Performance and Energy Assessment of Last-Level Cache Replacement Policies

Pierre-Yves Péneau, David Novo, Florent Bruguier, Gilles Sassatelli, Abdoulaye Gamatié

Performance and Energy Assessment of Last-Level Cache Replacement Policies

Pierre-Yves Péneau, David Novo, Florent Bruguier, Gilles Sassatelli and Abdoulaye Gamatié
LIRMM (CNRS & Université de Montpellier), Montpellier, France
first.last@lirmm.fr

***Abstract*—The Last-Level Cache (LLC) is a critical component of the memory hierarchy which has a direct impact on performance. Whenever a data requested by a processor core is not found in the cache a transaction to the main memory is initiated, which results in both performance and energy penalties. Decreasing LLC miss rate therefore lowers external memory transactions which is beneficial both power and performance-wise. The cache replacement policy has a direct impact on the miss rate. It is responsible of data eviction of cache lines whenever the cache runs full. Thus, a good policy should evict data that will be re-used in a distant-future, and favour data that are likely to be accessed in the near-future. The most common cache replacement policy is the Least-Recently Used (LRU) strategy. It has been used for years and is cheaper in terms of hardware implementation. However, researchers have shown that LRU is not the most efficient policy from a performance point of view, and is further largely sub-optimal compared to the best theoretical strategy. In this paper, we analyze a number of cache replacement policies that have been proposed over the last decade and carry out evaluations reporting performance and energy.**

I. Introduction

Computing systems are composed of several processing cores on the same chip and their performance is growing every year. In particular, this is a critical requirement in the High Performance Computing domain (HPC), where best-effort computation is highly desired. On the other hand, core computation capacity increases faster than memory capacity, leading to the well-known memory wall problem. It is widely accepted that the issue in such systems is no longer the computation but the memory access.

Therefore, an efficient memory system is key to providing compute systems with high performance capabilities. A key component of the memory hierarchy is the Last-Level Cache, or LLC. It is the last on-chip memory buffer where data can be stored before data exchanges reaches the off-chip levels, particularly the main memory. Data accesses that occur beyond the LLC are usually time and energy-consuming. Thus, an efficient LLC should decrease main memory access count. Such efficiency is directly related to the cache replacement policy of the LLC. This policy is responsible for deciding which cache line / data is to be evicted for other useful data to be cached. Thus, it is critical to suitably select which data must be discarded and which data should stay in a given cache level so as to maximize the amount of useful data in the cache.

In this paper, we review four popular cache replacement policies that have been proposed during the last decade in the literature. In order to support our analysis, we compare their performance against the classical Least-Recently Used (LRU) and the Random cache replacement policies. Furthermore, we evaluate the impact of each policy on the energy consumption of the LLC. To the best of our knowledge, no existing work provides a comprehensive comparison of these policies including energy consumption. In order to carry out our study, we use the ChampSim [1] simulator and the SPEC CPU2006 [2] benchmark suite. The energy model for the LLC is obtained by using the popular CACTI tool [3].

The remainder of this paper is organized as follows: Section II presents the metrics used to assess the cache replacement policies; Section III reviews the considered policies; Section IV details the experimental setup used for their evaluation, then discusses the obtained results; and finally, Section V gives some concluding remarks.

II. Background

A. Common metrics for replacement policy assessment

We review some commonly-used metrics for assessing the efficiency of CPUs and memory hierarchy. In the present paper, upper (or higher) cache levels correspond to those which are closer to the CPU. Thus, the L1 cache is the upper level and the main memory is a lower level in the memory hierarchy. A classical LLC cache organization supporting LRU is depicted by Figure 1, where the data structures used together with the LLC are illustrated. By default, the LRU cache replacement policy covers the components that are outside of the dashed box. The remaining components are used in the cache replacement policies presented later in this paper. In order to understand how the different policies affect the system performance, we introduce a few useful evaluation metrics.

One prominent issue in current compute systems lies is memory response time that cores must accommodate. This time has to be reduced as much as possible for performance reasons. A non negligible part of this time is due to misses in the LLC which often result in stalling of program execution until data is retrieved from the main memory. Thus, an efficient LLC helps reducing these expensive off-chip accesses such that execution can be resumed as early as possible.

A commonly used metric to assess the cache efficiency is the Miss Per Kilo Instructions metric, or MPKI, defined by formula (1). When assessing a replacement policy, the number of executed instructions remains the same, but the number of misses varies. As mentioned previously, a good policy should

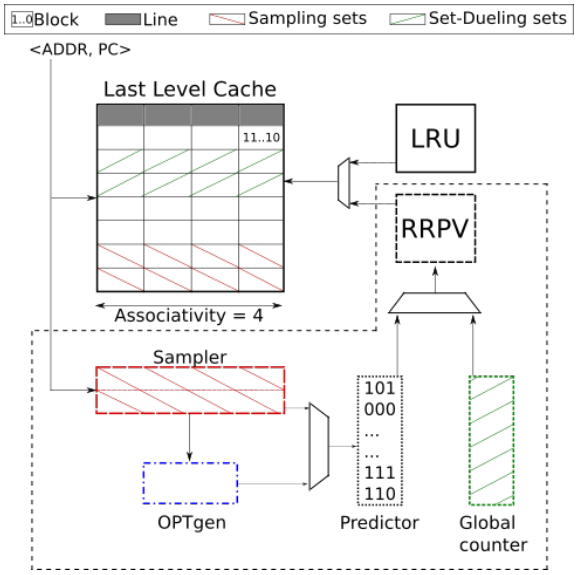


Fig. 1: Data structures for the LLC. Default configuration for LRU is outside the dashed box. SRRIP only uses RRPV; DRRIP uses RRPV, Set-Dueling and Global Counter; SHiP uses Sampler, Predictor and SRRIP structures; Hawkeye uses OPTgen in addition to SHiP structures.

reduce the number of cache misses. Thus, the MPKI should be decreased accordingly.

$$MPKI = \frac{Total\,Miss}{Total\,instructions\,/\,1000} \tag{1}$$

Another common metric used to evaluate replacement policies is the Instruction Per Cycle (IPC), which describes the efficiency of the CPU. As the number of executed instructions should stay identical while evaluating different replacement policies, only the number of cycles required by the execution of the instructions may be different. This number relies on the memory efficiency since a non-negligible part of the execution time is wasted waiting for data, even more so when the fetch extends all the way to the external memory, which is the critical path. An efficient LLC policy should reduce the MPKI, which decreases the probability for a request to follow the critical path, and finally shortens the overall time required for the computation. If this objective is reached, the IPC increases, as expressed in the following equation:

$$IPC = \frac{Total\,Instructions}{Total\,Cycles} \tag{2}$$

B. Re-use distance and memory pattern

The replacement policy is responsible of block eviction on cache lines when they are full. Upon a miss, a block is selected as the victim and is discarded. Such a decision is based on heuristics that the LLC builds during the system execution. A heuristic has to predict accurately when a block will be re-used in the future, also called the *re-use distance*. When the re-use distance is in the near future, the block should stay in the cache so as to avoid a miss. Otherwise, the block should have a high priority candidate whenever eviction is required. Thus, the heuristic of cache replacement policy has to avoid wrong decisions as much as possible.

Application workloads generally have memory access patterns that have a varying impact on the efficiency of replacement policies. These patterns are divided into four categories [4]: *recency-friendly, trashing, streaming* and *mixed*, respectively denoted by the formulas (3), (4), (5) and (6):

$$(a_1, a_2, ..., a_k, a_{k-1}, a_{k-2}, ..., a_2, a_1)^N \text{ for any k} \tag{3}$$

$$(a_1, a_2, ..., a_k)^N \text{ for k} > \text{ cache size} \tag{4}$$

$$(a_1, a_2, a_3, ..., a_k) \text{ for k} = +\infty \tag{5}$$

$$\{(a_1, ..., a_k)^A P_\varepsilon(a_1, a_2, ..., a_k, a_{k+1}, ..., a_m)\}^N \tag{6}$$
$$\text{for k}< \text{ cache size, m}> \text{ cache size, } 0 < \varepsilon < 1, \{A, N\} \geq 1$$

With a recency-friendly pattern, blocks have a near-immediate usage. A trashing pattern leads to a cyclic access to K blocks that repeats N times. A streaming pattern occurs when blocks have no locality in their usage (also referred to as scan pattern). Finally, mixed patterns are a combination of the other three.

III. Evaluated cache replacement policies

A. Least-Recently Used

The Least-Recently Used (LRU) cache replacement policy sorts the blocks of a line according to their usage over time. More precisely, they are ordered from 0 to $N-1$, where N is the cache associativity. Then, when a block B is used, it becomes the block number 0 and each block position between 1 and the last position of B is incremented by 1. Upon an eviction, the block $N-1$ becomes the victim. This heuristic predicts a short re-use distance for blocks that are recently used. Conversely, blocks with high position, i.e., near N, in the LRU list have a long distance re-use prediction. This policy works well with recency-friendly access pattern, and also with trashing pattern when the number K of blocks is less or equal than the number of blocks in the cache.

B. Static Re-Reference Interval Prediction

Proposed by Jaleel et al. [4], the goal of the Static Re-Reference Interval Prediction (SRRIP) cache replacement policy is to predict the usage interval of blocks to avoid filling the cache with blocks that have a distant re-use interval. For this purpose, a Re-Reference Prediction Value (RRPV) is assigned to each block of the cache (see Figure 1). This value is coded on M bits, and varies between 0 and 2^M-1. When $RRPV \leq 1$, the usage interval of the corresponding block is predicted as *immediate*. When $RRPV = 2^M-1$, the usage interval of the block is predicted as *distant*. Between these two cases, it is predicted as *intermediate*. On a block insertion, a RRPV value of 2^M-2 is assigned to the block. Upon an access, the SRRIP cache replacement sets RRPV to 0. Then, the block is predicted to be re-used in a near-immediate future. Upon an eviction, the SRRIP policy looks for a block such that $RRPV = 2^M-1$, i.e., a distant usage interval prediction. If no such a block exists in the line, each RPPV is incremented by 1 and the process is restarted until one RRPV at least

reaches this requirement. The advantage of inserting with an intermediate usage interval prediction of $2^M - 1$ is to give more time to the SRRIP policy to learn the access pattern of a block, since its RRPV value is not enough for an eviction. The main disadvantage is that SRRIP can be polluted by blocks that have only two accesses, since it predicts a near-immediate usage interval on a hit. Thus, with SRRIP, a block that has been used only once is more likely to be evicted compared to a re-used block, even though this happened less recently.

This policy is scan-resistant (i.e., efficient when dealing with scan cache access pattern), since non temporal accesses do not pollute the LLC. It is also trash-resistant when the number k of blocks is less or equal to the cache size.

C. Dynamic Re-Reference Interval Prediction

When the usage interval of blocks is larger than the cache size, i.e., a trashing access pattern with a number of blocks $k > associativity$, the above SRRIP policy becomes inefficient and causes more misses. To prevent this situation, Jaleel et al. [4] proposed the Dynamic Re-Reference Interval Prediction (DRRIP) cache replacement policy, as an extension of SRRIP.

Upon an insertion, the RRPV of a block is more likely to be set to $2^M - 2$. However, RRPV can be sometimes set to the maximum value $2^M - 1$. Then, for applications that have a trashing access patterns, each inserted block is evicted after its use and does not pollute the cache. Blocks that are re-used are set to 0 and stay in the cache. To accurately detect the memory access pattern of an application, authors used the Set-Dueling techniques [5] (see also Figure 1). They dedicated 32 lines of the LLC to the SRRIP policy only and 32 lines to the DRRIP policy only. A global counter, initialized to zero, is incremented or decremented by 1 respectively upon a cache miss on SRRIP or DRRIP dedicated cache lines. When the value of the counter is higher than 0, it means that the SRRIP policy leads to more cache misses than DDRIP. Conversely, when the counter is less or equal to 0, the DRRIP policy is less efficient. The cache uses the above global counter to dynamically adapt its replacement policy and to choose between SRRIP and DRRIP. This enables to select the most efficient replacement policy among the two when dealing with either scan or trashing pattern. One issue with DRRIP is the granularity of the prediction. According to the global counter, the entire cache has to follow the selected policy, i.e., the prediction is not at block granularity.

D. Signature-based Hit Prediction

One limitation of the SRRIP policy is that predictions made upon block insertion in the cache are static. The DRRIP policy addresses this issue by randomly alternating between two possibilities. This also relies on static choices upon a block insertion, and the learning phase is done after. The insertion strategy does not take into account the past activity of the cache. The *Signature-based Hit Predictor* (SHiP) policy [6] tackles this aspect by making predictions according to each block's access pattern. Authors added near the LLC a table of saturating counters, indexed by a hash called *signature* (see Figure 1). They evaluated signatures based on memory addresses, instruction sequences and program counters (PCs). Results show that the most efficient is the PC-based approach.

From a technical point of view, two fields are added in the cache meta-data: the signature and the *outcome bit*, both initialized to 0. Upon a block insertion, the corresponding signature is updated. Upon a hit on that block, the outcome bit is set to 1, meaning that since its insertion, this block has been re-used at least once. If a cache access results in a hit, the corresponding entry in the predictor is incremented by 1. In case of a cache miss, it is left unchanged. The entry is decremented when a block is evicted from the cache and its outcome bit is set to zero, i.e., this block has never been re-used since it was inserted. Hence, the PCs that generate a lot of misses are identified and can be treated. When inserting a new block, the predictor is checked. When its signature points at an entry with the value of 0, the re-use interval is predicted as a distant interval. Any other value is considered as intermediate. Thus, the re-use interval prediction is dynamically predicted according to the PC responsible for the access.

Assigning a signature to each cache blocks is unrealistic for hardware resource requirement reason. Hence, the proposed implementation samples 64 lines of the LLC in order to train the predictor. The predictor is updated only upon a cache access to one of these 64 lines, while it is consulted on each access. Thus, a small portion of the cache activity steers the predictions over the entire cache.

This policy offers two main contributions. Firstly, the possibility to make predictions on application behavior thanks to program counters. Secondly, each block has its own signature, making the prediction more accurate than with SRRIP or DRRIP policies, which affect all cache blocks. The SHiP policy has been used to improve the SRRIP mechanism [6]. Instead of always predicting intermediate re-use interval, SHiP is applied to take decisions. This makes SRRIP more resistant to trashing pattern thanks to the signature-based prediction.

E. Hawkeye Approach

The Hawkeye policy [7] is based on Belady's MIN [8] algorithm. For a given set of cache accesses, MIN classifies blocks in two categories: hit blocks and miss blocks. This classification has been proved optimal [9], hence the number of misses found with MIN cannot be reduced.

Upon a block eviction in a cache, one can know if this is a good decision only after the victim block is later re-used. For example, given a cache with a capacity of 2 blocks per line and only 1 line containing B_0 and B_1. B_1 is the victim block upon the insertion of a new block B_2 and the next access to the cache is B_1. The MIN algorithm would detect that B_1 should not be evicted and would classify B_1 as a "hit block". Jain et al. [7] proposed a mechanism called OPTgen that is able to reconstruct the MIN algorithm on past cache accesses (see Figure 1). Past accessed blocks are separated into two groups with the assumption that what happened is probably what will happen in the future. They verified this experimentally on the SPEC CPU2006 benchmark suite and

found that MIN's decisions for a block remains the same 90% of the time on average during an entire execution. Then, they used MIN predictions on the past to guide future decisions.

The Hawkeye predictor uses the same sampling idea as that of Wu et al. [6]. A subset of the LLC is monitored in order to minimize the implementation hardware cost, while still having an accurate representation of the cache activity. The predictor is only trained when these lines are accessed. A table of saturating counters is used, which is indexed by PCs that are updated on each OPTgen prediction. When OPTgen predicts a cache hit, the corresponding entry is incremented by 1. When OPTgen predicts a cache miss, meaning that whatever is the victim choice, the next access to this block will result in a cache miss under optimal replacement policy, the corresponding entry is decremented by 1. Thus, such a block can be discarded and replaced by a useful data. Blocks within the cache are ordered according to their corresponding RRPV values. A high RRPV value corresponds to a high cache eviction priority, while the opposite is for low eviction priority. Blocks that are predicted to miss in the future are set to $RRPV = 2^M - 1$, i.e., the maximum value. Those predicted to hit in the future are set to $RRPV = 0$. Upon a block insertion (i.e., a miss), if the block is predicted to hit in the future, each RRPV on the line are aged, i.e. incremented by 1. Blocks that are predicted to hit in the future can never reach the maximum RRPV value. Thus, those predicted to miss will always have the higher priority for eviction. If no such blocks exist, the block with the highest RRPV value is evicted.

This policy is efficient when dealing with scan, trash and mixed patterns. In fact, thanks to Belady's MIN algorithm which is able to accurately predict the future of blocks, each block behavior is detectable and can be anticipated. Thus, only useful cache blocks remain in the cache.

F. Summary

More generally, the data structures required by the above policies are summarized Figure 1. Rectangles with stripes in the LLC are the monitored lines. They are copied in the sampler, illustrated on the left of the LLC. The sampler contains meta-data about lines like tag or PCs. Upon an incoming request, the LLC is accessed and the sampler checks if the access is done on one of the sampled lines. If so, it is updated. The sampler is connected to the predictor. When the former is accessed, the later is updated. The Hawkeye policy requires an extra step between the sampler and the predictor called OPTgen. This step reproduces the MIN algorithm presented in Section III-E. The result is used to train the predictor. The predictor is used to change the value of the heuristics depicted by the three columns at the right of the LLC. The LRU updates the LRU bits only. SRRIP, DRRIP, SHiP and Hawkeye update the RRPV value. Set-Dueling mechanism is highlighted by the shaded rectangles. Rectangles in the LLC are the monitored lines and according to their activity, i.e hit or miss, the global counter (GC) is modified.

On the other hand, the reviewed cache replacement policies can be classified into two families: coarse-grained granularity versus fine-grained granularity. SHiP, SDBP and Hawkeye are considered as fine-grained prediction policies due to their usage of Program Counters that make prediction for each block. Conversely, LRU, SRRIP and DRRIP are considered as coarse-grained policies because they do not use an identification mechanism such as PC, and make a global prediction for the overall cache. The hardware implementation cost is however higher, even though sampling is used to reduce the storage overhead. Table I summarizes cache replacement policies w.r.t. their efficiency against the memory access patterns mentioned in this paper. Two features are also mentioned, i.e., Set-Dueling and Sampling, which enable to reduce the hardware overhead and to improve the accuracy in the implementation of the policies. We also report the hardware budget evaluated in the specific experimental setup used for our current study (thus, it may differ from the cost found in the corresponding literature). As an example, our SHiP implementation uses sampling on 256 lines while Wu et al. mentioned 64 lines.

	Receny resistant	Trash resitant	Scan resistant	Mixed resistant	Set-Dueling	Sampling	HW budget
LRU	+++	+	+	+			16KB
Random	+	+	+	+			0KB
SRRIP	+++	++	+	+			12KB
DRRIP	++	+++	++	+	x	x	13.6KB
SHiP	+++	+++	+++	++		x	35KB
Hawkeye	+++	+++	+++	+++		x	31.8KB

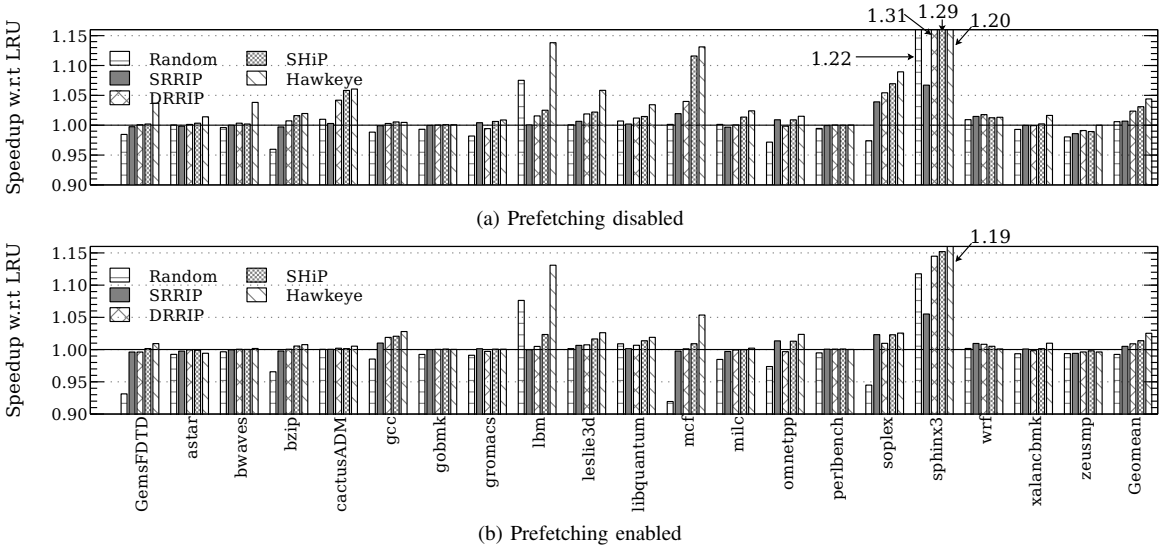
TABLE I: Summary of reviewed cache replacement policies. Hardware budget corresponds to a 16-associative 2MB LLC.

IV. Experimental setup and results

We evaluate the previous cache replacement policies by considering the ChampSim simulator [1] used during the Cache Replacement Championship at ISCA'17 conference. ChampSim models out-of-order (OoO) CPUs, a 3-level cache hierarchy and a main memory. This architecture is based on an Intel Core i7 system. More technical details are given in Table II. We executed 20 traces of the SPEC CPU2006 benchmark suite. Traces are obtained by isolating a single region of interest of 1 billion instructions with SimPoint [10] and collected with Pin [11]. Cores execute 1 billion instructions, with a warm-up period of 200 millions instructions. Average IPC is computed by applying a geometric mean on the IPCs measured for the 20 considered applications, as in previous work [7]. As the Random cache replacement policy is not deterministic (unlike the other policies), it is run 10 times and the resulting geometric mean and average value for IPC

L1 (I/D)	32KB, 8-way, LRU, Private, 4 cycles
L1 D prefetcher	next line
L2	256KB, 8-way, LRU, Unified, 8 cycles
L2 prefetcher	PC based
L3	2MB, 16-way, Shared, 20 cycles
L3 energy/access	0.217 nJ
L3 static power	79.34 mW
CPU number	1
CPU clock	Out-of-Order, 4GHz
DRAM size	4GB, hit: 55 cycles, miss: 165 cycles

TABLE II: Experimental setup configuration



(a) Prefetching disabled

(b) Prefetching enabled

Fig. 2: IPC speedup w.r.t. LRU for a monocore platform without and with prefetching.

and MPKI are reported. The energy model for the LLC is based on CACTI [3], a cache latency and power modeling tool. We use a leakage/area optimization with a temperature of 350K. The considered technology is 32nm. Data array cells are Low Standby operating Power (LSTP) and Mat interconnect is aggressive. Dynamic and static energy are computed according to the following formulas:

$$E_{dyn} = (N_{read} + N_{write}) \times E_{access} \quad (7)$$

$$E_{stat} = ExecTime \times W_{leak} \quad (8)$$

where E_{dyn} and E_{stat} respectively denote the dynamic and static energy; N_{read} and N_{write} are the total numbers of reads and writes on the LLC respectively; E_{access} is the energy of either a read or a write on the LLC; $ExecTime$ is the execution time and W_{leak} represents the static power.

Two architectures are addressed: *a)* monocore without prefetching and *b)* monocore with prefetching. Given these two architectures, we evaluate the performance and energy consumption of the LLC with the following policies: LRU, Random, SRRIP, DRRIP, SHIP and Hawkeye.

Figure 2a depicts IPC results with a monocore configuration without prefetching. On average, all policies outperform the LRU. The most efficient is Hawkeye, with an average speedup of 1.04. Applications that are memory-intensive such as *lbm*, *mcf* or *soplex* exhibit a greater speedup than others. The best performance improvement is achieved on the *sphinx3* application. Random, SRRIP, DRRIP, SHiP and Hawkeye policies respectively perform a speedup of 1.22, 1.06, 1.31, 1.29 and 1.20. In addition, Hawkeye provides consistent results with no performance loss for any of the benchmarks, unlike others such as for *zeusmp*. Performance results on a single-core configuration with prefetching w.r.t. LRU are shown in Figure 2b. Similar trends are observed, but the speedup is lower compared to the configuration without prefetching. Moreover, the Random policy no longer outperforms LRU. The average speedup for the four considered policies varies between 1.01 (for SRRIP) and 1.03 (for Hawkeye).

However, the cache is more efficient when prefetching is activated. Indeed, Figure 3 shows that the average MPKI is drastically reduced by 48.2% in this case. This improvement affects the IPC, which is increased by 16% on average for all replacement policies. Thus, an important insight here is that prefetching has a positive effect on both MPKI and IPC. We also notice that this positive impact on IPC and MPKI is more visible on LRU compared to the other policies, as shown in Figure 3. Thus, cache replacement policies can make eviction decisions that are in conflict with the prefetcher, then benefit less from the positive effect of the latter.

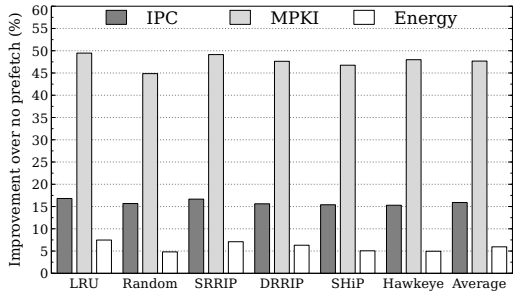
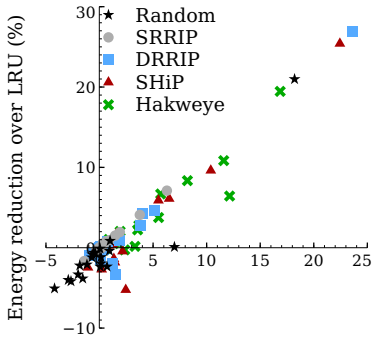
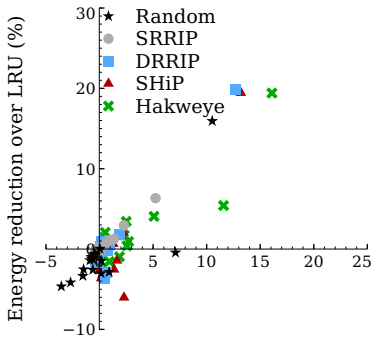


Fig. 3: IPC, MPKI and energy improvement w.r.t. a configuration without prefetching.

One reason for this observation lies in the fact the prefetching system generates extra traffic in the memory hierarchy, which is hardly tractable for the prediction system that heavily relies on Program Counters. In such a situation it becomes challenging to predict block usage. Moreover, except Hawkeye, all policies analyzed in this paper do not take into account prefetched blocks. As a result, the LLC can be polluted by blocks that are wrongly prefetched, or prefetched blocks can be inadequately evicted due to lack of information. Yet, Figure 3 shows that SRRIP, DRRIP and SHiP benefit more



(a) Prefetching disabled



(b) Prefetching enabled

Fig. 4: IPC and energy improvement of evaluated policies over LRU for the 20 considered applications.

from the prefetching than Hawkeye in terms of IPC despite its advanced mechanisms. This suggests that there is room for improvements about the adequate exploitation of prefetching with advanced cache replacement policy.

The energy impact of cache replacement policies has been studied only marginally in the literature. Yet, this is an important topic that deserves a deeper understanding. These policies try to reduce the number of cache misses in general. Thus, the computation can be faster while the static energy can be reduced. Figure 4a shows the IPC improvement as a function of the energy improvement, w.r.t. LRU. A linear trend is observed: when the IPC is improved, the energy consumption is reduced accordingly. This can be explained by a faster computation reducing the static energy of the LLC.

Figure 4b shows that a prefetching system has less impact on the energy consumption, compared to a similar configuration without prefetching. Moreover, the linear trend observed previously is less pronounced. This is mainly due to two factors. Firstly, the prefetching is poorly (or not) managed by the covered replacement policies. As depicted in Figure 3, despite its advanced features, the Hawkeye policy benefits less from prefetching than the other policies in terms of energy reduction. Secondly, prefetching increases the dynamic energy of the cache since more memory access requests have to be processed. For example, it increases by 40% the amount of requests received by the LLC for the *mcf* application, with all policies. On the one hand, the average static energy, which represents 78.4% of the energy consumption, is reduced by 9.4% thanks to a faster execution. On the other hand, the average dynamic energy is increased by 37.2%. Finally, the total energy consumption is reduced by only 5.5%.

V. Conclusion and Future Direction

In this paper, we reviewed four popular cache replacement policies proposed in the literature during the last decade. These policies are not currently implemented in real systems-on-chip, as they require extra hardware which comes with an additional design complexity. We considered their implementation models within the ChampSim simulator for a cross-evaluation. We assessed their impact on the performance and energy consumption of the LLC by measuring their induced Instructions Per Cycle, and their static and dynamic energy improvements. The SPEC CPU2006 benchmark suite has been considered in the experiments. While the obtained results confirmed the improvement of the evaluated policies over the commonly used LRU policy, an important gained insight is that their combination with the prefetching mechanism in modern architectures is very challenging for reaching improved performance. Moreover, the energy analysis shows that Hawkeye, the best replacement policy in the literature in terms of performance, benefits less from prefetching than other policies regarding energy improvement. Thus, future cache replacement policies should consider this issue in order to achieve the best performance/energy trade-off.

Acknowledgements

This work has been funded by the French ANR agency under the grant ANR-15-CE25-0007-01, within the framework of the CONTINUUM project.

References

[1] "The ChampSim simulator," https://github.com/ChampSim/ChampSim.
[2] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.
[3] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.
[4] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *ACM SIGARCH Comp. Arch. News*, vol. 38, no. 3, 2010, pp. 60–71.
[5] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Arch. News*, vol. 35, no. 2, 2007, pp. 381–391.
[6] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Int'l Symp. on Microarch. (MICRO)*, 2011, pp. 430–441.
[7] A. Jain and C. Lin, "Back to the future: leveraging Belady's algorithm for improved cache replacement," in *Int'l Symp. on Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual*. IEEE, 2016, pp. 78–89.
[8] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.
[9] L. A. Belady and F. P. Palermo, "On-line measurement of paging behavior by the multivalued min algorithm," *IBM Journal of Research and Development*, vol. 18, no. 1, pp. 2–19, 1974.
[10] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.
[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.