This is a pre-print of an article published in Journal of Supercomputing. The final authenticated version is available online at: https://doi.org/10.1007/s11227-012-0859-6

Replacement Techniques for Dynamic NUCA Cache Designs on CMPs

Javier Lira[a], Carlos Molina[b], Ryan N. Rakvic[c], Antonio González[d]

[a]*Computer Architecture Department. Universitat Politcnica de Catalunya*
[b]*Computer Engineering Department. Universitat Rovira i Virgili*
[c]*Department of Electrical and Computer Engineering. U.S. Naval Academy*
[d]*Intel Barcelona Research Center. Intel Labs - UPC*

Abstract

The growing influence of wire delay in cache design has meant that access latencies to last-level cache banks are no longer constant. Non-Uniform Cache Architectures (NUCAs) have been proposed to address this problem. Furthermore, an efficient last-level cache is crucial in chip multiprocessors (CMP) architectures to reduce requests to the offchip memory, because of the significant speed gap between processor and memory. Therefore, a bank replacement policy that efficiently manages the NUCA cache is desirable. However, the decentralized nature of NUCA has eliminated the effectiveness of replacement policies because banks operate independently of each other, and hence their replacement decisions are restricted to a single NUCA bank. In this paper, we propose three different techniques to deal with replacements in NUCA caches.

Keywords: D-NUCA, replacement, CMP

1. Introduction

Non-Uniform Cache Architecture (NUCA) caches often migrate accessed lines closer to the core that requested them. Consequently, the most frequently accessed lines are stored in the banks that are closer to the requesting cores, termed *hot banks*. A replacement in a *hot bank*, however, evicts a line whose probabilities of being accessed farther in the program are much higher than that of a line stored in other bank in the NUCA cache. Moreover, as banks in the NUCA cache work independently of each other,

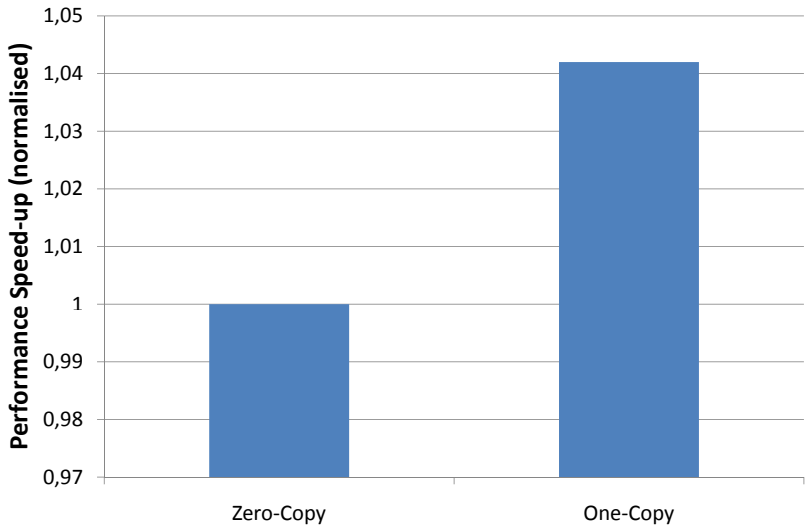


Figure 1: Performance results assuming one-copy and zero-copy replacement policies.

none of the less used banks can even know that a *hot bank* is constantly evicting data blocks that are being reused. Thus, a more sophisticated *replacement policy* that allows all banks in the NUCA cache to take part in data-replacement decisions is desirable. Evicted lines from the *hot banks* can be relocated to other banks in the NUCA cache, instead of being evicted from the NUCA cache permanently.

Unfortunately, most previous works have ignored the replacement issue or have adopted a replacement scheme that was originally designed for use in uniprocessors/uniform-caches. Kim et al. [1] proposed two replacement policies for NUCA caches in a uniprocessor environment: *zero-copy* and *one-copy* policies. For a *zero-copy* policy, the evicted line in the NUCA cache is sent back to the upper level of the memory hierarchy (the main memory in our studies). For a *one-copy* policy, the evicted line is demoted to a more distant bank. This policy gives a second chance to the evicted lines to stay within the NUCA cache. We have adapted these policies to work in our CMP simulation framework. Our version of the *one-copy* policy for CMP gives a second chance to the evicted lines by randomly relocating them to another bank in the NUCA cache where they can be mapped.

Figure 1 shows that the *one-copy* replacement policy improves

performance compared to the baseline configuration[1]. *One-copy*, however, is considered a *blind* replacement policy because it does not take into account the current cache state before relocating the evicted data to another NUCA bank. Thus, this approach may cause unfair data replacements that negatively impact performance.

This paper presents three approaches for dealing with replacements in NUCA caches:

- *Last Bank*: this policy utilizes an extra bank that acts as a victim cache by catching all evictions that happen in the regular NUCA banks.
- *LRU-PEA*: this replacement policy utilizes novel data block categories in order to make the replacement decision. The categories assumed in the LRU-PEA rely on the last migration action taken by a particular data block (e.g. promoted, demoted or none).
- *The Auction*: this replacement framework spreads replacement decisions from a single bank to all banks in the NUCA cache. Thus, other banks can take part in deciding which is the most appropriate data to evict within the whole NUCA cache.

The remainder of this paper is structured as follows. Section 2 describes the baseline configuration assumed in this paper. Section 3 presents our experimental methodology. Sections 4, 5 and 6 present and analyze the *Last Bank* policy, the *LRU-PEA* replacement policy, and *The Auction* policy, respectively. Related work is discussed in Section 7, and concluding remarks are given in Section 8.

2. Baseline NUCA cache architecture

We assume an inclusive fully-shared L2 cache with a NUCA, derived from the Dynamic NUCA (D-NUCA) design by Kim et al. [1]. As in their original proposal we partition the address space across cache banks which are connected via a 2D mesh interconnection network. As illustrated in Figure 2, the NUCA storage is partitioned into 128 banks. D-NUCA allows migration of data towards the cores that use it the most. This technique may reduce the access latency for future accesses to the same data.

[1]The experimental methodology is described in Section 3.

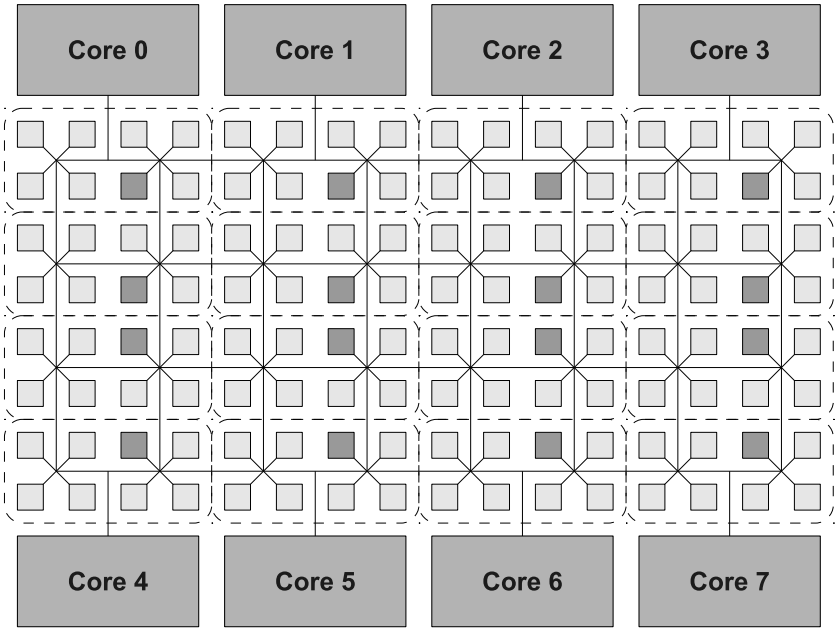


Figure 2: Baseline architecture layout.

Ideally, a data block would be mapped onto any cache bank in order to maximize placement flexibility. However, the overhead of locating a data block in this scenario would be increased as each bank would have to be searched, either through a centralized tag store or by broadcasting the tag to all banks. To mitigate this, the NUCA cache is treated as a set-associative structure, called *banksets*, with each bank holding one "way" of the set. Thus, data blocks can be mapped to any bank within a single bankset. The NUCA banks that make up a bankset are organized into bankclusters within the cache (the dotted boxes in Figure 2). Each bankcluster consists of a single bank from each bankset. As an example, the darker shaded NUCA banks in Figure 2 compose a bankset. As shown in Figure 2, we assume a 16-way bankset associative NUCA cache, organized in 16 bankclusters. The eight bankclusters that are located closest to the cores compose the *local banks*, and the other eight in the center of the NUCA cache are *the central banks*. Therefore, a data block has 16 possible placements in the NUCA cache (eight local banks and eight central banks). Note that a particular bank in the NUCA cache is still set-associative.

The behaviour of a NUCA cache is determined by the following four

policies: *placement*, *access*, *replacement* and *migration*. In order to fully describe the baseline architecture, we show how it behaves in each of the NUCA policies.

2.1. Placement policy

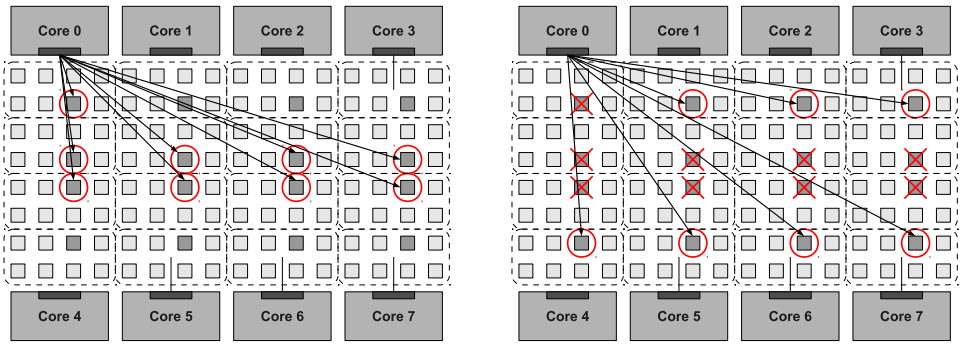
This policy determines where a particular data block can be mapped in the NUCA cache, as well as its initial location when it arrives from the off-chip memory. As previously mentioned, the NUCA cache implements a bankset organization to limit the potential banks where a data block can be mapped. In this case, a data block can be stored in 16 possible banks in the NUCA cache (eight local banks and eight central banks). An incoming data block from the off-chip memory is mapped to a specific bank within the cache. This is statically predetermined based on the lower bits from the data block's address.

2.2. Access Policy

Dynamic features provided by D-NUCA, like having multiple candidate banks store a single data block and migration movements, make access policy a key constraint in NUCA caches. The baseline D-NUCA design uses a two-phase multicast algorithm that is known as *partitioned multicast* [2]. Figure 3 shows the two steps of this algorithm for a request started from core 0. First, it broadcasts a request to the closest *local bank* to the processor that launched the memory request, and to the eight *central banks* (see Figure 3a). If all nine initial requests miss, the request is sent in parallel, to the remaining seven banks from the requested bankset (see Figure 3b). Finally, if the request misses all 16 banks, the request would be forwarded to the off-chip memory.

2.3. Replacement Policy

Upon a replacement in a NUCA bank, this policy determines which data block should be evicted from a particular bank (*data eviction policy*) and the position that the incoming data block will occupy in the LRU-stack (*data insertion policy*). In addition, this policy also determines the final destination of the evicted data block (*data target policy*). Regarding the baseline configuration, the replacement policy assumed within the NUCA banks is LRU, and the incoming data block is set as the MRU line in the bank. Moreover, the evicted data block is directly sent to the off-chip memory (*zero-copy* replacement policy [1]).



(a) First step (b) Second step

Figure 3: Scheme of the access algorithm used in the baseline configuration.

2.4. Migration Policy

Once the data block is in the NUCA cache, the migration scheme determines its optimal position. As a migration policy, we assume *gradual promotion* that has been widely used in the literature [2, 1]. This states that upon a hit in the cache the requested data block should move one-step closer to the core that initiated the memory request. Figure 4 illustrates an example to better understand how this migration policy works in the baseline configuration. In this example, core 0 accesses a data block which is stored in the local bank of core 6 (this is the most pessimistic situation). Then, the requested data block moves one step towards the requesting core, and thus stays in a central bank near core 6. After core 0 accesses the same data block for the second time, it leaves core 6 influence area, and arrives to the central bank of core 0. If the same data is accessed again by the core 0, it moves towards core 0 arriving at the optimal location in terms of access latency for accesses from core 0.

3. Methodology and Experimental Framework

We use the full-system execution-driven simulator, Simics [3], extended with the GEMS toolset [4]. GEMS provides a detailed memory-system timing model that enables us to model the NUCA cache architecture. Furthermore, it accurately models the network contention introduced by the simulated mechanisms. The simulated architecture is structured as a single CMP made up of eight in-order cores with CPI equal to one for non-memory instructions. During the simulation, when a memory instruction appears, Simics stalls,

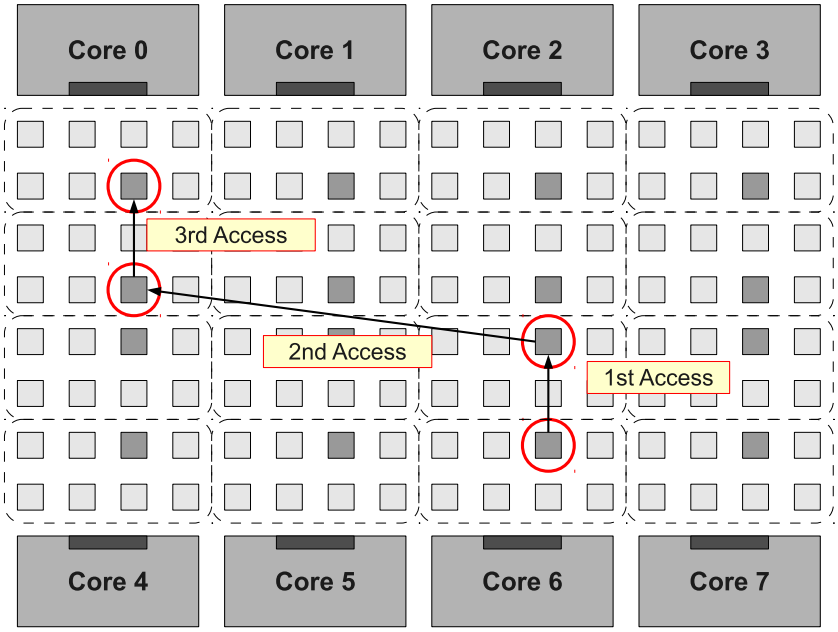


Figure 4: Scheme of the migration algorithm used in the baseline configuration (accesses from core 0).

and GEMS takes control of execution. It provides the number of cycles the core must stall due to the memory request. The processor cores emulate the UltraSPARC IIIi ISA. Each core is augmented with a split first-level cache (data and instruction). The second level of the memory hierarchy is the NUCA cache. This is inclusive and shared among all cores integrated into the chip. In order to exploit cache capacity, replications are not allowed in the NUCA (L2) cache. However, we used MESI coherence protocol to maintain coherency in all private L1 cache memories. Each cache line in the NUCA cache keeps track of which L1 cache has a copy of the stored data. This information moves, as with the whole data block, along the NUCA cache with migration movements. In case of replacement of clean data in the L1 cache, silent replacement is assumed. In other words, the NUCA cache will not be notified that a private L1 cache has evicted a non-dirty data block. Consequently, L1 caches could receive invalidation messages from the NUCA cache even when not having the data. With regard to the memory consistency model, we assume sequential consistency.

Table 1 summarizes the configuration parameters used in our studies. The

access latencies of the memory components are based on the models made with the CACTI 6.0 [5] modeling tool. This is the first version of CACTI that enables NUCA caches to be modeled.

Processors	8 - UltraSPARC IIIi
Frequency	1.5 GHz
Integration Technology	45 nm
Block size	64 bytes
L1 Cache (Instr./Data)	32 KBytes, 2-way
L2 Cache (NUCA)	8 MBytes, 128 Banks
NUCA Bank	64 KBytes, 8-way
L1 Latency	3 cycles
NUCA Bank Latency	4 cycles
Router Latency	1 cycle
Avg Offchip Latency	250 cycles

Table 1: Configuration parameters.

The methodology we used for simulation involved first skipping both the initialization and thread creation phases, and then fast-forwarding while warming all caches for 500 million cycles. Finally, we performed a detailed simulation for 500 million cycles. As a performance metric, we used the aggregate number of user instructions committed per cycle, which is proportional to the overall system throughput [6].

3.1. Energy Model

In this paper, we also evaluated the energy consumed by the NUCA cache and the off-chip memory. To do so, we used a similar energy model to that adopted by Bardine et al. [7]. This allowed us to also consider the total energy dissipated by the NUCA cache and the additional energy required to access the off-chip memory. The energy consumed by the memory system is computed as follows:

$$E_{total} = E_{static} + E_{dynamic}$$
$$E_{static} = E_{S\_noc} + E_{S\_banks} + E_{S\_mechanism}$$
$$E_{dynamic} = E_{D\_noc} + E_{D\_banks} + E_{D\_mechanism} + E_{off-chip}$$

We used models provided by CACTI [8, 9] to evaluate static energy consumed by the memory structures (E_{S_banks} and $E_{S\_mechanism}$). CACTI has been used to evaluate dynamic energy consumption as well, but GEMS [4]

support is required in this case to ascertain the dynamic behavior in the applications (E_{D_banks} and $E_{D\_mechanism}$). GEMS also contains an integrated power model based on Orion [10] that we used to evaluate the static and dynamic power consumed by the on-chip network (E_{S_noc} and E_{D_noc}). Note that the extra messages introduced by the mechanism that is being evaluated into the on-chip network are accurately modeled by the simulator. The energy dissipated by the off-chip memory ($E_{off-chip}$) was determined using the *Micron System Power Calculator* [11] assuming a modern DDR3 system (4GB, 8DQs, Vdd:1.5v, 333 MHz). Our evaluation of the off-chip memory focused on the energy dissipated during active cycles and isolated this from the background energy. Our study shows that the average energy of each memory access is 550 pJ.

As an energy metric we used the energy consumed per memory access. This is based on the energy per instruction (EPI) [12] metric which is commonly used for analysing the energy consumed by the whole processor. This metric works independently of the amount of time required to process an instruction and is ideal for throughput performance.

4. Last Bank

Cache memories take advantage of the temporal and spatial data locality that applications usually exhibit. However, the whole working set does not usually fit into cache memory, causing capacity and conflict misses. These misses mean that a line that may be accessed later has to leave the cache prematurely. As a result, evicted lines that are later reused return to the cache memory in a short period of time. This is more pronounced in a NUCA cache memory because data movements within the cache are allowed, so the most recently accessed data blocks are concentrated in a few banks rather than evenly spread over the entire cache memory. Therefore, we propose adding an extra bank to deal with data blocks that have been evicted from the NUCA cache, similar to the victim cache [13]. This extra bank, called *Last Bank*, provides evicted data blocks a second chance to come back to the NUCA cache without leaving the chip.

Last Bank, which is as large as a single bank in the NUCA cache, acts as the last-level cache between the NUCA cache and the off-chip memory. It is physically located in the center of the chip at about the same distance to all cores. When there is a hit on Last Bank, the accessed data block leaves the Last Bank and returns to the corresponding bank in the NUCA cache.

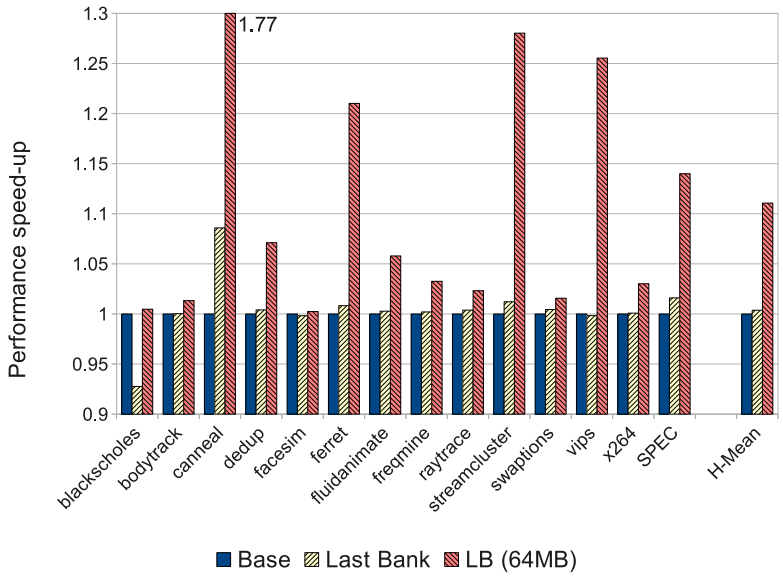


Figure 5: Performance improvement achieved with Last Bank

Figure 5 illustrates the performance results achieved when introducing the Last Bank to the baseline architecture. We observe that such small storage is not able to hold evicted data blocks before they are accessed again. Thus, the performance benefits of this proposals are negligible in most of the simulated applications. Assuming a Last Bank of 64 MBytes, however, we achieve an overall performance improvement of 11% and up to 75% with *canneal*. There are three main reasons that prevent the small version of Last Bank from being so effective as the 64-MByte configuration: cache pollution, potential bottleneck and size of the cache. Last Bank indistinctively catches all evicted data blocks from the regular NUCA banks, but only a small portion of them are going to be effectively reused further in the program. In the best case, the data blocks that are no longer accessed by the program would be directly sent to the upper-level memory. However, the reality is that Last Bank does not know whether an evicted data block is going to be accessed further by the program. This pollutes the Last Bank and provokes other data blocks that may be reused to be evicted from the Last Bank before they are accessed. Moreover, our design assumes a single Last Bank for the whole NUCA cache. Although this could be a potential bottleneck when executing applications

with large working sets that provoke lots of replacements, this design provides similar response latency to all the cores. Our evaluation also shows that a Last Bank of 64 KBytes is not large enough to hold all evictions from 128 NUCA banks, thus we conclude that the benefits of this approach are limited by the size of the Last Bank.

4.1. Last Bank Optimizations

This section describes two optimizations for the Last Bank approach. They exploit some of the drawbacks of the mechanism and allow it to achieve higher performance benefits at low implementation cost.

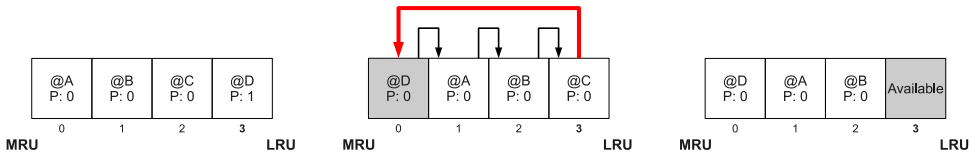
4.1.1. Selective Last Bank

Last Bank is not large enough to deal with all the evicted data blocks from the entire NUCA cache. So, Last Bank is polluted with *useless* data blocks that will not be accessed again and that provoke the eviction of *useful* data blocks from Last Bank before they are accessed. This fact leads us to propose a selection mechanism in Last Bank called *Selective Last Bank*. This selection mechanism allows evicted data blocks to be inserted into Last Bank by way of a filter.

Migration movements in D-NUCA cache make most accessed data blocks concentrate in the NUCA banks that are close to the cores (local banks). Consequently, the probabilities of a data block that have been evicted from a local bank to return to the NUCA cache are much higher than if it were evicted from a central bank. Therefore, we propose a filter that allows only the evicted data blocks that resided in a local bank before eviction to be cached.

4.1.2. LRU prioritising Last Bank

Because of the high locality found in most applications, the vast majority of evicted lines that return to the NUCA cache, do it at least twice. Thus, we propose modifying the data eviction algorithm of the NUCA cache in order to prioritise the lines that enter the NUCA cache from Last Bank. We call this *LRU prioritising Last Bank (LRU-LB)*. LRU-LB gives the lines that have been stored by the Last Bank and return to the NUCA cache an extra chance. Therefore, they remain in the on-chip cache memory longer. This requires storing an extra bit, called the *priority bit*, to each line in the NUCA cache.



(a) (b) (c)

Figure 6: LRU prioritising Last Bank (LRU-LB) scheme. (a) A priority line is in the LRU position. (b) The priority line resets its priority bit and updates its position to the MRU; the other lines move one position forward to the LRU. (c) The line in the LRU position is evicted since its priority is clear.

The LRU-LB eviction policy works as follows. When an incoming line comes to the NUCA cache memory from Last Bank, its priority bit is set. Figure 6a shows how this policy works when a line with its priority bit set is in the LRU position. The line that currently occupies the LRU position clears its priority bit and updates its position to the MRU. Thus, the other lines in the LRU stack move one position towards the LRU (Figure 6b). Finally, as the line that is currently in the LRU position has its priority bit cleared, it is evicted from the NUCA cache (Figure 6c). If the line that ends in the LRU position has its priority bit set, the algorithm described above is applied again until the line in the LRU position has its priority bit cleared.

4.2. Results and analysis

This section analyses the performance results obtained with the two optimizations for the Last Bank proposed in Section 4.1, *Selective Last Bank* and *LRU prioritising Last Bank (LRU-LB)*. With Selective Last Bank, the filter only allows blocks that have been evicted from a local bank to be cached.

As mentioned in Section 4, the potential performance improvement of this mechanism is strictly limited by the size of the Last Bank, however, we found that both optimizations could exploit the Last Bank features to achieve some performance improvement compared to the baseline configuration (by 2%). Figure 7 shows that the Selective approach outperforms Last Bank in almost all simulated applications. The reduction of pollution in the Last Bank allows data blocks to stay longer in the Last Bank. Thus, the hit rate in the Last Bank increases, and consequently it performs better.

With regard to the LRU-LB optimization, giving an extra chance to

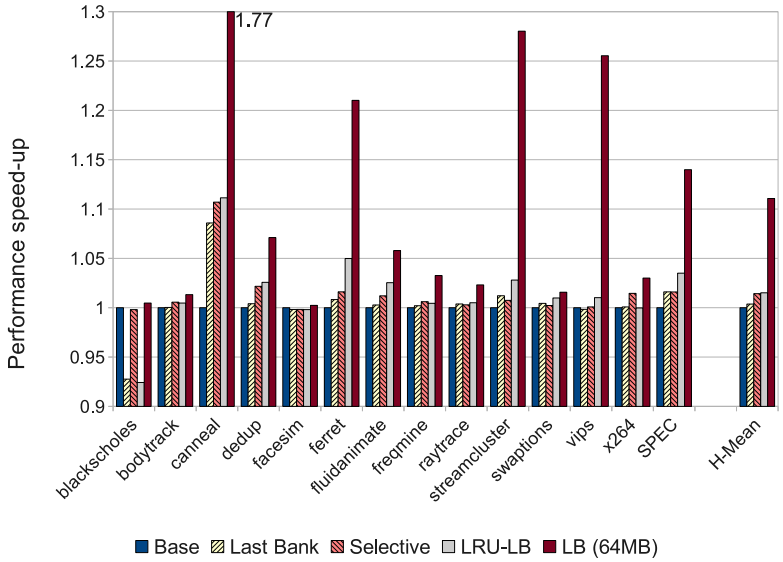


Figure 7: Speed-up achieved with Last Bank optimizations.

reused addresses before being evicted from the NUCA cache has two direct benefits. First, if accessed, they are closer to cores, which means lower access latency. Second, the number of reused addresses stored in the NUCA cache is higher. This translates into an overall performance improvement of 2%. Moreover, LRU-LB also outperforms the regular Last Bank configuration with most of simulated applications.

4.3. Summary

Last Bank is a simple mechanism that acts as victim cache for the regular NUCA banks. Moreover, we have also presented two optimizations for the Last Bank that exploit the features of this mechanism, resulting in performance benefits. However, our performance results show that a small Last Bank (64 KBytes) achieves negligible performance improvements while a larger (but expensive) implementation of 64 MBytes outperforms the baseline configuration by 11%.

Although this mechanism works well with small caches [14], Last Bank requires expensive hardware overheads to get significant benefits when a larger configuration is assumed. We now move on with additional proposals

that require similar hardware than the Last Bank, but obtain higher benefits in terms of both performance and energy consumption.

5. LRU-PEA

When an incoming data block enters into the NUCA cache, the placement policy determines in which NUCA bank it will be placed. Then, the replacement policy determines (1) which data block must be evicted from the bank to leave space for the new data (*data eviction policy*), and (2) which position within the replacement stack the incoming data will occupy (e.g. MRU or LRU). This decision is known as *data insertion policy*. Replacement policies in traditional cache memories are composed by these two sub-policies. However, D-NUCA incorporates one last decision to determine the final destination of the evicted data block, termed *data target policy*.

In this section we introduce the *Least Recently Used with Priority Eviction Approach (LRU-PEA)* replacement policy. This policy focuses on optimizing the performance of applications on a CMP-NUCA architecture by analyzing data behaviour within the NUCA cache and trying to keep the most accessed data in cache as long as possible. We now describe the two sub-policies that the LRU-PEA modifies: *data eviction policy* and *data target policy*. With regard to *data insertion policy*, the incoming data block will occupy the MRU position in the replacement stack.

5.1. Data Eviction Policy

Being able to apply migration movements within the cache is one of the most interesting features of NUCA cache memories. This enables recently accessed data to be stored close to the requesting core in order to optimize access response times for future accesses. We classify data into *promoted*, *demoted*, and *none* categories.

Figure 8 shows the percentage of resued and non-reused evicted addresses that belonged to each of these three categories at the moment of their replacement. We observe that the probabilities of a data block to return to the NUCA cache are higher if it belongs to the promoted category.

Based on this observation, LRU-PEA statically prioritizes the three categories, and evicts from the NUCA cache the data block that belongs to the lowest category. Having a static prioritization, however, could cause the highest-category data to monopolize the NUCA cache, or even cause a simple data block to stay in the cache forever. In order to avoid these

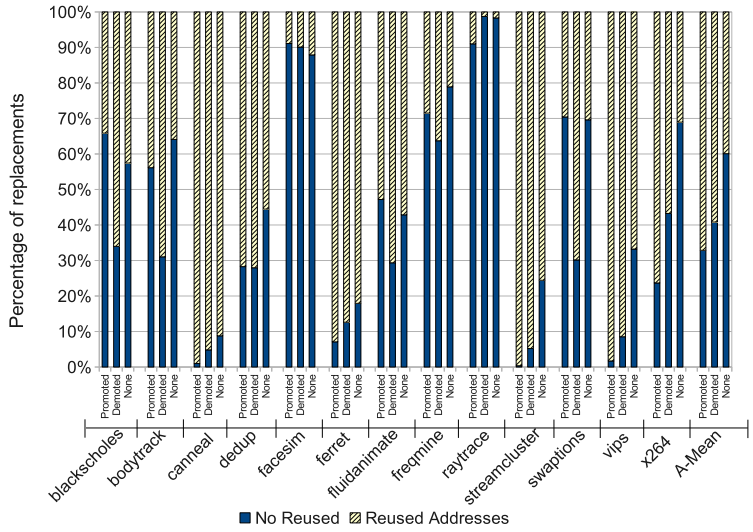


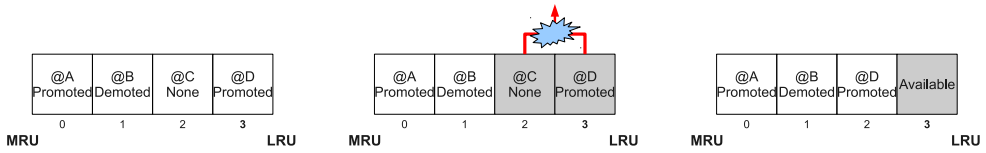
Figure 8: Distribution per categories of resued and non-reused evicted addresses at the moment of their replacement.

+	Promoted
Priority	None
-	Demoted

Table 2: Prioritization for LRU-PEA.

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 9 gives an example of how the *LRU-PEA* scheme works. First, we define the prioritization of the data categories. Based on the results showed in Figure 8, the final prioritization is as Table 2 outlines. When the LRU-PEA eviction policy is applied, the last two positions of the LRU-stack compete to find out which one is going to be evicted (see Figure 9b). 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 9c).



(a) (b) (c)

Figure 9: 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.

5.2. Data Target Policy

There are two key issues when a Dynamic-NUCA (D-NUCA) architecture [1] is considered: 1) a single data can be mapped onto 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, bank usage in a NUCA cache is heavily imbalanced, and a capacity miss in a heavily-used NUCA bank could cause constantly accessed data to be evicted from the NUCA cache while other NUCA banks are storing less frequently accessed data. The LRU-PEA addresses this problem by defining a *data target policy* that allows the replacement decision to be *spread* across all banks in the NUCA cache where evicted data can be mapped.

We propose Algorithm 1 as a data target policy for the LRU-PEA. The main idea of this algorithm is to find a NUCA bank whose victim data belongs to a lower priority category than that which is currently being evicted. In this 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: $\text{Local\_Bank\_Core}_i \rightarrow \text{Central\_Bank\_Core}_i \rightarrow \text{Local\_Bank\_Core}_{i+1} \rightarrow \text{Central\_Bank\_Core}_{i+1} \rightarrow ...$

The algorithm finishes when one of the following occurs: 1) the evicted data belongs 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 another NUCA bank. Then, whether the evicted data could not be relocated to another bank in the NUCA cache, it is written back to upper-level memory.

By using sequential access, the accuracy of the LRU-PEA is restricted to the NUCA banks that have been visited before finding a target bank. To address this problem, we introduce the *on cascade* mode. When this

```
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

mode is enabled, the algorithm does not finish when the evicted data finds a target bank. Instead, it uses the data that has been evicted from the target bank as evicted data. Thus, after visiting all NUCA banks we can assure that the current evicted data belongs to the current lowest priority category. In Section 5.4, we consider both configurations, with the *on cascade* mode enabled or disabled.

Figure 10 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 belongs to the *none* category, so the priority of the evicted data is 2 (see Table 2). First, the algorithm checks whether the evicted data can be relocated in the local bank of the next core (step 1 in Figure 10). However, the priority of the victim data in the current bank is higher than the evicted data, so the 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 the next bank needs to be checked. Finally, in the third step, the category of

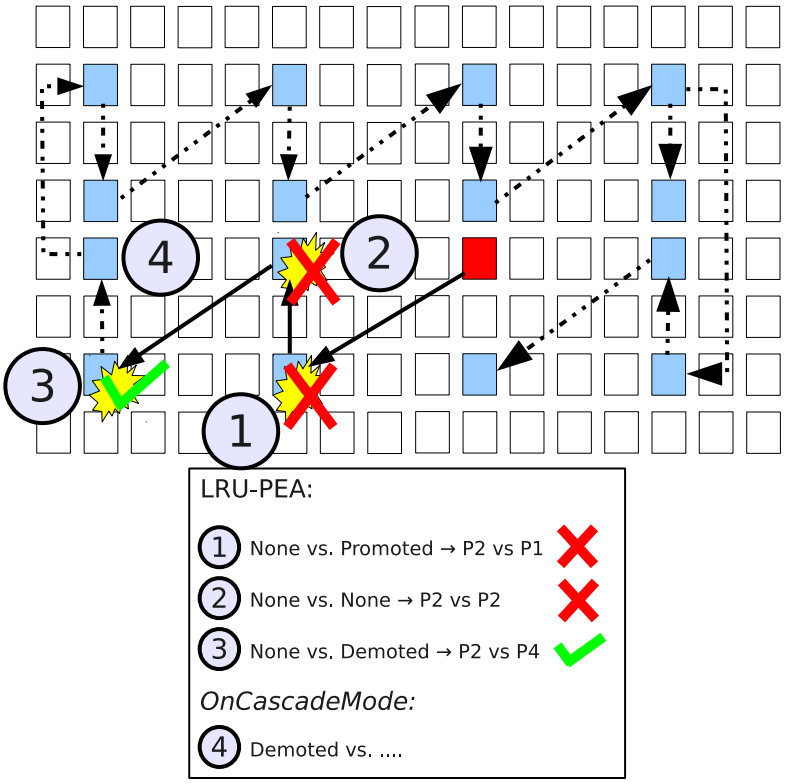


Figure 10: Example of how LRU-PEA behaves.

the evicted data has higher priority than the victim data of the current bank. Thus, the evicted data is relocated to this current bank. If the *on cascade* mode is enabled, the algorithm continues with the 4th step (see Figure 10), but uses the data that has been evicted from the current bank as evicted data. Otherwise, this data is directly evicted from the NUCA cache and sent back to the upper-level memory.

5.3. Additional Hardware

This mechanism requires the introduction of some additional hardware to the NUCA cache. In order to determine the data's category, we add two bits per line (there are three categories). Then, assuming that an 8 MByte NUCA cache described in Section 3 is used, LRU-PEA will need to add 32 KBytes, which is less than 0.4% of the hardware overhead. Furthermore, the proposed mechanism can be implemented without significant complexity.

	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 3: Number of extra messages introduced by both configurations of LRU-PEA to satisfy replacements.

5.4. Results and analysis

This section analyses the impact of the LRU-PEA as a *replacement policy*. Table 3 shows the average number of extra messages introduced by the LRU-PEA to satisfy a single replacement. When the *on cascade* mode is disabled, the communication overhead introduced by LRU-PEA is very low. On average, close to 80% of replacements are satisfied by adding up to 3 extra messages into the on-chip network. By enabling the *on cascade* mode, however, a significant percentage of replacements add the maximum number of messages into the network (the number of banks where the evicted data can be mapped minus one). The difference between the two modes can be explained by the high-accuracy provided by the LRU-PEA when the *on cascade* mode is enabled. In general, the data in NUCA banks have higher priority, and it is much more difficult to find a victim data with lower priority than the evicted data. In the following sections we analyze how the LRU-PEA behaves in terms of performance and energy consumption.

5.4.1. Performance Analysis

Figure 11 shows the performance improvement achieved when using the LRU-PEA as *replacement policy* in the NUCA cache. On average, we find that the LRU-PEA outperforms the baseline configuration by 8% if the

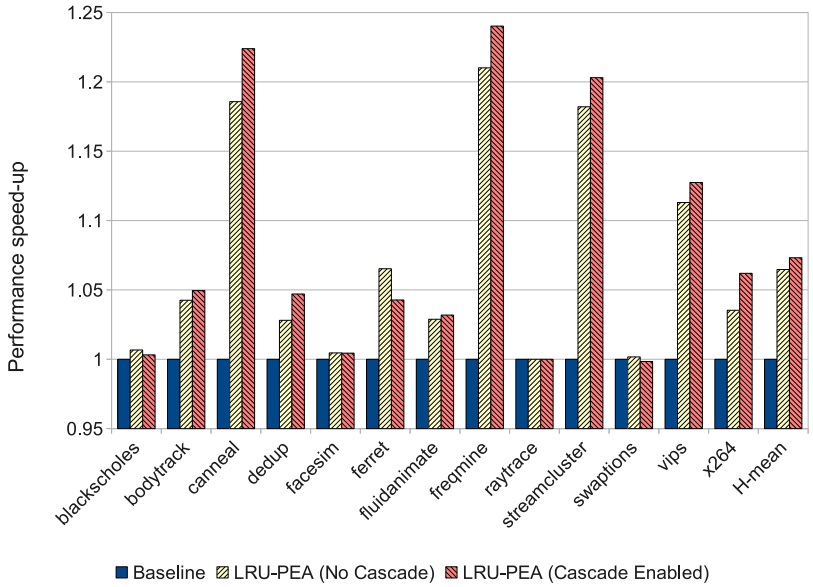


Figure 11: IPC improvement with LRU-PEA.

on cascade mode is enabled, and by 7% when it is disabled. In general, we find that the LRU-PEA significantly improves performance with most PARSEC applications, obtaining about 20% improvement in three of them (*canneal*, *freqmine* and *streamcluster*). On the other hand, 4 of the 13 PARSEC applications do not show perfomance benefits when using the LRU-PEA (*blackscholes*, *facesim*, *raytrace* and *swaptions*). We also observe that although the LRU-PEA is not harmful to performance either.

Figure 12 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 observed a significant reduction in MPKI when using the LRU-PEA, and even more when the *on cascade* mode is enabled. In general, we found that PARSEC applications that provide performance improvements, also significantly reduce MPKI. Moreover, we saw that *canneal*, *freqmine* and *streamcluster* (the applications that provide the highest performance improvement with LRU-PEA) also have the highest MPKI. In contrast, applications with an MPKI close to zero do not usually improve performance when the LRU-PEA is used.

Regarding those applications where the LRU-PEA does not improve

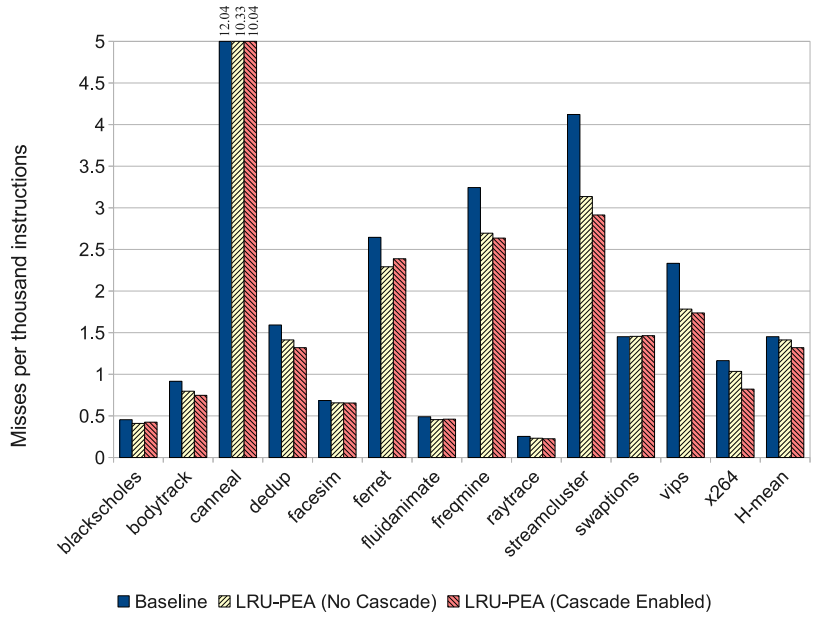


Figure 12: Misses per thousand instructions with LRU-PEA.

performance, *blackscholes* and *swaptions* are financial applications with small working sets, so their cache requirements are restricted. On the other hand, *raytrace* and *facesim* have very big working sets, but they are computationally intensive and mainly exploit temporal locality.

5.4.2. Energy Consumption Analysis

The energy consumption is analysed by using the Energy per Instruction (EPI) metric. Figure 13 shows that, on average, the LRU-PEA reduces the energy consumed per instruction compared to the baseline architecture by 5% for both configurations (with and without the *on cascade* mode enabled). In particular, the LRU-PEA significantly reduce energy consumption in PARSEC applications with large working sets, such as *canneal*, *freqmine* and *streamcluster*. Moreover, with the exception of *blackscholes* and *swaptions*, EPI was always reduced by the LRU-PEA.

As we can see in Figure 13, EPI is heavily influenced by static energy. Figure 14 shows the normalized EPI without taking into consideration the static energy consumed. We find that when *on cascade* mode is enabled, the dynamic energy consumed is 10% higher than the baseline configuration.

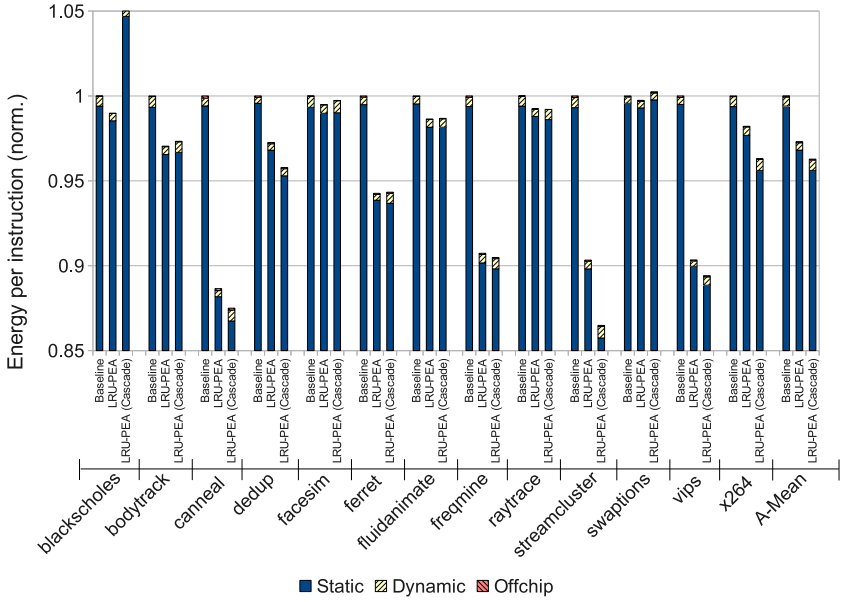


Figure 13: Normalized average energy consumed per each executed instruction.

However, the LRU-PEA with *on cascade* mode disabled still reduces EPI by more than 15%. This difference in EPI between the two LRU-PEA modes corresponds to the number of extra messages introduced into the on-chip network by each of them (see Table 3).

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.

6. The Auction

The Auction is an adaptive mechanism designed for the *replacement policy* of NUCA architectures in CMPs. It provides a framework for *globalizing* the replacement decisions in a single bank, and thus enables the replacement policy to evict the most appropriate data from the NUCA cache. Moreover, unlike the *one-copy* policy [1] or LRU-PEA (described in Section 5) which blindly relocate evicted data to other bank within the NUCA cache, The

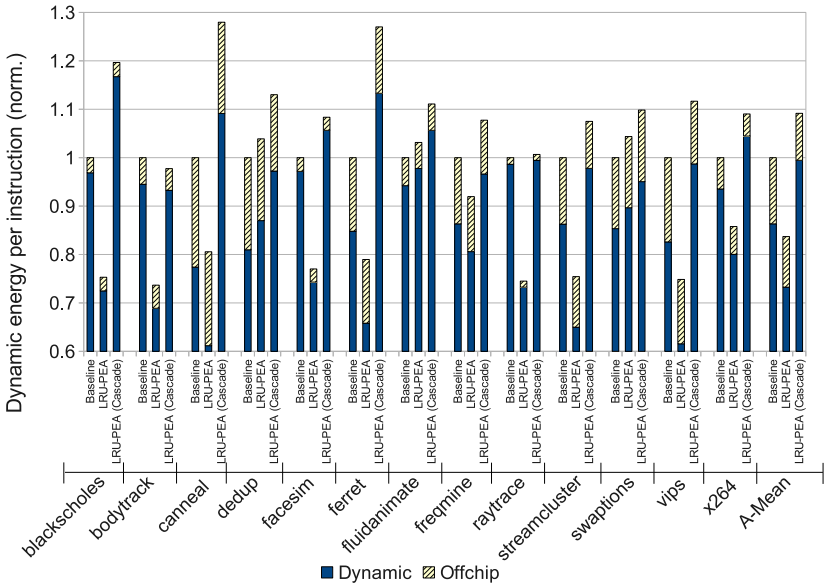


Figure 14: Normalized average dynamic energy consumed per each instruction.

Auction enables evicted data from a NUCA bank to be relocated to the most suitable destination bank at any particular moment. Thus, it takes into consideration the current load of each bank in the NUCA cache. This section describes in detail how *the auction* mechanism works.

6.1. Roles and components

In order to explain how *the auction* works, we will first introduce three roles that operate in the mechanism:

- **Owner:** It owns the item but wants to sell it, thus starting the auction. The bank in the NUCA cache that evicts the line then acts as the *owner* and the evicted line is the *item* to sell.
- **Bidders:** They can bid for the item that is being sold in the auction. In the NUCA architecture, the bidders are the banks in the NUCA cache where the evicted line can be mapped. They are the other NUCA banks from the *owner's* bankset.

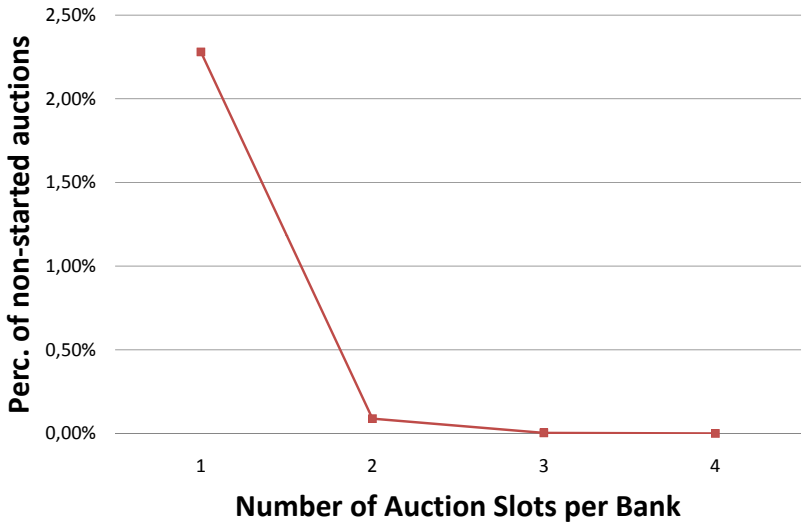


Figure 15: Percentage of non-started auctions when using up to four auction slots per NUCA bank.

- **Controller:** It stores the item while the auction is running. It also receives the bids for the item from the bidders and manages the auction in order to sell the item to the highest bidder.

The *auction controller* introduces a set of *auction slots* that are distributed among all banks in the NUCA cache. Each auction slot manages a single active auction by storing the evicted line that is being sold, the current highest bidder and the remaining time. When the auction finishes the corresponding auction slot is deallocated and becomes available for forthcoming auctions. Therefore, the number of active auctions per NUCA bank is limited by the number of auction slots. Figure 15 shows the percentage of auctions that cannot be started because there are no auction slots available when having up to four auction slots per NUCA bank. Assuming one auction slot per bank, just 2.3% of evicted lines can not be relocated. Moreover, we find that using more auction slots per NUCA bank, the percentage of non-started auctions dramatically decreases. At this point, the challenge is to determine the optimal number of auction slots that provides high accuracy without introducing prohibitive hardware overhead. In the remainder of the paper, we assume having *two auction slots per NUCA bank*. This configuration provides a good trade-off between auction accuracy

and hardware requirements.

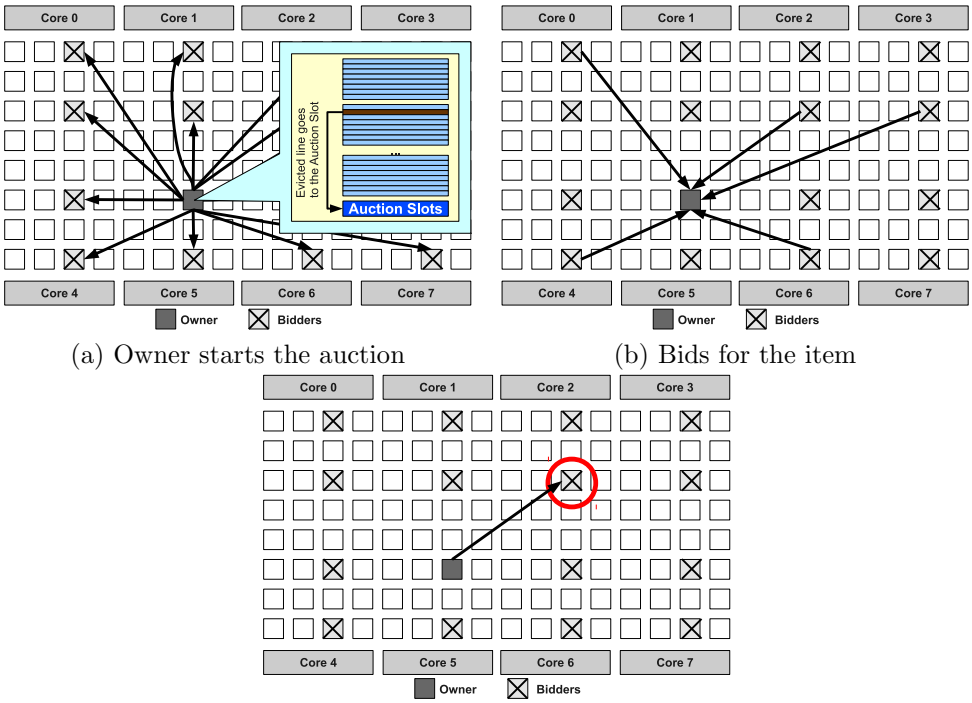
6.2. How The Auction works

Figure 16 shows the three steps of *the auction*. It starts when there is a replacement in a bank in the NUCA cache and it has at least one auction slot available. Otherwise the auction can not be started and the evicted line is directly sent to the main memory. The bank that is replacing data (the *owner*) moves the evicted line (the *item*) to the *controller* (the corresponding auction slot) and sets the auction deadline (Figure 16a). At the same time, the *owner* invites the other banks from the bankset (the *bidders*) to join the auction and bid for the *item*. Recall that an address maps to a bankset and can reside within any bank of the bankset. Thus, in our baseline architecture the evicted line can only be mapped to 16 banks. When a *bidder* finds out that a data block has been evicted from the NUCA cache, it decides whether to bid for it (Figure 16b). If the *bidder* is interested in getting the evicted data, it notifies the *controller* who manages *the auction*. Otherwise, the *bidder* ignores the current auction. Finally, when the auction time expires (Figure 16c), the *controller* determines the final destination of the evicted data based on the received bids and the implemented *heuristic*. It then sends it to the winning *bidder*. Moreover, in order to avoid recursively starting auctions, even if relocating the evicted data provokes a replacement in the winning bank, it will not start a new auction. In contrast, if none of the *bidders* bid for the evicted data when the auction time expires, the *controller* sends it to main memory.

Note that *The Auction* describes how to proceed when a replacement occurs in a bank in the NUCA cache. This is, therefore, a generic algorithm that must be customized by defining the decisions that each role can take on during the auction.

6.3. Hardware Implementation and Overhead

As described in Section 6.1, the auction mechanism introduces two *auction slots* per NUCA bank in order to manage the active auctions. Each auction slot requires 66 bytes (64 bytes to store the evicted line, 1 byte to identify the current highest bidder and 1 byte to determine the remaining time). The hardware overhead of this configuration is 33 KBytes (which is less than 0.4% of the total hardware used by the NUCA cache). Apart from hardware overheads, the auction also introduces extra messages onto



(a) Owner starts the auction

(b) Bids for the item

(c) Item is sold!

Figure 16: The three steps of the auction mechanism.

the on-chip network (i.e. messages to join the auction and bids). The impact of introducing these messages onto the network is analysed in Section 6.5.

6.4. Implementing an Auction Approach

The Auction algorithm defines how the owner and the controller behave while an auction is running, however, it does not define whether a bidder should bid for the evicted data block or not. The generality of this scheme opens a wide area to explore in which architects may use The Auction framework to implement smart auction-like replacement policies. This section describes two examples of auction approaches, termed *bank usage imbalance* (AUC-IMB) and *prioritising most accessed data* (AUC-ACC), that enable the determination of the *quality of data* during the auction, and thus enable bidders to compare the evicted data that is being sold with their own data.

Providing a significant number of bids per auction to the controller is crucial to having higher *auction accuracy* (i.e. the goodness of its final

decision). Thus, increasing the number of bits per auction, an auction-like replacement policy will provide controller more options to determine the most appropriate destination bank for the evicted data within the NUCA cache, and reduce the number of auctions that finish without receiving any bid.

6.4.1. Bank Usage Imbalance (AUC-IMB)

There are two key issues when a Dynamic-NUCA (D-NUCA) architecture [1] is considered. First, single data can be mapped in multiple banks within the NUCA cache. Second, the migration process moves the most accessed data to the banks that are closer to the requesting cores. Therefore, bank usage in a NUCA cache is heavily imbalanced, and a capacity miss in a heavily-used NUCA bank could cause constantly accessed data to be evicted from the NUCA cache, while other NUCA banks are storing less frequently accessed data.

We propose an auction approach that measures the usage rate of each bank. Thus, least accessed banks could bid for evicted data from banks that are being constantly accessed. We use the number of *capacity replacements* in each set of NUCA banks as our bank usage metric.

When a replacement occurs in a NUCA bank, the owner notifies the bidders that the auction has started, and sends them the current replacement counter. When a bidder receives the message from the owner, it checks whether its current replacement counter is lower than the counter attached to the message. If it is lower, the current bank bids for the evicted data by sending the controller the bank identifier and its replacement counter. At the same time, the controller that manages the auction is storing the current winner and its replacement counter. When a bid arrives to the controller, it checks if the replacement counter from the bid is lower than the one from the current winner. If so, the incoming bid becomes the current winner, otherwise the bid is discarded. Finally, when the auction time expires, controller sends the evicted data to the bidder with the lowest replacement counter.

Migration movements make most frequently accessed data to be stored in *local banks*, thus if the *controller* receives bids from both types of banks, local and central, it will always prefer to send the item to the central bank. If the first bid that arrives to the *controller* comes from a central bank, the auction finishes and the *item* is directly sent to the *bidder*. If the first *bidder* is a local bank, however, the *controller* sets the auction deadline to 20 cycles and waits for other bids from central banks. We have experimentally observed that most bids arrive at the controller in 20 cycles from the arrival of the

first bid.

Unfortunately, this approach is not affordable without restricting the number of bits used by each replacement counter. Therefore, in addition to restricting the bits dedicated to the replacement counter, we implement a reset system that initializes the replacement counters of other NUCA banks when one of them arrives at the maximum value. If this is not done, when a replacement counter overflows, it could not be compared with the other counters. Thus, when a replacement counter arrives at its maximum value, the owner sends the bidders the reset signal with the message that notifies that an auction has started.

We evaluate this approach by assuming an 8-bit replacement counter per cache-set in all NUCA banks. We have chosen this size on the basis of the following issues: additional hardware introduced (bits for the replacement counter and comparators), accuracy obtained, and reset frequency.

Hardware implementation and overheads: This approach requires the introduction of 8 bits in every cache set and auction slot. Thus, assuming the baseline architecture described in Section 2, this means adding an additional 16.5 KBytes to the hardware overhead required by The Auction (33 KBytes). Then, this approach requires introducing 49.5 Kbytes to the 8 MByte NUCA cache, which is less than 0.6% of the hardware overhead. Because these messages need to include the 8-bit replacement counter, they increase the size of both kind of auction messages, auction invitations and bids. The auction invitation message sent by the owner also requires one bit more for the reset signal. We take this overhead into account when evaluating this approach.

6.4.2. Prioritising most accessed data (AUC-ACC)

This auction approach focuses on keeping the most accessed data in the NUCA cache. When the bidder receives the auction start notification, it checks whether the evicted data has been accessed more times than the data that is currently occupying the last position in the LRU-stack. If this is the case, it bids for the evicted data by sending to the controller the bank identifier and the access counter of the LRU data block. As in AUC-IMB, when a bid arrives to the controller, it compares the access counter that comes with the incoming bid to the access counter of the current winner. If it is lower, then the incoming bid becomes the current winner. Finally, when the auction time expires, controller sends the evicted data to the bidder whose LRU data block has the lowest access counter. Note that as with AUC-IMB,

the auction deadline is set when the auction starts and then modified when the first bid arrives.

This approach assumes that each line in the NUCA cache has an access counter. It only keeps information regarding accesses made to the NUCA cache, which is updated just after a hit in this cache. However, as in the previous approach, having an unbounded counter per line is not affordable, thus we assume a 3-bit saturated counter per line. We choose this size for the counter because it is sufficiently accurate, and the additional hardware introduced is still affordable.

Hardware implementation and overheads: This approach requires the introduction of 3 bits per cache line and auction slot. Thus, assuming the baseline architecture described in Section 2, it adds 49.5 KBytes to the basic auction scheme. So, the overall hardware requirements of this auction approach in the 8 MBytes NUCA cache is 82.5 KBytes. As in the previous proposal, the auction messages are larger. In this case, the size of the messages is increased by 3 bits. These overheads are also considered when evaluating this approach.

6.5. Results and analysis

This section analyses the impact on performance and energy consumption of using different auction approaches (AUC-IMB and AUC-ACC) as *replacement policies* in the baseline architecture. Unfortunately, none of the mechanisms previously proposed for NUCA caches on CMPs properly addresses *data target policy*. Thus they could complement the improvements achieved by *The Auction* framework. As previously mentioned, Kim et al. [1] proposed two different approaches, *zero-copy* and *one-copy*. However, these alternatives were proposed in a single-processor environment. Therefore, in order to compare them with the auction we have adapted them our the CMP baseline architecture. Moreover, we evaluate the baseline architecture with an extra bank that acts as a victim cache [13]. This mechanism does not show performance improvements on its own and it does introduce the same additional hardware as the auction approach.

6.5.1. Performance analysis

Figure 17 shows the performance improvement achieved when using *The Auction* for the *replacement policy* in the NUCA cache. On average, we find that both auction approaches outperform the baseline configuration by 6-8%. In general, we observe the auction performs significantly well with most of the

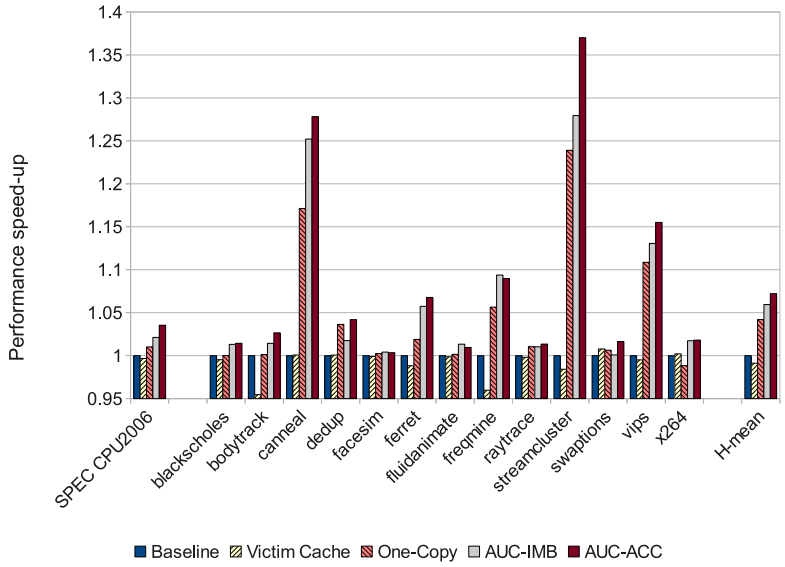


Figure 17: Performance improvement.

PARSEC applications. Three of them had improved performance by more than 15% (*canneal*, *streamcluster* and *vips*). On the other hand, assuming the multi-programmed environment (*SPEC CPU2006* in Figure 17), the auction approaches increase performance on average by 4%.

One-copy replacement policy always relocates evicted data without taking into consideration the current state of the NUCA cache. This enables one-copy to improve performance in those PARSEC applications with large working sets, such as *canneal*, *streamcluster* and *vips*. However, blindly relocating evicted data can be harmful in terms of performance. For example, if *x264* is used, one-copy has a 2% performance loss. The Auction, on the other hand, checks the current state of all NUCA banks from the bankset where the evicted data can be mapped, and thus does not relocate evicted data if no suitable destination bank has been found (i.e. if there are no bidders). This makes the auction a harmless mechanism in terms of performance even for applications with small working sets, such as *blackscholes* and *x264*. Moreover, we have experimentally observed that the performance benefits achieved using one-copy rely on the NUCA cache size, whereas the auction approaches do not correlate with the size of the cache.

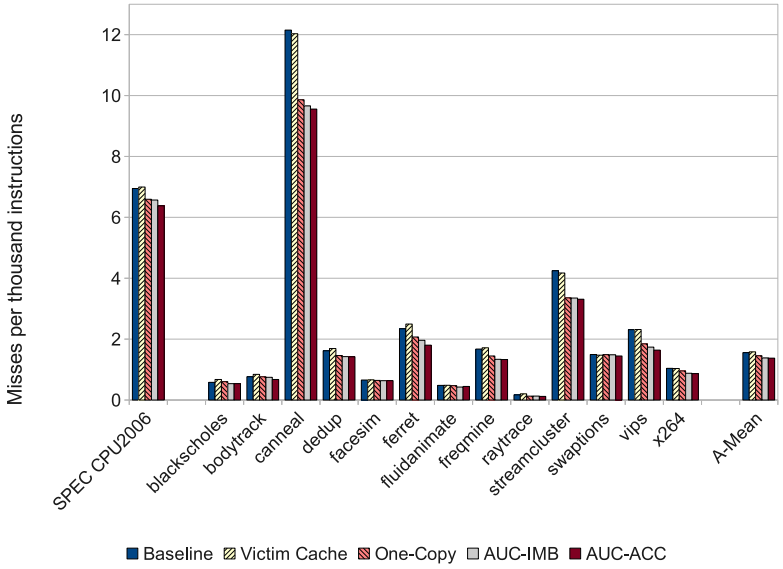


Figure 18: Misses per 1000 instructions.

Figure 17 shows that, on average, both auction approaches outperform one-copy by 2-4%. However, note that as a replacement policy, the auction takes advantage of workloads with large working sets because they lead to more data replacements. For example, the auction increases performance benefits by 10%, 8% and 5% compared to one-copy for benchmarks *streamcluster*, *canneal*, and *vips*, respectively. Unfortunately, most of PARSEC applications have small-to-medium working sets, and thus the average performance benefits achieved with a replacement policy are restricted.

Figure 18 shows the NUCA misses per 1000 instructions (MPKI) with both auction approaches (AUC-IMB and AUC-ACC). On average, we observe that there is a significant reduction in MPKI by using the auction. In general, we find that PARSEC applications that improve performance also significantly reduce MPKI. Moreover, we should emphasize the fact that *canneal*, *streamcluster* and *vips* are the applications that both provide the highest performance improvement with the auction and have the highest MPKI. In contrast, applications that have MPKI close to zero do usually not significantly improve performance when using this mechanism. In the

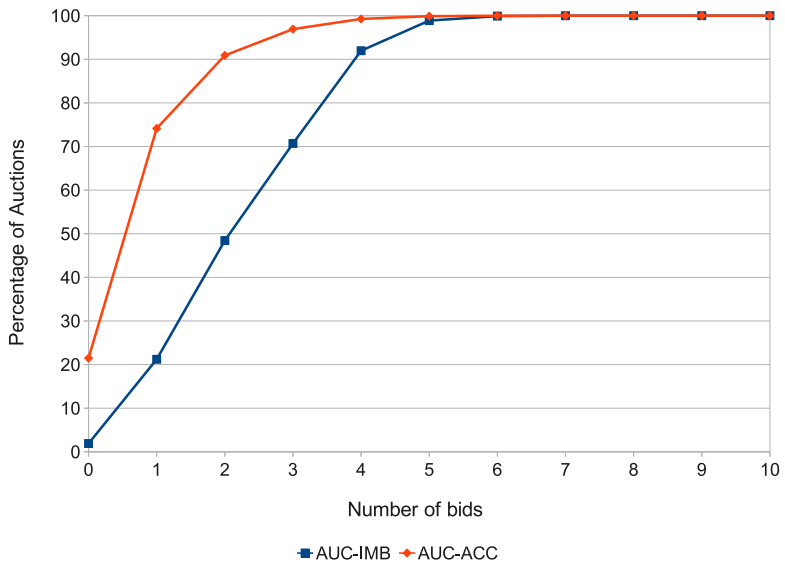


Figure 19: Received bids per auction.

multi-programmed environment, we find that the performance improvement is also related to an MPKI reduction.

Figure 19 shows that almost every auction launched when using AUC-IMB finishes with at least one bid. Assuming AUC-ACC, on the other hand, 20% of auctions finish without bids. Moreover, we observe that AUC-IMB achieves much higher auction accuracy than AUC-ACC. Actually, one in every two auctions get two or more bids assuming AUC-IMB as replacement policy, whereas only 10% of auctions get at least two bids with AUC-ACC. In an auction, the more bids the controller receives the more confident its final decision will be. Increasing the number of bids per auction, however, also increases the number of messages introduced to the on-chip network.

Figure 20 shows the auction message (auction invitation and bids) overhead introduced by both auction approaches. We find that network contention is a key constraint that prevents both auction approaches from achieving higher performance results. On average, AUC-IMB and AUC-ACC outperform the baseline configuration by 6% and 8%, respectively. AUC-ACC outperforms the other auction approach in most of workloads

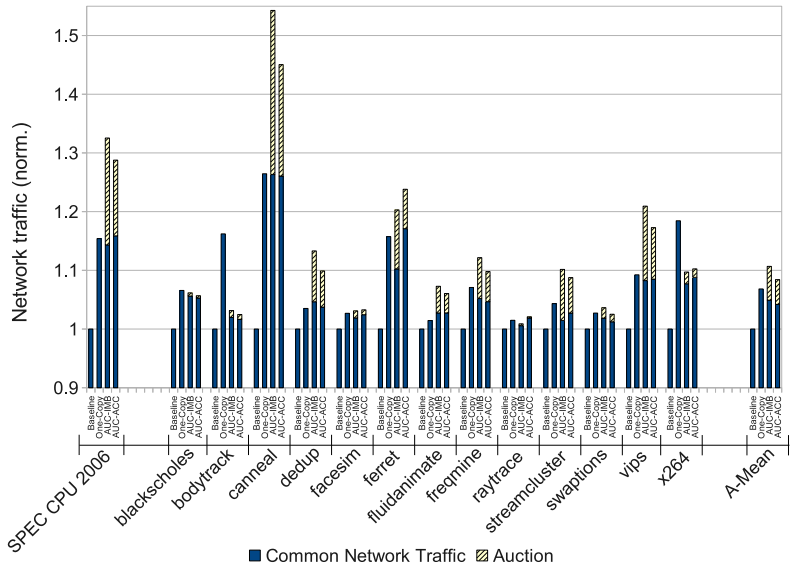


Figure 20: Network traffic.

including those with large working sets like *canneal*, *streamcluster* and *vips*. AUC-IMB is heavily penalized by the high on-chip network contention that this approach introduces. Therefore, this prevents AUC-IMB from obtaining higher performance results. Based on this observation, we conclude that the big challenge to implement a high-performance auction-like replacement policy is to balance *auction accuracy* and *network contention*.

6.5.2. Energy consumption analysis

In order to analyze the on-chip network contention introduced by the auction approaches, Figure 20 shows the traffic of the on-chip network normalized to the baseline configuration. On average, one-copy increases on-chip network traffic by 7%, while the increase caused by the auction is 8% for AUC-ACC, and 11% for AUC-IMB. Although both replacement mechanisms relocate evicted data, they also reduce miss rate in the NUCA cache compared to the baseline configuration. Therefore, increasing the on-chip network traffic is not as high as previously expected. On the other hand, the auction also introduces extra messages onto the on-chip network (auction invitations and bids). Figure 20 shows that the auction messages

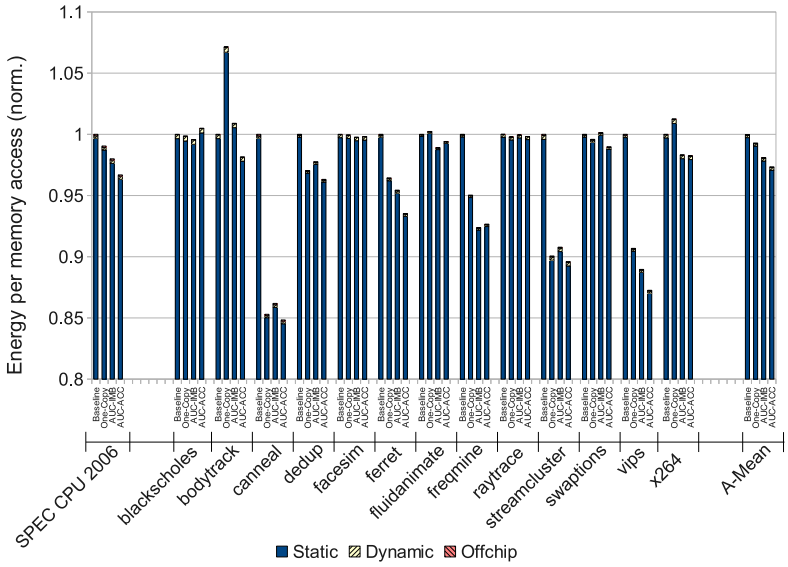


Figure 21: Energy per memory access.

represents less than 5% of total on-chip network traffic in both auction approaches.

Figure 21 shows that, on average, the auction reduces the energy consumed per each memory access by 3-4% compared to the baseline architecture. In particular, they significantly reduce energy consumption in PARSEC applications with large working sets, such as *canneal*, *freqmine*, *streamcluster* and *vips*. We find similar results for the multi-threaded applications with an energy reduction up to 4%.

6.5.3. Summary

We conclude that as a replacement policy, this mechanism takes advantage of workloads with the largest working sets because they lead to more data replacements and launch more auctions. On the other hand, we find that blindly relocating data in the NUCA cache without taking into consideration the current state of the banks, as is the case with one-copy, may cause unfair data replacements that can hurt performance.

7. 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 to replace. Traditionally, caches use the Most Recently Used (MRU) algorithm to insert data and the Least Recently Used (LRU) algorithm to evict data [15, 16].

Modifications to the traditional LRU scheme have also been proposed. Wong and Bauer [17] modified the standard LRU to maintain data that exhibited higher temporal locality. Alghazo et al. [18] proposed a mechanism called 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. [19] also proposed another LRU approach based on frequency of access in shared multiprocessor caches.

Kharbutli and Solihin [20] proposed a counter-based L2 cache replacement. This approach includes an event counter with each line that is incremented under certain circumstances. The line can be evicted when this counter reaches a certain threshold.

Recently, several papers have revisited data insertion policy. Qureshi et al. [21] 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 [22].

Dynamic NUCA cache memories also incorporate *data target policy*. This determines the final destination of the evicted data block. Several works have recently proposed in the literature to efficiently manage NUCA caches [2, 23, 24, 1, 25]. However, none of them properly addresses replacement policy in NUCA caches for CMPs.

8. Conclusions

The decentralized nature of NUCA prevents replacement policies previously proposed in the literature from being effective in this kind of cache. As banks operate independently from each other, their replacement decisions are restricted to a single NUCA bank. Unfortunately, *replacement policy* in NUCA-based CMP architectures has not been properly researched.

Most of previous works have ignored the replacement issue or adopted a replacement scheme that was originally designed for use with uniprocessors or uniform-cache memories. This paper describes three different approaches for dealing with replacements in D-NUCA achitectures on CMPs.

Last Bank mechanism introduces an extra bank into the NUCA cache memory that catches evicted data blocks from the other banks. Although this mechanism work well with small caches [14], Last Bank requires non-affordable hardware overheads to get significant benefits when a larger configuration is assumed. LRU-PEA, an alternative to the traditional LRU replacement policy, aims to make more intelligent replacement decisions by protecting these cache lines that are more likely to be reused in the near future. We observe that, on average, the baseline configuration's performance is increased by 8% when using LRU-PEA as replacement policy. Finally, we propose The Auction. This is a framework that allows architects for implementing auction-like replacement policies in D-NUCA cache architectures. The Auction spreads replacement decisions from a single bank to the whole NUCA cache. This enables the replacement policy to select the most appropriate victim data block from the whole NUCA cache. The implemented auction approaches increase performance benefits by 6-8% and reduce energy consumption by 4% compared to the baseline configuration.

References

[1] C. Kim, D. Burger, S. W. Keckler, An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches, in: Procs. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 2002.

[2] B. M. Beckmann, D. A. Wood, Managing wire delay in large chip-multiprocessor caches, in: Procs. of the 37th International Symposium on Microarchitecture, 2004.

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

[4] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, D. A. Wood, Multifacet's

general execution-driven multiprocessor simulator (gems) toolset, in: Computer Architecture News, 2005.

[5] N. Muralimanohar, R. Balasubramonian, N. P. Jouppi, Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0, in: Procs. of the 40th International Symposium on Microarchitecture, 2007.

[6] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, J. C. Hoe, Simflex: Statistical sampling of computer system simulation, IEEE Micro 26 (4) (2006) 18–31.

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

[8] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, N. P. Jouppi, Cacti 5.1, Tech. rep., HP (2008).

[9] N. Muralimanohar, R. Balasubramonian, N. P. Jouppi, Cacti 6.0: A tool to understand large caches, Tech. rep., University of Utah and Hewlett Packard Laboratories (2007).

[10] H. S. Wang, X. Zhu, L. S. Peh, S. Malik, Orion: A power-performance simulator for interconnection networks, in: Procs. of the 35th International Symposium on Microarchitecture, 2002.

[11] Micron, System power calculator, in: *http* : *//www.micron.com/*, 2009.

[12] E. Grochowski, R. Ronen, J. Shen, H. Wang, Best of both latency and throughput, in: Procs. of the 22nd Intl. Conference on Computer Design, 2004.

[13] N. P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, in: Procs. of the 17th annual international symposium on Computer Architecture, 1990.

[14] J. Lira, C. Molina, A. González, Last bank: dealing with address reuse in non-uniform cache architecture for cmps, in: Procs. of the 15th International Euro-Par Conference (Euro-Par), 2009.

[15] L. A. Belady, A study of replacement algorithms for virtual-storage computer, IMB Systems Journal 5 (2).

[16] A. J. Smith, Cache memories, ACM Computing Surveys 14 (3).

[17] W. Wong, J. Baer, Modified lru policies for improving second-level cache behavior, in: Procs. of the 6th International Symposium on High-Performance Computer Architecture, 2000.

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

[19] H. Dybdahl, P. Stenström, L. Natvig, An lru-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches, Computer Architecture News 35.

[20] M. Kharbutli, Y. Solihin, Counter-based cache replacement algorithms, in: Procs. of the 23rd International Conference on Computer Design, 2005.

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

[22] M. K. Qureshi, A. Jaleel, Y. N. Patt, Adaptive insertion policies for high-performance caching, in: Procs. of the 34th International Symposium on Computer Architecture, 2007.

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

[24] M. Kandemir, F. Li, M. J. Irwin, S. W. Son, A novel migration-based nuca design for chip multiprocessors, in: Procs. of the International Conference on Supercomputing, 2008.

[25] N. Muralimanohar, R. Balasubramonian, Interconnect design considerations for large nuca caches, in: Procs. of the 34th International Symposium on Computer Architecture, 2007.