LRU-PEA: A Smart Replacement Policy for Non-Uniform Cache Architectures on Chip Multiprocessors

Javier Lira[ϒ], Carlos Molina[Φ] and Antonio González[Ψ]
[ϒ] *Universitat Politècnica de Catalunya*
[Φ] *Universitat Rovira i Virgili*
[Ψ] *Intel Barcelona Research Center, Intel Labs - UPC*
javier.lira@ac.upc.edu, carlos.molina@urv.net and antonio.gonzalez@intel.com

***Abstract*— The increasing speed-gap between processor and memory and the limited memory bandwidth make last-level cache performance crucial for CMP architectures. Non Uniform Cache Architectures (NUCA) has been introduced to deal with this problem. This memory organization divides the whole memory space into smaller pieces or banks allowing nearer banks to have better access latencies than further banks. Moreover, an adaptive replacement policy that efficiently reduces misses in the last-level cache could boost performance, particularly if set associativity is assumed. Unfortunately, traditional replacement policies do not behave properly as they were assumed for single-processors. This paper focuses on *Bank Replacement*. This policy involves three key decisions when there is a miss: where to place a data within the cache set, which data to evict from the cache set and finally, where to place the evicted data. We propose a novel replacement technique that enables more intelligent replacement decisions to be taken, based on the observation that some type of data are less commonly accessed depending of the bank where they reside. We call this technique as LRU-PEA (Least Recently Used with a Priority Eviction Approach). We show that the proposed technique significantly reduces the requests to the off-chip memory by increasing the hit ratio in the NUCA cache. This translates into an average IPC improvement of 8% and into an Energy per Instruction (EPI) reduction of 5%.**

I. Introduction

Memory system is a pivotal component which can boost or decrease performance dramatically. Chip Multiprocessors (CMPs) typically incorporate large and shared last-level caches (LLCs) with a homogeneous access time. However, the increasing influence of wire delay in cache design means that access latencies to the last-level cache banks are no longer constant [1]. Non-Uniform Cache Architectures (NUCAs) have been proposed [2] to address this problem. A NUCA divides the whole cache memory into smaller banks and allows nearer cache banks to have lower access latencies than farther banks, and thus mitigates the effects of the cache's internal wires. Therefore, each bank behaves as a regular cache and all of them are connected by means of an interconnection network.

When an incoming data arrives to a NUCA cache, first is determined the bank where a new data should be placed by means of a *placement policy*. Then, *replacement policy* must make three decisions considering a set-associative bank: (1) where to insert new data into the bank, (2) which data to evict from the bank and (3) where to place the evicted data. Notice that a direct-mapped bank only needs to determine what to do with the evicted data. As the number of cores on chip increases, so does the contention caused by applications sharing the LLC. Thus, performance of such systems is heavily influenced by how efficiently the shared cache is managed. Therefore, an efficient and adaptive *replacement policy* could boost performance.

One of the key challenges in a *replacement policy* is choosing the most appropriate data to evict. For this reason, an oracle that knows the future of the program would be the best data eviction policy. Unfortunately, this is not affordable. Thus, most policies already proposed in the literature choose the data to be evicted from the cache on the basis of history, for example, whether the data was accessed after its allocation.

In this work, we propose a novel technique called LRU-PEA (*Least Recently Used with a Priority Evicted Approach*) that selects evicted data based on an explicit NUCA data characterization. This approach relies on the traditional LRU scheme but introducing priorisation of data within a single bank of the NUCA cache. Furthermore, this mechanism *globalize* the replacement decisions that have been taken in a single bank to multiple banks within the NUCA cache.

The remainder of this paper is structured as follows. Section II describes the baseline architecture used in our studies. Section III motivates this work by analyzing the characteristics of cache accesses, followed by Section IV that presents the experimental methodology. Section V introduces the proposed mechanism, and Section VI analyses its performance and energy consumption. Related work is discussed in Section VII and concluding remarks are given in Section VIII.

II. Baseline Architecture

As illustrated in Figure 1, the baseline architecture consists of an eight-processor CMP based on that of Beckmann and Wood [3]. The processors are located on the edges of the NUCA cache, which occupies the central part of the chip. Each processor provides the first-level cache memory, composed of two separated caches: one for instructions and one for data. The NUCA cache is then the second-level cache memory and is shared by the eight processors. The NUCA

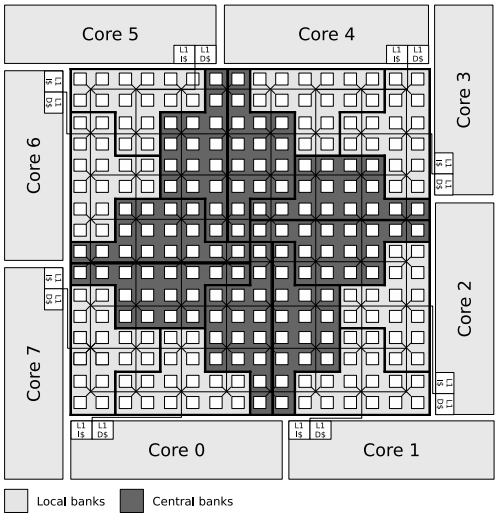

Fig. 1. Baseline architecture layout.

cache is divided into 256 banks structured in a 16x16 mesh that are connected via a 2D mesh interconnection network. The banks in the NUCA cache are also *logically* separated into 16 banksets that are either *local banks* (shaded light in Figure 1) or *central banks* (shaded dark in Figure 1) according to their physical distance from the processors. A NUCA model can be characterized by describing how it behaves with the following four policies: *Bank Placement Policy*, *Bank Migration Policy*, *Bank Access Policy* and *Bank Replacement Policy*. Thus, we can describe the behaviour of the baseline NUCA model for each of the four NUCA policies:

a) ***Bank Placement Policy****:* This policy determines in which bank of the NUCA cache memory a data element should be placed when it comes from the off-chip memory or from other caches. We assume that an address can only be mapped to a single bank within the bank-set. Therefore, a data has *16 possible placements in the NUCA cache* (eight *local banks* and eight *central banks*). The incoming data from the main memory is placed statically within a bank, whereas the incoming data from the first-level cache is placed in the closest *local bank* of the requestor core.

b) ***Bank Migration Policy****:* This policy triggers when a hit has been produced in the cache. It determines if a data element is allowed to change its placement from one bank to another bank, which data should be migrated, when this data should be migrated and to which bank it should be moved. We assume *gradual promotion*. Thus, when there is a hit in a NUCA bank, the accessed data is promoted to a bank that is one-step closer to the processor that has just accessed it.

c) ***Bank Access Policy****:* This policy determines the bank-searching algorithm in the NUCA cache memory space. We assume a two-step data search algorithm in the NUCA cache. The first phase of this algorithm broadcasts a request to the *local bank* that is closest to the processor that launched the search, and to the appropriate eight *central banks*. If all nine initial requests miss, the request is broadcasted to the remaining seven banks where the address can be placed in the NUCA cache. Only after a request misses all 16 banks will a request be sent off-chip.

d) ***Bank Replacement Policy****:* This policy triggers when a miss has been produced in the cache or when a data comes from a lower memory level. Once the *Bank Placement Policy* determines the bank where data should be placed, *Bank Replacement Policy* determines how data is inserted within the bank and which data is evicted from that bank. Therefore, several decisions must to be taken, particularly if set-associativity is assumed. For the sake of simplicity, this policy can be split into three more separated sub-policies:

- *Data Insertion Policy:* It determines the position within the replacement stack of the cache set where the incoming data must be placed. We assume the Most-Recently Used (MRU) mechanism, which is the most commonly used approach in cache organizations.
- *Data Eviction Policy:* It determines the data that must leave the cache when an incoming data arrives. We assume the Least-Recently Used (LRU) mechanism, which is also the most commonly used approach in all type of cache organizations.
- *Data Target Policy:* It determines the destination of the data that has been replaced. We assume that the evicted data is sent directly back to the off-chip memory.

III. Motivation

Applying a *bank migration policy* that moves data within the cache is one of the most interesting features of NUCA caches. This enables recently accessed data to be stored close to the requesting core in order to optimize access response times for future accesses. These movements, however, often concentrate most recently accessed data in the NUCA banks that are next to the cores. As a result, NUCA banks behave differently according to their physical location within the cache. Whereas banks that are close to the cores (*local banks*) usually store the most recently accessed data, *central banks*, which are physically located in the centre of the NUCA cache, store lines that are moving into or leaving *local banks*, because the bank migration policy implemented uses gradual promotions. Apart from this, *bank placement policy* also introduces a difference between *local banks* and *central banks*. Thus, when data is evicted by a first-level cache, it is relocated to the closest local bank in the NUCA cache. Furthermore, an incoming line from the off-chip memory is statically placed within a NUCA bank.

On the basis of these two features, we classify data in the NUCA cache into four categories: (1) data that has just been promoted (*promoted*), (2) data that has just been demoted (*demoted*), (3) data that has just arrived from the off-chip memory (*offchip*) and (4) data that has just arrived from a first-level cache (*L1 replacement*).

We analysed the behaviour of the NUCA cache with regard to these four categories, and we found that nearly all accesses that were satisfied by local banks were *L1 replacement* data. However, the vast majority of accesses satisfied by central banks were *promoted* data.

Based on this observation, we propose a novel replacement policy that benefits from the type of data (promoted, demoted, off-chip and L1 replacement) that is most commonly accessed from local and central banks. We refer to this mechanism as LRU-PEA (Least Recently Used with a Priority Eviction Approach).

IV. Experimental Methodology

A. Simulation Environment

We use the full-system execution-driven simulator, Simics [4], extended with the GEMS toolset [5]. GEMS provides a detailed memory-system timing model that enables us to model the NUCA cache architecture. Furthermore, it perfectly models network contention introduced by all simulated mechanisms. The simulated architecture is structured as a single CMP made up of eight UltraSPARC IIIi homogeneous cores. With regard to the memory hierarchy, each core provides a split first-level cache (data and instructions). The second level of the memory hierarchy is the NUCA cache. We used the MOESI token-based coherence protocol to maintain correctness and robustness in the memory system. Table I summarizes the configuration parameters used in our studies. The access latencies of the memory components are based on the models done with the CACTI 6.0 [6] modelling tool, that is the first version of CACTI that provides support for modelling NUCA caches.

Processors	8 - UltraSPARC IIIi
Frequency	1.5 GHz
Block size	64 bytes
L1 Cache (Instr./Data)	32 KBytes, 2-way
L2 Cache (NUCA)	8 MBytes, 256 Banks
NUCA Bank	32 KBytes, 8-way
L1 Latency	3 cycles
NUCA Bank Latency	4 cycles
Router Latency	1 cycle
Avg NUCA Miss Latency	250 cycles

TABLE I
Configuration parameters.

We simulated the whole set of applications from the PARSEC benchmark suite [7] with the *simlarge* input data sets. This suite contains 13 programs from many different areas such as image processing, financial analytics, video encoding, computer vision and animation physics, among others. Regarding the methodology used for the simulations, first we skipped both the initialization and thread creation phases, then we fast-forwarded while warming all caches for 500 million cycles. Finally, we performed a detailed simulation for 200 million cycles.

As performance metric, we use the aggregate number of user instructions committed per cycle, which is proportional to overall system throughput [8].

B. Energy Model

We deal with a similar energy model that is assumed by Bardine et al [9]. Therefore, we also consider the static and dynamic energy dissipated by the NUCA cache and the additional energy required to access the off-chip memory. Total energy dissipated by the NUCA cache is the addition of all three components:

$$E_{total} = E_{static} + E_{dynamic} + E_{off-chip}$$

NUCA cache is modeled by means of the CACTI 6.0 tool [6] to obtain the static energy (E_{static}). The dynamic energy ($E_{dynamic}$) is modeled by the GEMS toolset [5] that deals with the Orion simulator to take into account the energy per bank access, the energy required to transmit a flit on the network link and the energy required to switch a flit through a network switch. The extra network traffic introduced by our proposal is also considered and perfectly modeled into the simulator.

The energy dissipated by the off-chip memory is derived from the *Micron System Power Calculator* [10] assuming a modern DDR3 system (4GB, 8DQs, Vdd:1.5v, 333 MHz). We focus our evaluation of the off-chip memory on the energy dissipated during active cycles isolating it from the background one. This study concludes that the average energy of each access is 550 pJ.

Energy per instruction (EPI) [11] is assumed as a metric to analyze the consumption results. This metric is independent of the amount of time required to process an instruction and ideal for throughput performance.

V. LRU with Priority Eviction Aproach (LRU-PEA)

As described in Section II, a *Bank Replacement Policy* can be divided into the following three sub-policies: *data insertion policy*, *data eviction policy* and *data target policy*. In this section we introduce *Least Recently Used with Priority Eviction Approach (LRU-PEA)* replacement policy. This policy focuses on optimizing performance of applications on Non-Uniform Cache Architectures for Chip Multiprocessors by analyzing data behaviour within the NUCA cache and trying to keep the most accessed data in cache as much time as possible. In order to describe how this policy works, we describe separately the two sub-policies that LRU-PEA modifies: *data eviction policy* and *data target policy*. With regard to *data insertion policy*, we assume MRU mechanism (same as baseline).

A. Data Eviction Policy

LRU-PEA statically prioritises the previously defined categories (*promoted*, *demoted*, *offchip* and *L1 replacement*). However, the two groups of banks are too different, so LRU-PEA defines the prioritisation for both local and central banks. Having a static prioritisation, however, could cause the highest-category data to monopolize the NUCA cache, or even cause a simple data to stay in the cache forever. In order to avoid these situations, we restrict the category comparison to the two last positions in the LRU-stack. In this way, even data with the lowest category will stay in the cache until it arrives at the LRU-1 position in the LRU-stack.

Figure 2 gives an example of how the *LRU-PEA* scheme works. First, we define the prioritisation of the data categories. For instance, the prioritisation of the example

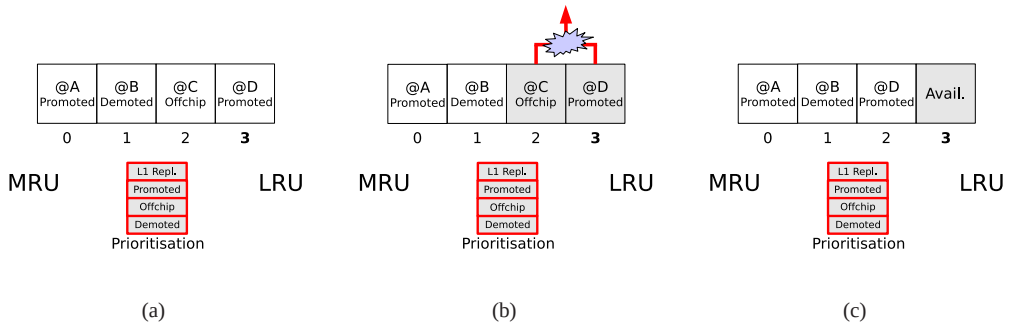

Fig. 2. LRU-PEA scheme. (a) Initial state of the LRU-stack. (b) The last two positions of the LRU-stack compete to avoid being evicted. (c) The lowest category data has been evicted.

is as follows (see Figure 2(a)): 1) L1 Replacement, 2) Promoted, 3) Offchip and 4) Demoted. When the LRU-PEA eviction policy is applied (see Figure 2(b)), the last two positions of the LRU-stack compete to find out which one is going to be evicted. Thus, we can compare their categories. If they are different, the data with the lower category is evicted. But, if both have the same category, the line that currently occupies the LRU position is evicted. Finally, the data that has not been evicted updates its position within the LRU-stack (see Figure 2(c)).

By analysing NUCA cache behaviour, we found that local banks access more frequently to data that owns to *L1 Replacement* category, while central banks mainly access to *promoted* data. Thus, we evaluate LRU-PEA assuming the prioritisation described in Table II.

	BANK	
	Local	**Central**
+	L1 Replacement	Promoted
Priority	Promoted	Offchip
	Offchip	Demoted
-	Demoted	L1 Replacement

TABLE II
PRIORITISATION FOR LRU-PEA.

We have also experimentally observed that modifying the priority order of the lower categories does not introduce significant differences in terms of performance.

B. Data Target Policy

There are two key issues when a Dynamic-NUCA (D-NUCA) architecture [2] is considered: 1) a single data can be mapped in multiple banks within the NUCA cache, and 2) the migration process moves the most accessed data to the banks that are closer to the requesting cores. Therefore, banks usage in a NUCA cache is heavily imbalanced, and a capacity miss in a heavy-used NUCA bank could provoke the eviction from the NUCA cache of constantly accessed data, while other NUCA banks are storing less frequently accessed data. LRU-PEA addresses this problem by defining a *data target policy* that allows to *spread* the replacement decision that has been taken in a single bank to all banks in the NUCA cache where evicted data can be mapped.

```
Input: initial_bank: Bank that started the replacement process
Input: ev_data: Evicted data
Output: Final data to be evicted from the cache
begin
    final = false;
    if Category(initial_bank, ev_data) == LOWEST_CATEG then
        return ev_data;
    end
    next_bank = NextBank(initial_bank);
    ev_bank = initial_bank;
    while !final and next_bank ≠ initial_bank do
        may_evict_data = ApplyLRU-PEA(next_bank, ev_data);
        if Category(ev_bank, ev_data) > Category(next_bank,
        may_evict_data) then
            InsertIntoBank(next_bank, ev_data);
            ev_data = may_evict_data;
            ev_bank = next_bank;
            if IsCascadeModeEnabled() == false then
                final = true;
            else if Category(ev_bank, ev_data) > LOWEST_CATEG then
                next_bank = NextBank(next_bank);
            else
                final = true;
            end
        else
            next_bank = NextBank(next_bank);
        end
    end
    return ev_data;
end
```

Algorithm 1: LRU-PEA scheme

The algorithm that we propose as data target policy of LRU-PEA is described in Algorithm 1. The main idea of this algorithm is to find a NUCA bank whose victim data owns to a category with less priority than the one that is currently being evicted. In that way, while the target NUCA bank is not found, all NUCA banks where the evicted data can be mapped are sequentially accessed in an statically defined order. In our evaluation we use the following order: Local_Bank_Core$_i$ $\rightarrow$ Central_Bank_Core$_i$ $\rightarrow$ Local_Bank_Core$_{i+1}$ $\rightarrow$ Central_Bank_Core$_{i+1}$ $\rightarrow$ ...

The algorithm finishes when one of these three conditions succeeds: 1) the evicted data owns to the lowest priority category, 2) all NUCA banks where the evicted data can be mapped have been already visited, and 3) the evicted data has been relocated to other NUCA bank. Then, whether the evicted data could not be relocated to other bank into the

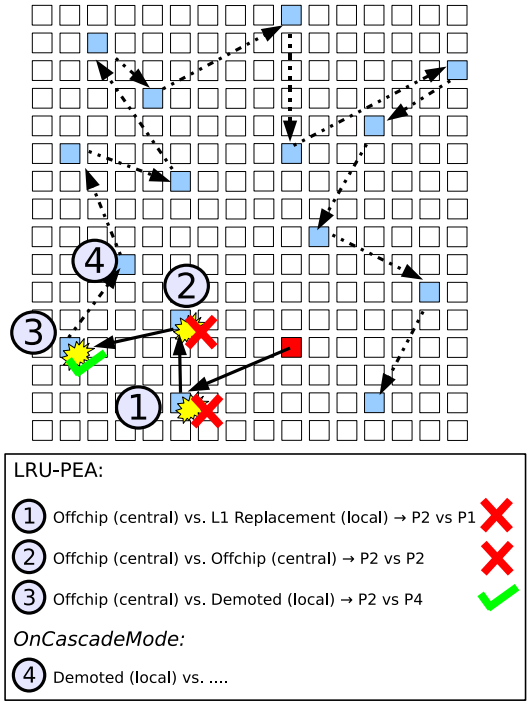

Fig. 3. Example of how LRU-PEA behaves.

NUCA cache, it is written back to the upper-level memory.

By using sequential access, however, the accuracy of LRU-PEA is restricted to the NUCA banks have been visited before finding a target bank. To address this problem, we introduce the *on cascade* mode. Enabling this mode the algorithm does not finish, when the evicted data finds a target bank. From that point, it uses as evicted data, the data that has been evicted from the target bank. Thus, after visiting all NUCA banks we can assure that the current evicted data owns to the category with the lowest priority. In Section VI, we consider both configurations, with *on cascade* mode enabled and disabled.

Figure 3 shows an example of how the LRU-PEA's data target policy works. In this example, the algorithm starts in a central bank and the evicted data owns to the *Offchip* category, so the priority of the evicted data is 2 (see Table II). First, it checks whether the evicted data can be relocated in the local bank of the next core (step 1 in Figure 3). However, the priority of the victim data in the current bank is higher than the evicted data, so LRU-PEA tries to relocate the evicted data into the next bank. In the second step, it visits another central bank. In this case, the category of the victim data in the current bank is the same as the evicted data, and so next bank needs to be checked. Finally, in the third step, the category of the evicted data has higher priority than the one of the victim data of the current bank. Thus, the evicted data is relocated to the current bank. If *on cascade* mode is enabled, the algorithm continues with the 4th step (see Figure 3), but using as evicted data the one that has been evicted from the current bank. Otherwise, this data is directly evicted from the NUCA cache and sent it back to the upper-level memory.

	No Cascade	Cascade Enabled	
		Direct	*Provoked*
1 message	64	54	20
2 messages	12	7	7
3 messages	4	2	4
4 messages	3	2	4
5 messages	3	2	3
6 messages	2	1	4
7 messages	2	1	3
8 messages	2	1	4
9 messages	1	1	3
10 messages	1	1	4
11 messages	1	1	3
12 messages	1	1	6
13 messages	1	1	6
14 messages	1	1	30
15 messages	3	21	-

Values in percentage (%)

TABLE III
NUMBER OF EXTRA MESSAGES INTRODUCED BY BOTH CONFIGURATIONS OF LRU-PEA TO SATISFY REPLACEMENTS.

C. Additional Hardware

This mechanism requires to introduce some additional hardware to the NUCA cache. In order to know which is the category of data, we add two bits per line (there is four categories). Then, assuming the 8 MBytes NUCA cache described in Section IV, LRU-PEA will require to add 32 KBytes, which results less than 0.4% of hardware overhead. Furthermore, behaviour of the proposed can be easily implemented without significant complexity.

VI. Results and Analysis

This section analyses the impact of assuming LRU-PEA as *bank replacement policy*. LRU-PEA takes advantage of on-chip network introduced by CMPs to provide a sophisticated algorithm that allows to *globalize* the replacement decisions that have been taken in a single bank. Although this approach may increase contention in the on-chip network, it is perfectly modeled in our simulator. Table III shows the average number of extra messages introduced by LRU-PEA to satisfy a single replacement. When disabling *on cascade* mode, the communication overhead introduced by LRU-PEA is very low. On average, close to 80% of replacements are satisfied by introducing up to 3 extra messages into the on-chip network. By enabling *on cascade* mode, however, a significant percentage of replacements introduce the maximum number of messages into the network (the number of banks where the evicted data can be mapped minus one). This difference between the two modes can be reasoned by the high-accuracy provided by LRU-PEA when the *on cascade* mode is enabled. In general, data in NUCA banks has higher priority, and then is much more difficult to find a victim data with lower priority than the evicted one. In the following sections we analyse how LRU-PEA behaves in terms of performance and energy consumption.

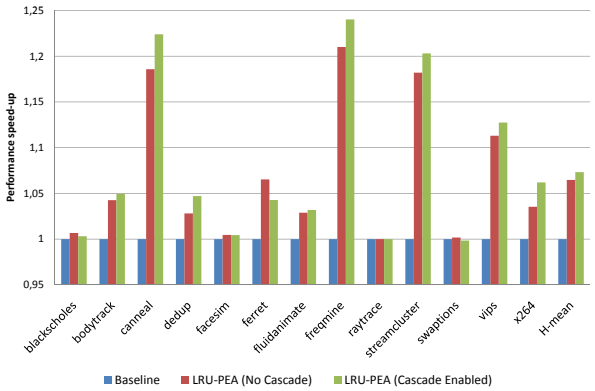

Fig. 4. IPC improvement with LRU-PEA.

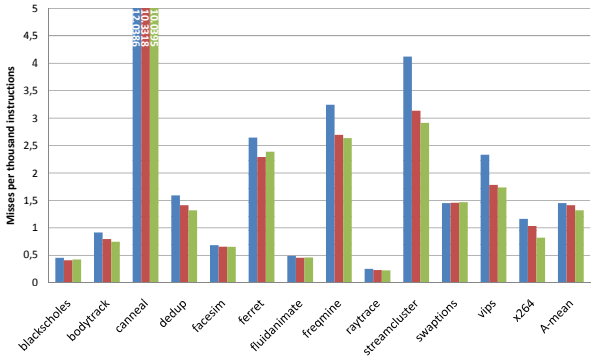

Fig. 5. Misses per thousand instructions with LRU-PEA.

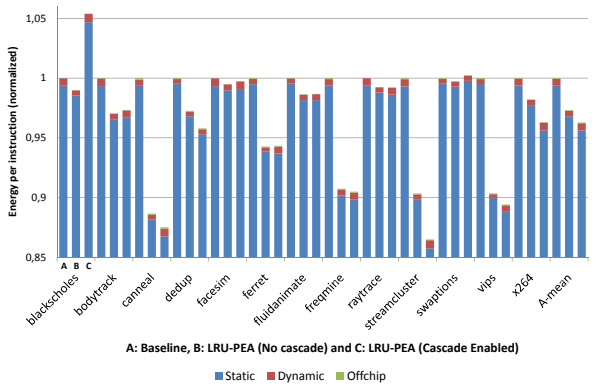

Fig. 6. Normalized average energy consumed per each executed instruction.

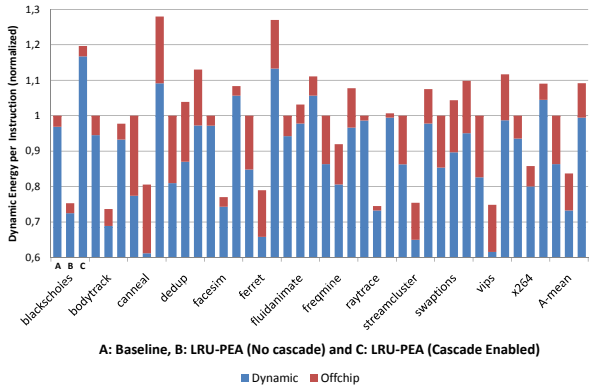

Fig. 7. Normalized average dynamic energy consumed per each instruction.

A. Performance Analysis

Figure 4 shows the IPC improvement achieved when using LRU-PEA as *bank replacement policy* in the NUCA cache. On average, we find that LRU-PEA increases 8% IPC with respect the baseline architecture if *on cascade* mode is enabled, and 7% when it is disabled. In general, we find that LRU-PEA achieves significant IPC improvement with most PARSEC applications, obtaining near 20% of improvement in three of them (*canneal*, *freqmine* and *streamcluster*). On the other hand, only 4 of 13 PARSEC applications do not show perfomance benefits when using LRU-PEA (*blackscholes*, *facesim*, *raytrace* and *swaptions*). We also observe that although LRU-PEA does not achieve significant performance improvement in some of the PARSEC applications, this mechanism is not harmful in terms of performance.

Figure 5 shows the NUCA misses per 1000 instructions (MPKI) with the three evaluated configurations: baseline, LRU-PEA and LRU-PEA with *on cascade* mode enabled. On average, we observe that there is a significant reduction in MPKI by using LRU-PEA, and even more when *on cascade* mode is enabled. In general, we find that PARSEC applications that provide performance improvements, also provide significant MPKI reduction. Moreover, we highlight that *canneal*, *freqmine* and *streamcluster*, that are the applications that provide the highest IPC improvement with LRU-PEA, also have the highest MPKI. In contrast, applications that have MPKI close to zero usually do not introduce performance improvements when using LRU-PEA.

With regard to the applications in which LRU-PEA does not obtain performance benefits, *blackscholes* and *swaptions* are financial applications with small working sets, so their cache requirements are restricted. On the other hand, *raytrace* and *facesim* have huge working sets, but they are computational intensive and mainly exploit temporal locality.

B. Energy Consumption Analysis

The energy consumption is analysed by using the Energy per Instruction (EPI) metric. Figure 6 shows that, on average, LRU-PEA reduces the energy consumed per each instruction compared to the baseline architecture by 5% with both configurations (with and without *on cascade* mode enabled). In particular, LRU-PEA obtains significant energy consumption reduction in PARSEC applications with large working sets, such as *canneal*, *freqmine* and *streamcluster*. Moreover, we observe that, with the exception of *blackscholes* and *swaptions*, EPI is always reduced by using LRU-PEA.

As we can see in Figure 6, EPI is heavily influenced by static energy. Figure 7 shows the normalized EPI without taking into consideration static energy consumed. We find that when *on cascade* mode is enabled, dynamic energy consumed is 10% higher than assuming the baseline configuration. However, LRU-PEA with *on cascade* mode disabled still reduces EPI in more than 15%. This difference

between the two LRU-PEA modes responds to the amount of extra messages introduced into the on-chip network by each of them (see Table III).

Finally, we highlight that although LRU-PEA increases the on-chip network contention, the average energy consumed per instruction is still reduced due to the significant performance improvement that this mechanism provides.

VII. Related Work

Replacement policy brings together two decisions that can be seen as two more policies: data insertion and data eviction. The former decides where to place data and the latter decides which data is replaced. Traditionally, caches deal with the Most Recently Used (MRU) algorithm to insert data and the Least Recently Used (LRU) algorithm to evict data [12], [13].

Modifications to the traditional LRU scheme have been also proposed. Wong and Bauer [14] modified the standard LRU to maintain data that exhibit higher temporal locality. Alghazo et al. [15] proposed a mechanism called as SF-LRU (Second-Chance Frequency LRU). This scheme combines both the recentness (LRU) and frequency (LFU) of blocks to decide which blocks to replace. Dybdahl et al. [16] also proposed another LRU approach based on frequency of access in shared multiprocessor caches. Kharbutli and Solihin [17] proposed a counter-based L2 cache replacement. This approach includes an event counter with each line that is incremented under certain circumstances. Then, the line can be evicted when this counter achieves a certain threshold.

Recently, several papers have revisited data insertion policy. Qureshi et al. [18] propose *Line Distillation*, a mechanism that tries to keep frequently accessed data in a cache line and to evict unused data. This technique is based on the observation that, generally, data is unlikely to be used in the lowest priority part of the LRU stack. They also proposed LIP (LRU Insertion Policy), which places data in the LRU position instead of the MRU position [19].

Kim et al. [2] introduced the concept of Non-Uniform Cache Architecture (NUCA). They observed that the increase in wire delays would make cache access times no longer a constant. Instead, latency would become a linear-function of the line's physical location within the cache. From this observation, several NUCA architectures were designed by partitioning the cache into multiple banks and using a switched network to connect these banks. However, the introduction of CMP architectures posed additional challenges to the NUCA architecture leading Beckmann and Wood [3] to analyse NUCA for CMP. Recent studies have explored policies for bank placement [20], bank migration [21], bank access [22] and bank replacement [2] in NUCA caches. None of these studies properly addresses bank replacement policy in a CMP environment.

VIII. Conclusions

The increasing gap between processor and memory speed and the limited memory bandwidth make last-level cache performance crucial for CMP architectures. Reducing last-level cache misses, therefore, will provide significant performance benefits. In this paper we propose a novel alternative to the traditional LRU replacement policy. It aims to make more intelligent replacement decisions by protecting the cache lines that are likely to be reaccessed. On average, LRU-PEA replacement policy obtains 8% IPC improvement with respect to the baseline configuration, and reduces the energy consumption per instruction by 5%.

In conclusion, in this paper we demonstrate that minimal modifications in the bank replacement policy to enable more intelligent eviction and target decisions will result in significant performance and consumption benefits.

IX. Acknowledgements

This work is supported by the Spanish Ministry of Science and Innovation (MCI) and FEDER funds of the EU under the contracts TIN 2007-61763 and TIN 2007-68050-C03-03, the Generalitat de Catalunya under grant 2005SGR00950, and Intel Corporation. Javier Lira is supported by the MCI under FPI grant BES-2008-003177.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate vs. ipc: The end of the road for conventional microprocessors," in *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[2] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[3] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," in *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[4] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, *Simics: A Full System Simulator Platform*. Computer, 2002, vol. 35-2, pp. 50–58.

[5] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," in *Computer Architecture News*, Sept. 2005.

[6] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[8] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[9] A. Bardine, P. Foglia, G. Gabrielli, and C. A. Prete, "Analysis of static and dynamic energy consumption in nuca caches: Initial results," in *Proceedings of the 2007 Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, 2007.

[10] Micron system power calculator. [Online]. Available: *http://www.micron.com/support/part_info/powercalc*

[11] E. Grochowski, R. Ronen, J. Shen, and H. Wang, "Best of both latency and throughput," in *Proceedings of the 22nd International Conference on Computer Design*, 2004.

[12] L. A. Belady, "A study of replacement algorithms for virtual-storage computer," *IMB Systems Journal*, vol. 5, no. 2, 1966.

[13] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, 1982.

[14] W. Wong and J. Baer, "Modified lru policies for improving second-level cache behavior," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, 2000.

[15] J. Alghazo, A. Akaaboune, and N. Botros, “Sf-lru cache replacement algorithm,” in *Records of the International Workshop on Memory Technology, Design and Testing*, 2004.

[16] H. Dybdahl, P. Stenström, and L. Natvig, “An lru-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches,” *Computer Architecture News*, vol. 35, 2007.

[17] M. Kharbutli and Y. Solihin, “Counter-based cache replacement algorithms,” in *Proceedings of the 23rd International Conference on Computer Design*, 2005.

[18] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, “Line distillation: Increasing cache capacity by filtering unused words in cache lines,” in *Proceedings of the 13th International Symposium of High-Performance Computer Architecture*, 2007.

[19] M. K. Qureshi, A. Jaleel, and Y. N. Patt, “Adaptive insertion policies for high-performance caching,” in *Proceedings of the 34th International Symposium on Computer Architecture*, 2007.

[20] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, “A nuca substrate for flexible cmp cache sharing,” in *Proceedings of the 19th ACM International Conference on Supercomputing*, 2005.

[21] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son, “A novel migration-based nuca design for chip multiprocessors,” in *Proceedings of the ACM/IEEE conference on Supercomputing*, 2008.

[22] N. Muralimanohar and R. Balasubramonian, “Interconnect design considerations for large nuca caches,” in *Proceedings of the 34th International Symposium on Computer Architecture*, 2007.