Improving PeLIFO Cache Replacement Policy: Hardware Reduction and Thread-Aware Extension

Abstract

Studying blocks behavior during their lifetime in cache can provide useful information to reduce the miss rate and therefore improve processor performance. According to this rationale, the peLIFO replacement algorithm [1], which learns dynamically the number of cache ways required to satisfy short-term reuses preserving the remaining ways for long-term reuses, has been recently proposed. In this paper, we propose several changes to original peLIFO policy in order to reduce the implementation complexity involved, and we extend the algorithm to a shared-cache environment considering dynamic information about threads behavior to improve cache efficiency. Experimental results confirm that our simplification techniques reduce the required hardware with a neglegible performance penalty, while the best of our thread-aware extension proposals reduces average CPI by 4.9% and 10.5% on average compared to original peLIFO and LRU respectively for a set of 35 multi-programmed workloads on an 8MB 16-way set associative shared L2 cache.

Key words: Cache memories, replacement policies, peLIFO, miss rate

1. Introduction

The great gap between processor and memory speed, which continues to increase nowadays, has forced researchers to develop solutions to mitigate this problem. During the last few years many techniques have been proposed, ranging from processor level, aiming for memory instructions to generate no stalls –like dependence prediction [2], value prediction [3], address prediction [4], etc– to DRAM level, trying to schedule accesses to improve the global performance [5]. In between there is cache level, with many proposals in literature attempting to improve its management. Examples of these proposals are prefetching techniques and non-blocking caches, which try to overlap cache misses with previous or subsequent independent instructions respectively [6].

One of the main methods for reducing the miss rate of an N-way set associative cache is through an efficient cache replacement policy: when the insertion of a new block involves an eviction, the policy decides which block is to be replaced. In general, as Belady established in [7], the best decision is to replace the block that will not be referenced again for the longest time. Since knowing the future is for now still impossible, the different policies proposed in literature try to predict which one is such block, based on past behavior analysis.

One of the most extended replacement policies is the well-known Least Recently Used policy (LRU), which discards the least recently used block, under the philosophy that, due to temporal locality, it is also the block that will not be required for the longest time in the future. This policy behaves satisfactorily for many applications, but exhibits a malfunction for others: when a datum is used only once, or will not be referenced again for a long time, it should be evicted from the cache as soon as possible; however, the LRU policy retains the corresponding block during its traversing

from MRU (Most Recently Used) to LRU positions. Besides, for workloads with a working set larger than the available cache size, this policy could move on to a pathological behavior.

In the last few years many policies have emerged trying to overcome these and other problems [1, 8, 9, 10, 11, 12]. This work extends the proposal from [1]. According to the authors and our own results, this proposal improves LRU and most of the latest cache replacement policies appeared in literature. It is called Probabilistic-Escape-LIFO (peLIFO), and it is based on the Last In First Out policy (LIFO). LIFO maintains a Fill Stack for each set to keep the block insertion order[1]. When a replacement is required, LIFO selects as victim the most recently inserted block. The authors observed that for most applications the number of short-term reuses is over one for many blocks, which means that a newly arrived block should not be evicted (as LIFO incorrectly does), since it will likely be referenced again soon. However, the number of short-term reuses is on average much lower than the associativity of the cache, so LRU may maintain the blocks in the cache for too long. Considering both properties, peLIFO tries to learn dynamically the amount of ways required to satisfy the short-term reuses, preserving the remaining ways for long-term reuses. Taking advantage of the Fill Stack, peLIFO confines the replacement activity to the positions close to the top of the stack –learning dynamically the optimal exact position to perform a replacement– and keeps the blocks from the bottom area of the stack untouched.

Our first goal in this paper is to improve the original peLIFO by reducing the implementation complexity without penalizing performance. The second objective consists in improving peLIFO behavior in the shared cache level of a 4-way multiprogrammed environment, extending it to dynamically collect information about each thread nature and take that information into consideration. peLIFO has been already evaluated in such an environment with satisfactory results [1]; nevertheless, it did not take into account any kind of information about threads properties. As we show in the evaluation section, making the policy thread-aware improves performance.

The rest of the paper is organized as follows: Section 2 reviews the original peLIFO replacement policy. Sections 3 and 4 detail our proposals to reduce peLIFO complexity and to efficiently implement the policy in a shared cache environment respectively. Section 5 describes the experimental setup employed, Section 6 gathers experimental results and analyses. Section 7 recapitulates related work. Finally, Section 8 concludes.

2. PeLifo

In this section we summarize the operation of the original peLIFO policy [1] in which our work is based. The authors start from a LIFO replacement policy: using a Fill Stack, the block that entered the cache in the last place is the candidate for replacement. They propose several optimizations to improve its poor performance. The key idea behind peLIFO is to keep the bottom part of the Fill Stack for long-term reuses as LIFO does. Figure 1 illustrates the difference in operation: (a) LIFO performs all replacements at the head of the stack, (b) peLIFO dynamically selects an intermediate position that guarantees short-term reuses to be fulfilled. The policy is built around two main concepts:

1. *Escape Probability:* The *Escape Probability* for the k^{th} position of the Fill Stack is defined as the probability that cache blocks experience hits at Fill Stack positions bigger than k.

[1] Position 0 contains the most recently inserted block, and is called top/head of the stack.

2. *Escape Point (*EP*):* It is the position in the Fill Stack where the *Escape Probability* decreases below a certain threshold when compared to that of the previous position. Blocks placed in this position become suitable candidates to leave the cache when a replacement occurs, since from then on, they will probably experiment no more short-term reuses.

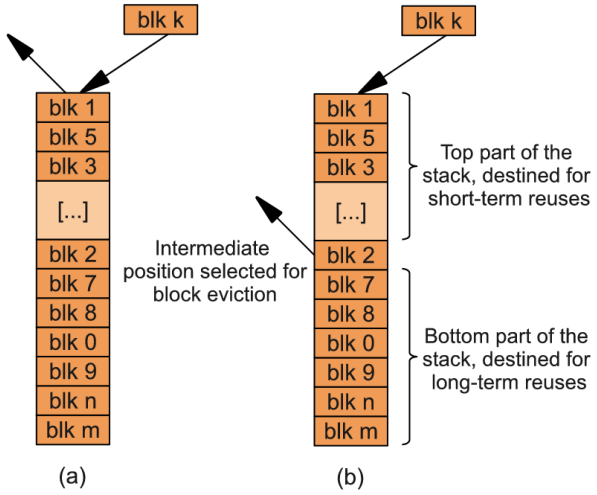

Figure 1: Basic replacement operation in LIFO (a) and peLIFO policies (b).

PeLIFO computes the *Escape Probabilities* by employing several arrays that are dynamically updated according to the Fill Stack positions in which hits take place. Specifically, every N^{th} refills to the cache, these arrays are processed and the *Escape Probabilities* and the *EP*s are calculated (details of the whole process can be found in [1]). Among all the *EP*s obtained, only those 3 closest to the top of the Fill Stack are selected.

Based on these 3 computed *EP*s, replacement decisions are taken. Particularly, the authors use four competing policies, one per escape point, denoted P_1, P_2 and P_3, and P_4, that corresponds to a simple LRU. The policy P_i (for $i \in \{1, 2, 3\}$), victimizes the block closest to the top of the Fill Stack that satisfies the following two criteria:

- Its current Fill Stack position is greater than or equal to the i^{th} *Escape Point*.
- The block has not experienced a hit in its current Fill Stack position.

In order to dynamically select one policy among P_1, P_2, P_3 and P_4, a Set Dueling mechanism is employed. It dedicates a small number of sample sets to each policy just for collecting statistics [9]. Based on this information, the "best" policy is applied for all the non-sampled sets. More details can be found in [1].

As most applications go through different stages along execution, the *Escape Probabilities* and the *EP*s must be recalculated periodically. For this purpose, the authors define the concepts *epoch* and *phase*: an *epoch* is determined as the elapsed time between two *Escape Probabilities* computations, i.e. every N refills to the cache (in [1] $N = 2^{13}$). The current *Escape Probabilities*

are compared with those corresponding to the previous *epoch*. If the difference exceeds a certain threshold, then a *phase* change occurs and the *EP*s are recomputed.

Figure 2 illustrates the basic operation of peLIFO. Inside a *phase*, the first *epoch* always employs the LRU, thus giving the low part of the Fill Stack a chance to get flushed out so that the new cache blocks accessed in the starting *phase* can be brought in. Once this first *epoch* ends, 3 new *EP*s for the current *phase* are calculated and the cache starts to use peLIFO until a *phase* change is detected; at this point an LRU *epoch* starts again and the whole process repeats.

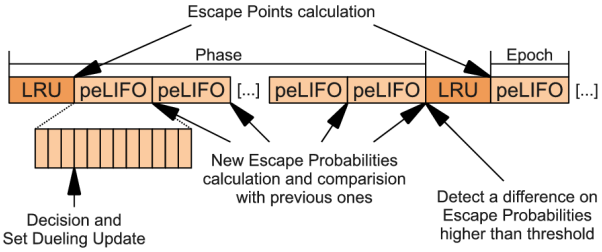

Figure 2: PeLIFO algorithm.

2.1. Implementation Overhead

A new replacement policy usually comes with several types of overhead, namely, extra storage, extra logic, and impact on the critical path. Moreover, the new policy affects the chip power consumption, but this issue exceeds the goal of the paper so we do not consider here the associated power impact.

Extra Storage:. In peLIFO the *Escape Probabilities* and *EPs* calculation are expensive processes both in terms of algorithm complexity and hardware resources involved. Specifically, they require some structures to record the current and previous *Escape Probabilities* values (referred to as *epCounter* and *EpCounterLastEpoch* respectively) as well as to store the last hit position for each block inside the cache (denoted as *lastHitPosition*).

Table 1 recaps the extra hardware that peLIFO implementation requires compared to a conventional LRU policy, where A is the cache associativity, N the amount of sets, epCS the *epCounter* counters size, SDS the Set Dueling counters size and EES the *end of epoch* counter size [2]. According to this table, peLIFO adds around 18KB to the L2 (2MB) of Memory-System 1 (see specific configurations detailed in Section 6), and around 72KB to the L2 (8MB) of Memory-System 2, which means that the storage overhead with respect to LRU, in terms of extra bits required, is lower than 1%.

Algorithm Complexity:. In this section, we state in detail the new tasks that the algorithm must deal with compared to LRU, providing the reader a rough idea about the additional logic that would be required to support the mentioned tasks.

[2]Here we assume for simplicity that the *Escape Probabilities* are learned for the whole cache (a finer granularity would imply more *epCounter*, *epCounterLastEpoch*, Set Dueling Counters, etc.).

	Bits in a general configuration	*Bits in Memory-System 1*	*Bits in Memory-System 2*
Fill Stack	A*log_2A*N	16*4*1024	16*4*4096
LastHitPosition	A*log_2A*N	16*4*1024	16*4*4096
Hit in current Fill-Stack position?	A*1*N	16*1*1024	16*1*4096
epCounter and epCounter-LastEpoch	A*epCS	16*20	16*20
Escape Points	3*A	3*16	3*16
Set Dueling Counters	6*SDS	6*30	6*30
End of Epoch Counter	1*EES	1*13	1*13

Table 1: peLIFO storage overhead with respect to LRU.

- *Upon each hit:*
  1. *epCounter* updating: it implies the reading of *LastHitPosition* of the involved block (namely i), as well as the reading of its current Fill Stack position (namely j); then *epCounter*[i] to *epCounter*[j-1] are incremeted.
  2. *LastHitPosition* of the block is updated to its current position.
  3. The bit that identifies a hit in the current position (*Hit in current Fill Stack position*) is set to '1' for this block.

- *Upon each miss:*
  4. The Fill Stack corresponding to the set in which the miss occurs must be updated. Also, the *Hit in current Fill Stack position* bit must be reset for every block that varied its position in the Fill Stack.
  5. If a replacement is required, the candidate block is chosen: starting from the *EP* considered, the first block with the hit bit set to '0' is selected.
  6. *End of epoch* counter updating.
  7. If the involved set belongs to the pool of those reserved for Set Dueling:
     (a) Update the 6 Set Dueling counters.
     (b) Perform the corresponding 6 comparisons, and update the winner policy accordingly.

- *When an epoch ends:*
  8. *epCounter* processing in order to calculate the new *Escape Probabilities*.
  9. Perform a comparison between *epCounter* and *epCounterLastEpoch* for detecting a *phase* change.
  10. Reset the *End of Epoch* counter.

- *When a phase ends:*
  11. The new *EPs* are computed.
  12. Reset the Set Dueling counters.

Impact on the Critical Path:. The proposed policy does not impact the critical path. All the updating processes required when a hit occurs (*epCounter*, *LastHitPosition*, *Hit in current Fill Stack position*) may be performed off the critical path. Besides, even if we assume that these operations are carried out on the critical path, they are fast enough for overlapping them with the data array access after the hit is detected in the tag array. The required tasks when a miss occurs are slightly more complex. Still, like in the hit case, they can be performed off the critical path, and even assuming the contrary, again there would be time enough to overlap them with the miss handling, which is much slowler than the hit handling, since we must search the next level of the hierarchy in order to obtain the corresponding data.

3. PeLIFO simplification techniques

As the authors show in [1], peLIFO outperforms most of the other algorithms it is compared to. However, this is obtained at the cost of a complex and resource-expensive hardware implementation. Therefore, strategies that aim to reduce the implementation overhead of the policy, with a minimal performance degradation, are desired. In this section we propose two different approaches to achieve this goal.

3.1. Fixed Escape Points*: peLIFO-fixEP*

An analysis of the *EP*s usage per program reveals that a major percentage of the selections concentrate on very reduced sets of *EP*s. According to [1], 13 out of the 14 benchmarks employed select, in the 74% of the *phases*, among just 3 of all the 16 possible *EP*s (a 16-way set associative cache is employed). Moreover, for 8 of the benchmarks, this percentage rises above 90%.

The first of our proposals is maintaining the same 3 *EP*s unchanged during the whole execution, not recalculating them at the beginning of each *phase*. The 3 points, as well as the duration of a *phase*, are determined through profiling. We set the 3 *EP*s to be those positions of the Fill Stack where the largest number of replacements take place during the profiling. Since the LRU *epoch* at the beginning of each *phase* is used in peLIFO to feed the cache with data for determining the *Escape Points* for the starting *phase*, we can also avoid it. Figure 3 shows how the resultant algorithm works.

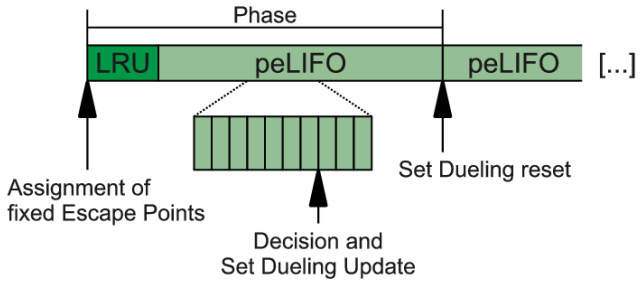

Figure 3: peLIFO-fixEP algorithm.

When an application starts, the 3 *EP*s to be used are set and an LRU *epoch* begins in order to warm-up the cache. Then, the system switches to peLIFO-fixEP until the application ends. When

a replacement is required, the *Escape Point* selection and the Set Dueling mechanism updating are done as in peLIFO. Specifically, the counters used for Set Dueling are reset at the end of each *phase*. This way, we manage to remove all the extra storage required for *Escape Probabilities* and *EP*s calculations, and also simplify the logic.

3.2. Coarse grain decision: peLIFO-CG

Although the aforementioned proposal achieves considerable simplifications at the expense of a small performance degradation, compared to original peLIFO (see evaluation section), peLIFO-fixEP requires an undesired profiling step that could be avoided. To this end, we introduce a second technique, denoted as peLIFO-CG, which implements further simplifications, delivering a very similar performance as peLIFO-fixEP without needing a profiling step.

When a memory reference incurs in a cache block eviction, peLIFO decides which policy must be followed (recall that 4 possibilities are examined: P_1, P_2, P_3, corresponding to the 3 used *EPs*, and P_4 corresponding to LRU). We have observed that for most of the applications, the Set Dueling victor is the same policy for long periods of time, that may even span several *epochs*. Figure 4 shows the policy selection for a random region in 171.swim (from the SPEC CPU2000 suite [13]). During some intervals (e.g. from replacements 0 to 379160) the selection oscillates rapidly among several policies (specifically peLIFO with *EPs* 0, 1 and 2). However, for other periods (e.g. from replacements 379160 to 1440453) the selection remains completely unchanged (peLIFO with EP=1). Thus, we conclude that such a fine grain policy selection may not be necessary. Instead, peLIFO-CG chooses just one policy, making all the replacements based on this decision for a fixed (long) time lapse, and periodically revises the selected policy.

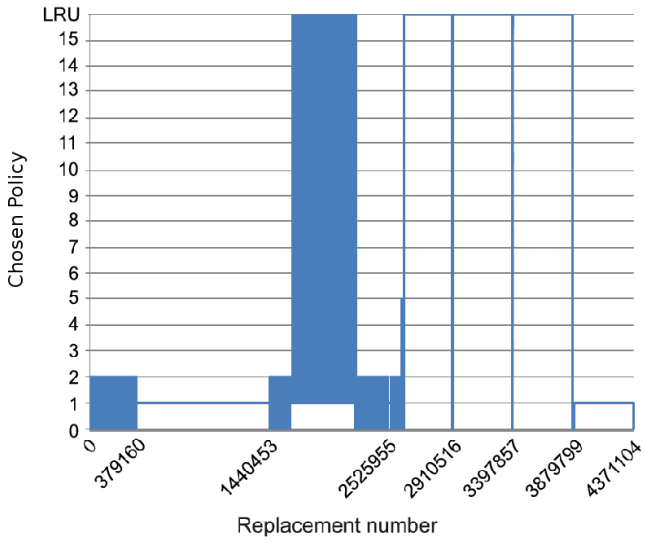

Figure 4: *Escape Points* chosen in 171.swim.

For peLIFO-CG, we split each *phase* into two different stages (see Figure 5). First, we execute a *training stage* consisting of several *epochs*, each using peLIFO with a different fixed *EP* and one more *epoch* employing LRU. We employ hit rate (ratio between cache hits and cache accesses) for

evaluating each policy[3]. The policy exhibiting the highest hit rate is selected to be used along the second stage (*production stage*) of the *phase*. In our simulations we train all the 16 possible *EPs* –L2 cache associativity is 16– leading to a *training stage* of 17 *epochs*. Then, the selected policy is employed during the next 128 *epochs* (this number is empirically determined, the experiment is detailed in Section 7.3.2). After that time, a new *phase* starts and a new policy is selected based on the information extracted from a new *training stage*. It is worth noting that, as shown in Figure 5, the first *training stage* includes an LRU *epoch* at first, aiming to prevent compulsory misses to unfairly degrade the hit rate delivered in the first *EP* evaluation *epochs*. Subsequent *phases* do not require this initial LRU *epoch*, since the effect of compulsory misses becomes negligible.

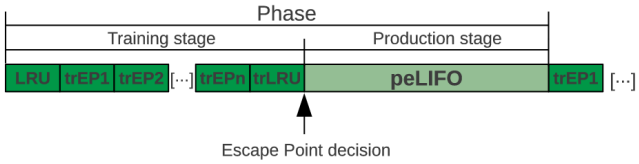

Figure 5: PeLIFO-CG algorithm.

3.2.1. Implementation Overhead

As done in the previous section for original peLIFO, in this section we detail the peLIFO-CG overhead (the peLIFO-fixEP overhead analysis would be analogous) with respect to LRU and we perform a comparison with the overhead of original peLIFO.

Extra Storage:. Table 2 recaps the additional hardware that peLIFO-CG requires compared to LRU, being the size of the *end of training and production stages* counter denoted as ETPS, and the size of the counters recording hits and accesses within each *training stage* denoted as HS and AS respectively. As shown, compared with peLIFO, we manage to remove *epCounter*, *epCounterLastEpoch* and *lastHitPosition*, as well as the Set Dueling mechanism and 2 of the 3 *EPs*. On the contrary, we need to include some additional counters (*Amount of Hits*, *Amount of Accesses*, and *End of Training and Production Stages*). Overall, according to this table, peLIFO-CG adds around 10KB to the L2 (2MB) of Memory-System 1, which means that the storage overhead with respect to LRU, in terms of extra bits required, is lower than 0.5%.

Algorithm Complexity:. In the following we summarize the new tasks that peLIFO-CG must handle compared to LRU. As we detail, we manage to remove all complex logic related with *Escape Probabilities* and *EP*s calculation that original peLIFO requires, as well as the logic associated with the Set Dueling mechanism (i.e. tasks 1, 2, 7, 8, 9, 11 and 12 from Section 3.1). On the contrary, we add some simple tasks for supporting the new policy (tasks 6 to 9).

- *Upon each hit:*
  1. The only required task here is to set to '1' the bit that identifies a hit in the current position (*Hit in current Fill Stack position*) for the involved block.

[3]In order to allow the cache to warm-up for the evaluated policy, in each *training epoch*, data required to calculate the ratio is collected just in the final part of the *epoch*.

	Bits in a general configuration	*Bits in Memory-System 1*
Fill Stack	A*log_2A*N	16*4*1024
Hit in current Fill-Stack position?	A*1*N	16*1*1024
Escape Point	1*A	1*16
Amount of Hits Counter	1*HS	1*30
Amount of Accesses Counter	1*AS	1*30
End of Train. and Prod. Stages Counter	1*ETPS	1*7

Table 2: peLIFO-CG storage overhead with respect to LRU.

- *Upon each miss:*
  2. The Fill Stack corresponding to the set in which the miss occurs must be updated.
  3. If a replacement is necessary, the candidate block is chosen: starting from the proper *EP*, the first block with the corresponding hit bit set to '0' is selected.
  4. The *End of Epoch* counter is updated.
- *When an epoch ends:*
  5. Reset the *End of Epoch* counter.
  6. The *end of training and production stages* counter is incremented.
- *When a training stage ends:*
  7. The *Amount of Hits* and *Amount of Accesses* counters are reset.
  8. The *end of training and production stages* counter is reset.
- *When a phase ends:*
  9. The *end of training and production stages* counter is reset.

Impact on the Critical Path:. Applying the same reasoning detailed in section 3.1, and given that the algorithm complexity is considerably lower than in peLIFO, again the policy has no impact on the critical path.

4. Extending peLIFO to shared caches

In this section we introduce an extension of peLIFO focused on the shared cache of a multi-core processor environment. A shared cache accomodates blocks from different threads, each with its own access pattern. We propose modifications to peLIFO, leveraging information about each thread nature, with the objective of devoting more cache resources to those threads that are using it more efficiently, thus improving the global hit rate in the shared level of the cache hierarchy and system performance.

Particularly, we propose two different changes to the original policy. The underlying idea in both approaches is evicting from the shared cache those blocks belonging to threads that are not taking advantage of long-term reuses. This way we make space available for accommodating blocks from threads that do take advantage of that kind of reuses. As the original single-core policy tries to dedicate the bottom part of the Fill Stack to hold blocks that experience long-term hits (the top part keeps blocks with short-term reuses), we bound the thread behavior analysis to that zone.

4.1. peLIFO with lists (peLIFO-ls)

To start with, we partition each set in 4 zones according to the *EP*s. The first zone goes from the top of the Fill Stack to the first *EP*, while the second, third and fourth zones go from each *EP* down to the bottom. We loosely refer to the first zone as the top part of the Fill Stack, and to the last three zones as the bottom part of the Fill Stack.

In this first technique, for each thread we dynamically calculate –at the end of each *epoch*– the *hits per block ratio* in the bottom part of the Fill Stack. This is, number of hits that the thread has experienced in the bottom part during the current *epoch*, divided by the average number of blocks allocated to this thread and accommodated in the bottom part during this *epoch*. Then, lists are generated with the threads ordered increasingly according to the obtained values: the threads reporting lowest and highest ratios are placed in the top and bottom respectively. When a block insertion requires an eviction, the victim block is chosen according to these ordered lists as we detail later. Thus, blocks corresponding to threads exhibiting a lower amount of *hits per block* (i.e. poor cache usage) are prioritized for eviction, on the foundation that each of these blocks is using the cache less efficiently than others. Among the different metrics we considered, this one seemed the most reasonable.

In practice, to obtain such *hits per block ratio* we need to track, per thread, the number of blocks accommodated in the cache when every replacement occurs, as well as the amount of hits experienced along the *epoch* for the 3 last zones of the Fill Stack. Therefore, for an N-way multi-programmed workload (N threads), we require *3*N* counters for accounting the amount of thread hits experienced by thread t_i in blocks between *EP* p_j and the bottom of the Fill Stack (denoted as *hits*(t_i,p_j)). We also require *2*3*N* counters for computing the average number of thread blocks in cache, from *EP* p_j to the bottom, corresponding to thread t_i (denoted as *blocksTot*(t_i,p_j) and *blocksPrevRepl*(t_i,p_j), and recording the current and previous number of thread blocks respectively). Consider the following example: A 2-set, 8-way cache, shared by 2 threads, presents the Fill Stack state shown in Figure 6 at a given moment.

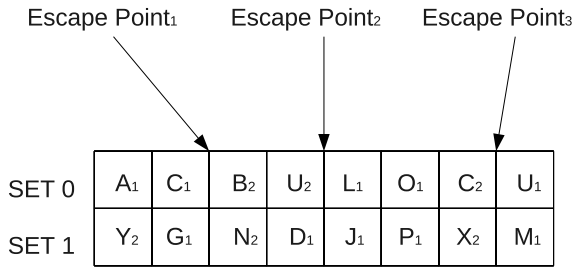

Figure 6: Fill Stack state at a given moment. The top of the stack corresponds to the block further to the left. Thus, in this case, A_1 (that represents block A from thread 1) and Y_2 occupy the top of the stack for SET_0 and SET_1 respectively.

According to this figure, the current state of the *blocksPrevRepl*(t_i,p_j) counters is the following:

- *blocksPrevRepl*(t_1,p_1)=7
- *blocksPrevRepl*(t_1,p_2)=6
- *blocksPrevRepl*(t_1,p_3)=2

- *blocksPrevRepl*(t_2,p_1)=5
- *blocksPrevRepl*(t_2,p_2)=2
- *blocksPrevRepl*(t_2,p_3)=0

Next, we illustrate the counters updating operation under different situations:

1. If thread 1 demands block A, it is found in set 0. In this case, all the counters remain unchanged since the corresponding hit takes place at the top part of the Fill Stack.
2. If thread 1 demands block J, it can be found in set 1, so a hit in the bottom part occurs. The corresponding counters are updated as follows:
   - *hits*(t_1,p_1)=*hits*(t_1,p_1)+1
   - *hits*(t_1,p_2)=*hits*(t_1,p_2)+1
3. If thread 2 demands block L, the cache misses. Suppose that the new block (L) replaces block O from thread 1. The cache state is updated as shown in Figure 7.

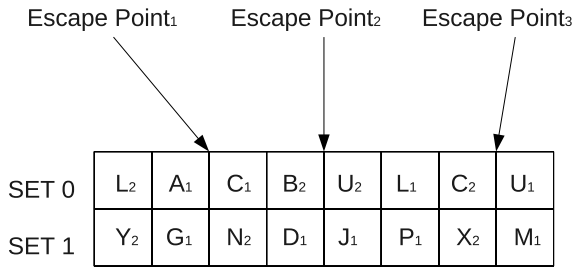

Figure 7: Fill Stack state after L_2 insertion and O_1 eviction.

To update the counters, an inspection of the Fill Stack from the cache set where the replacement took place is required. Thus, the counters are updated as follows:

- *blocksPrevRepl*(t_1,p_2)=6-1=5
- *blocksPrevRepl*(t_2,p_2)=2+1=3
- *blocksTot*(t_1,p_1)=*blocksTot*(t_1,p_1)+7
- *blocksTot*(t_1,p_2)=*blocksTot*(t_1,p_2)+5
- *blocksTot*(t_1,p_3)=*blocksTot*(t_1,p_3)+2
- *blocksTot*(t_2,p_1)=*blocksTot*(t_2,p_1)+5
- *blocksTot*(t_2,p_2)=*blocksTot*(t_2,p_2)+3
- *blocksTot*(t_2,p_3)=*blocksTot*(t_2,p_3)+0

At the end of every *epoch*, the *hits per block ratio* for each thread t_i and from each *EP* p_j down to the bottom (*Ratio*(t_i,p_j)) is computed using equation 1 [4]. With the obtained results, an ordered list is built for each of the 3 *EPs* (L_1, L_2 and L_3, for p_1, p_2 and p_3 respectively).

$$Ratio(t_i, p_j) = \frac{hits(t_i, p_j)}{blocksTot(t_i, p_j)} \quad (1)$$

PeLIFO-ls policy operates as follows: when the Set Dueling mechanism decides which *EP* p_j should be used in a replacement, the closest block to the top of the Fill Stack satisfying the next criteria is selected as the victim block:

- It corresponds to the thread in the top of the L_j list.
- Its current Fill Stack position is greater than or equal to the chosen p_j.
- It has not experienced a hit in its current Fill Stack position.

If no block satisfying these requirements is found, we restart from the first condition but using next thread in the list. If the entire list is traversed without finding a suitable block according to the criteria, the LRU block is selected for eviction.

4.1.1. Hardware simplifications

Although this peLIFO-ls scheme reports mild benefits (see Section 6.2) in terms of performance at an admissible hardware cost, we list below some simplifications to reduce the implementation complexity without a significant performance degradation.

- First, instead of tracking the number of thread blocks from each *EP* to the bottom, we just track the total amount of thread blocks in the entire cache (equation 2). Thus, the number of counters per thread required to compute *blocksTot* and *blocksPrevRepl* is reduced from 6 to 2. Besides, the inspection of the Fill Stack from the cache set where the replacement takes place is no longer necessary.

$$Ratio_{simplified}(t_i, p_j) = \frac{hits(t_i, p_j)}{blocksTot(t_i, 0)} \quad (2)$$

  Going back to the previous example, the state of the *blocksPrevRepl*(t_i,*0)* counters before the replacement in situation 3 would be the following:

  - *blocksPrevRepl*(t_1,*0)*=10
  - *blocksPrevRepl*(t_2,*0)*=6

  The counter updating after the replacement is shown next (Fill Stack inspection is now not necessary):

[4] It is worth noting that to obtain the average thread blocks from a *EP* during an *epoch*, we just need to divide *blocksTot*(t_i,p_j) by the total number of replacements occurred in the *epoch* (recall that it is defined as a power of 2). However, as we only care about the relative order between *Ratio* values from different threads, performing the division turns unnecessary.

- *blocksPrevRepl*(t_1,0)=10-1
- *blocksPrevRepl*(t_2,0)=6+1
- *blocksTot*(t_1,0)=*blocksTot*(t_1,0)+9
- *blocksTot*(t_2,0)=*blocksTot*(t_2,0)+7

- Second, instead of updating *blocksTot* at every replacement, it is done at a much coarser granularity (just 8 times per *epoch*, i.e., every 1024 refills to the cache), preventing the counter size to grow indiscriminately.

- Third, in equation 2, we approximate the denominator to the closest power of two. Thus, the required division turns into a simple right shift operation.

4.2. Proportional peLIFO (peLIFO-prop)

This second technique consists in replacing, per thread, an amount of blocks inversely proportionate to the current cache usage. Specifically, at the end of each *epoch*, the inverse of equation 1 (equation 3) is computed. Then, the amount of blocks to be replaced during the next *epoch*, belonging to thread t_i and located between *EP* p_j and the bottom, is given by equation 4, where N is the number of threads. For this purpose, *3*N* counters are initialized with the corresponding *blocksToReplace* values[5].

$$invRatio(t_i, p_j) = \frac{blocksTot(t_i, p_j)}{hits(t_i, p_j)} \tag{3}$$

$$blocksToReplace(t_i, p_j) = \frac{invRatio(t_i, p_j)}{min_{k=[1,N]}(invRatio(t_k, p_j))}; \tag{4}$$

PeLIFO-prop operates as follows: when the Set Dueling mechanism decides which *EP* p_j should be used in a replacement, the closest block to the top of the Fill Stack satisfying the next criteria is selected as the victim block:

- It corresponds to a thread with the *blocksToReplace* counter, corresponding to that *EP* value, higher than zero.

- Its current Fill Stack position is bigger than or equal to the chosen p_j.

- It has not experienced a hit in its current Fill Stack position.

If a cache block satisfying these conditions is found, the corresponding *blocksToReplace* counter is decremented. When all the counters associated to a particular *EP* reach zero, they are reset to the starting *blocksToReplace* values computed for the current *epoch*. If no suitable block is detected, the original peLIFO policy is employed.

[5]If the resulting value is not an integer number, it is rounded to the nearest one.

4.2.1. Hardware simplifications

Our peLIFO-prop scheme provides satisfactory performance results (see Section 6.2) at an admissible hardware cost. However, as well as for peLIFO-ls, we propose some simplifications that reduce the cost and implementation complexity required without a significant performance degradation.

- The first three approximations are analogous to those proposed for peLIFO-ls.
- The fourth simplification is applied to equation 4, approximating the denominator to the closest power of two, thus turning the division into a simple right shift operation.

4.2.2. Implementation Overhead

In this section we analyze the storage and algorithm complexity as well as the impact on the critical path for peLIFO-prop (for peLIFO-ls the analysis would be fully analogous).

Extra Storage:. Table 3 shows the additional hardware that peLIFO-prop requires compared to the original peLIFO policy, being TN the number of threads of our multi-programmed workload (*i.e.* TN-way), HS and BS the size of hits and blocks per thread counters respectively, and BRS the size of *blocksToReplace* counters. The first row in the table refers to an identifier of the thread that each block belongs to (that may be already present in a common multi-core policy). According to this table and to Table 1, with respect to LRU, peLIFO-prop adds around 88KB (16KB [6] from efficiently extending the policy to shared caches + 72KB added by the original peLIFO) to the L2 (8MB) of Memory-System 2, which means that the overhead with respect to LRU, in terms of extra bits required, is lower than 1.1%.

	Bits in a general configuration	*Bits in Memory-System 2*
Thread Id	A*log_2TN*N	16*2*4096
Amount of Hits Counters ($hits(t_i,p_j)$)	3*TN*HS	3*4*30
Amount of Blocks Counters ($blocksTot(t_i,0)$ and $blocksPrevRepl(t_i,0)$) (recall the first simplification detailed in Section 4.1.1)	2*NT*BS	2*4*30
blocksToReplace	3*TN*BRS	3*4*6

Table 3: peLIFO-prop storage overhead with respect to peLIFO.

Algorithm Complexity:. Next we summarize the new tasks that peLIFO-prop must solve with respect to original peLIFO.

- *Upon each hit in the bottom part of the Fill Stack:*

[6] Note that this overhead would reduce to just a few bytes in case that the shared cache already includes thread ID information

1. The *Amount of Hits* counter (*hits(t_i,p_j)*) is updated.

- *Upon each miss:*

2. When a replacement is required, a block is selected for eviction. A new condition joins to those from peLIFO: the candidate block must belong to a thread with the *blocksToReplace* counter higher than 0. In our case, we can tolerate to perform a sequential search through the Fill Stack (instead of an associative search, that would be more hardware consuming), since the task is out of the critical path.
3. Blocks counters (*blocksTot(t_i,0)* and *blocksPrevRepl(t_i,0)*)) are updated.
4. The *blocksToReplace* counter of the corresponding thread is decremented.

- *When an epoch ends:*

5. Reset *hits(t_i,p_j)*, *blocksTot(t_i,0)* and *blocksPrevRepl(t_i,0)*.
6. Compute the *blocksToReplace* value for each thread (note that considering the hardware simplifications detailed in Section 4.1.1 and Section 4.2.1, the required logic is extremely simple).

Impact on the Critical Path:. Again, the proposed policy has not any impact on the critical path. Upon a hit, just one additional task compared to peLIFO is needed: to update the *Amount of Hits* counter, which is an extremely fast and straightforward process. When a miss occurs, we need to include the new condition for the selection of the block to replace. However, again all these tasks can be performed off the critical path, so they are not a concern. Even assuming that they were within the critical path, there would be time enough for overlapping the tasks with the miss handling, just using associative logic for making the comparisons introduced by the new condition.

5. Simulation environment

We simulate two types of memory systems, one to evaluate our proposals from Section 4 over single-threaded applications (Memory-System 1) and the other to evaluate our proposals from Section 5 over 4-way multiprogrammed workloads (Memory-System 2). Details about the specific configurations are shown in Table 4.

Parameter	Memory-System 1	Memory-System 2
L1 Cache Size	32KB	4 private 32KB
L1 Cache Associativity	4 ways	4 ways
L1 Cache Block Size	32B	32B
L2 Cache Size	2MB	8MB (shared)
L2 Cache Associativity	16 ways	16 ways
L2 Cache Block Size	128B	128B

Table 4: Cache hierarchy configuration.

As single-threaded applications we selected several benchmarks from SPEC CPU2000 [13] and SPEC CPU2006 [14] suites, whereas 35 mixes of applications from the same distributions are built for the 4-way multiprogrammed workloads.

For evaluating the techniques described in Section 4, we execute the single-threaded applications until completion. We use *train* entries for the peLIFO-fixEP profiling phase and *reference* entries for the subsequent production runs.

For evaluating the designs from Section 5, given that the execution of multiprogrammed workloads turns too slow, we have employed the SimPoint toolset [15], selecting a single point from where we execute a billion instructions. The selected mixes are shown in Table 5 (we only show the SPEC benchmark id for each member of a mix). We tried to construct the mixes pool in a balanced and homogeneous fashion. Thus, each of the 28 considered application appears exactly in 5 mixes. Each workload mix is simulated until each thread commits the representative one billion dynamic instructions. A thread that completes this representative set early continues execution so that we can correctly simulate the cache contention for all the threads. However, all results reported in Section 7.2 take into account only the first one billion committed instructions from each thread.

MIX1: 171,172,173,181	MIX2: 183,401,403,410
MIX3: 429,433,434,435	MIX4: 437,444,445,450
MIX5: 453,454,456,458	MIX6: 462,464,465,470
MIX7: 471,473,481,482	MIX8: 171,183,429,437
MIX9: 172,453,462,471	MIX10: 401,433,444,454
MIX11: 173,403,464,473	MIX12: 434,445,456,465
MIX13: 181,410,435,481	MIX14: 450,458,470,482
MIX15: 171,410,445,464	MIX16: 172,429,450,465
MIX17: 173,433,453,470	MIX18: 183,435,456,473
MIX19: 401,437,458,481	MIX20: 403,444,462,482
MIX21: 171,444,445,482	MIX22: 172,437,450,481
MIX23: 183,433,456,470	MIX24: 401,429,458,465
MIX25: 403,410,462,464	MIX26: 171,410,462,482
MIX27: 172,429,458,481	MIX28: 173,433,456,473
MIX29: 181,434,454,471	MIX30: 183,435,453,470
MIX31: 401,437,450,465	MIX32: 403,444,445,464
MIX33: 181,434,454,471	MIX34: 173,181,434,435
MIX35: 453,454,471,473	

Table 5: Groups of benchmarks used in the simulation of multiprogrammed workloads.

As simulation infrastructure we use Pin [16] –a dynamic instrumentation tool that allows us to inject C or C++ code at arbitrary points of an executable during runtime– and a modified version of SESC [17] –a cycle-accurate microprocessor architectural simulator– to model the cache hierarchy. In the original SESC, cache communication between consecutive cache levels is done through a bus. Since we were not interested about the bus behaviour, we extended the model to support direct communication between levels (i.e. without interaction with the bus). Besides, we implemented a write back policy –the most employed nowadays– and an inclusive cache hierarchy was enforced. As Pin is designed to execute just a single application instance, we also had to adapt the simulator structure to support multiprogrammed environments. To face this inconvenient, we implemented a shared memory solution using the <sys/shm.h> library. Thus, a simulator instance for each evaluated application is created, being the required shared data –like cache state and statistics– stored in shared memory, so each instance can easily consult the information. Figure 8 shows a diagram of the simulator. Something similar was made in [18] with considerably satisfactory results.

Finally, to obtain execution time results, which the previous infraestructure does not provide, we have used the simulator provided in the *First JILP Workshop on Computer Architecture Competitions* [19]. This simulation framework is based on CMP$im simulator [18] and models a simple

out-of-order core with basic parameters that can be found in [19].

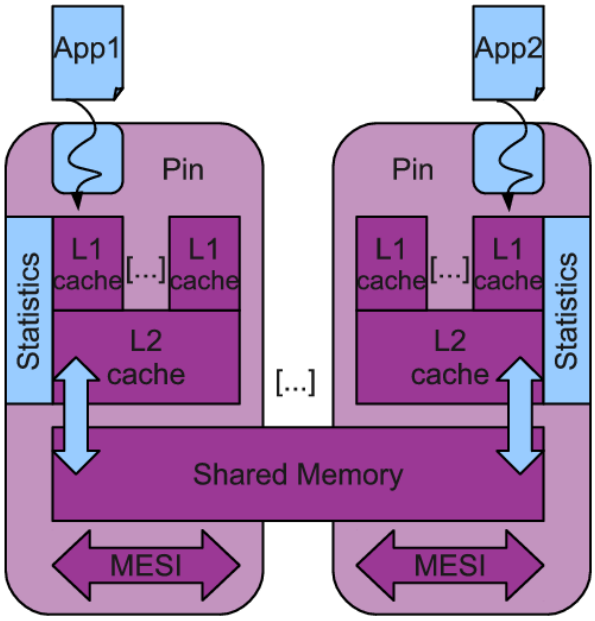

Figure 8: Modified simulator.

6. Evaluation

6.1. PeLIFO simplification

This section presents the results derived from the evaluation of the peLIFO simplification techniques described in Section 3. We have tested these proposals over the L2 Cache from the Memory-System 1 –detailed in the previous section– which receives blocks from a single-threaded application. Figure 9 shows performance results (CPI normalized to the baseline peLIFO policy) obtained for all considered applications employing LRU, peLIFO, and our two simplification proposals, as well as the average (geometric mean) results[7].

As illustrated, execution time reported by peLIFO reduces that of LRU by around 4.4% on average for the considered applications. In the case of our two simplification policies, both of them perform very similarly to original peLIFO (average execution time only increases by 0.1% and 0.2% for peLIFO-fixEP and peLIFO-CG respectively), so the impact of the proposed simplifications can be considered almost negligible.

Zooming into peLIFO-fixEP and peLIFO-CG results, we observe from the figure that for some benchmarks (like *mcf*, *milc*, or *art* and *gcc* with some entries) the CPI reported is even lower than

[7]When the gmean is calculated, the contribution weight of each benchmark should be exactly the same. To this end, those benchmarks that provide several reference entries (*art*, *bzip2* and *gcc*), contribute to the final gmean with their particular gmean among the different entries.

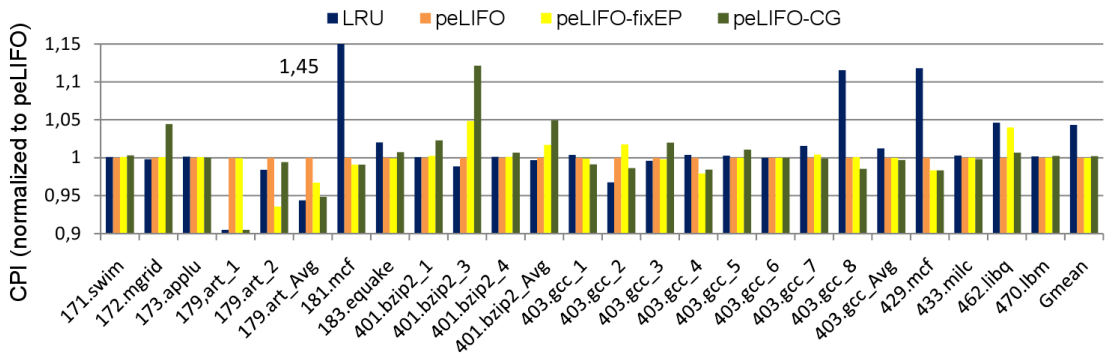

Figure 9: Performance of the different policies in a single-threaded environment.

that of peLIFO. The explanation lies on the fact that peLIFO chooses the *EPs* based on the first three drops in *Escape Probabilities*. However, in some cases, the optimum *EP* is located somewhere further inside the Fill Stack. In this situation, our two simplification techniques manage to identify the most adequate *EP* to be employed, leading consequently to better performance.

Finally, comparing the evaluated policies, we can conclude that peLIFO-CG is the most suitable among the policies studied for a single-threaded environment for several reasons: first, unlike peLIFO-fixEP, it requires no profiling, since the optimal *EP* is selected dinamically during each training stage. Besides, as explained in section 3, peLIFO-CG achieves the largest hardware reduction, and given that performance delivered by the two simplificaton techniques is very similar, peLIFO-CG manages to maximize the cost-performance ratio.

6.2. PeLIFO extension to shared caches

This section evaluates the techniques described in Section 5. We have tested these proposals over the shared L2 Cache from the Memory-System 2 –detailed in Section 6– which receives blocks from 4 threads (belonging to different applications each). Figure 10 reports performance results (average CPI[8] normalized to the baseline peLIFO policy) for all considered mixes when LRU, peLIFO, peLIFO-ls and peLIFO-prop (and the corresponding hardware simplifications proposed, denoted in the experiments as peLIFO-ls-simple and peLIFO-prop-simple) are employed, as well as the correspondig geometric mean.

As the figure illustrates, average CPI using peLIFO drops by around 5.8% on average compared to that of LRU for the considered mixes. Taking into account thread cache usage information, peLIFO-ls-simple reduces average CPI with respect to peLIFO and LRU by 1.8% and 7.5% respectively. In the case of peLIFO-prop, this reduction rises to 4.9% and 10.5% respectively. Given the best cost-performance ratio achieved, we consider peLIFO-prop-simple as the most suitable among all studied policies for a multi-threaded environment.

[8]For each considered mix, the average CPI is calculated as the arithmetic mean of individual CPI values reported by each of the four threads.

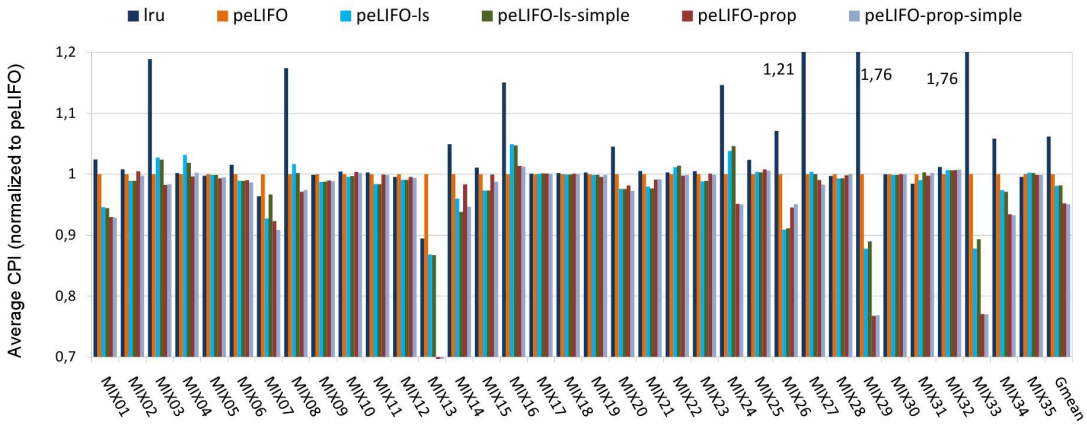

Figure 10: Performance of the different policies in a four-threaded environment.

Regarding the hardware simplifications proposed for peLIFO-ls and peLIFO-prop, we observe that they barely impact on performance, which suggests that they are totally adequate, as we will analyze in more detail in Section 7.3.3.

6.3. Other experimental results

In this section we extend our experimental analysis to clarify some important aspects of our approaches.

6.3.1. Analysis of peLIFO-fixEP sensitivity to different inputs

To determine the fixed *EPs* to be used in peLIFO-fixEP, a *train* entry is employed with the peLIFO policy. Then, with a *reference* entry, those selected *EPs* are used in order to obtain the corresponding peLIFO-fixEP final results. The efficiency of the policy relies on the profiling ability to select fixed *EPs* that work well for the same application with different inputs. In this section, we perform an analysis of the sensitivity of peLIFO-fixEP to different *reference* inputs.

As the suite of benchmarks employed in our simulations only provides three applications (*art*, *bzip2* and *gcc*) with different *reference* inputs, we restrict our analysis to these benchmarks. In Table 6 we show the miss rates obtained with peLIFO and peLIFO-fixEP considering different *reference* inputs, as well as the difference between each couple. From this table, we calculate the standard deviation of the difference values. The obtained results –0.06, 0.32 and 1.43 for *art*, *bzip2* and *gcc* respectively– are significantly low (especially for *art* and *bzip2*), so we conclude that peLIFO-fixEP is quite insensitive to the use of different *reference* inputs for the same benchmark.

	art.1	art.2	bzip2.1	bzip2.2	bzip2.3	bzip2.4	gcc.1	gcc.2	gcc.3	gcc.4	gcc.5	gcc.6	gcc.7	gcc.8
peLIFO	8.29	8.54	6.53	23.15	24.13	7.13	31.17	18.73	29.02	25.57	25.92	30.12	26.9	11.37
peLIFO-fixEP	9.05	9.22	6.45	23.02	24.65	6.99	32.63	18.94	29.75	26.68	28.87	31.97	31	11.18
Difference	0.76	0.68	-0.08	-0.13	0.52	-0.14	1.46	0.21	0.73	1.11	2.95	1.85	4.1	-0.19

Table 6: Miss rate obtained (%) with peLIFO and peLIFO-fixEP using different inputs.

6.3.2. Determining peLIFO-CG production stage length

In order to provide justifying data to support the choice of 128 *epochs* as the length of the peLIFO-CG *production stage*, Figure 11 shows the average miss rate obtained with different lengths. As inferred from the figure, the lowest miss rate is reported when 128 *epochs* are employed, so this length was used in all previous peLIFO-CG experiments.

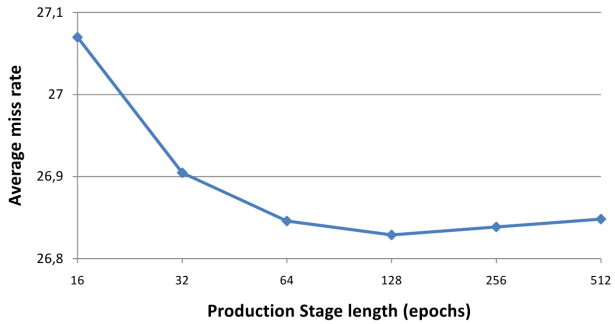

Figure 11: Average miss rate in peLIFO-CG with different *production stage* lengths.

6.3.3. Strictness of the first peLIFO-ls and peLIFO-prop HW simplifications

As we have illustrated in Section 6.2, the hardware simplifications proposed for peLIFO-ls and peLIFO-prop approaches barely impact performance. However, for the sake of completeness, in this section we carry out an analysis that confirms that the first simplification from Sections 4.1.1 and 4.2.1 (the one that could seem more imprecise) is adequate.

We perform the following study over peLIFO-ls (analogous results are obtained for peLIFO-prop): we define $K(t_i,p_j)$ as the ratio between $blocksTot(t_i,p_j)$ and $blocksTot(t_i,0)$ (see equation 5); if we merge equation 2 and equation 5, we gather equation 6. Thus, if $K(t_i,p_j)$ becomes very similar among all the threads for a given p_j, the ordered list derived from equation 1 remains unchanged, since the simplification just scales the original *Ratio* values by a constant. Figure 12 shows, for 3 random mixes and 3 random *epochs* each, the 3 K values per thread obtained from equation 5. As expected, $K(t_i,p_1)$ remains very stable among threads, and so do $K(t_i,p_2)$ and $K(t_i,p_3)$.

$$K(t_i,p_j) = \frac{blocksTot(t_i,p_j)}{blocksTot(t_i,0)} \tag{5}$$

$$\begin{aligned} Ratio_{simplified}(t_i,p_j) = K(t_i,p_j)\frac{hits(t_i,p_j)}{blocksTot(t_i,p_j)} = \\ = K(t_i,p_j)Ratio(t_i,p_j) \end{aligned} \tag{6}$$

6.3.4. Cache usage metric employed in peLIFO-ls and peLIFO-prop

We have considered the *hits per block ratio* (equation 1) –and the corresponding inverse *blocks per hit* (equation 3)– as the most suitable metric for measuring the thread usage of the bottom part

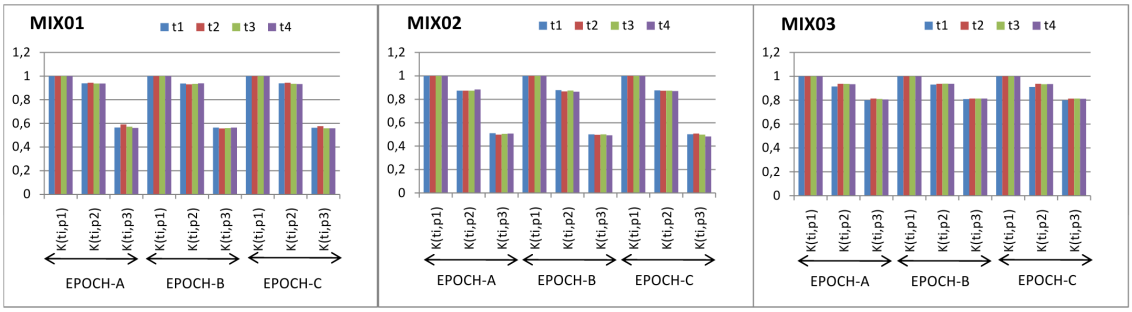

Figure 12: $K(t_i,p_j)$ values for 3 mixes and 3 random *epochs* each using peLIFO-ls policy.

of the Fill Stack. However, other metrics could be employed. One that easily may come to our mind is the hit rate (*hits per access*), so frequently used when analysing cache policies. For comparing them, Table 7 shows –for 3 random mixes– the average miss rate increase for peLIFO-prop (results are analogous for peLIFO-ls) when using *hits per access* as the thread usage estimation metric instead of the *hits per block* metric. As shown, the metric employed in this paper works better for all the cases evaluated.

MIX	**Miss Rate increment using the *hits per access* metric**
MIX01	0.14%
MIX02	2.61%
MIX03	3.48%

Table 7: Average miss rate increment for peLIFO-prop when employing *hits per access* as metric for estimating cache usage instead of *hits per block*.

7. Related work

Being a key aspect for performance, cache replacement policies have evolved and improved in recent years. Starting from basic proposals –LRU, NRU, LFU, FIFO, Round-robin and random replacement–, several proposals have been developed aiming to obtain better cache usage through more clever replacement policies.

A first body of work tries to get efficient LRU implementations [20, 21, 22]. The algorithm known as *pseudo LRU* or *Tree-LRU* [20] considers a binary search tree for cache items. Each node in the tree contains a one-bit flag denoting which one is the correct direction to find the next item to replace. To find this element, the tree is traversed according to the values of these flags. When an access to an item occurs, the tree is updated by looking up the item, setting the corresponding flags in traversed nodes to denote the direction opposite to the one taken in the search. This technique is used in the on-chip cache of the Intel 486 and in many processors in the Power Architecture family (formerly PowerPC). A different approach is the Not Recently Used policy (NRU)[23]. This algorithm works on the following principle: when a block is referenced, a referenced-bit is set for that block. Similarly, when a block is modified, a modified-bit is set. After a fixed-length time

interval, the clock interrupt triggers and clears the referenced-bit of all the blocks, so only blocks referenced within the current clock interval have the referenced-bit set. When a blocks has to be replaced, the hardware divides the blocks into four classes: 1) not referenced, not modified, 2) not referenced, modified, 3) referenced, not modified, 4) referenced, modified. The NRU algorithm picks a random block from the lowest-numbered class for removal. Note that this algorithm considers a modified (within clock interval) but not referenced block less important than a not modified block that is intensely referenced.

Another body of work uses *Dead Block Prediction*. Only a small fraction of cache lines actually hold data that will be referenced before eviction. Traditionally, a cache block that will be referenced again before eviction is called *live block*; otherwise it is called *dead block*. At replacement time, it is desirable for the policy to choose a dead block instead of a live block. Many different approaches concerning dead block prediction techniques have been proposed [24, 25, 26, 27, 28, 29, 30].

Other approaches pursue to settle the common problems that arise when using LRU. In [8], the authors observe that the performance loss resulting from a cache miss is reduced when multiple misses are served in parallel, since the idle cycles waiting for memory get amortized over all the concurrent misses. On the other hand, isolated misses are the ones that hurt performance the most because the processor is stalled waiting for the cache to serve just a single miss. Considering this, they propose a new replacement policy, LIN, that holds in cache the blocks that would incurr an isolated miss. Although this policy obtains good results, the main contribution of the paper is a technique called *Set Sampling*: a hybrid (LIN and LRU) policy is proposed. For each replacement, this approach decides which policy should be used, based on their particular performance dynamic evaluation. To carry out the evaluation, the tag array for a small number of cache set (the samplig sets) is duplicated to provide statistics for both policies. The policy that reports the best results in the sampling sets is used for the remaining sets. Another recent piece of work [9] tries to mitigate the well-known LRU thrashing problem when the working set is larger than the cache size. The authors introduce new replacement policies both static and dynamic. On the static side, a straightforward policy called LIP is proposed. According to it, the block evicted is the least recently used –as in LRU– but the new block is not considered the MRU, but in the LRU. If the block is referenced again, then it becomes the MRU. Thus, if a recently inserted block is not referenced again soon, it will be most likely evicted. This way, the cache is able to hold those blocks with long-term reuses. Based on LIP, the authors propose another static policy, called BIP, in which just a fraction of new blocks are inserted in cache as LRU and the remaining as MRU. Regarding dynamic proposals, authors describe the dynamic insertion policy, DIP, which dynamically combines BIP and LRU using a mechanism based on Set Sampling. This mechanism is called Set Dueling: a competition takes place between a small amount of sets that use LRU and another group of sets that uses just LIN. The winner is the one with the lowest number of misses, and imposes its policy to the rest of the cache. In [31] authors detect an additional problem of LRU that appears when applications show a memory access pattern consisting of a burst of references to data whose re-reference will ocurr in the distant future. This pattern is called scan. They propose a new policy, called Static-RRIP (SRRIP), that aims to fight against this quite common problem: every block arriving to the cache is predicted to have an *intermediate* re-reference interval (that sits between *near-immediate* re-reference and *distant* re-reference). If the block is soon re-referenced, then its interval is changed to *near-immediate*. However, if the block is not re-referenced soon (as it happens with scans), it will quickly be evicted from the cache, not polluting it. The authors also propose another policy, called Dynamic-RRIP, that is both thrash-resistant and scan-resistant: using Set-Dueling, it selects the best of two policies: SRRIP, to fights against scans, and BRRIP (a RRIP policy based on the

philosophy under BIP [9]), that fights against thrashing.

With the proliferation of multi/many-core systems in recent years, some cache policies have emerged to efficiently adapt to this new environment. The main challenge consists in obtaining a cache replacement policy that works efficiently in the shared levels of the memory hierarchy, where different threads can access concurrently, leading to a destructive interference. Although at the beginning of the multi-core era uniprocessor policies –such as LRU– were directly applied to shared cache levels, in many cases destructive interference leads to a significant slowdown, making necessary new management techniques that include thread information to achieve right replacement decissions.

In [11] the authors propose a runtime mechanism that partitions a shared cache amonng multiple applications. Depending on the rate of cache misses, each application obtains a given amount of cache resources. The distribution of memory space is the one that minimizes the total number of misses.

In another piece of work, described in [32], the authors propose a hybrid approach, called Thread-Aware DIP (TADIP), merging [9] and [11]. Similarly to DIP, TADIP dynamically choses between LRU and BIP for each application executing on a CMP core. When the number of concurrent applications is small, we can use Set Dueling to perform a runtime comparison of all possible binary strings and select the best performing one. However, when it is large, the number of combinations increases exponentially making this bruteforce approach impractical. To avoid the exponential increase in the number of Set Dueling mechanisms, the authors propose two scalable approaches called TADIP-Isolated and TADIP-Feedback . These two approaches leverage the fact that some of the bits in the best-performing insertion string can be determined independently.

Finally, in [12] the authors propose a new mechanism, called Promotion/Insertion Pseudo Partitioning PIPP. Instead of explicitly partitioning the cache by ways, sets or total occupancy, PIPP implicitly partitions (or pseudo-partitions) the cache by simply managing the insertion and promotion policies of the cache. The insertion policy determines where, in the eviction priority (e.g., LRU stack), a line should initially be installed, and allows a new block not to be always inserted at the top of the recency stack (as in LRU). The promotion policy determines how eviction priority should be changed on a cache hit and allows a hit block not to go to MRU (as in LRU). This flexibility makes the shared cache behave as a virtual partitioned cache, exhibiting significantly high hit rates.

8. Conclusions

In this paper we have suggested several enhancements to the recently proposed peLIFO cache replacement policy [1]. First, we propose different techniques for reducing its implementation overhead with a negligible performance impact.

Second, we propose an efficient peLIFO extension to the shared cache of a multi-core processor environment, where several applications with opposed features may coalesce. Thus, we gather information about each thread nature, favoring that those threads exhibiting a better cache usage may profit from cache resources. At the expense of negligible extra hardware, our suggested thread-aware peLIFO-prop-simple policy manages to reduce average CPI by 4.9% and 10.5% on average compared to peLIFO and LRU respectively. For peLIFO-ls-simple mechanism, these percentages are 1.8% and 7.5% respectively. Moreover, we introduce *hits per block* as an alternative metric to classify the threads accessing a shared cache in terms of their cache usage, and it reveals as an accurate and useful metric to guide peLIFO policy, being extensible to any other cache replacement mechanism that seeks for an effcient shared cache partition among different threads.

Finally, it is worth noting that both the simplification techniques and the extension to shared caches proposals are completely independent and could be merged straightforwardly.

References

[1] M. Chaudhuri, Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches, Proceedings of MICRO-09, New York, NY, USA, 12-16 December. (2009) 401–412. ACM, New York, NY, USA.

[2] G. Z. Chrysos, J. S. Emer, Memory Dependence Prediction Using Store Sets, Proceedings of ISCA-98, Barcelona, Spain, 27 June-2 July (1998) 142–153. IEEE Computer Society Washington DC, USA.

[3] M. Lipasti, C. Wilkerson, J. Shen, Value Locality and Load Value Prediction, Proceedings of ASPLOS-96, Cambridge, MA, USA, 1-4 October (1996) 138–147. ACM, New York, NY, USA.

[4] J. Gonzalez, A. Gonzalez, Speculative Execution via Address Prediction and Data Prefetching, Proceedings of ICS-97, Vienna, Austria, 7-11 July (1997) 196–203. ACM, New York, NY, USA.

[5] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, J. D. Owens, Memory Access Scheduling, Proceedings of ISCA-2000, Vancouver, BC, Canada, 10-14 June (2000) 128–135. ACM, New York, NY, USA.

[6] T. Chen, J. Baer., Effective Hardware-based Data Prefetchting for High Performance Processor, IEEE Transactions on Computers, Vol. 44, No. 5 (1995) 609–623.

[7] L. A. Belady, A Study of Replacement Algorithms for a Virtual-storage Computer, IBM Systems Journal, Vol. 5, No. 2 (1966) 78–101.

[8] M. K. Qureshi, D. N. Lynch, O. Mutlu, Y. N. Patt, A Case for MLP-aware Cache Replacement, Proceedings of ISCA-06, Boston, MA, USA, 17-21 June (2006) 167–177. IEEE Computer Society, Washington DC, USA.

[9] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, J. Emer, Adaptive Insertion Policies for High Performance Caching, Proceedings of ISCA-07, San Diego, CA, USA, 9-13 June (2007) 381–391. ACM, New York, NY, USA.

[10] D. Rolán, B. B. Fraguela, R. Doallo, Adaptive Line Placement with the Set Balancing Cache, Proceedings of MICRO-09. New York, NY, USA, 12-16 December (2009) 529–540. ACM, Mew York, NY, USA.

[11] M. K. Qureshi, Y. N. Patt, Utility-based Cache Partitioning: A Low-overhead High-performance Runtime Mechanism to Partition Shared Caches, Proceedings of MICRO-06, Orlando, FL, USA, 9-13 December (2006) 423–432. IEEE Computer Society, Washington DC, USA.

[12] Y. Xie, G. H. Loh, PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches, Proceedings of ISCA-09, Austin, TX, USA, 20-14 June (2009) 174–183. ACM, New York, NY, USA.

[13] SPEC 2000 home page, http://www.spec.org/cpu2000.

[14] SPEC 2006 home page, http://www.spec.org/cpu2006.

[15] Simpoint home page, http://cseweb.ucsd.edu/ calder/simpoint.

[16] Pin home page, http://www.pintool.org.

[17] MultiCacheSim Homepage, http://github.com/blucia0a/MultiCacheSim.

[18] A. Jaleel, R. Cohn, C. Luk, B. Jacob, CMP$im - A Pin-Based On-The-Fly Multi-Core Cache Simulator, Proceedings of MOBS-08, Beijing, China, 22 June (2008) 28–36. IEEE Computer Society, Washington DC, USA.

[19] First JILP Workshop on Computer Architecture Competitions, http://www.jilp.org/jwac-1.

[20] J. Handy, The Cache Memory Book, Academic Press, Boston, MA, USA, 1993.

[21] J. T. Robinson, Generalized Tree-LRU Replacement, IBM Research Report RC23332 (W0409-045), Yorktown Hts., NY, USA.

[22] H. Ghasemzadeh, S. Mazrouee, M. R. Kakoee, Modified Pseudo LRU Replacement Algorithm, Proceedings of ECBS-06, Postdam, Germany, 27-30 March (2006) 368–376. IEEE Computer Society, Washington DC, USA.

[23] UltraSPARC T2 supplement to the UltraSPARC architecture 2007, Draft D1.4.3. 2007.

[24] Z. Hu, S. Kaxiras, M. Martonosi, Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior, Proceedings of ISCA-02, Anchorage, AK, USA, 25-29 May (2002) 209–220. IEEE Computer Society, Washington DC, USA.

[25] J. B. Sartor, S. Venkiteswaran, K. S. McKinley, Z. Wang, Cooperative Caching with Keep-Me and Evict-Me, Proceedings of INTERACT-05, San Francisco, CA, USA, 13 February (2005) 46–57. IEEE Computer Society, Washington DC, USA.

[26] Z. Wang, K. S. McKinley, A. L. Rosenberg, C. C. Weems, Using the Compiler to Improve Cache Replacement Decisions, Proceedings of PACT-02, Charlottesville, VA, USA, 21-25 September (2002) 199–208. IEEE Computer Society, Washngton DC, USA.

[27] M. Kharbutli, Y. Solihin, Counter-Based Cache Replacement and Bypassing Algorithms, IEEE Transactions on Computers, Vol. 57, No. 4 (2008) 433–447.

[28] J. Abella, A. Gonzalez, X. Vera, M. O'Boyle, IATAC: A Smart Predictor to Turn-Off L2 Cache Lines, ACM Transactions on Architecture and Code Optimization, Vol. 2, No. 1 (2005) 55–77.

[29] H. Liu, M. Ferdman, J. Huh, D. Burger, Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency, Proceedings of MICRO-08, Como, Italy, 8-12 November (2008) 222–233. IEEE Computer Society, Washington DC, USA.

[30] A. Lai, C. Fide, B. Falsafi, Dead-block Prediction & Dead-block Correlating Prefetchers, Proceedings of ISCA-01, Gothenburg, Sweden, 30 June- 4July (2001) 144–154. ACM, New York, NY, USA.

[31] A. Jaleel, K. Theobald, S. Steely, J. Emer, High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP), Proceedings of ISCA-10, Saint-Malo, France, 19-23 June (2010) 60–71. ACM, New York, NY, USA.

[32] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, J. Emer, Adaptive Insertion Policies for Managing Shared Caches, Proceedings of PACT-08, Toronto, Canada, 25-29 October (2008) 208–219. ACM, New York, NY, USA.