Write-aware replacement policies for PCM-based systems

R. Rodríguez-Rodríguez, F. Castro, D. Chaver, R. Gonzalez-Alberquilla, L. Piñuel, F. Tirado

ArTeCS Group, Facultad de Informatica, University Complutense of Madrid
Email: {rrodriguezr,fcastror,dani02,rgalberquilla,lpinuel,ptirado}@ucm.es

The gap between processor and memory speeds is one of the greatest challenges that current designers face in order to develop more powerful computer systems. In addition, the scalability of the Dynamic Random Access Memory (DRAM) technology is very limited nowadays, leading to consider new memory technologies as candidates for the replacement of conventional DRAM. Phase-Change Memory (PCM) is currently postulated as the prime contender due to its higher scalability and lower leakage. However, compared to DRAM, PCM also exhibits some drawbacks, like lower endurance or higher dynamic energy consumption and write latency, that need to be mitigated before it can be used as the main memory technology for the next computers generation. This work addresses the PCM endurance constraint. For this purpose, we present an analysis of conventional cache replacement policies in terms of the amount of writebacks to main memory they imply and we also propose some new replacement algorithms for the last level cache (LLC) with the goal of cutting the write traffic to memory and consequently to increase PCM lifetime without degrading system performance. In this paper we target general purpose processors provided with this kind of non-volatile main memory and we exhaustively evaluate our proposed policies in both single and multi-core environments. Experimental results show that on average, compared to a conventional Least Recently Used (LRU) algorithm, some of our proposals manage to reduce the amount of writes to main memory up to 20-30% depending on the scenario evaluated, which leads to memory endurance extensions up to 20-45%, reducing also the energy consumption in the memory hierarchy up to 9% and hardly degrading performance.

Keywords: PCM, endurance, failing cells, cache replacement policies

1. INTRODUCTION

The current trend of increasing the number of cores in a single chip allows various threads or applications to execute simultaneously, which increases the demand on the main memory system to retain the working set of all the concurrently executing streams. This leads to the requirement of a larger main memory capacity in order to maintain the expected performance growth. However, although DRAM has been the prevalent building block for main memories during many years, scaling constraints have been observed when DRAM is used with small feature sizes. Moreover, the increase in memory size makes its leakage current to grow proportionally, and as a result, its energy consumption has become a major portion of the overall energy consumption in the system. Consequently, current research is focused on exploring new technologies for designing alternative memory systems in response to these energy and scaling constraints observed in DRAM technology. Among these technologies, Phase-Change Memory (PCM) is clearly the prime contender.

PCM is a low-cost and non-volatile memory technology that almost removes the static power consumption and provides higher density and therefore much higher capacity within the same budget than DRAM. Nevertheless, several obstacles restrict the adoption of PCM as main memory for the next computer generation: long write access latency, high write power and limited *endurance*.

The *endurance* is related with the amount of writes that a cell is likely to sustain before it fails, and in PCM technology this number is significantly lower than in DRAM. Specifically, a PCM cell fails after around 10^8 writes while a DRAM cell supports over 10^{15} writes. After a cell fails, it is not possible to change its value anymore and consequently the corresponding block and even the whole page it belongs to must be discarded. A natural solution is to cut the write traffic to PCM in order to extend the lifetime of PCM-based systems. For this purpose, several architectural techniques have been proposed recently [1, 2, 3, 4].

In this paper, which extends our previous work [5], we deal with the PCM endurance problem at a cache controller level by focusing on the LLC replacement policy. Conventional policies make their replacement decisions with the only objective of increasing the hit rate in the cache. Our goal is to redesign these policies so that they report a satisfactory trade-off between the memory lifetime and performance. We first analyze the behavior of classical and current *performance-oriented* cache replacement policies [6, 7, 8, 9] regarding the write traffic to memory that they entail. Then, we propose several new *write-aware* policies, based on the previous *performance-oriented* ones. We evaluate all these policies in a single-core environment using the SPEC CPU2006 suite and also in a multi-core scenario using the PARSEC parallel applications suite and multiprogrammed workloads. The results obtained in the single-core system reveal that some of our proposed policies significantly reduce the number of writes to main memory, thus improving memory endurance, and also the energy consumption in the memory hierarchy with respect to a conventional LRU, while they suffer a negligible performance drop. Furthermore, in the multi-core system our proposals behaves similarly and in some cases even better than in the single-core environment. We have directly ported our proposals to the multi-core system without any change, which suggests that it still remains opened a broad avenue for improvement.

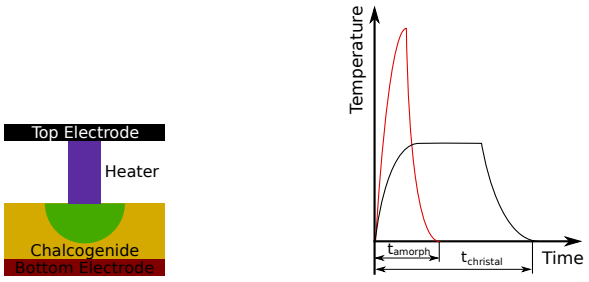
Notably, in this paper we make the following contributions:

- We evaluate some representative *performance-oriented* cache replacement policies regarding the amount of dirty blocks evicted from the LLC (i.e. the number of writes transmitted to main memory).
- We propose several modifications to these *performance-oriented* policies, aimed to cut down the write traffic to main memory and consequently to extend the memory lifetime.
- We qualitatively and quantitatively compare, in single and multi-core environments, our *write-aware* replacement algorithms to both classical and recent *performance-oriented* policies, as well as to other *write-aware* policies recently proposed.
- We quantitatively compare the write traffic reduction ability of our proposed *write-aware* policies with that of an optimal policy, in order to estimate how far we are from the maximum reduction feasible.

The rest of the paper is organized as follows: Section 2 outlines the main concepts of PCM operation and also recaps some related work. Section 3 presents the algorithms proposed to increase the lifetime of PCM-based systems. Section 4 describes other *write-aware* policies previously proposed. Sections 5 and 6 detail the experimental framework used and the obtained results respectively. Finally, Section 7 concludes.

2. BACKGROUND

Before addressing our main goal of improving PCM endurance by delaying the failure of the cells, we must briefly describe their operation.



(a) (b)

FIGURE 1. (a) PCM cell. A heating element (purple) is attached to a chalcogenous material (yellow/green), and enclosed between the two electrodes. The bit of material attached to the heater forms the programmable volume (green), *i.e.* the part of material that will experiment the phase change, (b) Heat pulses used to set and reset.

A PCM cell consists of two enveloping electrodes, a thin layer of chalcogenide and a heating element as Figure 1(a) illustrates. This heater is just a material that produces Joule heat when an electrical current is driven through, warming the chalcogenous material. In order to write a logical value in the cell, the heating element is employed to apply electrical pulses to the chalcogenide, which changes the properties of the material resulting in two different physical states: *amorphous* (high electrical resistivity) and *crystalline* (low resistance). Notably, if a high-intensity current pulse is applied, the material reaches over 600^oC and melts. Then, it is cooled down quickly, making it amorphous (RESET process). If the pulse is longer and with lower intensity, the material goes through an annealing process allowing the molecules to re-crystallize, lowering the electrical resistance (SET process). Thus, the chalcogenide switches easily, rapidly and in a reliable way between both states. Figure 1(b) shows graphically the heat pulses used to set and reset a PCM cell.

The process for reading the stored value just consists in applying a low current to the cell in order to measure the associated resistance. The limited endurance of PCM relies on the fact that after a certain number of writes on a PCM cell (around 10^8), the heating element is detached from the cell as a consequence of the continuous expansions/contractions derived from the writing process, leaving the cell in a *stuck at failure* state. From that moment on, although the cell is still readable, the stored valued can not be changed anymore.

Next we briefly describe some of the most representative proposals oriented to extend the PCM lifetime by reducing the number of writes.

- Eliminating redundant bit writes [1, 10, 11]: In a conventional DRAM write access, all bits in the row must be updated. However, a great portion of these writes are redundant (i.e. the bit before and after the write remains unchanged). Hence, taking advantage of the fact that in PCM reads are much faster and less power consuming than writes, every write is preceded by a read and a bitwise comparison, writing only those bits that have changed.
- *Flip-N-Write* [2]: Before performing a write to PCM, both the data to write and its bitwise inverse are compared with the data stored in the row, writing the one that involves less bit flips. The row incorporates an extra bit to indicate whether the stored data is the original one or its opposite.
- Hybrid memory [3, 4, 12, 13]: The idea here is to combine PCM with other technology not so sensible to writes (for example, DRAM). Thus, in [3] the authors combine a large PCM storage with a fast and small DRAM memory which acts as a cache for main memory. In [12], the same memory combination is studied in a Digital Signal Processor (DSP), whereas in [13], the scenario is quite different: CMPs with local Scratch Pad Memories (SPM) and PCM as main memory. Finally, in [4], the authors propose to combine DRAM and PCM in a large and flat memory and migrate pages between them, allocating in DRAM those pages that are most frequently written.

All these techniques pursue our same goal of extending the PCM lifetime by reducing the amount of writes performed to PCM, but through quite different avenues. Although there are also some other recent works much more closer to ours [14, 15, 16, 17], we have postponed the corresponding descriptions to Section 4, since we consider that the reader will much better understand these proposals after reading Section 3.

3. PROPOSED POLICIES

As stated above, it becomes essential to restrict the number of writes performed to memory in order to improve the PCM lifetime. Writes can reach the memory via two channels: from the upper level in the hierarchy (the disk) for loading the code or data, or from the lower level in the hierarchy (the LLC) for updating those blocks that have been modified by the processor. This work focuses on the second type of writes.

The observed LLC-to-memory write pattern is very dependent on the memory updating policy employed: if a *write-through* policy is used, every time a block is modified in the LLC it is also updated in the main memory level; conversely, if a *write-back* policy is employed, a block is updated in memory only when it leaves the LLC in a dirty state. In this paper we will employ the latter policy, which is the most frequently used in real systems due to its better tolerance to high memory latency and lower bandwidth requirements than the former one. Moreover, the *write-back* policy implies a significantly lower amount of writes to memory, which makes it even more adequate to our scenario.

In a *write-back* policy, a block can be modified several times in cache before eviction. Our aim is to coalesce as many modifications to a block as possible in the LLC. For this purpose we focus on the cache replacement policy, which mainly determines the lifetime of the blocks inside the cache. When a cache miss implies the eviction of a block, the replacement algorithm decides which block must be replaced/victimized. Conventional policies –conceived for systems with several cache levels backed up by a DRAM main memory– make their decisions with the only goal of increasing the cache hit rate and hence the system performance. However, in our PCM scenario, the replacement policy for the LLC should not only aim to improve performance but also to reduce the number of writes to memory that it implies.

For developing such write-aware replacement policy we should pay attention to the following general considerations:

1. First, a clean block leaving the LLC can simply be discarded, generating no writeback at all. Therefore, a *write-aware* replacement policy should give a higher priority to the eviction of clean blocks over dirty ones.
2. Second, among dirty blocks, we should distinguish whether they will be modified again in the future or not. A block that will be modified later again should stay in cache in order to merge future modifications with the previous ones into a single writeback. Conversely, a block that will never be modified again will not be able to reduce the amount of writebacks even though staying in the LLC. Consequently, our policy should give a higher priority to the eviction of the latter blocks.
3. Finally, among dirty blocks that will be modified in the future, we should consider the following two aspects:
   - First, based on Belady's conclusions [18], the evicted block should be the one that will be modified again furthest in the future, given that this is the block with the lowest probability of merging the future modifications with the preceding ones into the same writeback.
   - Second, based on the fact that a dirty block may have from only one to all its words modified, the evicted block should be the one with the least amount of modified words, since this is the block with the lowest probability of overwriting a dirty word in the next write to the block.

The first issue is quite easy to accomplish, since the dirtiness information of blocks is available in the cache.

However, the two remaining points are more complex to achieve, as they require extra hardware and some knowledge about the future. Besides, the third issue takes into account two conflicting aspects. Hence, as done in other similar situations, and in particular in conventional cache replacement policies, the solution in this case will be to build a prediction based on the previous behavior/state of the blocks.

Obviously, we have to take into account that an appropriate block replacement policy is essential to guarantee a high hit rate in the cache, and if we only pay attention to writeback reduction we can severely impact performance. Therefore, we must look for a satisfactory trade-off between performance and writes reduction.

Prior to detailing our proposed *write-aware* replacement policies, in the next subsection we describe several conventional algorithms –only oriented to improve performance– in which our proposals are based.

3.1. LLC *performance-oriented* replacement policies

When an incoming block implies an eviction, the cache replacement policy must decide which block to replace. In general, as Bélády established in [18], the best decision –in terms of performance– is to choose the block that will not be referenced again for the longest time. Since knowing the future in advance is not possible, the different policies proposed in the literature try to identify and *victimize* such a block by gathering information at different points during the lifetime of the blocks (specifically *insertion* and *promotion*). Each block has a *state* associated for collecting this information. Thus, the cache replacement algorithms can be split into three different sub-policies:

- *Insertion sub-policy:* This sub-policy determines the initial *state* to assign to a block when it is filled into the cache.
- *Promotion sub-policy:* This sub-policy determines how to update the *state* of a block when it experiences a hit (due to a miss in the next level closer to the processor).
- *Victimization sub-policy:* This sub-policy chooses the victim block when a replacement is required, by comparing the *states* of the candidate blocks.

The access pattern to the various cache levels is different. For example, in the lowest level a strong temporal locality is observed for most applications, which leads to replacement policies trying to exploit such locality. However, when a block reaches the LLC, the temporal locality has been almost totally filtered by the lower levels, so that the replacement policy must also exploit other features. Although many LLC replacement algorithms have been proposed recently, we next describe only those directly related with our work:

3.1.1. Least Recently Used (LRU)

It constitutes the baseline algorithm to which every proposal compares to, being implemented in most commercial systems under different simplified versions. LRU arranges blocks using a *recency stack*, in which the block that occupies the LRU position is the furthest referenced block in the past, while the one at the MRU (Most Recently Used) position is the nearest referenced block in the past.

- *Insertion sub-policy:* a new block is inserted into the *recency stack* as the MRU block, moving the remaining blocks one step closer to the LRU position.
- *Promotion sub-policy:* a block experiencing a hit is moved to the MRU position inside the *recency stack*.
- *Victimization sub-policy:* the block occupying the LRU position is selected for eviction, under the philosophy that, due to temporal locality, it is also the block that will not be required again for the longest time.

3.1.2. Pseudo Last In First Out (peLIFO)

This policy [8] builds on a LIFO (Last In First Out) replacement policy [6], in which, making use of a Fill Stack, the last block entering the cache is the candidate for replacement. With this scheme, some blocks will remain at the bottom part of the stack, being able to exploit *long-term* reuses. In peLIFO the bottom part of the Fill Stack is reserved for *long-term* reuses as well. However, unlike LIFO, peLIFO selects dynamically intermediate stack positions (called Escape Points) for replacement, that guarantee that *short-term* reuses are also fulfilled.

3.1.3. Static, Bimodal and Dynamic Re-Reference Interval Prediction (SRRIP, BRRIP and DRRIP, respectively)

These are the algorithms [7] in which the best of our proposals are based on, so we will next explain them in detail: each block has an associated state that represents the prediction of how far in the future it will be referenced again (Re-Reference Interval Prediction, or RRIP). This state is codified with M bits, that represent 2^M different RRPVs (Re-Reference Prediction Values). A block with RRPV=0 is predicted to be referenced again soon (*near-immediate* RRIP), whereas a block with RRPV=2^M-1 is predicted to be referenced again far in the future (*distant* RRIP). Based on this state information, the *insertion*, *promotion* and *victimization* sub-policies of SRRIP operate as follows:

- *Insertion sub-policy of SRRIP:* on cache fills SRRIP assigns to the new block an intermediate prediction state of RRPV=2^M-2 (denoted as *long* RRIP).
- *Promotion sub-policy of SRRIP:* on re-reference of a block, there are two different options: HP (Hit

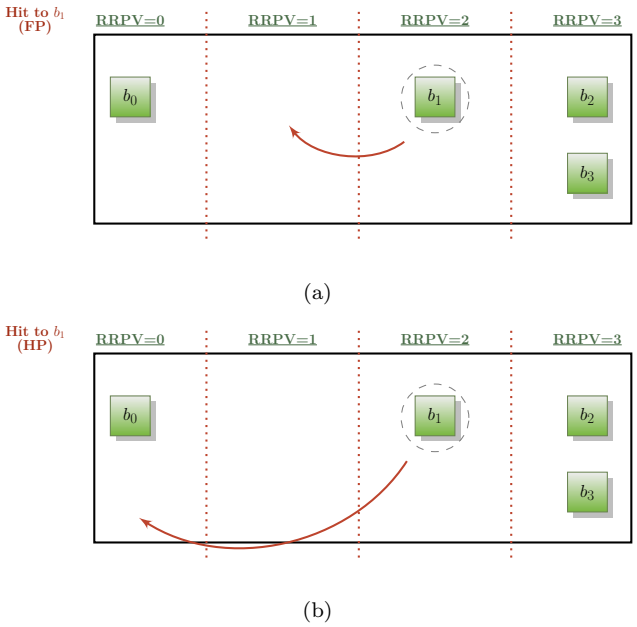


FIGURE 2. SRRIP promotion schemes: (a) SRRIP-FP, (b) SRRIP-HP.

Priority) that sets the RRPV of the block to zero and FP (Frequency Priority) that decrements it by one. Figure 2 illustrates an example of both promotion schemes for block b_1 using M=2.

- *Victimization sub-policy of SRRIP:* for eviction, SRRIP selects one block with a *distant* RRIP (RRPV=2^M-1). If there is not such a block, SRRIP increments the RRPV of all the blocks in the set and repeats the search. Figure 3 illustrates the victimization process when using M=2.

In some cases, for example when the re-reference interval of all the blocks is larger than the available cache size, SRRIP utilizes the cache inefficiently. In such scenarios, SRRIP generates cache *thrashing* and results in no cache hits at all. To avoid this situation, the authors propose BRRIP, that modifies the *insertion sub-policy* of SRRIP:

- *Insertion sub-policy of BRRIP:* it inserts majority of cache blocks with a *distant* RRIP (RRPV=2^M-1) and infrequently inserts new blocks with a *long* RRIP (RRPV=2^M-2).

This policy helps to preserve some of the working set in cache, improving performance under such scenario. However, for non-thrashing access patterns, always using BRRIP can significantly hurt cache performance. In order to be robust across all kind of cache access patterns, the author also propose a third policy (DRRIP) which follows an *insertion sub-policy* that combines those of SRRIP and BRRIP:

- *Insertion sub-policy of DRRIP:* it includes a Set Dueling mechanism [19] that identifies which

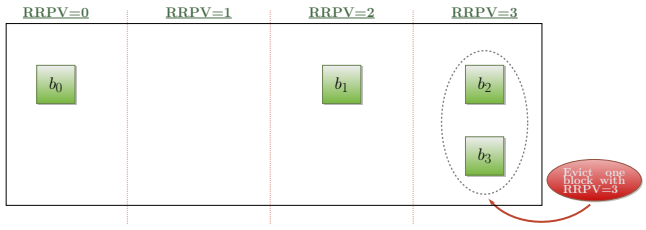


FIGURE 3. SRRIP Victimization.

insertion policy among SRRIP and BRRIP –based on the current miss rates reported– is best suited for the application (Figure 4).

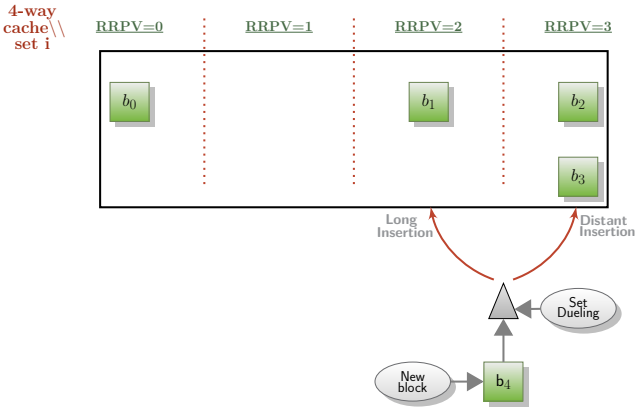


FIGURE 4. DRRIP Insertion.

3.1.4. Signature-based Hit Predictor (SHiP)
This proposal [9] is not a policy per se, but a predictor for the insertion of new blocks that can be used in conjunction with any other policy. SHiP associates each cache reference with a distinct signature, learns dynamically the re-reference interval of each one, and employs this information in the *insertion sub-policy*. The authors incorporate a Signature History Counter Table (SHCT) of saturating counters to learn the re-reference behavior of each signature. Every time a block is re-referenced (hit), the corresponding counter is incremented. If a block is never re-referenced during its residency in cache, the corresponding counter in SHCT is decremented when the block leaves the cache. The authors evaluate SHiP in conjunction with an SRRIP replacement policy: when a new block enters the cache, SHiP assigns it a *distant* RRIP if the associated counter is zero; otherwise a *long* RRIP is assigned.

3.2. Proposed LLC *write-aware* replacement policies

Before delving into our *write-aware* LLC replacement policies, we first evaluated (Section 6.1) all conventional algorithms described in the previous sub-section in order to determine how they impact the writeback account. Given that all assessed algorithms report quite

poor results for our considered PCM-based scenario, we studied several modifications looking for a reduction in the amount of writes to memory.

We have proposed many policies based on the conventional ones by modifying them in several ways. However, DRRIP-based policies proved to achieve the best trade-off between writeback reduction and performance, so hereafter we stick to these. Below we detail those changes to DRRIP *insertion*, *promotion* and *victimization* sub-policies, that demonstrated to be efficient enough. Note that we always use 2-bit DRRIP (RRPV ranges between 0 and 3).

1. *Changes to the insertion sub-policy of DRRIP:* the only change that reveals as satisfactory deals with the set-dueling mechanism, which decides the policy to employ in each insertion (SRRIP or BRRIP). This mechanism makes the decision by comparing the number of misses that each policy generates in a few *dedicated sets*. Specifically, for the insertion, some cache sets always follow SRRIP while other cache sets always follow BRRIP; the remaining sets (*follower sets*) use the policy determined by the mechanism, so that the block is inserted according to the scheme reporting a lower amount of misses at that moment. Our proposal is to change the metric employed to make the decision: instead of using the number of misses (which is reasonable for a performance-oriented policy), we compare the number of writebacks to memory that both SRRIP and BRRIP generated in the dedicated sets, inserting the block according to the policy currently exhibiting a lower number of writebacks. We will refer to this change as SD.
2. *Changes to the promotion sub-policy of DRRIP:* we make the following three proposals:
   - PL (Promotion Low-aggressiveness): as explained before, clean blocks leaving the LLC are not harmful at all for PCM, whereas dirty blocks cause a writeback to main memory when evicted from the LLC. Consequently, we propose to promote more aggressively a dirty block than a clean one. Notably, PL promotes clean blocks using the FP option (decrementing the RRPV), while dirty blocks are promoted using the HP option (setting the RRPV to 0). Note that a more aggressive approach would be not to promote clean blocks at all; however, according to our experiments this leads to an excessive performance drop, due to not exploiting whatsoever the temporal locality of clean blocks. Summarizing:
     – Clean Blocks→ FP promotion.
     – Dirty Blocks→ HP promotion.
   - PM (Promotion Medium-aggressiveness) and PH (Promotion High-aggressiveness): based on both the second and third general considerations previously detailed and the temporal locality principle, we propose to promote a block that experiences a write hit[1] with a very aggressive policy under the intuition that, if a block is written, it will probably be written again soon. Notably, in both PM and PH, blocks experiencing a write hit promote under the HP option. However, whereas PM promotes a block that experiences a read hit using the FP option in order to not impact performance in excess, PH does not promote a block at all under a read hit, for giving even more protection to writes at the cost of some performance degradation. To sum up:
     PM:
     – Read→ FP promotion.
     – Write→ HP promotion.
     PH:
     – Read→ RRPV unchanged.
     – Write→ HP promotion.
3. *Changes to the victimization sub-policy of DRRIP:* for eviction, the only extra issue (with respect to DRRIP) to which we will pay attention is the dirtiness of the blocks. According to the first general consideration previously described, we propose the following three modifications:
   - VL (Victimization Low-aggressiveness): at first, only clean blocks with a *distant* RRIP are considered to replacement[2]. In the event that the policy is unable to find such a block, a dirty block with a *distant* RRIP is victimized. As in the original DRRIP policy, if no blocks with a *distant* RRIP exist, VL increments the RRPV of all the blocks and repeats the same process. For the sake of clarity, Figure 5 illustrates the flow chart of VL policy.
   - VM (Victimization Medium-aggressiveness): we can view this policy as a two-stage process. At the first stage, only clean blocks are considered for the replacement, so that the clean block with the highest RRPV is victimized. Notably, when no clean block with a *distant* RRIP is found, the policy augments the RRPV of clean blocks. In the event that VM is unable to find clean blocks in this first stage, a second stage takes place, in which a conventional search is performed considering

[1] Note that, unlike L1, LLC reads and writes do not correspond directly to program loads and stores respectively. In our system, the LLC is written to only when a dirty block is evicted from L2, which can take place both due to a load or a store in the processor. Conversely, LLC is read when L2 misses, which again can occur both due to a load or a store.

[2] Note that, as done in original DRRIP, our three victimization sub-policies break ties by always starting the victim search from a fixed location (the left in our studies).

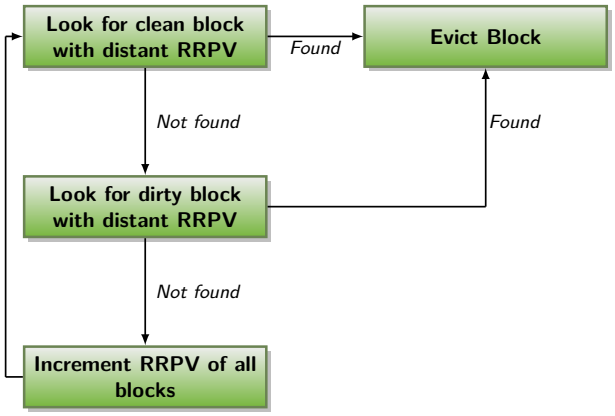


FIGURE 5. VL Flow Chart.

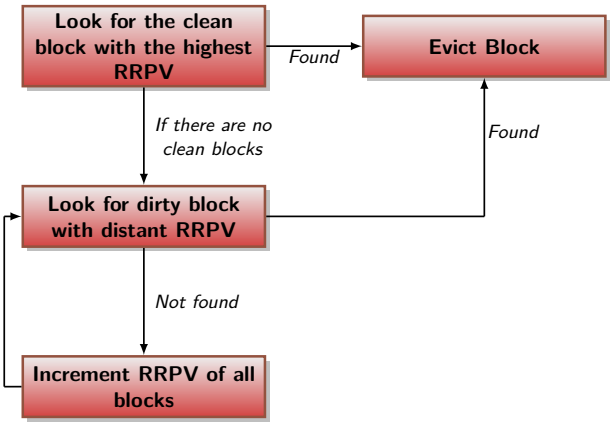


FIGURE 7. VH Flow Chart.

all blocks in the set. Figure 6 shows the flow chart of VM policy.

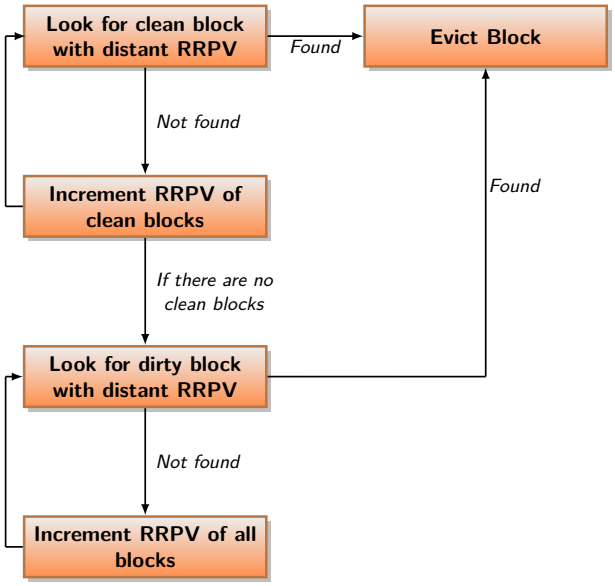


FIGURE 6. VM Flow Chart.

- VH (Victimization High-aggressiveness): this is also a two-stage process. At the first stage, as in the case of VM, only clean blocks are considered for the replacement, so that the clean block with the highest RRPV is victimized. However, when no clean block with a *distant* RRIP is found, the policy maintains the RRPV of all blocks unchanged during this first stage. When no clean blocks exist in the set, a second stage starts, in which a conventional search is performed considering all blocks in the set. Figure 7 illustrates the flow chart of VH policy.

As a result of all the possible combinations of the 7 proposed sub-policies, we introduce a full set of LLC replacement policies, denoted by the the names of the sub-policies they consist of. Although we have evaluated all of them, in the evaluation section and for the sake of simplicity we only show results for those providing a satisfactory trade-off among PCM lifetime and performance.

In order to further clarify our proposals, Figure 8 illustrates an example of the operation of three of our policies (PL-VL-SD, PM-VM-SD, PH-VH-SD) and original DRRIP. For simplicity, we make several assumptions: first, we consider a 4-ways LLC and that all accesses are mapped to the same set. Second, a write-back updating policy is used. Third, in the sub-insertion policy, the SD mechanism is supposed to always select SRRIP (i.e. blocks inserted with RRPV=2). Finally, HP is employed as the promotion sub-policy for DRRIP.

We should highlight that our proposal stacks on top of any other technique applied at main memory level, making it even more appealing in the presence of other techniques to extend the lifetime of a PCM device.

3.2.1. Endurance model

The policies proposed in the previous section are conceived with the aim of extending the lifetime/endurance of a PCM main memory without sacrificing too much performance. For achieving such a goal, we modified conventional performance-oriented replacement policies trying to reduce the number of LLC-to-memory write-backs. Note that the lifetime extension of the device is closely related to the reduction of writes, assuming the presence of wear leveling mechanisms, both at page level and inside each cache block, to prevent hot memory locations to wear memory unevenly, as [1, 20, 21, 22, 23]. This claim holds in the reasoning that if we write to memory, *e.g.* half often, then each cell is worn half as much, which translates directly into twice ($1/0.5 = 2$) the memory lifetime.

There is a caveat though; any conventional PCM-based memory would implement the redundant bit writes technique [1, 10, 11], based on which only those bits that have changed need to be updated. Therefore, we cannot just look at the amount of writebacks to memory we reduce, because that could artificially make our results look better than they are, but we need to

Blk_{way0} $_{RRPV,DirtyBit}$ Blk_{way1} $_{RRPV,DirtyBit}$ Blk_{way2} $_{RRPV,DirtyBit}$ Blk_{way3} $_{RRPV,DirtyBit}$

Initial state: $A_{2,1}$ $E_{2,0}$ $C_{2,0}$ $F_{3,1}$

Ref	DRRIP	PL-VL-SD	PM-VM-SD	PH-VH-SD
write A	Hit. Promote with HP, i.e. $RRPV_A$=0 $A_{0,1}$ $E_{2,0}$ $C_{2,0}$ $F_{3,1}$	Hit on dirty block. Promote with HP, i.e. $RRPV_A$=0 $A_{0,1}$ $E_{2,0}$ $C_{2,0}$ $F_{3,1}$	Hit and write access. Promote with HP, i.e. $RRPV_A$=0 $A_{0,1}$ $E_{2,0}$ $C_{2,0}$ $F_{3,1}$	Hit and write access. Promote with HP, i.e. $RRPV_A$=0 $A_{0,1}$ $E_{2,0}$ $C_{2,0}$ $F_{3,1}$
Read C	Hit. Promote with HP, i.e. $RRPV_C$=0 $A_{0,1}$ $E_{2,0}$ $C_{0,0}$ $F_{3,1}$	Hit on clean block. Promote with FP, i.e. $RRPV_C$=$RRPV_C$-1=1 $A_{0,1}$ $E_{2,0}$ $C_{1,0}$ $F_{3,1}$	Hit and read access. Promote with FP, i.e. $RRPV_C$=$RRPV_C$-1=1 $A_{0,1}$ $E_{2,0}$ $C_{1,0}$ $F_{3,1}$	Hit and read access. Do not promote at all. $A_{0,1}$ $E_{2,0}$ $C_{2,0}$ $F_{3,1}$
write B	Miss. One block with RRPV=3 (F). Evict F. $A_{0,1}$ $E_{2,0}$ $C_{0,0}$ $B_{2,1}$	Miss. Zero clean blocks with RRPV=3. One dirty block with RRPV=3 (F). Evict F. $A_{0,1}$ $E_{2,0}$ $C_{1,0}$ $B_{2,1}$	Miss. 1st stage: Zero clean blocks with RRPV=3. Increment RRPVs of clean blocks. One clean block with RRPV=3 (E). Evict E. $A_{0,1}$ $B_{2,1}$ $C_{2,0}$ $F_{3,1}$	Miss. 1st stage: There are clean blocks. Evict clean block with the highest RRPV (E). Evict E. 2nd stage unnecessary. $A_{0,1}$ $B_{2,1}$ $C_{2,0}$ $F_{3,1}$
write C	Hit. Promote with HP, i.e. $RRPV_C$=0 and set dirty bit. $A_{0,1}$ $E_{2,0}$ $C_{0,1}$ $B_{2,1}$	Hit on clean block. Promote with FP and set dirty bit. $A_{0,1}$ $E_{2,0}$ $C_{0,1}$ $B_{2,1}$	Hit and write access. Promote with HP and set dirty bit. $A_{0,1}$ $B_{2,1}$ $C_{0,1}$ $F_{3,1}$	Hit and write access. Promote with HP and set dirty bit. $A_{0,1}$ $B_{2,1}$ $C_{0,1}$ $F_{3,1}$
Read G	Miss. Zero blocks with RRPV=3. Increment all RRPVs. Now two blocks with RRPV=3 (E and B). Evict E. $A_{1,1}$ $G_{2,0}$ $C_{1,1}$ $B_{3,1}$	Miss. Zero clean and dirty blocks with RRPV=3. Increment all RRPVs. Now one clean block with RRPV=3 (E). Evict E. $A_{1,1}$ $G_{2,0}$ $C_{1,1}$ $B_{3,1}$	Miss. 1st stage: No clean blocks at all. Go to 2nd stage: One dirty block with RRPV=3 (F). Evict F. $A_{0,1}$ $B_{2,1}$ $C_{0,1}$ $G_{2,0}$	Miss. 1st stage: There are zero clean blocks. Go to 2nd stage: One dirty block with RRPV=3 (F). Evict F. $A_{0,1}$ $B_{2,1}$ $C_{0,1}$ $G_{2,0}$

FIGURE 8. Example of different policies operation: original DRRIP and three of our proposals.

take into account how many bits are affected by each writeback. Given the size of the task, we have relaxed the model slightly as follows:

- Whenever a cache block is written to main memory, we account for how many of the 64-bit words within the block have been modified. Unmodified do not wear off PCM cells.
- We assume that when a word is modified half of its bits are flipped and the other half preserves its value. This assumption is based on the fact that we use both *integer* and *FP* applications; on the one hand control variables have a small variability, on the other hand pointers to the heap and FP-numbers have much larger variability. Therefore, we take 0.5 as a general, broad approximation.

Based on this reasoning, we can derive the following equations, which we will employ in the evaluation section for calculating endurance results.

First, we need to account for how many bits in the block are affected by each writeback. In other words, the probability of each bit to flip, denoted as BFP (*Bit Flip Probability*). This is needed because reducing the number of writes to main memory could has the side effect of increase the dirtiness of a block and increase the probability of bits being flipped.

$$BFP = \frac{PMBW * BW * \sum_{n=1}^{WB} n * NWMM_n}{NWMM * BB} \quad (1)$$

where:

PMBW = *Percentage of Modified Bits per Word.* Recall that our relaxed model assumes PMBW=0.5.

BW = *Bits per Word*. In our scenario, BW=64.
WB = *Words per Block*. In our scenario, WB=8.
n = *Number of words modified within a written-back block*. In our scenario, n can be an integer from 1 to 8.
NWMM = *Total Number of Writebacks to Main Memory*.
$NWMM_n$ = *Number of Writebacks to Main Memory* with n words modified.
BB = *Bits per Block*. In our scenario, BB=512).

Based on the fact that memory wear is proportional to NWMM corrected with BFP, and that memory endurance is inversely proportional to memory wear, we can derive the wear reduction and endurance extension (denoted as WR and EE respectively) achieved by a *policy* with respect to *lru* algorithm when running a particular application (we are using the relation WB=BB/BW):

$$\frac{1}{WR_{policy/lru}} = EE_{policy/lru} = \frac{1}{\frac{BFP_{policy}}{BFP_{lru}} * \frac{NWMM_{policy}}{NWMM_{lru}}} = \frac{1}{\frac{\sum_{n=1}^{WB} \frac{n}{WB} * NWMM_{n,policy}}{\sum_{n=1}^{WB} \frac{n}{WB} * NWMM_{n,lru}}} \quad (2)$$

Note that this equation is perfectly coherent with the reasoning from this section and the assumptions of our model: it weights each write to main memory with the percentage of dirty words that the written-back block contains. Note also that, due to the normalization vs LRU, the EE obtained in Equation 2 is independent of the constant factors we assumed above (PMWB, BB, BW and WB).

4. OTHER WRITE-AWARE POLICIES

Recently, some other works have also addressed the PCM endurance constraint at the LLC controller level by redesigning the LLC cache replacement policy. Next we recap two distinguished proposals:

- CLean-Preferred victim selection policy (CLP) [14]: Like us, CLP aims to maintain dirty blocks in the cache to increase the probability that writes were coalesced. It implements a modified LRU that gives preference to clean blocks when choosing a victim. The authors propose a family of clean-preferred replacement policies, called *N-Chance*. The N parameter reflects how much preference is given to clean blocks[3]. The algorithm selects as victim the oldest clean block among the N least recently used ones. If such a block does not exist, the LRU block is used.
- Read-Write Aware (RWA) and Improved RWA (I-RWA) [16]: These policies are based on SRRIP-HP and they both use RRPV values of $log_2 assoc$ bits, being *assoc* the associativity of the cache. RWA modifies neither the *victimization* nor the *promotion* sub-policies with respect to SRRIP-HP, but the *insertion* sub-policy is modified as follows: when a read misses in the LLC, the RRPV of the inserted block is set to $assoc - 2$; when a writeback from L2 to the LLC misses, the RRPV of the inserted block is set to 0. Conversely, I-RWA distinguishes between single and multiple-use dirty lines, trying to protect multiple-use lines. The SRRIP-HP *victimization* sub-policy is not modified, but the *insertion* sub-policy is changed so that any read miss sets the RRPV of the filled block to $assoc - 2$ while a write miss sets it to $assoc - 3$. Besides, the *promotion* sub-policy is changed so that a read hit sets the RRPV of the block to a medium value –instead of 0 as done in SRRIP-HP– while a write hit sets it to 0.

Also [15] shares our same objective of reducing the number of LLC-to-memory writebacks at the LLC controller level. However, in this case, the solution has to do with the partitioning algorithm for sharing the cache among multiple applications. Notably, the authors accomplish the goal by adapting [24] to a PCM scenario.

Finally, in [17], a new *write-aware* replacement policy is presented, but in this case not for the replacement of LLC blocks but for the swapping of memory pages between PCM and DRAM in a hybrid main memory architecture. The authors propose a new memory management policy, based on the well-known CLOCK algorithm [25], that predicts if memory pages will receive future write references soon or not, and depending on that prediction maps the pages either to DRAM or to PCM.

5. EXPERIMENTAL FRAMEWORK

For our main experiments we use *gem5* [26]. We use the classic memory model provided by the simulator and we modify it by including a new cache level (L3) and encoding the proposed cache replacement policies. We simulate both a single and a multi-core scenario, using the simulator in its Syscall Emulation mode (SE) or Full System mode (FS) respectively. For the sake of a better accuracy in both execution modes, an O3 processor type (detailed mode of simulation) was used.

The cache hierarchy is modeled after that of an Intel i7 [27], formed by three cache levels, being L2 and L3 non-inclusive/non-exclusive. In the case of the multi-core scenario we model 2 and 4-core CMPs, with private L1 and L2, and a shared L3.

[3]In our paper, although we have evaluated CLP with different N values, we just report data of CLP with N equal to the cache associativity, since it is the policy achieving the highest writes reduction maintaining also a satisfactory trade-off with the performance delivered.

Figure 9 illustrates the experimental system used in both single-core and multi-core scenarios. Recall that for the evaluation of our proposed policies we implement them in the L3 while L1 and L2 use both a classical LRU algorithm. For modeling the PCM main memory we use DRAMSIM2 [28]. The integration between DRAMSIM2 and *gem5* is done using the patch developed for *gem5* [29]. We adapt the read and write latencies according to the PCM target. We consider a PCM main memory with 1 channel with 2 ranks of 8 banks each. Each bank has a 8-entry read queue and a 32-entry write queue for pending requests. When a writeback of a cache line from the LLC occurs, a write request is sent to the PCM, which is queued at a write queue. The application progresses without delay if the write queue has available entries since the writebacks are not on the critical path. Otherwise, the application stalls. Both lantencies and buffers size were selected according to the system used in [15].

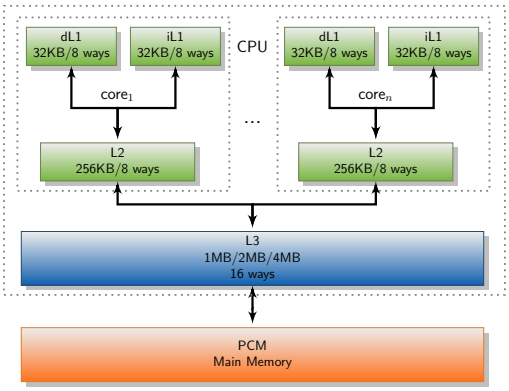


FIGURE 9. Simulated system, for all caches a 64 bytes/block is used.

Since we target both uniprocessor and multiprocessor architectures, our experiments make use of the SPEC CPU2006 [30] and the PARSEC [31] benchmark suites. When using the former suite in a single core scenario (L3 1MB size) we employ *train* inputs –we also tried with *reference* inputs, obtaining very similar results but at the expense of huge simulation times– and we simulate 1 billion instructions from the checkpoint determined using PinPoints [32]. Note that results from 9 out of 29 benchmarks are not considered in the evaluation section due to experimental framework constraints. We also report results of 12 multiprogrammed mixes using SPEC CPU2006 programs (Table 1) in a 4-CMP system with a 4MB L3. In this case, we fast forward 100M instructions, warm up caches for 200M instructions and then report results for 1B instructions. Finally, when the PARSEC workload is employed, we use *large* inputs, which exhibit memory footprints much more similar to those of *train* inputs for SPEC applications than *small* ones, and each simulation run is fast forwarded to the predefined checkpoint at the code region of interest (ROI), warmed-up by 100 million instructions, and then simulate 1 billion instructions for all threads or to ROI completion, whichever comes first.

For selecting the multiprogrammed mixes from Table 1, we employ the following methodology: we execute each benchmark alone, using an L3 of 1MB and an LRU replacement policy for all cache levels, and we measure the amount of LLC-to-memory writebacks that it generates. We then obtain for each benchmark the *writebacks to main memory per instruction* ratio (WPI). Based on these values, we include each benchmark into the *high*, *medium* or *low* category. Specifically, the *high* category includes benchmarks with a WPI higher than $3{*}10^{-3}$, the *medium* those with a WPI satisfying $3 * 10^{-3} < WPI < 10^{-3}$ and finally, in the *low* category we include the programs with a WPI lower than 10^{-3}. Table 2 shows this classification. Based on this classification, and as we will further detail in Section 6.3.1, we build some mixes with high WPI, some with medium WPI, some with low WPI, and some combining applications from different WPI categories.

We should also highlight here that *gem5* supports both x86 and Alpha ISAs. However, given that we found restrictions when using the FS mode for x86, we only simulated the PARSEC suite compiled for the Alpha ISA. In the case of SPEC CPU2006 suite, we run the simulations with the benchmarks compiled for both architectures, but since reported results were in the same ballpark, we chose to only show those corresponding to the x86 architecture.

Memory endurance simulator: Although *gem5* yields the values required to estimate the memory endurance extensions according to the model detailed in Section 3.2.1, we also employ a statistical simulator in order to plot the available memory ($\#alive\_pages$) under each evaluated policy as time passes. Doing faithful, cycle accurate simulation is unfeasible due to the amount of time PCM requires to experience stuck-at faults. Therefore, we have developed an in-house montecarlo simulator following the description in [33]. Table 3 recaps the main parameters employed in this simulator.

Each proposal is simulated by creating a number of memory pages ($\#starting\_pages$). Each bit inside every page is created with a lifetime randomly distributed according to a Gaussian distribution $N(\mu = 10^8, \sigma = 2.0 \cdot 10^7)$. The simulator uses as input parameters for each proposal the following (provided by the *gem5* simulator): the average BFP, the total number of writes and the average write reduction vs LRU. Initially, the wear rate w is calculated as a function of BFP and the number of pages in the system, as Equation 3 shows. The BFP expresses how much each bit is worn per-write, and the part involving the number of pages expresses how much extra wear alive pages have to absorb on behalf of those faulty pages

Mixes	Applications	Mixes	Applications
MIX0	lbm, mcf, milc, soplex	MIX1	lbm, mcf, milc, GemsFDTD
MIX2	mcf, milc, soplex, GemsFDTD	MIX3	bzip2, zeusmp, cactus, leslie3d
MIX4	zeusmp, cactus, leslie3d, gobmk	MIX5	cactus, leslie3d, gobmk, calculix
MIX6	perlbench, gcc, gromacs, namd	MIX7	hmmer, h264ref, omnetpp, astar
MIX8	lbm, mcf, gromacs, sphinx3	MIX9	mcf, milc, perlbench, h264ref
MIX10	cactus, hmmer, h264ref, lbm	MIX11	mcf, cactus, hmmer, h264ref

TABLE 1. SPEC 2006 multiprogrammed mixes

High	Medium	Low
lbm, mcf, milc, soplex, gemsFDTD	bzip2, zeusmp, cactus, leslie3d gobmk, calculix	perlbench, gcc, gromacs, namd, hmmer, h264ref, omnetpp, astar, sphinx3

TABLE 2. Writes to PCM per instruction categories

Page size	4KB
Row size	64 Bytes
Chips per rank	8
Bit lines per chip	$x8$
Lifetime distribution	$N(\mu = 10^8, \sigma = 2.0 \cdot 10^7)$
Pages	2000

TABLE 3. Memory endurance simulator parameters.

that have been discarded from the system.

$$w = BFP \cdot \frac{\#starting\_pages}{\#alive\_pages} \quad (3)$$

Then, according to the characterization obtained from *gem5*, we start simulating writes to the cells. At some point in time t a cell will wear out, the page containing it is discarded, the simulator updates the $\#alive\_pages$ and the wear rate accordingly, and the simulation proceeds until all pages are discarded. As a result of this simulation, a curve is generated that shows the amount of memory available (expressed in percentage of available pages) as a function of the number of writes.

Energy model: Finally, we should mention that in the evaluation section we will show results about energy consumption in the memory hierarchy[4], following a model that includes both dynamic and static contributions. The static component is calculated using CACTI 6.5 [34, 35], which reports a leakage number for each cache level. In the case of the main memory, as it is built with PCM technology, the static power can be considered as negligible. Thus, adding the power contributions of each cache level, we obtain the total static power consumed in the memory hierarchy. Finally, we multiply that number by the execution time of the program to obtain the total static energy consumed in the execution of that application. In the case of the dynamic component, we again use CACTI 6.5 for determining the dynamic energy consumption per access to each cache level, whereas for computing the dynamic energy consumption associated with the accesses to main memory we follow [15], employing 1J/GB and 6J/GB per PCM read and PCM write respectively (Table 4 includes data regarding latencies and energy consumption per memory hierarchy level). Then, the equation employed to determine the dynamic energy consumption in the memory hierarchy is:

$$\begin{aligned} \textit{Dynamic Energy} = \sum_{i=1}^{n} (RHL_i * REL_i + WHL_i * WEL_i + \\ (RML_i + WML_i) * (TEL_i + WEL_i)) + \\ + RPCM * REPCM + WPCM * WEPCM \end{aligned}$$

where n is the amount of cache levels, RHL_i and WHL_i denote the number of read and write hits in cache level i, RML_i and WML_i denote the number of read and write misses in cache level i, RPCM and WPCM correspond to the amount of reads and writes to PCM, REPCM and WEPCM denote the energy consumption per read and write to PCM and finally REL_i, WEL_i and TEL_i correspond to the energy consumption of a read, a write and a tag array consult in cache level i.

6. EVALUATION

This section is divided into four parts. In Section 6.1 we evaluate how classical *performance-oriented* policies behave in terms of the amount of writes they involve. Sections 6.2 and 6.3 assess the effectiveness of our proposed LLC replacement policies and that of other *write-aware* policies in cutting the write memory traffic and extending the memory lifetime in single-core and multi-core environments respectively. Finally we also report some additional results in Section 6.4. Note that throughout this section, when providing results about writes, we refer to LLC-to-memory writebacks (recall that we do not deal with writes from the disk to memory) at a *block-level*. Note also than the y-axis in all charts displayed in this section does not start from 0. Instead, we opted to start it from other different values in order to properly explode the differences among the various policies evaluated.

[4]In the energy consumption analysis we do not account for the implementation overhead of the replacement policies, which, according to the analysis from Section 6.4.2, constitutes a reasonable approximation. In any case, including this overhead in the energy study would be beneficial to our interests.

Level	Latencies (cycles)	Energy (Read/Write/Tag) (nJoules)	Leakage (mW/bank)
L1	1	0.222466/0.211604/0.00174426	4.01624
L2	10	0.530476/0.542389/0.0055894	11.6566
L3 1MB	30	2.25868/2.50327/0.019461	38.6125
L3 2MB	30	1.98142/2.16845/0.034619	115.595
L3 4MB	30	2.71345/2.98494/0.0673199	202.533
PCM read	100	59.6046447754	0
PCM write	700	357.6278686523	0

TABLE 4. Latencies and Energy Consumption

6.1. Performance-oriented policies

In this section we evaluate the behavior of conventional and recently proposed *performance-oriented* cache replacement policies when applied to the LLC regarding the number of dirty blocks evicted from this level, which corresponds to the amount of writes to main memory that each algorithm involves. Notably, we evaluate LRU, RANDOM, SRRIP, DRRIP, SHiP – in combination with SRRIP, as done in the original paper [9]– and peLIFO policies using the SPEC CPU2006 benchmark suite. Figure 10 illustrates the corresponding amounts of writes to memory normalized to the LRU baseline. Note that the figure shows results per benchmark as well as the geometric mean obtained. From the figure we observe that DRRIP ranks only second to LRU as the policy that on average delivers the lowest number of writes to main memory, although DRRIP outperforms LRU for most applications.

6.2. *Write-aware* policies in a single-core scenario

In this section we deeply analyze the behavior of our proposed LLC replacement policies and other algorithms in a PCM-based system within a single-core scenario. First, we explore the isolated contribution of each of our proposals and, based on this evaluation, we justify the decision of reporting results from just some of our policies, which we consider as representative, along this section. Then, we report data about the number of writes to memory and endurance that each evaluated proposal involves as well as the performance delivered. Besides, being energy consumption one of the main motivations for adopting PCM as main memory technology, we also include results about the involved energy consumption in the memory hierarchy according to the model detailed in Section 5. Finally, we expose a brief discussion about the trade-off between the memory endurance and the performance that the evaluated policies deliver.

6.2.1. Contribution of each proposed change

The various modifications we have proposed over original DRRIP impact the write memory traffic differently. Figure 11 quantifies this impact for each change isolated and also when we combined them. First, we observe that modifying the *insertion* sub-policy by changing the criteria that rules the Set Dueling mechanism (SD label) reports low reductions (around 1%) in the amount of writes to memory. Second, the impact of our *promotion* sub-policies is also limited. The low, medium and high-aggressive versions (PL, PM and PH respectively) only manage to cut the write traffic to memory between 0.5 and 1.5% with respect to DRRIP. Third, the *victimization* sub-policies clearly report the major benefits. Notably, VL, VM and VH are able to reduce the amount of writes generated by original DRRIP by around 10, 24 and 32% respectively. Therefore, we arrange our proposals into *high-aggressive*, *medium-aggressive* and *low-aggressive* categories depending on whether they include the VH, VM or VL label respectively. It is worth to note that when different sub-policies are combined, in some cases the writes reduction achieved is higher than that we would expect considering their isolated contributions.

Next we explain the criteria followed to choose the most representative policies across the board (highlighted in Figure 11 with green bars and north-east lines). First, we discarded all policies that do not reduce the write traffic to memory with respect to conventional LRU. We also ruled out most of the algorithms that achieve modest writes reductions at the expense of performance drops. Finally, among those policies reporting similar results in both amount of writes to memory and performance, we pick one of them within each group. As a result, we choose just one algorithm from each class of victimization sub-policies (i.e., VH, VM and VL) combined with one of the three possible promotion sub-policies (the best performing one in each case, either PH, PM or PL) and with the SD insertion sub-policy. Notably, we choose PM-VH-SD, PM-VM-SD and PL-VL-SD.

From now on, in all the following figures and for each evaluated policy we report the geometric mean of the normalized metrics (writes reduction, endurance, performance and energy consumption in the memory hierarchy) with respect to LRU considering both all the benchmarks (we labeled as *All*) and only the most memory-intensive programs (we labeled as *memory-intensive*). The rationale behind including this group of applications with high memory footprints is twofold: first, to reveal potential biased values in the *All* numbers due to programs in which the amount of writes is so low that minimal writes reductions in absolute values lead to high percentage numbers, polluting the *All* number, and second, to stress the benefits

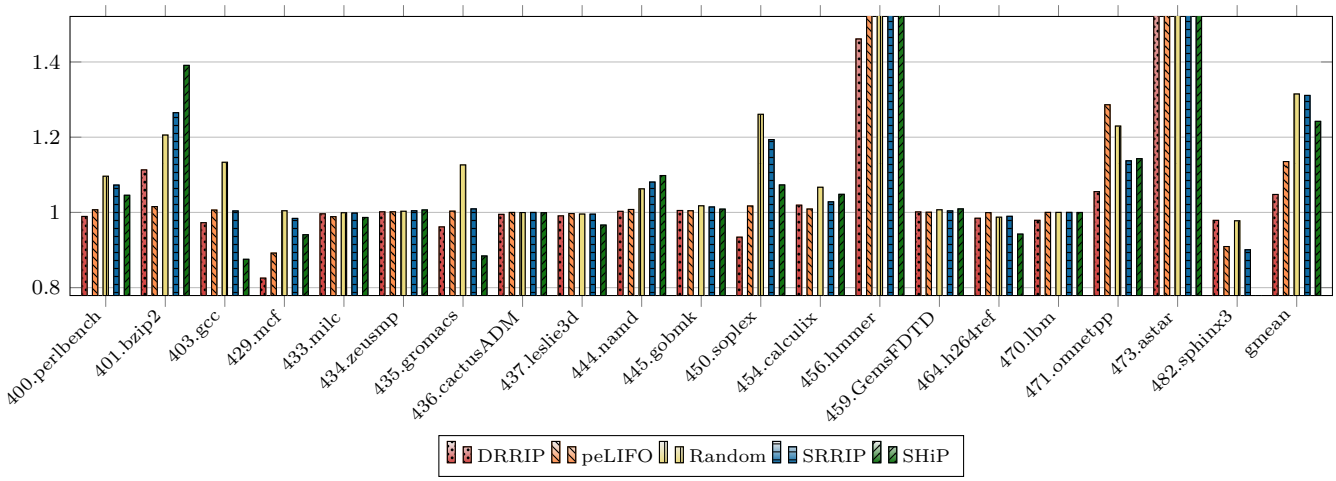


FIGURE 10. Writes to main memory normalized to LRU for performance-oriented policies: SPEC CPU2006 suite. *hmmer* and *astar* applications report numbers ranging between 7 and 16x for most policies.

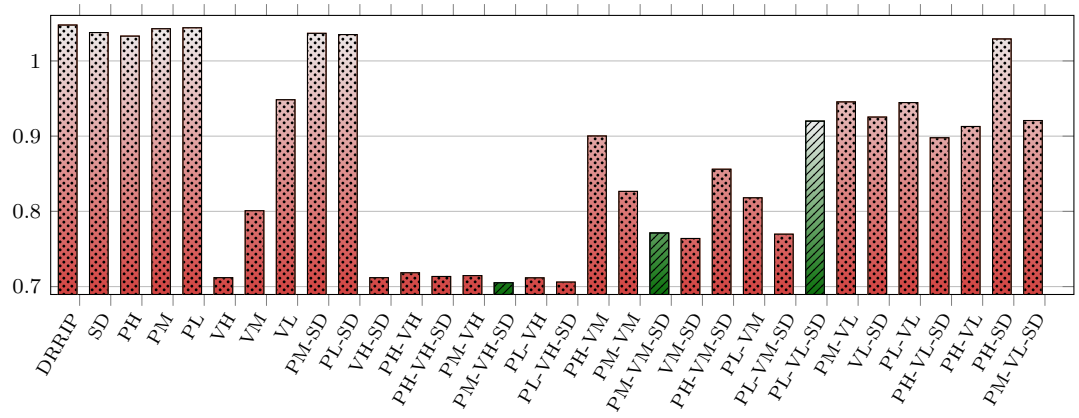


FIGURE 11. Amount of writes to main memory normalized to LRU: contribution of each proposed policy.

derived from our techniques over those applications that, performing high amounts of writes to memory, are more harmful to PCM lifetime. In order to define this second group of applications, we sort all the programs according to the WPI values exhibited in the baseline policy, and, starting from the benchmark with the highest WPI, we pick applications until the accumulated WPI of the selected programs reaches at least the 75% of the total WPI obtained when considering all applications. In the experimental setting we employ in this section, the *memory-intensive* group is conformed by *lbm*, *mcf*, *milc* and *soplex*.

Note that in this scenario we also report the geometric mean of the evaluated metrics when all the applications except the *sphinx3* program are considered (we label this group as *All w/o sphinx3*). The reason for this is that this application exhibits an special behavior that may lead the evaluated metrics to be somehow biased. Notably, while for the rest of applications the evaluated policies are able to reduce the amount of writes to memory by a factor ranging from 1 to 2X, for *sphinx3* many policies under evaluation are able to cut the write traffic to memory in an unusually large fashion (up to a hundredth part, 100X).

6.2.2. Amount of writes and Endurance

Figure 12 illustrates the number of writes to main memory generated by different proposals: from left to right we show results of some *performance-oriented* policies (DRRIP, peLIFO and SHiP), some previously proposed *write-aware* approaches (RWA, IRWA and CLP) and –ordering by decreasing aggressiveness– our chosen algorithms.

First, as shown, all our proposals significantly outperform original DRRIP. Second, they also exhibit higher ability in cutting the write traffic to main memory than other *write-aware* policies (RWA and IRWA and similar to that of CLP). The rationale behind the deficient behavior of both RWA and IRWA is twofold: 1) they are based on SRRIP instead of DRRIP, which implies a higher number of writes as Figure 10 illustrates, 2) they do not modify the SRRIP victimization sub-policy, that, as demonstrated in Section 6.2.1, constitutes the sub-policy with the highest impact (Section 6.2.1). CLP reduces the amount of writes by around 33% and

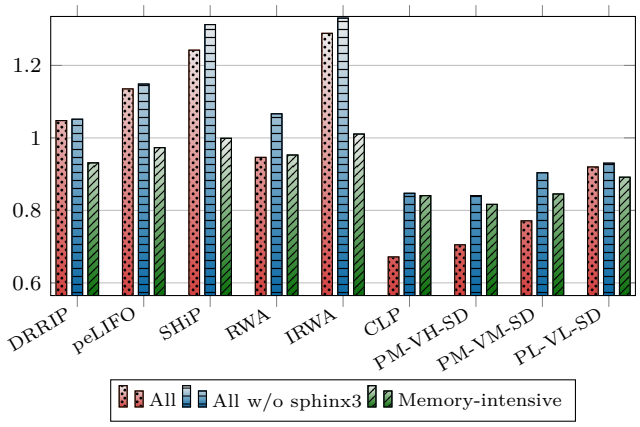


FIGURE 12. Amount of writes to main memory normalized to LRU: SPEC CPU2006 suite.

15.3% when *sphinx3* is considered or not respectively, while the best-performing of our proposals (PM-VH-SD) reaches the 30 and 16% respectively. Third, the three *performance-oriented* policies augment the number of writes compared to LRU. Fourth, zooming into the *memory-intensive* programs, we observe that, although the results follow a similar tendency, the differences among policies get reduced[5]. Besides, our most aggressive policy even improves CLP for these applications.

Next, according to the model detailed in Section 3.2.1, we show how the achieved reductions in writes to memory translate into extensions of the PCM lifetime for each evaluated proposal. Notably, Table 5 illustrates the memory lifetime improvement (percentages) with respect to LRU.

As shown, the trends observed in the values of memory lifetime extension closely follow those observed for the writes reduction numbers when considering all applications. This is due to the fact that the ratios between the BFP of each evaluated policy and that of LRU, which modulates the contribution of the writes reduction factor to the memory endurance extension number as shown in Equation 2, are very similar across the board. Notably, the average of dirty words per writeback when considering all applications ranges between 5.95 in SHiP and 5.62 in PM-VM-SD, exhibiting LRU a value of 5.86, which implies BFPs ratios in the range 1.02-0.95. This makes the writes reduction number the major contribution to the memory improvement obtained. However, those policies exhibiting a BFP value lower than that of LRU obtain an additional memory lifetime improvement. Thus, PM-VM-SD, reducing 6% less memory writes than CLP when considering all applications except *sphinx3*, is able to almost match its endurance extension (17.1% vs 17.6%) since the geomean of the ratios between BFP and the BFP of LRU is 0.94 in our proposal while it is just 1.0 in CLP. When considering the *memory-intensive* programs, the memory lifetime improvements observed are further below from the values of writes reductions than when all applications are considered. This is due to the fact that the ratios obtained between the BFPs of the evaluated policies and that of LRU are higher than 1 across the board (up to 1.06 in the case of CLP), hence reducing the memory endurance extensions achieved. For these programs the average of dirty words per writeback is moderately lower, ranging from 3.9 in LRU to 4.17 in CLP.

In order to visually observe the memory lifetime improvements achieved, we use the memory endurance simulator detailed in 5, which reports the amount of memory available (expressed in percentage of available pages) as a function of the number of writes. We apply the change of variable in the x-axis from number of writes to time (expressed in generic units). Figure 13 illustrates the curves obtained when considering both all the applications except *sphinx3* and the *memory-intensive* programs. We observe how in both cases our PM-VH-SD policy reports the highest amount of memory available for a given time. For instance, when considering all applications except *sphinx3*, after 1.8 billions of programs execution PM-VH-SD maintains around 58% of pages surviving while CLP just maintains around 42%. Note that for the sake of clarity we omit some proposals (peLIFO, SHiP and IRWA) that do not provide satisfactory results according to Table 5.

Note also that the time reached for the *memory-intensive* programs until the whole memory fails is higher than that of all the applications, since, although the amount of writes performed is higher, the average of dirty words that exhibit the writebacks in this kind of applications is significantly lower than when considering all applications.

6.2.3. Performance

Although our prime goal is to extend the PCM lifetime, it is clear that a proposal meeting this requirement could not be adopted if it comes at the expense of a significant performance drop. In order to further evaluate the benefits that each policy reports, Figure 14 shows the performance (execution time) delivered.

First, we observe that most of our DRRIP-based proposals almost match the execution time of original DRRIP and even some of our algorithms –those including a medium-low *victimization* sub-policy (VM or VL)– manage to outperform it. Second, all our proposals outperform the other *write-aware* policies (especially RWA and IRWA, which significantly degrade performance, 4.1 and 10.2% over LRU respectively).

Third, the two most aggressive policies in reducing the amount of writes to memory (CLP and PM-VH-SD) do provide a moderately satisfactory trade-off with performance since they all penalize it around 1-2%,

[5]The percentages of writes reduction exhibited by most *write-aware* policies decrease for the *memory-intensive* programs (as analyzed in Section 6.4.1, this occurs even for an optimal policy).

Benchs/Policies	DRRIP	peLIFO	SHiP	RWA	IRWA	CLP	PM-VH-SD	PM-VM-SD	PL-VL-SD
All/All w/o sphinx3	-5.9/-6.3	-11.5/-12.2	-20.9/-24.9	5.4/-7.2	-23.1/-25.1	49.1/17.6	41.4/19.0	36.3/17.1	7.4/6.5
Memory-intensive	2.0	0.9	-3.7	3.6	-4.2	11.9	16.1	11.6	5.8

TABLE 5. Memory lifetime improvements (percentages) with respect to LRU: SPEC CPU2006 suite.

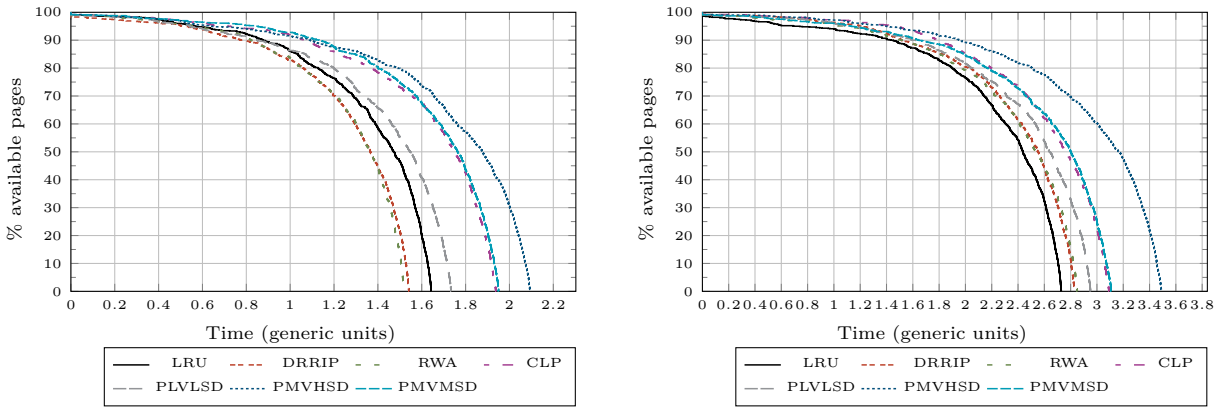


FIGURE 13. Available memory vs time for SPEC CPU2006 suite: (a) All, (b) Memory Intensive.

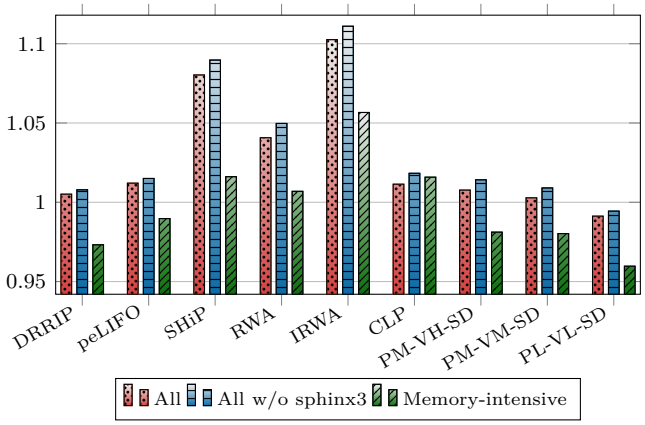


FIGURE 14. Execution Time normalized to LRU: SPEC CPU2006 suite.

depending on considering the *sphinx3* program or not. Intuitively, we would expect that this performance penalty was higher. Therefore, it is worth to note the key factors that may affect the performance delivered by the various *write-aware* approaches. Note that two opposing factors turn up:

1) Our proposed replacement algorithms are mainly focused on reducing the amount of dirty blocks evicted, being partially unaware of the consequent impact on performance, so it would be expected to lead to an increase on the number of read misses, hence augmenting the amount of accesses to main memory and hurting performance. However, our results reveal that for the PM-VH-SD policy –our algorithm achieving the highest reduction of writes to memory– the amount of LLC read misses in 8 out of 20 benchmarks decreases with respect to LRU. In fact, in the original DRRIP algorithm (the policy our schemes are based on) more than half of the benchmarks exhibit a lower amount of LLC read misses than LRU. The rationale behind this decrease in the amount of LLC read misses is that the temporal locality has been almost totally filtered at the LLC by the lower cache levels, thus a policy focusing only on temporal locality exploitation, like LRU, may provide poor results at this level of the hierarchy for some applications. Overall, the total amount of read misses in L3 considering all applications (and hence the events of read queue filling) has slightly increased in our proposals compared to LRU. This makes the net contribution on performance essentially negligible for our proposals.

2) Conversely, reducing the number of writes to memory reduces the pressure over the write queues, which leads to performance improvements (as explained in Section 5, once the corresponding write queue is full, the application stalls when a writeback from the LLC occurs). As for the write queues filling, our results illustrate that in the LRU baseline only in 8 out of 20 programs some write queue fills up at least once. For these 8 benchmarks, PM-VH-SD is able to significantly reduce the chances of filling up write queues and therefore mostly cancel the performance drops associated with the higher LLC read misses observed in 6 of these programs. For the remaining two applications (*soplex* and *h264ref*), as the amount of LLC read misses is also lower than that of the baseline, performance improvements of 16 and 5% respectively are delivered.

Fourth, among the *performance-oriented* policies, DRRIP and peLIFO provide satisfactory results (as expected) while SHiP surprisingly exhibits the second

worst number across the board (largely due to the performance drop delivered in *hmmer*, *astar*, *bzip2* and *soplex* applications). Finally, zooming into the *memory-intensive* programs, we observe that all our policies manage to significantly improve the performance of the other *write-aware* proposals and also that of some *performance-oriented* algorithms (note that DRRIP performs especially well for this kind of applications). Notably, our most aggressive algorithm, providing very similar writes reductions as CLP, manages to outperform it by more than 3%. Moreover, for these applications (and also considering all benchmarks) our less-aggressive approach even reports the best numbers across the board, exhibiting also the lowest amounts of LLC read misses among all evaluated policies.

6.2.4. Memory energy consumption

As for the energy consumption on the memory hierarchy for the different evaluated policies, our experimental results reveal that our proposals outperform all the other policies, reporting energy savings ranging between 4% (medium and less-aggressive algorithms) and 2% (high-aggressive policies) with respect to LRU. Only DRRIP and CLP are able to also report energy savings with respect to LRU. The other two *write-aware* policies significantly augment the energy consumption. Considering only the *memory-intensive* programs, the energy savings are greater across the board, being again our three proposals (and also CLP) those reporting the highest values (in the range from 9 to 8%). We should highlight that these savings mainly come from a lower number of writes to main memory, which are highly energy-consuming in the PCM technology. However, note also that for the SPEC applications the most aggressive policies in cutting the write traffic to memory are not the best in terms of energy consumption, since they penalize the execution time with respect to the less-aggressive schemes, leading the static energy to grow. Thus, the energy saving for these high-aggressive policies is partially canceled.

6.2.5. Putting it all together

Although considering a policy as the best one depends on the particular requirements of the system and the user, here we try to extract some insights about the trade-offs reported based on the different metrics evaluated. We consider that our high-aggressive algorithm provides satisfactory trade-offs between the memory endurance extension and performance. Notably, it exhibits a high number in memory lifetime extension (around 19% when the *sphinx3* application is not considered) without significantly degrading performance (around 1.4% penalty), reducing also the memory energy consumption around 2%. Our medium-aggressive policy also reports satisfactory trade-offs, improving memory endurance in a slightly more modest fashion (17%) but maintaining the system performance largely unchanged with respect to original DRRIP and LRU, and also reporting around 4% energy savings. Our less aggressive algorithm –although it contributes least to PCM lifetime extension (around 6.5%)– exhibits the best performance and energy numbers. As for the other *write-aware* policies, while RWA and IRWA clearly report poor trade-offs, CLP exhibits similar numbers to our most-aggressive proposal when considering all the applications except *sphinx3* (although CLP achieves a lower endurance extension and delivers lower performance), but performs significantly worse when we focus on the *memory-intensive* programs (our PM-VH-SD is able to improve memory lifetime by 4% more than CLP while also exhibits a performance value more than 3% better than CLP). Finally, the *performance-oriented* policies –although exhibiting satisfactory performance numbers, except SHiP– fail in improving the memory endurance.

6.3. *Write-aware* policies in a multi-core scenario

In this section we extend the prior study about the behavior of different cache replacement policies to a multi-core scenario. We just move to this new and more realistic setting without any change in our proposed algorithms. We evaluate the same policies that in the single-core environment for both multiprogrammed workloads (Section 6.3.1) and multithreaded applications (Section 6.3.2). For this purpose, we measure again the number of writes to main memory, endurance, execution time and energy consumption in the memory hierarchy that each policy involves normalized to LRU. However, due to the inherent non-determinism that all simulators exhibit (especially in multi-core environments, where the number of instructions executed across different policies are not stable owing to the random interleaving among memory accesses of different threads) and for the sake of higher accuracy, we opted to employ in this scenario the geometric mean of the metrics above mentioned but divided by the total number of instructions executed. Note that, conversely, in the single-core scenario both kind of metrics match, since all the benchmarks execute the same amount of instructions (1B) in all the runs. Finally, in Section 6.3.3 we analyze the trade-off between memory endurance and performance that the evaluated policies exhibit.

6.3.1. Multiprogrammed workloads

We employ 12 mixes made up of applications from SPEC CPU2006 chosen accordingly to the WPI categories illustrated in Table 2. We randomly compose 3 mixes made up of applications with high values of WPI (mixes 0, 1 and 2, referred to as the *memory-intensive* ones), 3 made up of programs exhibiting a medium WPI ratio (mixes 3, 4 and 5) and 2 composed

by benchmarks with low WPI values (mixes 6 and 7). We also evaluate four mixes merging applications with WPI corresponding to different categories (mixes 8 to 11). The detailed mixes are illustrated in Table 1. Besides, like in the single-core configuration, we report data considering both all applications and only the *memory-intensive* programs.

6.3.1.1. Amount of writes and Endurance Figure 15 shows the number of writes to memory *per instruction* that each evaluated policy performs normalized to LRU. As shown, considering all mixes, our most-aggressive proposal exhibits the best behavior. Notably, PM-VH-SD achieves a writes reduction of around 19%. The rest of our techniques manage to cut the write traffic to memory in the range from 14-12%. The other *write-aware policies* (except CLP that reduces writes by around 16%) as well as the *performance-oriented* ones behave significantly worse. Indeed some of them even augment the amount of writes with respect to the baseline.

When we just consider the 3 *memory-intensive* mixes, our proposals with medium and low-aggressiveness are able to further increase the writes reduction capability, mainly due to the numbers that they exhibit for the *mcf* application (better than those of all the other policies evaluated), which appears in these 3 mixes. Besides, our three techniques –and also SHiP and peLIFO– achieve the best numbers across the board (reductions ranging from 22 to 15%), significantly outperforming the other *write-aware* evaluated policies. Even 2 of our proposals exhibit an amount of writes more than 10% lower than that of CLP, the best-performing of the other *write-aware* techniques.

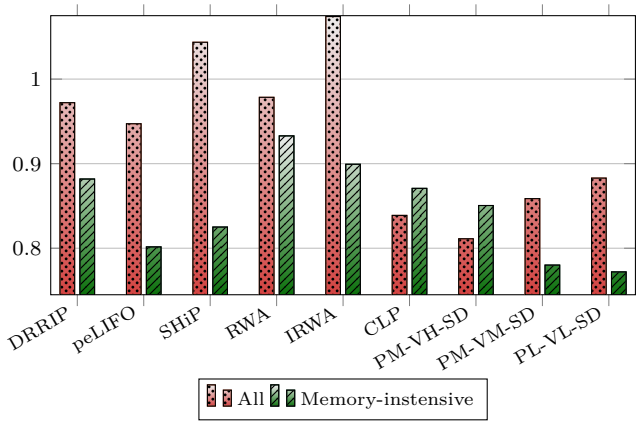


FIGURE 15. Writes to main memory per instruction normalized to LRU: multiprogrammed workloads.

Table 6 illustrates the memory lifetime improvements with respect to LRU for the different evaluated policies.

As in the single-core scenario, the general trend observed is similar to that of writes reduction numbers. However, in this case we observe a higher variability in the ratios of BFPs when considering all applications (from 1.02 in CLP to 1.15 in SHiP), which mitigates the memory endurance extensions. These numbers are even higher when considering just the *memory-intensive* programs, ranging from 1.04 in PM-VH-SD to 1.18 in peLIFO. The average of dirty words per writeback are slightly lower than in the single-core scenario, oscillating between 5.31 in CLP and 5.98 in SHiP.

Figure 16 shows the memory available when using each proposal over the time, where we observe that, according to data of Table 6, PM-VH-SD exhibits the best behavior when considering all applications and our three proposals are the best-performing across the board when considering just the *memory-intensive* programs. Note also that in this scenario, unlike in the single-core environment, we do not observe the effect of a significant higher amount of programs execution for the *memory-intensive* programs until the entire memory fails. This is due to the fact that the average of dirty words per writeback for these applications is just slightly lower than that when considering all the applications, ranging between 4.73 and 5.57.

6.3.1.2. Performance In order to evaluate the performance delivered by each proposal when executing multiprogrammed workloads, we analyze the *Instruction Throughput* (IT) and the *Weighted Speedup* (WS) metrics. The IT metric is defined as the sum of all the number of instructions committed per cycle in the entire chip ($\sum_{i=1}^{n} IPC_i$, being n the number of applications/threads), while the WS is defined as the slowdown experienced by each application in a mix, compared to its run under the same configuration when no other application is running on other cores ($\sum_{i=1}^{n}(IPC_i^{shared}/IPC_i^{alone})$). Figure 17 illustrates the IT that each evaluated policy delivers normalized to LRU. Note that, unlike in the case of CPI metrics, now values higher than 1 imply performance improvements with respect to LRU.

First, we observe that although our medium-aggressive policy suffers a performance drop of around 1%, our high-aggressive algorithm slightly improves LRU performance and our less-aggressive scheme outperforms LRU by more than 2% and also all the other algorithms evaluated. Second, all our techniques behave better than RWA and IRWA *write-aware* policies, while CLP performs slightly worse than our most aggressive proposal. Third, zooming into the *performance-oriented* policies, the figure reveals that DRRIP and peLIFO slightly improve the LRU performance, while SHiP suffers a moderate penalty due to the high performance drop exhibited in mixes 7, 10 and 11, which include the *hmmer* application, for which, as stated in Section 6.2.3 SHiP performs especially poor.

Fourth, for the *memory-intensive* mixes, we observe that all evaluated policies, except the most aggressive ones (CLP and PM-VH-SD) and RWA, improve their IT with respect to considering all the mixes. Note also that original DRRIP performs significantly well

Benchmarks/Policies	DRRIP	peLIFO	SHiP	RWA	IRWA	CLP	PM-VH-SD	PM-VM-SD	PL-VL-SD
All	-5.0	-4.9	-16.7	-1.6	-17.1	16.8	19.9	7.8	4.1
Memory-intensive	4.7	6.0	4.3	0.7	-1.1	7.5	12.9	14.4	13.1

TABLE 6. Memory lifetime improvements (percentages) with respect to LRU: multiprogrammed workloads.

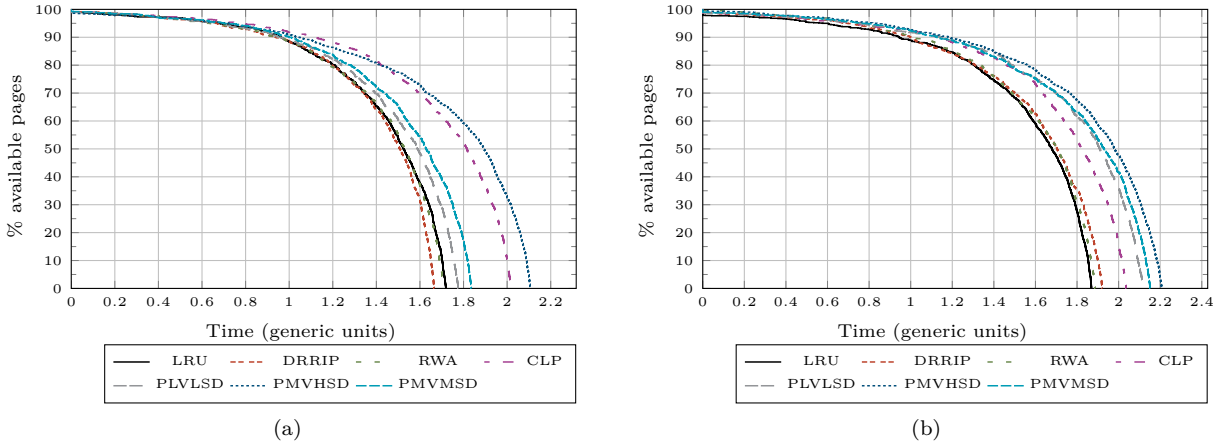


FIGURE 16. Available memory vs time for multiprogrammed workloads: (a) All, (b) Memory Intensive.

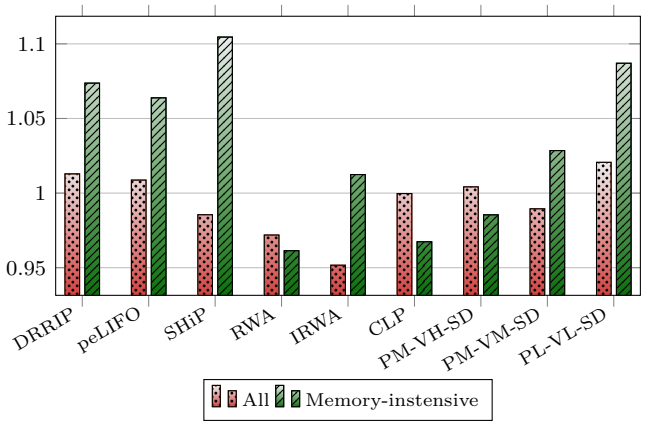


FIGURE 17. Instruction throughput normalized to LRU: multiprogrammed workloads.

for these mixes. Our less-aggressive proposal reports performance numbers that improve LRU by more than 8%, still outperforming all the other techniques –except SHiP–. All our proposals also outperform RWA, while IRWA is able to report for these mixes an IT improvement over LRU of around 1%. Finally, we should also highlight that all our techniques clearly outperform CLP –even up to more than 10%– for these *memory-intensive* workloads.

For the sake of simplicity and since in our context the WS does not constitute a metric as significant as the IT, we do not show the WS results obtained. Anyway, these results follow an analogous trend that those obtained when we evaluate the *instruction throughput*.

6.3.1.3. Memory energy consumption As for the energy savings *per instruction*, our results reveal that our three evaluated proposals and CLP achieve the best results with numbers around 9% (our most aggressive policy) and 7% (CLP and our medium-less aggressive schemes) with respect to LRU. The results from the other *write-aware* and *performance-oriented* policies are much more modest even augmenting the energy consumption as in the case of RWA, IRWA and SHiP. For the *memory-intensive* mixes, our PM-VH-SD policy reports energy savings of around 5% whereas our medium and less-aggressive schemes are around 12 and 15% respectively. The best of the other *write-aware* techniques (IRWA) hardly reaches 7% while CLP is around just 3%.

6.3.2. Multithreaded applications

In this section we inspect the behavior of our proposals when using the parallel applications from the PARSEC suite running in a system with two private cache levels (being L1 32KB/8-Way and L2 256KB/8-Way) and a shared level (L3). We explore both a 2-CMP system where L3 is 1MB/16-Way and a 4-CMP system with a 2MB/16-Way L3 . Again, we report data considering both all applications and only the *memory-intensive* programs. Following the same criteria defined in Section 6.2, this group of benchmarks includes *vips*, *facesim*, *dedup*, *ferret* and *fuidanimate* for the 2-CMP system while in the 4-CMP it consists of *vips*, *facesim*, *dedup*, *ferret* and *canneal*. Note that using the PARSEC applications we evaluate exactly the same algorithms as when employing the SPEC benchmark suite.

6.3.2.1. Amount of writes and Endurance Figure 18 shows the number of writes to memory *per instruction* that each policy generates. We can observe that in both

CMP systems all our proposals significantly outperform the *performance-oriented* policies (that roughly match and even exceed the number of writes of LRU) as well as the IRWA *write-aware* algorithm. Notably, our most aggressive policy reduces the write traffic to memory with respect to LRU by around 31 and 28% in the 2-CMP and 4-CMP evaluated systems respectively, improving CLP numbers by around 2 and 5% respectively. Moreover, when considering the *memory-intensive* programs, our best-performing policy is able to even involve amounts of writes 8% and 7% lower than those of CLP in the 2-CMP and 4-CMP systems respectively.

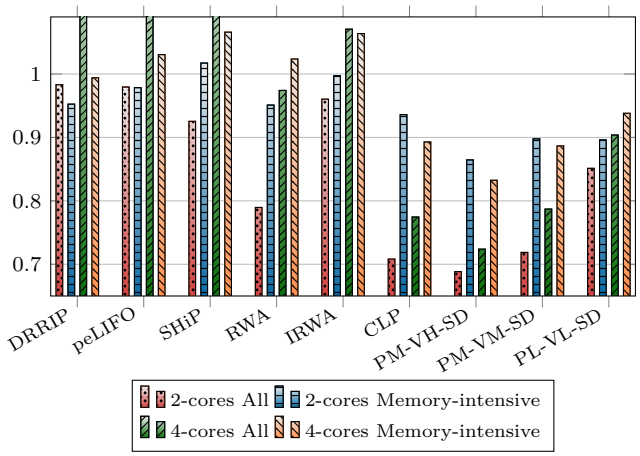


FIGURE 18. Writes to main memory per instruction normalized to LRU: PARSEC suite.

These amounts of writes leads to the memory lifetime improvements with respect to LRU shown in Table 7.

As in the case of the single-core scenario, in the 2-CMP system the trends observed in the memory lifetime improvements when considering all applications closely follow those of the writes reduction numbers since the BFP ratios just ranges between 0.99 and 1.02. For the *memory-intensive* programs, these values are lower than 1 for all the policies across the board (except RWA), which further extends the memory lifetime, with PM-VM-SD and PL-VL-SD exhibiting the best numbers (0.95 and 0.96 respectively), just outperformed by SHiP with 0.93. Thus, SHiP, even augmenting the amount of writes to memory with respect to LRU by 1.7% is able to extend the memory lifetime by 5.1% compared to the baseline. The average of dirty words per writeback ranges between 5.5 and 5.7 in this scenario.

The 4-CMP system is the scenario in where our policies benefit the most from the BFP ratios, that considering all applications range between 0.95 and 0.98 and considering just the *memory-intensive* programs range among 0.87 and 0.91. Notably, PL-VL-SD, reducing writes with respect to LRU by 9.6 and 6.2% for *All* and *memory-intensive* groups respectively, manages to improve the memory lifetime by 16.2 and 21.8% respectively. The average of dirty words per writeback ranges between 5.0 and 5.5 in this scenario.

Figures 19 and 20 show the memory available over the time for each evaluated proposal in the 2-CMP and 4-CMP systems respectively, where we observe that in all the cases PM-VH-SD exhibits the best behavior, significantly outperforming CLP. Furthermore, our three proposals behave especially well for the *memory-intensive* programs in both scenarios.

Note also here how the program executions number reached for *All* applications in the 2-CMP system before the entire memory fails is higher than that of the *memory-intensive* programs, since the difference between the memory lifetime improvements obtained by these two groups of benchmarks is the highest across all the scenarios evaluated, and, although the average of dirty words is slightly lower for *memory-intensive* applications, it is not enough to compensate the lower writes reduction obtained in these applications.

6.3.2.2. Performance Figure 21 shows the geometric mean of the *cycles per instruction* (CPI) reported by all evaluated policies.

First, we observe that our proposals behave slightly better in the 2-CMP system than in the 4-CMP one. Indeed, in the 2-CMP system most of our schemes even outperform original DRRIP.

Second, RWA performs worse than all of our proposals while IRWA outperforms most of them in both scenarios at the expense of reporting no memory lifetime extension or even augmenting it. CLP, the other *write-aware* policy under evaluation, reports worse numbers than all our proposals in both 2 and 4-CMP systems (up to 5 % performance degradation with respect to LRU in the 4-CMP system while our PM-VH-SD is able to improve LRU performance). Third, as expected, the *performance-oriented* policies –except peLIFO in the 4-CMP system– deliver satisfactory performance numbers.

Fourth, when considering the *memory-intensive* programs, in both CMP systems our algorithms that report higher writes reductions than CLP are also able to exhibit better performance numbers.

6.3.2.3. Memory energy consumption As for the energy consumption in the memory hierarchy *per instruction* reported by the evaluated policies, in the 2-CMP scenario the energy savings achieved are moderate across the board, being the the best-performing policy our low-aggressive PL-VL-SD, with reductions by around 5% compared to LRU. In the 4-CMP system our proposals obtain more modest numbers, up to 2.5%. In both scenarios they all outperform the other evaluated policies unless DRRIP in the 4-CMP system, which also reduces the energy consumption by around 2.5%.

For the *memory-intensive* programs, in the 2-CMP system our proposals obtain numbers between 8 and 6.5% while CLP reports energy savings of just 2.3% with respect to LRU. In the 4-CMP, our proposals report energy savings in the range 5-4.5%, while CLP

Benchmarks/Policies	DRRIP	peLIFO	SHiP	RWA	IRWA	CLP	PM-VH-SD	PM-VM-SD	PL-VL-SD
2-CMP All	3.2	2.5	9.3	24.9	4.2	38.5	44.1	39.3	18.3
2-CMP Memory-intensive	9.1	4.7	5.1	5.0	2.7	6.9	18.3	17.1	16.3
4-CMP All	-8.7	-15.2	-23.9	-2.5	-10.4	35.1	40.7	29.0	16.2
4-CMP Memory-intensive	11.2	-6.8	-2.3	-4.5	-4.6	24.2	30.7	22.9	21.8

TABLE 7. Memory lifetime improvements (percentages) with respect to LRU: PARSEC suite.

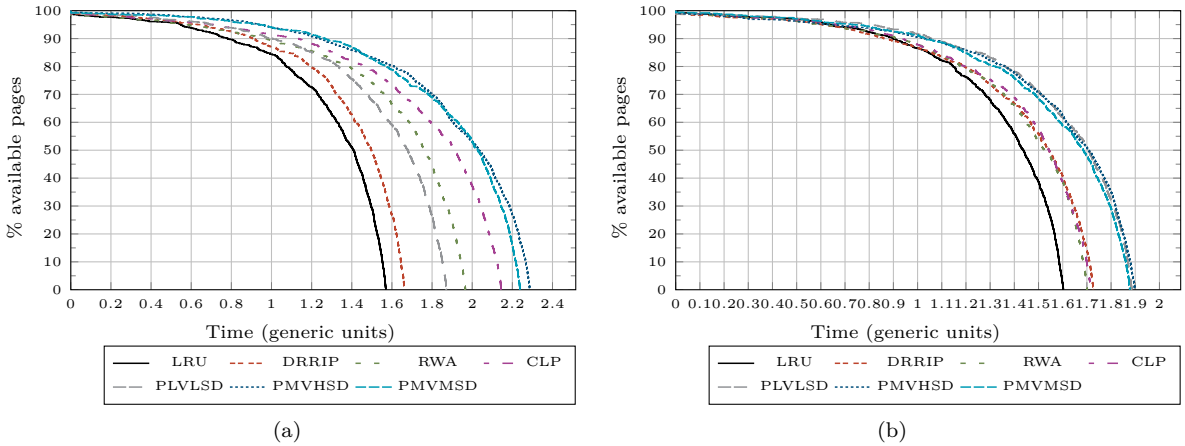


FIGURE 19. Available memory vs time for PARSEC suite in a 2-CMP: (a) All, (b) Memory Intensive.

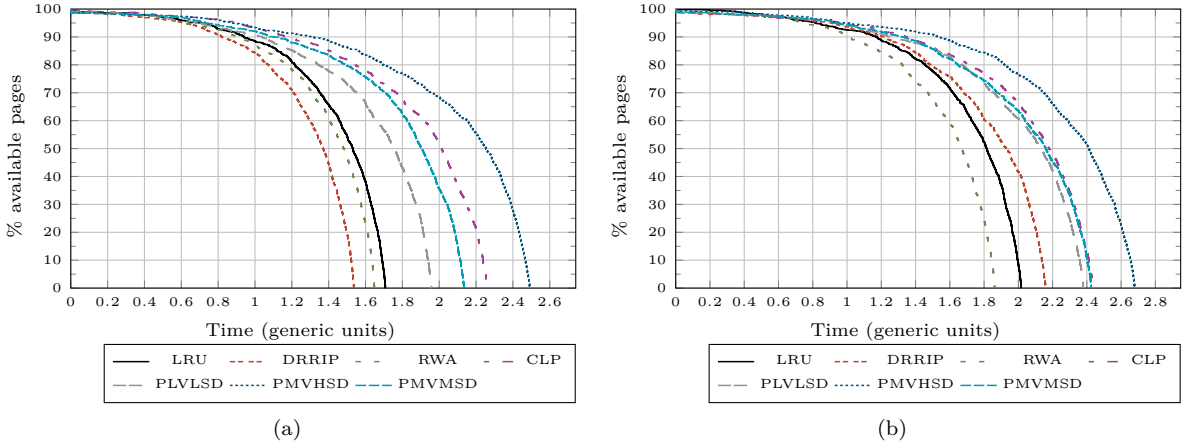


FIGURE 20. Available memory vs time for PARSEC suite in a 4-CMP: (a) All, (b) Memory Intensive.

reduces the energy consumption by 1%.

6.3.3. Putting it all together

Analyzing the trade-off between memory endurance and performance exhibited by the evaluated proposals in the multi-core scenario, we observe that when running multiprogrammed workloads, our PM-VH-SD is able to report the highest memory lifetime improvement (19.9%) and the highest reduction in the energy consumption in the memory hierarchy (more than 9%) without degrading performance. We must also highlight our low-aggressive PL-VL-SD policy, which clearly reports the best trade-off across the illustrated proposals[6] when considering only the *memory-intensive* programs. In such scenario, it achieves the second-highest memory lifetime improvement (13.1%, just slightly outperformed by PM-VM-SD), the second-highest throughput value (8.7% improvement with respect to LRU, just outperformed by SHiP) and the highest energy savings (more than 15%). In the same scenario CLP reports 7.5% of endurance extension, 3.3% of throughput degradation and just 3.3% of

[6]In the multiprogrammed workload scenario, other of our low-aggressiveness schemes report better numbers than the chosen algorithm. Notably, PM-VL-SD, delivering the same performance and energy numbers as PL-VL-SD, is able to reach 8.1% and 15.6% memory endurance extension for *All* and *memory-intensive* programs respectively vs 4.1 and 13.1% obtained by PL-VL-SD respectively.

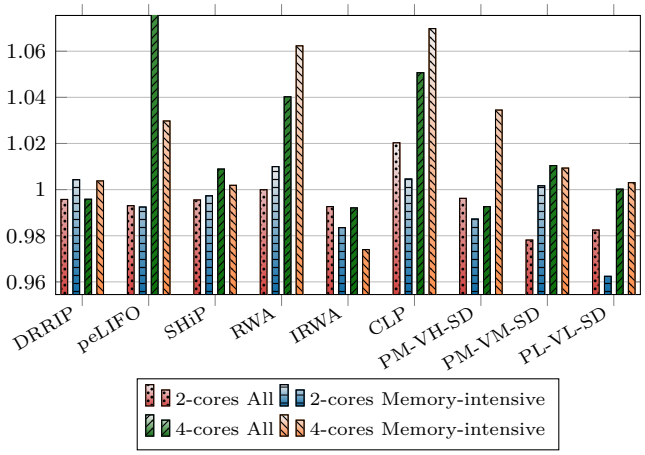


FIGURE 21. CPI normalized to LRU: PARSEC suite.

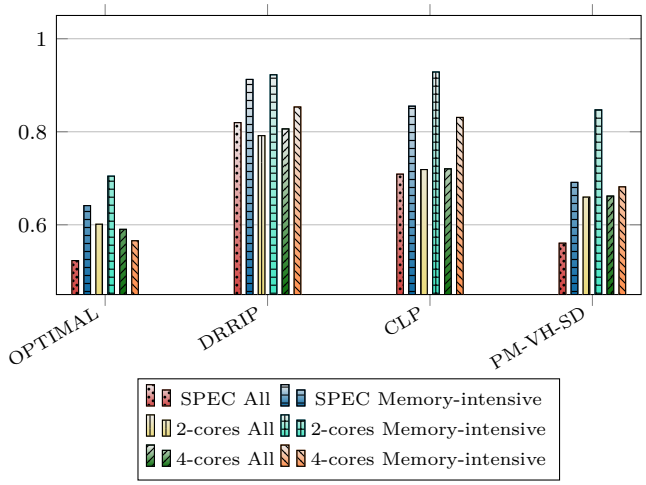


FIGURE 22. Amount of writes to main memory normalized to LRU: optimal policy and ours.

memory energy savings.

In the 2-CMP system running parallel applications, our most-aggressive policy, extending the memory lifetime up to 44.1%, provides also satisfactory performance numbers (around 1% improvement over LRU). Our medium-aggressive algorithm, that improves the memory endurance by around 39%, experiences also a performance improvement of around 2%. Finally, our less-aggressive policy, although extending memory endurance by a moderate 18%, manages to also improve performance by around 2%. They also are able to deliver energy savings of 3, 4 and 5% respectively. Thus, for our purpose of extending the memory lifetime without penalizing performance PM-VH-SD seems to be the the best option in this scenario. Note that CLP improves memory lifetime by 6% less than our proposal and also behaves worse (2.5%) than our PM-VH-SD in terms of performance. Moreover, for *memory-intensive* programs our three proposals significantly outperform CLP in memory lifetime while also deliver higher performance. In the 4-CMP, PM-VH-SD clearly provides again the best trade-off across the board (40% memory lifetime extension, performance improvement of around 1% and energy savings of 1.5%). Note that CLP improves endurance by 35%, but at the expense of a performance drop of around 5%. Finally, it is worth to note that the same trends, even augmented in favour of our proposals, are maintained when only the *memory-intensive* programs are considered.

Overall, we consider that in all the multicore-systems analyzed we always may find at least one of our policies that is able to clearly deliver a better trade-off (higher lifetime extension and also higher performance) than the other *write-aware* proposals and also *performance-oriented* algorithms. Notably, most *performance-oriented* policies decrease the memory lifetime with respect to LRU and even exhibit worse performance numbers than some of our proposals, while the RWA and IRWA *write-aware* policies clearly exhibit poor trade-offs. Finally, for CLP, which also reports significant extension in memory endurance, we perform a direct comparison with our proposals along the evaluation section to demonstrate our better trade-offs.

6.4. Additional analysis

Next we extend our study with some additional experiments.

6.4.1. Comparison to an optimal policy

In order to find out how far we are from the maximum writes reduction feasible, we compare the *performance-oriented* policies which report the lowest amount of writes –DRRIP and LRU–, our most-aggressive *write-aware* algorithm evaluated in the previous sections (PM-VH-SD) and also CLP with a straightforward optimal policy (in terms of writes to main memory), that operates as follows. For a given cache set, we allocate an array with an amount of entries matching the associativity (n) of the LLC, and traverse the trace of writebacks from L2 to L3. We first fill the array with the first n different blocks that are written back. Then, for every new block written back from L2 to L3 and not contained in the array, we do the following: (1) we analyze the trace onward to find the block in the array that will be re-written the furthest in the future; (2) we replace the found block with the incoming one (or we just bypass the new block if it is re-written further than all the other blocks in the array); and (3) we increment a counter. When the entire trace has been processed, the counter stores the total number of writes to main memory generated in this set under an optimal replacement policy.

Given that the simulation of the optimal policy is a time-consuming process, we opted to restrict the analysis to a pool of sets (32) conveniently scattered across the cache. Obviously, the other policies are also evaluated in the same cache sets.

Figure 22 illustrates the amount of writes to memory that each evaluated policy implies normalized to LRU for both SPEC and PARSEC benchmarks (2 and 4-CMP systems). Although the analysis is limited due to the reduced number of sets considered, we can infer a couple of relevant qualitative conclusions. First, our most aggressive policy is moderately close to the maximum theoretical reduction. Notably, PM-VH-SD is the algorithm closest to the optimal in all the scenarios evaluated for both *All* and *memory-intensive* benchmark sets. Second, for both SPEC applications and parallel programs in the 2-CMP system, the maximum writes reduction feasible for the *memory-intensive* programs is lower than that when considering all the benchmarks (in the 4-CMP system these numbers for both groups of benchmarks remain laregely unchanged). Overall we can extract the conclusion that although the optimal policy is unfeasible and our proposals are close to the optimal numbers, it seems that it still remains opened some avenue for improvement in order to even further reduce the current gap with this optimal policy and even improving the performance delivered.

6.4.2. Implementation Overhead

A new policy may induce two types of overhead, specifically, extra hardware (both extra storage needed for book keeping and extra logic needed for implementing the new algorithm) and impact on the critical path. We should start mentioning that, as many authors have previously pointed out, the updating of the replacement policy state is completely off the critical path [7, 8], which makes the critical path delay remain unaltered by our proposed changes. However, for the sake of completeness, we will prove in this section, by means of deeply analyzing the algorithm complexity of our proposals, that the delay generated by our changes, as well as the extra logic they involve, are both negligible compared to DRRIP. Besides, we will analyze the extra storage involved by our policies. To complete the section, we will also analyze the overhead of DRRIP, LRU, CLP, RWA and IRWA.

Algorithm Complexity: Let us analyze the algorithm complexity of each sub-policy for the different replacement policies.

- Insertion sub-policy: With respect to DRRIP, our policies only modify the *set-dueling evaluating metric*, so it comes "for free".
  Moreover, comparing DRRIP with LRU, note that, whereas the former policy only requires to update the RRPV of the new block, the latter one needs to update the position in the Recency Stack of all the blocks in the set.
- Promotion sub-policy: Our policies incorporate cleanness/dirtiness (PL) or read/write (PM and PH) information to the promotion process, something extremely simple. Besides, all this information is already present in the block or comes along with the access itself.
  Furthermore, compared to LRU, DRRIP promotion is much easier, since, like in the insertion sub-policy, DRRIP just updates the RRPV of the promoted block, whereas LRU promotion updates also the blocks between the promoted one and the MRU position.
- Victimization sub-policy: As established in [7], original DRRIP searches for the first block with the highest RRPV by replicating the Find First One (FFO) logic, requiring four FFO circuits that operate in parallel to find pointers to the first "0", "1", "2", and "3" RRPV registers. Then, a priority MUX chooses the output of the appropriate FFO circuit as the victim. When a block with *distant* RRIP is not found, DRRIP also requires additional logic to age the RRPV registers, using state machine logic for this purpose. Our policies could either duplicate the number of FFO circuits for distinguishing between clean and dirty blocks, leaving the delay intact vs DRRIP, or access the four FFO circuits sequentially, first for clean blocks and then for dirty blocks (if needed), leaving the logic overhead almost untouched with respect to DRRIP. Besides, VM entails some changes on the aging logic for being able to increment only clean blocks during the first stage of this sub-policy.
  Now let us move to the comparison among LRU and DRRIP. Similarly to DRRIP, the former policy requires a search for finding the LRU block. However, LRU needs no aging logic, making its complexity slightly lower.

To sum up, the time delay and extra logic that our policies introduce with respect to DRRIP can be considered negligible, whereas they are higher in LRU than in DRRIP.

Extra Storage: Original DRRIP just requires 2 bits per cache block for storing the state associated to the replacement policy. In our policies, no extra bits need to be added to those required by DRRIP.

Furthermore, as stated by the authors in [7], DRRIP implies less hardware overhead than LRU. Notably, the number of bits required per cache set, assuming an N-way associative cache, is $N * logN$ in LRU and $2 * N$ in DRRIP, which, for example, in our setting (LLC associativity = 16), would lead LRU to need twice as much storage overhead as DRRIP.

Finally, note that the implementation complexity of CLP, given that this policy is based on a conventional LRU, would be the same (or even higher) as that of LRU. Other approaches in which CLP was based on efficient implementations of LRU (such as Tree-LRU) would be possible, but this would come at the expense of some performance degradation. RWA and IRWA, like in our case, are DRRIP-based policies so they

entail a negligible implementation overhead compared to original DRRIP. Indeed, the overhead is even smaller than in our policies due to the fact that RWA and IRWA preserve the victimization sub-policy unchanged with respect to DRRIP.

6.4.3. Sensitivity to LLC size

In order to further evaluate the impact of our proposals, we inspect the reduction in the number of writes to memory[7] and also the performance delivered when larger sizes of LLC are considered. To analyze our proposals operation we scale the problem by augmenting the LLC size without changing the number of cores. We choose a 4-CMP system and explore multiprogrammed workloads (using SPEC CPU2006 applications) with LLC sizes of 4, 8 and 16 MB and parallel applications (PARSEC) employing 2, 4, 8 and 16 MB LLC sizes. We just evaluate LRU, DRRIP, CLP and our three chosen policies (PM-VH-SD, PM-VM-SD and PL-VL-SD). Figures 23 and 24 show the amount of writes and the performance (CPI) respectively when using multiprogrammed workloads, whereas Figures 25 and 26 illustrate the same information (using the throughput metric instead of CPI to measure performance) when multithreaded programs are employed.

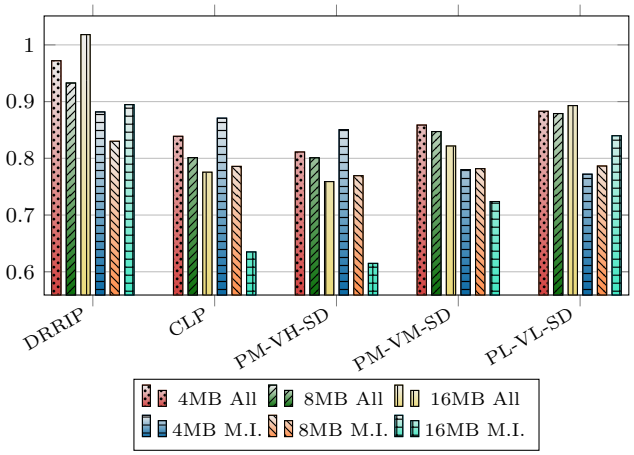


FIGURE 23. Writes to main memory per instruction normalized to LRU for different LLC sizes: multiprogrammed workloads.

For the evaluated mixes we observe that the same trends are still valid when we increase the LLC size. First, our PM-VH-SD policy reports again the highest writes reduction across the board when considering all the benchmarks, slightly outperforming CLP, while also delivers a higher throughput than CLP. Second, for the *memory-intensive* programs (labeled here as M.I. due to space constraints), although the differences get reduced with respect to a smaller LLC size, most of our

[7]For the sake of simplicity we omit endurance results, that follow a similar trend to writes reduction as previously demonstrated.

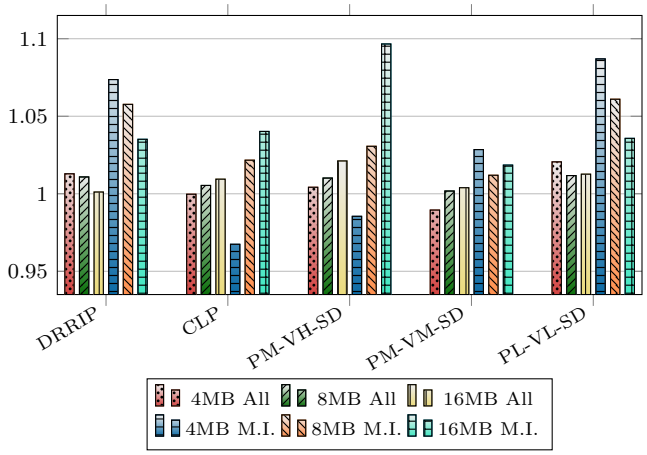


FIGURE 24. Instruction Throughput normalized to LRU for different LLC sizes: multiprogrammed workloads.

proposals still outperform CLP in both writes reduction capability and throughput. Thus, for this kind of applications, our PL-VL-SD policy, matching the CLP number regarding the amount of writes to memory in the 8MB scenario, is able to deliver a throughput 4% higher than that of CLP. Conversely, in the 16MB scenario, CLP is just outperformed by our PM-VH-SD policy.

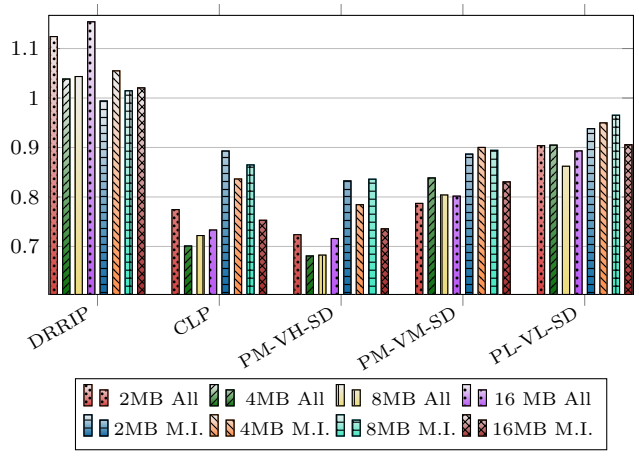


FIGURE 25. Writes to main memory per instruction normalized to LRU for different LLC sizes: PARSEC suite.

For multithreaded applications, as shown, our PM-VH-SD policy always reports the best numbers in cutting the write traffic to memory across the board for the four LLC sizes analyzed and for both *All* and *memory-intensive*[8] groups of benchmarks, moderately outperforming CLP. Furthermore, our PM-VH-SD scheme, being the technique that provides the highest reduction in amount of writes to memory, also behaves correctly in terms of performance, significantly outperforming CLP (up to 5%) in all the scenarios

[8]Note that the group of memory-intensive applications, chosen according to the criteria explained in Section 6.2.1 is not made up of exactly the same programs for all the LLC sizes considered.

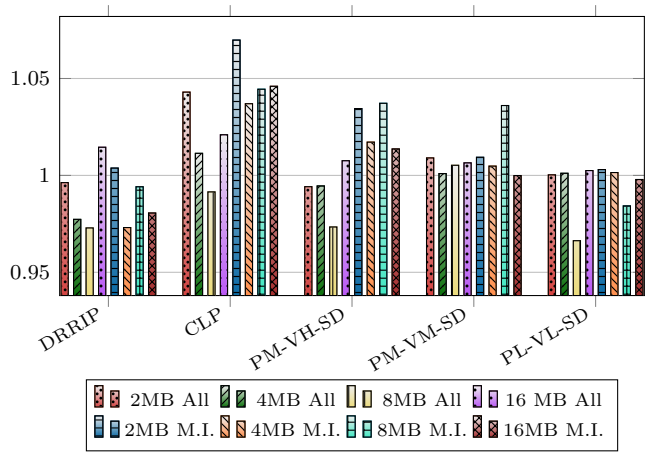


FIGURE 26. CPI normalized to LRU for different LLC sizes: PARSEC suite.

evaluated. Finally, note that our medium and less aggressive schemes analyzed also exhibit better performance numbers than CLP for all the LLC sizes and groups of benchmarks under evaluation.

7. CONCLUSIONS

In this paper we addressed the endurance constraint of the phase-change memories by means of the last level cache replacement policy. First, we evaluated the operation of classical replacement algorithms in terms of writes to main memory and then we proposed new policies with the main goal of minimizing this number in order to extend the PCM lifetime. The foundation behind these proposals is to merge as many modifications to a block as possible in a single writeback, while maintaining the system performance.

According to our results, the conclusions are triple. First, as the conventional *performance-oriented* replacement algorithms are absolutely unaware of the amount of writes performed to memory, they entail low PCM lifetime values, which suggests that they must be adapted to a potential future scenario where PCM-based systems prevail. Second, combining efficiently the proposed changes to the insertion, promotion and victimization sub-policies leads to algorithms that significantly cut the write traffic to memory and hence increase its lifetime, with low impact over performance. Specifically, our most aggressive policies deliver the best trade-offs between endurance and performance in both the single and the multi-core scenarios, reporting memory lifetime improvements in the range 20-45% without hardly penalizing performance. Besides, the write traffic reduction they achieve is not far from optimal. Concerning our medium and low-aggressive algorithms, they also report satisfactory results in both scenarios evaluated, managing to moderately improve the PCM lifetime and also reduce the energy consumption while delivering satisfactory performance results. Third, as demonstrated in Section 6, previously proposed *write-aware* policies (especially RWA and IRWA) clearly fail to achieve satisfactory trade-offs.

ACKNOWLEDGEMENTS

This work has been supported in part by the Spanish government through the research contract CICYT-TIN 2008/508, TIN2012-32180, and the HIPEAC-3 European Network of Excellence. Also it was supported by a grant scholarship from the University of Costa Rica and Costa Rican Ministry of Science and Technology MICIT and CONICIT.

REFERENCES

[1] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. (2009) A durable and energy efficient main memory using phase change memory technology. *ACM SIGARCH Computer Architecture News*, **37**, 14.

[2] Cho, S. and Lee, H. (2009) Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. *MICRO*, pp. 347–357.

[3] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. (2009) Scalable high performance main memory system using PCM technology. *ACM SIGARCH Computer Architecture News*, **37**, 24–33.

[4] Ramos, L. E., Gorbatov, E., and Bianchini, R. (2011) Page placement in hybrid memory systems. *ICS*, pp. 85–95.

[5] Rodríguez-Rodríguez, R., Castro, F., Chaver, D., Piñuel, L., and Tirado, F. (2013) Reducing writes in phase-change memory environments by using efficient cache replacement policies. *DATE*, pp. 93–96.

[6] Kim, C. (2001) LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, **50**, 1352–1361.

[7] Jaleel, A., Theobald, K. B., Steely, S. C., and Emer, J. S. (2010) High performance cache replacement using re-reference interval prediction (RRIP). *ISCA*, pp. 60–71.

[8] Chaudhuri, M. (2009) Pseudo-lifo: the foundation of a new family of replacement policies for last-level caches. *MICRO*, pp. 401–412.

[9] Wu, C.-J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely, S. C., and Emer, J. S. (2011) Ship: signature-based hit predictor for high performance caching. *MICRO*, pp. 430–441.

[10] Lee, B. C. et al. (2010) Phase-change technology and the future of main memory. *IEEE Micro*, **30**, 143.

[11] Qureshi, M. K., Gurumurthi, S., and Rajendran, B. (2011) Phase change memory: From devices to systems. *Synthesis Lectures on Computer Architecture*, **6**, 1–134.

[12] Hu, J., Xue, C. J., Tseng, W.-C., He, Y., Qiu, M., and Sha, E. H.-M. (2010) Reducing write activities on non-volatile memories in embedded cmps via data migration and recomputation. *DAC*, pp. 350–355.

[13] Liu, T., Zhao, Y., Xue, C. J., and Li, M. (2011) Power-aware variable partitioning for dsps with hybrid pram and dram main memory. *DAC*, pp. 405–410.

[14] Ferreira, A. P., Zhou, M., Bock, S., Childers, B. R., Melhem, R. G., and Mossé, D. (2010) Increasing pcm main memory lifetime. *DATE*, pp. 914–919. IEEE.

[15] Zhou, M., Du, Y., Childers, B., Melhem, R., and Mossé, D. (2012) Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM TACO*, **8**, 1–21.

[16] Zhang, X., Hu, Q., Wang, D., Li, C., and Wang, H. (2011) A read-write aware replacement policy for phase change memory. *Advanced Parallel Processing Technologies*, pp. 31–45. Springer.

[17] Lee, S., Bahn, H., and Noh, S. (2013) Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures. IEEE Transactions on Computers.

[18] Belady, L. A. (1966) A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, **5**, 78–101.

[19] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. S. (2007) Adaptive insertion policies for high performance caching. *ISCA*, pp. 381–391.

[20] Ban, A. (2004). Wear leveling of static areas in flash memory. US Patent 6,732,221.

[21] Kgil, T., Roberts, D., and Mudge, T. N. (2008) Improving nand flash based disk caches. *ISCA*, pp. 327–338. IEEE.

[22] Gal, E. and Toledo, S. (2005) Algorithms and data structures for flash memories. *ACM Comput. Surv.*, **37**, 138–163.

[23] Ben-Aroya, A. and Toledo, S. (2011) Competitive analysis of flash memory algorithms. *ACM Transactions on Algorithms*, **7**, 23.

[24] Qureshi, M. K. and Patt, Y. N. (2006) Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *MICRO*, pp. 423–432. IEEE Computer Society.

[25] Corbato, F. J. (1969) A Paging Experiment with the Multics System. *In Honor of P.M. Morse*, pp. 217–228. MIT Press.

[26] Binkert, N. et al. (2011) The gem5 simulator. *ACM SIGARCH Computer Architecture News*, **39**, 1.

[27] Intel (2013). http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html.

[28] Rosenfeld, P., Cooper-Balis, E., and Jacob, B. (2011) Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, **10**, 16 –19.

[29] http://www.cse.psu.edu/ xydong/software.html.

[30] (2013). http://www.spec.org/cpu2006/.

[31] Bienia, C. (2011) Benchmarking Modern Multiprocessors. PhD thesis Princeton University.

[32] Patil, H., Cohn, R. S., Charney, M., Kapoor, R., Sun, A., and Karunanidhi, A. (2004) Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. *MICRO*, pp. 81–92. IEEE Computer Society.

[33] Schechter, S. E., Loh, G. H., Strauss, K., and Burger, D. (2010) Use ecp, not ecc, for hard failures in resistive memories. *ISCA*, pp. 141–152.

[34] (2013). http://www.hpl.hp.com/research/cacti/.

[35] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. P. (2009) Cacti 6.0: A tool to understand large caches.