	1.3	0.7	0.3	0	11.6	0	3.1	1.6	6.8	8.4	8.3
4	0	0.9	0.2	0.5	0.3	0	6.9	0	3.4	1.1	4.8	6.8	1.8
5	0	0.4	0.1	0.9	0.3	0	3.6	0	4.5	1.2	4.5	5.8	0.4
6	0	0.5	0.1	2.8	0.2	0	1.9	0.1	4.7	1.7	4.5	5.2	0.1
7	0	1.0	0.1	2.5	0.2	0	1.2	69.2	5.3	1.6	4.4	4.6	0.1
8	0	6.4	0.1	3.9	0.2	11.1	0.9	7.7	7.0	2.8	4.2	4.1	0.1
9	0	0.6	0.1	3.7	0.2	45.7	0.8	3.1	7.6	2.9	3.8	3.7	16.5
10	0	6.8	0.1	4.9	0.2	7.0	0.7	2.5	7.7	3.6	3.2	3.4	20.6
11	0	6.3	0.1	3.1	0.1	6.0	0.6	2.2	7.9	3.6	2.8	3.2	4.7
12	0	0.1	0.1	3.5	0.1	7.1	0.6	1.6	7.8	4.7	2.4	3.0	0.9
13	0	0	0.1	5.2	0.1	5.4	0.5	1.9	9.4	4.7	2.2	2.8	0.1
14	0	0	0.1	6.9	1.2	4.7	0.5	1.4	11.1	5.0	2.0	2.7	0
15	0	0	0.1	7.0	3.8	3.2	0.5	3.5	13.2	5.8	1.9	2.5	0
P_{ret}	24.6	69.1	8.4	59.0	91.4	13.5	21.2	47.2	5.6	37.5	75.1	84.9	89.8

Set Dueling as done in Qureshi et al. [2007], resulting in the *Dynamic RRIP (DRRIP)* policy. These three policies have been also studied in this work.

3. WHERE LRU PERFORMANCE LOSSES COME FROM?

With LRU, when a block is not being referenced it goes step by step descending the LRU stack. During this *walk*, if it is referenced again, it returns to the MRU position. However, for caches with high associativities, the likelihood of coming back to the MRU position is not uniformly distributed among the locations of the stack. Table I shows this statement for a set of the SPEC2000 benchmark suite [SPEC] and a 1MB-16way L2 cache. Some applications have been skipped following the criteria discussed in Section 6. The P_{ret} row refers to the probability for a block to come back to the MRU position. For instance, $P_{ret} = x$ means that a given block has a likelihood of $x\%$ to return and $(100 - x)\%$ to be evicted before being referenced again. The other rows indicate the probability of being referenced on a given stack location (over 100%). Location 0 is the MRU location and it is not represented, while location 15 refers to the bottom of the stack.

As observed, for most applications, blocks closer to the MRU location (mainly locations 1 and 2) have a high probability to be referenced again. This information is useful to analyze why LRU achieves good or bad performance for each application, as experimental results will show. For instance, LRU will not work well in applications like *art* or *ammp*, since the probability for a block to be referenced again is higher in lower positions (closer to the bottom) than in the middle or upper positions.

Furthermore, these results also show that stack location is not important for most applications in high-associative caches. Nevertheless, the LRU algorithm spends a significant number of bits (e.g., 4-bit in a 16-way cache) to maintain the order of reference for each cache line in the set.

4. MRUT-BASED PREDICTION AND REPLACEMENT

This section presents the MRU-Tour (MRUT) concept and characterizes current benchmarks according to this concept. Then, the family of MRUT-based algorithms is presented, beginning by the simplest one, which will be referred to as the baseline.

Fig. 2. Generation time of the A cache block.

Fig. 3. Number of replacements split into single and multiple MRUTs.

4.1. Overview

The concepts of live and dead times of a block [Wood et al. 1991] have been widely used in cache research. The generation time of a block defines the elapsed time since the block is fetched into the cache until it is replaced. This amount of time can be divided in live and dead times. The live time refers to the elapsed time since the block is fetched until its last access before it is replaced, and the dead time refers to the time from its last access until eviction.

Figure 2 shows the concept of MRUT in the context of the live time of the A cache block. Assuming the LRU replacement policy, block A is initially allocated at time t_1 in the MRU position at the top of the LRU stack. The block maintains this position while it is being accessed. Then, the block leaves this position because block B is accessed. At this point, we say that block A has finished its first MRUT. After accessing block B, block A is referenced again so returning to the MRU position and starting a second MRUT. At time t_2 , block A finishes its third MRUT, which is the last MRU-Tour of this block before leaving the cache at time t_3 .

We also define the MRUT length as the number of times that a block is referenced in a given MRUT. In the example, block A has three MRUTs, with a length of four, three, and two accesses, respectively.

To explore the potential benefits of the MRUT concept on the replacement algorithm, we first split the accessed blocks depending on whether they exhibit a single or multiple MRUTs at the time they are evicted. Figure 3 depicts the results for a 1MB-16way L2 cache under the LRU algorithm. As observed, blocks having a single MRUT dominate those having multiple MRUTs, especially in those applications having a high number of replacements. For completeness reasons, Figure 4 breaks down multiple MRUTs

Fig. 4. Percentage of number of MRUTs under the LRU algorithm.

```

Algorithm: Baseline MRUT
Cache hit in block x:
    if (x is not in the MRU position)
        set the MRUT-bit of x to 1
Cache miss:
    a) select the block to be replaced
        if (there are candidates with MRUT-bit=0)
            randomly among candidates (except the MRU)
        else
            randomly among all blocks (except the MRU)
    b) set the MRUT-bit of the incoming block to 0
End Algorithm

```

Fig. 5. Baseline MRUT algorithm implementation.

according to the number of performed MRUTs. Blocks having a number of MRUTs greater than 4 have been grouped together.

4.2. MRUT-Based Algorithms

MRUT-based replacement algorithms are aware of the number of MRUTs (one or multiple) for each cache line. Figure 5 shows the baseline algorithm. This scheme uses one bit attached to each cache line, referred to as the MRUT-bit, to indicate if the block has experienced one or multiple MRUTs. This control bit is updated each time the block reaches the MRU position during its live time. The algorithm works as follows. Each time a block is fetched into the cache, its associated MRUT-bit is reset to indicate that the 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 a new MRUT starts. This is indicated by setting the MRUT-bit to one. That is, an MRUT-bit value of one indicates that the block had multiple MRUTs, but no additional bits are included to record how many.

This simple algorithm aims to avoid that those blocks with only one MRUT stay in the cache, resulting in a negative impact on performance, by choosing them as candidates for eviction. If a block exhibits a good locality, it will come back to the MRU position, so this block will not be considered for eviction. In this way, this approach acts as a catalyst to expel those blocks accessed during only one MRUT. To make hardware simple, the victim block is randomly selected among the candidate blocks except the MRU block. If there is no block with its associated MRUT-bit cleared, all the blocks

Table II. MRUT Patterns
Numbers in the cells indicate the number of MRU-Tours exhibited during the generation times

Pattern	Generation time								
	1st	2nd	3rd	4th	5th	6th	7th	8th	9th
Regular-1	1	1	1	1	1	1	1	1	1
Regular-n	3	3	3	3	3	3	3	3	3
Irregular-n	5	2	2	5	2	2	5	2	2
Irregular-1-n	1	1	1	8	1	1	1	8	1

(except the MRU one) will be considered as candidates and the victim will be randomly selected among them.

On the other hand, as shown in Table I, storing the order of the last referenced blocks may be also important for performance improvements in most applications. We define the MRUT- x family, which extends the baseline algorithm to exploit both the MRUT behavior and also recency of information. Recency of information is exploited by storing the order of the x last referenced blocks. These x blocks will not be considered as candidates for eviction. For instance, MRUT-2 will not consider as candidates the current MRU block and the following one, while MRUT-1 refers to the baseline MRUT algorithm. Notice that complexity is largely reduced with respect to the LRU policy which keeps the order of all the blocks of the stack.

5. ANALYZING MRUT PATTERNS OF ACCESSED BLOCKS

In order to improve the performance of the MRUT algorithm, we should gather the MRUT patterns of the blocks during their lives, and analyze whether the scheme will be able to capture such behavior or not.

Table II shows a sample of different patterns that we found in the *art* application. Patterns show the number of MRU-Tours exhibited during nine consecutive generation times. Longer patterns (30 generation times) were analyzed, but only a subset is shown for representative purposes. This analysis was carried out in the *art* application because of its low performance in any replacement policy. Patterns' values were gathered with the LRU algorithm. Four different patterns have been identified:

- Regular-1*. Blocks exhibiting this pattern only have a single MRUT during their live time.
- Regular-n*. This pattern is presented by blocks having a constant number of MRUTs but greater than one.
- Irregular-n*. Blocks exhibiting this pattern present a different number of MRUTs during their live times (cyclic or not) but all of them are greater than one.
- Irregular-1-n*. This pattern is exhibited by blocks that present a variable number of MRUTs, some of them equal to one, and some other greater than one.

The proposed algorithm, by design, is able to capture the three first patterns; but it is unlikely that catches the Irregular-1-n pattern. In other words, if some or most of the blocks in a given set have an Irregular-1-n pattern, blocks with a potential number of MRU-Tours might be discarded. In such a case, these blocks would be referenced again soon so incurring in performance losses. Notice that regardless the patterns' mix, the LRU algorithm would work well if the blocks are referenced again before reaching the bottom of the stack. That is, the LRU performance does not depend on the pattern mix but in the number of active blocks that are being referenced by the program.

Nevertheless, the potential performance loss due to the random component of our proposal can be solved by simply adding a small victim cache for the LLC. In this way, evicted blocks are placed close to the LLC, and, if the random strategy fails, these blocks

Fig. 6. Block diagram of the MRUT-3 policy working with a victim cache.

Table III. Machine Parameters

Microprocessor core	
Issue policy	Out of order
Branch predictor type	Hybrid gShare/Bimodal: gShare has 14-bit global history plus 16K 2-bit counters Bimodal has 4K 2-bit counters Choice predictor has 4K 2-bit counters
Branch predictor penalty	10 cycles
Fetch, issue, commit width	4 instructions/cycle
ROB size (entries)	256
# Int/FP ALUs	4/4
Memory hierarchy	
L1 instruction cache	64B-line, 16KB, 2-way, 2 cycles
L1 data cache	64B-line, 16KB, 2-way, 2 cycles
L2 unified cache	128B-line, 1MB, 16-way, 10 cycles
Memory access latency	Unlimited size, 300 cycles

can be quickly returned to the LLC with the MRUT-bit set to one. Figure 6 shows a block diagram of the proposed scheme.

6. PERFORMANCE EVALUATION

This section presents the simulation environment and benchmarks used to evaluate the proposed replacement algorithms. These policies have been modeled on an extended version of the SimpleScalar framework simulator [Burger and Austin 1997]. Experiments have been performed for the Alpha ISA and running the SPEC2000 benchmark suite with the *ref* input set. Statistics were collected simulating 500M instructions after skipping the initial 1B instructions. Table III summarizes the main architectural parameters used through the experiments.

First, we removed the benchmarks that do not stress the L2 cache; that is, those benchmarks where negligible performance benefits can come from any replacement policy. To this end, we obtained the percentage of compulsory misses for each benchmark. Figure 7 shows the results for a 1MB-16way 128B-line L2 cache. Applications having an MPKI less than one or a percentage of compulsory misses higher than 75% were skipped for this study. The MPKI differences showed by all the replacement algorithms analyzed in this work are less than 0.4 for the skipped benchmarks.

6.1. Performance of the Selective Random Strategy Working Alone

This section evaluates the performance of the proposed baseline MRUT algorithm. To this end, its performance is compared against the LRU algorithm. Figure 8 plots the MPKI of both policies. The MRUT algorithm shows, on average, the best results and reduces MPKI by 15% compared to LRU. As observed, MRUT performs much better than LRU in applications showing a high MPKI (see average for MPKI > 10), while it performs a bit worse in applications achieving low MPKI (see average for MPKI < 10).

Fig. 7. Percentage of compulsory misses of an L2 1MB-16way 128B-line cache.

Fig. 8. MPKI of MRUT and LRU algorithms for a 1MB-16way cache.

Two cases are worth to be analyzed in detail. The MPKI in *ammp* is around 68.7 and 49.7 in LRU and MRUT, respectively. The poor performance of LRU in this case can be explained by looking at the results presented in Table I. For instance, in the case of *ammp*, blocks have a high probability to be referenced again when they are on the lower positions of the stack (e.g., from 8 to 15); and, in this case, the LRU is replacing blocks with high probability to be referenced again. The behavior of *art* can be similarly reasoned. In this case, the baseline MRUT reduces the MPKI by 37% compared to LRU.

6.2. Combining the Selective Random Strategy with Recency of Information

Although the MRUT algorithm works well for some benchmarks, the MPKI is a bit worse than the achieved by LRU in some others (e.g., *facerec*). There are several reasons that could explain this behavior. First, baseline MRUT selects the victim among all the blocks having only one MRU-Tour, regardless the recency of information. Second, blocks change dynamically their behavior with time. For instance, a block can have several MRUTs at a given period of time when it exhibits good temporal locality. In such a case, the block will not be considered as candidate for eviction. However, if its locality changes or expires with time, the block will not be considered for eviction (unless all the blocks in the set are in the same conditions). Despite this fact, the block is occupying a cache line although it does not exhibit locality any more.

The first problem is attacked with MRUT- x algorithms, where the last x referenced blocks are not candidates to be replaced regardless their MRUT-bit value. Notice that values of x have been evaluated ranging from 1 to 4 (MRUT-1 refers to the baseline

Fig. 9. MPKI of MRUT- x algorithms varying x from 1 to 4.

Table IV. MPKI of MRUT-3 Varying the Reset Interval

Reset interval	Benchmarks												
	swi.	mgr.	app.	vpr	gal.	art	mcf	fac.	ammp	luc.	bzi.	two.	apsi
32K	11.6	4.9	11.5	1.8	1.2	66.8	16.5	6.0	61.5	8.2	2.0	3.5	2.3
256K	11.6	4.8	11.4	1.8	1.2	59.2	16.0	5.9	59.9	8.2	2.0	3.2	2.4
2M	11.6	5.3	11.2	1.9	1.1	50.5	15.6	6.3	58.9	8.2	2.1	3.1	5.4
8M	11.6	5.5	11.2	2.2	1.1	49.2	15.6	6.8	58.1	8.2	2.2	3.2	6.7
∞	11.6	5.5	11.2	2.3	1.1	48.3	15.6	6.6	51.1	8.3	2.2	3.2	7.3

MRUT algorithm). Figure 9 shows the results. As observed, MRUT-3 shows, on average, a slightly better MPKI than the others. Again, the reason can be explained by looking at the results presented in Table I. Most of the applications show a high probability to reference blocks near the MRU position (e.g., *swim* and *mcf*). Compared to the baseline and LRU, the MRUT-3 algorithm reduces MPKI by 4% and 19%, on average, respectively. Nevertheless, notice that increasing x does not always improve performance. This is the case of *art* and *ammp*. The reason is that the likelihood of accessing one of the last x referenced blocks is quite poor (about 10% for $x = 3$). Thus, if they are not considered for eviction, they can hurt the performance because a block with higher probability of reuse could be wrongly replaced. As opposite, increasing x slightly improves performance, on average, in those applications showing a low MPKI (mainly due to *apsi*). To sum up, there is no evidence that increasing x will improve or damage the performance since MRUT-3 improves, on average, both MRUT-2 and MRUT-4 replacement algorithms.

To deal with the second problem (i.e., block behavior changing with time), the MRUT-bit of each block should be updated at run-time according to its behavior. With this aim, we enhance the MRUT-3 scheme to address two main issues, i) reset the MRUT-bit at regular intervals, and ii) choose the best interval length at run-time for the next execution period.

The first enhancement pursues to clear the MRUT-bit of a block when its temporal locality expires. Different intervals have been evaluated ranging from 32K to 8M number of committed instructions. Table IV shows the MPKI achieved by MRUT-3 for each regular interval length. An infinite-length interval means that no reset is performed. As observed, 256K instructions is the interval that provides, in general, the best performance for those applications showing an MPKI less than 10. On the other hand, for the remaining applications, the performance increases with the interval length.

Fig. 10. MPKI achieved by MRUT-3-adaptive, MRUT-3, and LRU algorithms.

Based on this empirical information, the *reset* algorithm has been extended to be adaptive as follows. The policy resets the MRUT-bits every 256K committed instructions in those workloads with an MPKI less than 10, while MRUT-bits are never reset for the remaining applications. This new policy is referred to as MRUT-3-adaptive. Notice that the number of committed instructions is usually available in the set of performance counters of processors. Thus, storing the LLC misses during a given interval is enough to distinguish if the MPKI is higher or lower than 10. Figure 10 shows the obtained results. The MRUT-3 and LRU algorithms have been plotted for comparison purposes. As observed, MRUT-3-adaptive shows, on average, an MPKI reduction of about 3% and 22% compared to MRUT-3 and LRU, respectively.

Notice that this variation of the proposal provides noticeable MPKI improvements on applications with low MPKI like *mgrid*, *vpr*, *facerec* or *apsi*. Moreover, results are slightly better than LRU, on average, for those applications having an MPKI less than 10, while much better in those exhibiting a higher MPKI.

6.3. Introducing the Victim Cache

This section evaluates the effect of adding a victim cache to the L2 cache working with the MRUT-3-adaptive algorithm. In this way, evicted blocks due to inaccurate random strategy will have another chance to return to L2 without accessing to main memory. It has been assumed a fixed size of 64KB for the victim cache, which is much smaller (sixteen times) than the size of L2 (1MB in our testbed). We have tested both a full-associative and a 32-way victim caches with 3-cycle access time. For comparison purposes, we also analyze the effect of adding the victim cache when using the LRU algorithm in L2. Results of MPKI, hit rate of the victim cache, speedup, and dynamic energy consumption are presented for the analyzed schemes.

Figure 11 shows the MPKI results split into benchmarks with MPKI greater and lower than 10. Remark that, to these MPKI values, the corresponding number of hits in the victim cache have been subtracted. Otherwise, the MPKI values would be roughly the same with and without victim cache. As can be seen, the use of a victim cache helps in reducing the impact of evicting some blocks that are likely to be referenced sooner. These blocks are those exhibiting an Irregular-1-n pattern (see Section 5). In particular, MRUT with a victim cache reduces the MPKI of those benchmarks where LRU obtained better results. This is the case of *lucas*, *twolf* and *apsi*, where the MPKI values obtained by MRUT with victim cache are better than the ones obtained by LRU with victim cache. In some other cases, although the use of a victim cache does not allow MRUT to outperform LRU, the differences are reduced. This is the case of *mgrid* or *bzip2*. On the other hand, as expected, the use of a victim cache also helps

(a) Benchmarks with MPKI greater than 10.

(b) Benchmarks with MPKI less than 10.

Fig. 11. MPKI when using a victim cache.

Fig. 12. Hit rate (%) of the victim cache.

improving the performance of LRU in some benchmarks (*vpr* and *galgel*). Finally, it can be appreciated that it is not necessary a fully-associative victim cache, instead a 32-way cache reaches similar performance with simpler complexity.

Figure 12 shows the hit rate of the victim cache when applying both replacement algorithms in L2. Of course, the applications in which the MPKI is strongly reduced

Fig. 13. Speedup/slowdown (%) of adding a victim cache with respect to strict LRU.

Fig. 14. Normalized dynamic energy (%) with respect to strict LRU.

with a victim cache are those with a high hit rate. For instance, the best hit rate for MRUT is obtained for *lucas* (more than 10%), which obtains a reduction in MPKI (see Figure 11) from > 8 to 6. For LRU, the best hit rate corresponds to *galgel*, improving MPKI from 4 to < 0.5 .

An interesting observation is the impact of associating a victim cache to L2 in performance. Figure 13 shows the speedup (or slowdown) of the analyzed schemes compared to strict LRU. As observed, although the addition of a victim cache allows improving LRU, this effect is more noticeable in MRUT, which, on average, obtains the best performance. Taking into account that the MRUT replacement algorithm is simpler to implement (see Section 7), these results demonstrate that the MRUT-3-adaptive replacement algorithm combined with a victim cache is an efficient replacement for LRU in LLCs.

Finally, we quantified the dynamic energy consumed by the analyzed replacement algorithms. We used the CACTI tool [Thoziyoor et al. 2008a, 2008b] to compute the dynamic energy per access for each memory structure. We assumed a 1MB-16way SRAM LLC, a 64KB-32way SRAM victim cache, and a 1GB DRAM main memory. Main memory energy has been taken into account to estimate the energy costs of satisfying a miss in both the LCC and the victim cache. Results provided by CACTI were used with the number and type of memory operations measured during the benchmark execution to calculate the total dynamic energy of each algorithm.

Fig. 15. MPKI achieved by MRUT-3-adaptive, pseudo-LRU, LRU, and other recent policies (DRRIP, BIP, and DC-Bubble).

Figure 14 depicts the normalized dynamic energy for the analyzed algorithms with respect to strict LRU without victim cache. Regarding the victim cache exempt policies, a miss in the LLC means accessing the power-hungry main memory. Thus, the MRUT-3-adaptive policy achieves more energy savings than LRU in those applications where LLC misses are reduced. The proposed policy reduces dynamic energy by 3% compared to LRU. On the other hand, the schemes with victim cache consume less energy than those without victim cache, since a hit in the victim prevents main memory from being accessed. The LRU policy using a victim cache obtains energy savings by 6% on average, whereas this reduction is by 8% in the proposal.

6.4. Comparing MRUT with Recent Approaches from the Literature

This section evaluates the performance of MRUT-3-adaptive without victim cache against a set of the most representative state-of-the-art approaches such as Bubble [Zhang and Xue 2009], adaptive insertion policies [Qureshi et al. 2007], and Reference Interval Prediction (RRIP) [Jaleel et al. 2010]. Results for both strict and pseudo-LRU algorithms are also shown in this study. A binary tree-based variant of the pseudo-LRU algorithm has been implemented. Since most recent approaches usually propose several adaptive variants, we tuned the algorithms for our testbed and chose, for comparison purposes, the best performing variant of each proposal. In particular, the Bubble algorithm uses the *divide-and-conquer technique* (DC) and 4-block groups; the best adaptive insertion policy is BIP with $\epsilon = \frac{1}{32}$; and the 3-bit DRRIP HP with 32-entry SDMs, 10-bit PSEL counter, and $\epsilon = \frac{1}{32}$ is the best cache replacement with RRIP.

Figure 15 shows the results. As observed, LRU performs slightly better than pseudo-LRU for some benchmarks such as *facerec*, *lucas*, and *twolf*. However, both policies achieve, on average, almost the same MPKI. On the other hand, the other recently proposed approaches perform better than LRU on average, and, like MRUT-based algorithms, the major benefits are in those applications showing a high MPKI, whereas the MPKI remains almost the same, on average, for applications showing an MPKI less than 10. Nevertheless, the MRUT-3-adaptive is the policy achieving more MPKI reduction, on average. In particular, it reduces MPKI by 10%, 11%, and 11% compared to DRRIP, BIP, and DC-Bubble, respectively.

7. HARDWARE COMPLEXITY

This section analyzes the hardware complexity in terms of area for the studied policies. In addition, results for the counter-based predictor proposed in Kharbutli and Solihin [2008] (AIP and LvP approaches) are presented. We assumed that area overhead is mainly dominated by the control bits and additional hardware structures (e.g., tables and counters) required to implement the replacement/placement strategies.

The strict LRU replacement algorithm requires $\log_2(n)$ bits per cache block (LRU counters) to maintain the order of the blocks of the LRU stack in an n -way set-associative cache. That results in $\log_2(n) \times n$ control bits per cache set. For instance, in a 16-way cache, up to 4 bits per block and 64 bits per set are required. In addition, keeping the order of all the stack requires a circuitry to update the counters, which is expensive in terms of area. In contrast, the pseudo-LRU scheme implemented with a binary tree only uses $n - 1$ bits per set. This is the studied algorithm requiring less control bits.

Concerning the counter-based cache replacement algorithms, the LvP approach saves area with respect to AIP, since the former requires 17 control bits per cache block instead of 21 that uses the latter scheme. However, both approaches have a hardware overhead larger than the required by strict LRU. In addition, both policies use a prediction table to store the counter values of the victimized blocks. This table has 256×256 entries, each one storing 5 bits, resulting in an area overhead of 40KB.

Regarding the Bubble algorithm, it requires as many control bits as the LRU, since the entire stack order must be maintained. In addition, a bit per set is required to indicate whether the previous access to the set resulted in a hit or in a miss. Splitting the set in groups helps Bubble to reduce the number of control bits. DC-Bubble requires $\log_2(n/g)$ control bits per block, where g is the number of groups in each set. For example, in a 16-way cache, each 4-block group requires 2 control bits per block. Like Bubble, it also uses the aforementioned bit per set to record the previous access result.

In the adaptive insertion policies, the number of control bits of each policy is at least equal to the number required by LRU since they are based on it. LIP does not require more control bits since the only difference with respect to LRU is that the incoming block is placed in the LRU position and then promoted to the MRU. The BIP policy requires as many control bits as LIP, plus a 5-bit counter that is increased on each cache miss to indicate which incoming block is inserted in the MRU position. Finally, the DIP algorithm requires the same amount of bits as BIP, plus a 10-bit saturating counter to implement the *duel* between the competing algorithms.

In contrast to the adaptive insertion policies, the y -bit SRRIP policy uses y control bits per block regardless the number of ways. On the other hand, the y -bit BRRIP algorithm requires as many control bits as y -bit SRRIP and, similarly to BIP, a 5-bit counter. Finally, the y -bit DRRIP policy requires as many control bits as y -bit BRRIP and, analogously to DIP, a 10-bit counter to choose between SRRIP and BRRIP. For example, when $y = 2$, SRRIP, BRRIP, and DRRIP require 32, 37, and 47 control bits per set, respectively, whereas they require 48, 53, and 63 bits, respectively, when $y = 3$.

The baseline MRUT algorithm reduces the hardware complexity compared to the policies analyzed above, since it only requires 2 bits per block regardless the cache associativity: the MRUT-bit and an extra bit that indicates the MRU block (MRU-bit). In addition, by randomly selecting the victim among those candidate blocks, circuit complexity is largely simplified, not only compared to the LRU policy but also, to the best of our knowledge, with any recent proposal. On the other hand, the MRUT- x family of replacement algorithms maintains the order of the last x referenced blocks using z bits. A special combination of these bits is also used to register that the block order is not kept for the remaining $n - x$ blocks. Thus, $z = \lceil \log_2(x+1) \rceil$ MRU-bits per

Table V.
Area overhead (in number of control bits) in an n -way cache (per set) and a 1MB-16way cache for the studied algorithms

Algorithm	n -way cache	1MB-16way cache
LRU	$\log_2(n) \times n$	32Kb
pseudo-LRU	$n - 1$	7.5Kb
AIP	$21 \times n$	488Kb
LvP	$17 \times n$	456Kb
DC-Bubble	$\log_2(n/g) \times n + 1$	16.5Kb
BIP	$\log_2(n) \times n$	32Kb + 5b
3-bit DRRIP	$3 \times n$	24Kb + 15b
Baseline MRUT	$2 \times n$	16Kb
MRUT-3-adp.	$3 \times n$	24Kb

block are required. For instance, MRUT-2 and MRUT-3-adaptive policies need 3 control bits per block (2 bits for the block order and the MRUT-bit).

Table V summarizes the area overhead (in number of control bits) per cache set for the analyzed policies in an n -way cache. Results for the 1MB-16way cache account the total control bits (in Kbits) for the whole cache. As in the previous section, only results for the best variant of each proposal in terms of performance are shown.

As mentioned above, the binary tree implementation allows pseudo-LRU to be the cheapest algorithm in terms of area. However, the performance of this algorithm might be even worse than strict LRU in LLCs. The DC-Bubble is the policy, among the recent proposals, which most reduces the number of control bits due to it applies the *divide-and-conquer* technique, which considers each set divided in several groups (i.e., only the order in the group needs to be maintained). Nevertheless, the baseline MRUT algorithm obtains even better reduction in area in spite of considering the whole cache sets. Finally, only keeping the recency of information for three blocks is quite enough to occupy smaller area than LRU, AIP, LvP, BIP, and DRRIP policies.

8. CONCLUSIONS

Strict LRU is not being commonly implemented in LLCs of commercial processors because of its hardware complexity, and pseudo-LRU algorithms are the norm since they reduce such complexity but their performance deviate from the strict LRU policy. In this article it is shown that, combining recency of information, but only for just a few blocks, with a selective random strategy, hardware complexity can be hugely reduced while even improving the performance of the strict LRU policy and a set of the most representative state-of-the-art recent proposals.

The random strategy presented in this article is based on the concept of MRU-Tour (MRUT) of a block, which is related to the number of times that a block occupies the MRU position. This strategy was devised based on empirical observations, where we found that most of the blocks have a single MRUT when they are replaced. Variants of this scheme have been also studied aimed at exploiting the recency of information in the last x ($x \leq 3$) referenced blocks as well.

Evaluation results showed that combining recency of information with the selective random strategy, MPKI can be reduced by 15% on average over LRU (for $x = 1$). This reduction is by 25% for applications with a high number of misses (MPKI > 10), and up to 37% in some of these applications. For $x = 3$ the MPKI reduction is up to 19% compared to LRU. Since workloads change their behavior with time and blocks with multiple MRUTs may occupy the cache for long until they are evicted, an adaptive version of the MRUT replacement algorithm was devised. It periodically resets the MRUT-bit in those applications achieving an MPKI lower than 10. Results showed that, on average, the MPKI is reduced by 22% with respect to LRU. Compared to a

set of the most representative approaches, MPKI reduction falls in between 10% and 11%. Finally, in order to palliate possible inaccuracy of the random strategy in specific workloads, the effect of adding a small victim cache has been also studied. This cache has been already proven to be effective in L1 caches; however, when adding this cache to the LLC it can act even in a more effective way since the performance of both applications with large MPKI (usually due to large working sets) or applications with low MPKI (usually having low working sets but temporality is not well exploited by replacement algorithms) are improved. Regarding the proposal, this cache reduces the MPKI of the MRUT in those workloads with low MPKI values where LRU works better, although, average values fall close in both schemes.

Regarding speedup improvements, the MRUT-3—adaptive with victim cache is the one that achieves the best performance results. Speedup improvements reached by this scheme can be as high as 30% for particular benchmarks and, on average, by 4% and 2% with respect to LRU and LRU with victim cache.

This work has also evaluated the energy consumption of the MRUT algorithms. Experimental results showed that the best version of MRUT policy achieves dynamic energy savings up to 3% with respect to LRU. When applying this policy together with the victim cache, the reduction is up to 8%. In addition, the MRUT algorithms have been proven as low-cost approaches in terms of area requirements. Hardware complexity is largely reduced when using the baseline MRUT policy compared to strict LRU and the other existing proposals.

Finally, identifying sharing patterns and evaluating the potential of the MRUT concept in CMP systems is planned as for future work.

REFERENCES

- BAER, J.-L. 2010. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press.
- BASU, A., KIRMAN, N., KIRMAN, M., CHAUDHURI, M., AND MARTÍNEZ, J. 2007. Scavenger: A new last level cache architecture with global block priority. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 421–432.
- BELADY, L. A. 1966. A study of replacement algorithms for virtual-storage computer. *IBM Syst. J.* 5, 2, 78–101.
- BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar Tool Set, Version 2.0. *ACM SIGARCH Comput. Archit. News* 25, 13–25.
- CHAUDHURI, M. 2009. 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*. ACM, New York, NY, 401–412.
- DYBDAHL, H., STENSTRÖM, P., AND NATVIG, L. 2007. An LRU-based Replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. *ACM SIGARCH Comput. Archit. News* 35, 45–52.
- JALEEL, A., THEOBALD, K. B., STEELY, JR., S. C., AND EMER, J. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ACM, New York, NY, 60–71.
- JIANG, S. AND ZHANG, X. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, New York, NY, 31–42.
- JOHNSON, T. L., CONNORS, D. A., MERTEN, M. C., AND HWU, W.-M. W. 1999. Run-time cache bypassing. *IEEE Trans. Comput.* 48, 1338–1354.
- KALLA, R., SINHARROY, B., STARKE, W. J., AND FLOYD, M. 2010. Power7: IBM's next-generation server processor. *IEEE Micro* 30, 7–15.
- KHARBUTLI, M. AND SOLIHIN, Y. 2008. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Comput.* 57, 433–447.
- LIN, W.-F. AND REINHARDT, S. K. 2002. Predicting last-touch references under optimal replacement. Tech. Rep. CSE-TR-447-02, University of Michigan.

- LIU, H., FERDMAN, M., HUH, J., AND BURGER, D. 2008. Cache Bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 222–233.
- PETOUMENOS, P., KERAMIDAS, G., AND KAXIRAS, S. 2009. Instruction-based reuse-distance prediction for effective cache management. In *Proceedings of the 9th International Symposium on Systems, Architectures, Modeling, and Simulation*. IEEE, 49–58.
- QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY, S. C., AND EMER, J. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, New York, NY, 381–391.
- RIVERS, J. A., TAM, E. S., TYSON, G. S., DAVIDSON, E. S., AND FARRENS, M. 1998. Utilizing reuse information in data cache management. In *Proceedings of the 12th international conference on supercomputing*. ACM, New York, NY, 449–456.
- SMITH, A. J. 1982. Cache memories. *ACM Comput. Surv.* 14, 473–530.
- SPEC. Standard Performance Evaluation Corporation. <http://www.spec.org/cpu2000>.
- STACKHOUSE, B., BHIMJI, S., ET AL. 2009. A 65 nm 2-billion transistor quad-core itanium processor. *IEEE J. Solid-State Circ.* 44, 1, 18–31.
- THOZIYOOR, S., AHN, J. H., MONCHIERO, M., BROCKMAN, J. B., AND JOUPPI, N. P. 2008a. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. IEEE, 51–62.
- THOZIYOOR, S., MURALIMANOVAR, N., AHN, J. H., AND JOUPPI, N. P. 2008b. CACTI 5.1. Tech. Rep. Hewlett-Packard Development Company, Palo Alto, CA.
- TYSON, G., FARRENS, M., MATTHEWS, J., AND PLESZKUN, A. R. 1995. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. IEEE, 93–103.
- WONG, W. A. AND BAER, J.-L. 2000. Modified LRU policies for improving second-level cache behavior. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*. IEEE, 49–60.
- WOOD, D., HILL, M. D., AND KESSLER, R. E. 1991. A model for estimating trace-sample miss ratios. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, New York, NY, 79–89.
- ZHANG, C. AND XUE, B. 2009. Divide-and-Conquer: A bubble replacement for low level caches. In *Proceedings of the 23rd International Conference on Supercomputing*. ACM, New York, NY, 80–89.

Received July 2011; revised October 2011, November 2011; accepted March 2012