Reducing Writebacks Through In-Cache Displacement

MOHAMMAD BAKHSHALIPOUR, Sharif University of Technology, Iran and Institute for Research in Fundamental Sciences (IPM), Iran
AYDIN FARAJI, Sharif University of Technology, Iran
SEYED ARMIN VAKIL GHAHANI, Sharif University of Technology, Iran
FARID SAMANDI, Sharif University of Technology, Iran
PEJMAN LOTFI-KAMRAN, Institute for Research in Fundamental Sciences (IPM), Iran
HAMID SARBAZI-AZAD, Sharif University of Technology, Iran and Institute for Research in Fundamental Sciences (IPM), Iran

Non-Volatile Memory (NVM) technology is a promising solution to fulfill the ever-growing need for higher capacity in the main memory of modern systems. Despite having many great features, NVM's poor write performance remains a severe obstacle, preventing it from being used as a DRAM alternative in the main memory. Most of the prior work targeted optimizing writes at the main memory side and neglected the decisive role of upper-level cache management policies on reducing the number of writes.

In this paper, we propose a novel cache management policy that attempts to maximize write-coalescing in the on-chip SRAM last-level cache (LLC), for the sake of reducing the number of costly writes to the off-chip NVM. We decouple a few physical ways of the LLC to have a dedicated and exclusive storage for the dirty blocks, after being evicted from the cache and before being sent to the off-chip memory. By *displacing* dirty blocks in the exclusive storage, they are kept in the cache based on their *rewrite distance* and are evicted when they are unlikely to be reused shortly. To maximize the effectiveness of the exclusive storage, we manage it as a *Cuckoo Cache* to offer associativity based on the various applications' demands. Through detailed evaluations targeting various single- and multi-threaded applications, we show that our proposal reduces the number of writebacks, on average, by 21% over the state-of-the-art method and enhances both performance and energy efficiency.

CCS Concepts: • **Hardware** → **Memory and dense storage**; • **Computer systems organization** → *Multicore architectures*;

Authors' addresses: Mohammad Bakhshalipour, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran, School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran, bakhshalipour@{ce.sharif.edu/ipm.ir}; Aydin Faraji, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran, afaraji@ce.sharif.edu; Seyed Armin Vakil Ghahani, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran, vakil@ce.sharif.edu; Farid Samandi, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran, samandi@ce.sharif.edu; Pejman Lotfi-Kamran, School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran, plotfi@ipm.ir; Hamid Sarbazi-Azad, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran, School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran, azad@{sharif.edu/ipm.ir}.

1084-4309/2019/4-ART1 $15.00
https://doi.org/10.1145/3289187

Additional Key Words and Phrases: Non-Volatile Memory, Phase Change Memory, Read-Write Disparity, Writeback, Cache Management.

ACM Reference Format:
Mohammad Bakhshalipour, Aydin Faraji, Seyed Armin Vakil Ghahani, Farid Samandi, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Reducing Writebacks Through In-Cache Displacement. *ACM Trans. Des. Autom. Electron. Syst.* 24, 2, Article 1 (April 2019), 21 pages. https://doi.org/10.1145/3289187

1 INTRODUCTION

Non-volatile memories (NVMs), such as Phase Change Memory (PCM), Spin Torque Transfer RAM (STT-RAM), and Resistive RAM (ReRAM), have emerged as promising solutions for overcoming the scalability limitations of traditional SRAM and DRAM modules. Near-zero leakage power and high density are the common features of these technologies [17, 19, 51, 62, 73], making them attractive for being used in the memory hierarchy of future chips.

Among traditional technologies, DRAM has a high leakage power [48, 52] and poor technology scalability [26, 34, 42, 50]. Due to the large capacity of the main memory, DRAM has been considered to be replaced with alternative storage technologies. PCM, in particular, due to its zero standby power, high density, low read latency, and soft error resilience is a suitable candidate for replacing or augmenting DRAM in the main memory [3, 6, 35, 37, 45, 59, 61, 63, 85].

Despite having many great features, NVMs suffer from poor write performance and endurance, which are the main limitations, preventing their widespread adoption. The poor write performance and endurance of NVMs emanate from their intrinsic structures. For example, the structure of PCM requires driving a large current for a long time to modify the state of a cell, resulting in high latency and energy of write operations. Furthermore, writes to PCM cells diminish the injection contacts, giving rise to the fact that its endurance is limited to at most 10^9 writes per cell at the contemporary processes [7, 17, 71].

There is a large body of work that attempts to diminish the harmful effects of costly writes in the context of NVMs. *Wear leveling* techniques (e.g., [46, 60, 85]) enhance the lifetime of memory by balancing the distribution of writes. These methods remap the frequently written cells to rarely written ones, and hence, mitigate the wear-out in hotspot cells of the memory. *Wear limiting* techniques (e.g., [21, 38, 40, 76]) reduce the number of writes by coalescing and/or canceling the writes in the memory. While these methods are useful, they concentrate on the main memory itself, ignoring the management of upper-level caches. However, as writes are sent from higher[1] levels of the memory hierarchy, managing upper-level caches can significantly affect the behavior of main memory writes.

On the other hand, some approaches [13, 18, 28, 79, 80, 84] exploit the locality of accesses in the higher levels of memory hierarchy [9–11, 78] and attempt to reduce the number of off-chip NVM writes by efficiently managing upper-level caches. These approaches usually use predictors for identifying frequently written pieces of data and try to keep such data in the cache, in an attempt to reduce the number of main memory writes. While these techniques are effective at reducing the number of writes, their performance is significantly restricted by the accuracy of their predictors. Whenever a wrong prediction is made, an extra costly write is imposed on NVM or a block is unnecessarily kept in the cache for a long time, wasting its limited capacity.

[1]We use the term higher (lower) levels of the memory hierarchy to refer to the levels closer to (further away from) the core(s), respectively.

This work aims to improve the effectiveness of the last-level cache (LLC) management in the presence of NVM as the main memory of the system. Corroborating prior work [18, 79, 84], we observe that a considerable fraction of dirty blocks is rewritten several accesses after their eviction from the LLC. Based on this observation, we propose a cache management policy that attempts to maximize write-coalescing in the on-chip SRAM caches, in an attempt to reduce the number of costly off-chip NVM writes. Instead of employing a predictor for identifying frequently written cache blocks, we propose to *displace*[2] *all dirty blocks in the cache, until they are unlikely to be rewritten in the near future*. For enabling displacement, we decouple a few physical ways of the LLC and dedicate them exclusively to the dirty blocks *after the eviction, and before writeback*. By displacing dirty blocks, we build a structure that provides *associativity based on the demand of various applications*, and consequently, dirty blocks would remain in the cache based on their *rewrite distance*[3]. Thus, many write operations coalesce in the cache and writebacks are performed only when the blocks *are not likely to be rewritten shortly*.

In this paper, we make the following contributions:

- We corroborate prior work [18, 79, 84] that a significant fraction of dirty blocks is rewritten several accesses after the eviction from the LLC. We exploit this phenomenon for reducing the number of NVM writes via further keeping the dirty blocks in on-chip SRAM caches.
- We make the observation that the rewrite distance of cache blocks significantly vary among applications. As such, we show that techniques that rely on fixed associativity to further maintain dirty cache blocks are unable to considerably reduce the number of memory writebacks.
- For overcoming the limitations of prior approaches, we propose a novel architecture that attempts to maintain dirty blocks in the LLC based on the application's rewrite distance. We decouple a few physical ways of the LLC and keep dirty blocks there, after the eviction and prior to writeback. By displacing blocks in the decoupled physical ways, we provide associativity based on applications' demands.
- Finally, we evaluate our proposal using various single- and multi-threaded applications and compare it against state-of-the-art proposals. We show that our proposal substantially reduces the number of NVM writes (26% on average, and up to 87%) and improves both performance and energy efficiency. Moreover, we demonstrate that the proposed method outperforms the best-performing previous proposal by 21% on reducing the number of writes while imposing less overhead.

2 MOTIVATION

Whenever a dirty block is evicted from the LLC, it should be written to the off-chip NVM. As writes to NVM consume significant energy and lessen the lifetime of the cells, decreasing the number of writes is crucial for energy-efficiency and durability. Moreover, long-latency NVM writes can potentially interfere with performance-critical read requests, and hence, diminish the overall system performance [4].

The write can be discarded if LLC chooses to evict a clean block instead of a dirty block. However, naively evicting clean and keeping dirty blocks in the LLC would result

[2] *Displacement* is referred to changing the location of a piece of data in the storage. In this paper, we use this term for referring to the event of changing the location of a block in the cache. As is discussed in this paper, as well as in prior work [27, 66], displacement is feasible only when more than one hash function is used for indexing the cache.

[3] By *rewrite distance*, we mean how many LLC accesses it takes for a dirty cache block to be written again.

in substantial performance degradation, as evicted blocks may be re-referenced shortly. It is useful if LLC seeks to *maintain dirty blocks that will be re-referenced soon*. This way, multiple writes to the same block would coalesce in the LLC, and whenever the block is *unlikely to be re-referenced in the near future*, a *single* writeback would be accomplished by evicting the dirty block.

Figure 1 shows the cumulative distribution of LLC accesses between a writeback (i.e., a dirty eviction) and the subsequent write to the *same* block in the LLC[4]. The figure implies that for achieving a certain amount of write-coalescing (y-axis), how long, in the number LLC accesses, the dirty blocks should be *further* kept in the LLC (x-axis). As shown, a considerable fraction of writebacks is rewritten several accesses after their evictions. For example, in `calculix`, 27% of dirty evictions are rewritten within hundred accesses following the eviction. This phenomenon suggests that we can reduce a considerable fraction of writebacks via moderately further keeping the dirty blocks in the LLC. The figure, moreover, demonstrates that the rewrite distance of blocks drastically varies across workloads. For example, coalescing half of the writes in `blackscholes` requires keeping the dirty blocks in the LLC for 10^5 accesses after the eviction, while it is 10^6 in `vips`. This phenomenon also suggests that the time that we should further keep the dirty blocks in the LLC, in order to reduce the writebacks to a certain extent, differs across workloads.

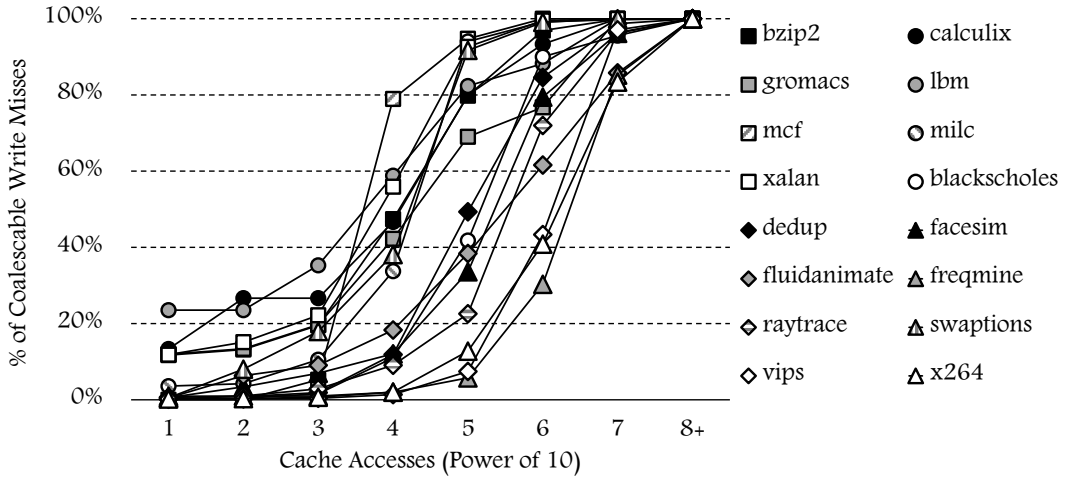

Fig. 1. The cumulative distribution of LLC accesses between a writeback and the subsequent write to the same block in the LLC. The figure implies that for achieving a certain amount of write-coalescing (y-axis), how long the dirty blocks should be further kept in the LLC (x-axis).

The discrepancy in rewrite distance of workloads calls for a general method for identifying blocks that are likely to be rewritten in the near future. One solution is using a predictor to classify evicted dirty blocks into two groups (i.e., "will be rewritten in the near future" or "will be rewritten in the far future"), and then speculatively perform replacement decisions based on the predictions. Such an approach, which is employed by many pieces of prior work in different ways, however, suffers from fundamental obstacles:

(1) The performance of such a technique heavily depends on the *accuracy* of the predictions. Whenever the prediction is wrong, an extra costly write would be imposed on NVM or

[4]The details of the methodology can be found in Section 4.

a block would unnecessarily be kept in the cache for a long time, wasting its limited capacity. For workloads with little access predictability (e.g., workloads that create their datasets on-the-fly during the execution [10]), significant performance degradation is expected with these methods, as we show in this paper.

(2) Such *binary* classifications, even with perfect accuracy, are still incapable of choosing the *best* victim upon cache replacements [30]. The problem is manifested when all of the blocks in a given cache set are (even correctly) predicted to be rewritten in the near future. In such situations, these techniques are unable to effectively choose the best victim (i.e., the dirty block which will be rewritten later than the others) for replacement.

(3) To enable the prediction, the history of access patterns should be maintained in the system. As such, such an approach needs precious SRAM storage for saving the metadata (e.g., 20 KB per core [79] or 34 KB [84] based on published numbers).

Instead of using a predictor, we propose to *partition the LLC into two portions* in order to provide an *exclusive storage for dirty blocks, after the eviction and prior to writeback*. Through *displacing* blocks in the exclusive storage, we provide associativity based on the dissimilar rewrite distance of various applications, paving the way for keeping them in the LLC, as long as they are prone to be rewritten shortly. This way, write operations are greatly coalesced in the cache and writebacks are performed only when the blocks are *not likely to be rewritten in the near future*.

3 THE PROPOSAL

We propose to partition the last-level cache into two parts where one part acts as a cache, and the other part is used as a special storage for dirty blocks, after being evicted from the cache and prior to being sent to the main memory. The LLC capacity is partitioned into two portions: **Normal Portion** and **Dirty Portion**. For this purpose, we leverage way-partitioning [16, 20, 22, 25, 57, 75, 77, 81], as it does not impose considerable hardware cost[5]. Fortunately, modern processors have LLCs with numerous ways (e.g., 8–32), as such, dedicating a few of these ways (say, 3) solely to dirty blocks does not result in substantial degradation in the hit ratio of the LLC, as we show in this paper (see Section 5.1).

Figure 2 shows an overview of the proposed design. Cache ways are partitioned into normal and dirty portions. The normal portion is the same as the baseline cache, but with fewer ways. It inherits the replacement policy of the baseline cache, without any modification. Dirty portion keeps dirty blocks that were evicted from the normal portion *until they are reused or written back*. As the dirty portion is used for write coalescing, it should keep dirty blocks based on their rewrite distance. Therefore, it should provide high associativity to capture the rewrite distance of various workloads. To offer high associativity, we borrow the ideas of Cuckoo Hashing [56] and Skewed-Associative Caches [72] to build a space- and

[5]Way-partitioning (sometimes called Column Caching [20]) was extensively utilized by many pieces of prior work that primarily aimed to improve fairness and provide quality of service. In a typical way-partitioning, cache ways are divided among applications, and each application is *restricted* to use only its assigned subset of ways. While way-partitioning is simple and can be implemented with minimal logic, it reduces the *effective associativity* of each partition, leading to performance degradation [67]. As such, way-partitioning is not readily applicable in the context of writeback reduction (target of this work) since a naive implementation not only does not reduce the writebacks but also is prone to increase them because of reducing the effective associativity for dirty blocks. In this paper, we propose a novel architecture that partitions the cache into two portions (normal and dirty) and uses the concept of *displacement* in order to provide high associativity for the *dirty* portion while not significantly reducing the associativity of the *normal* portion.

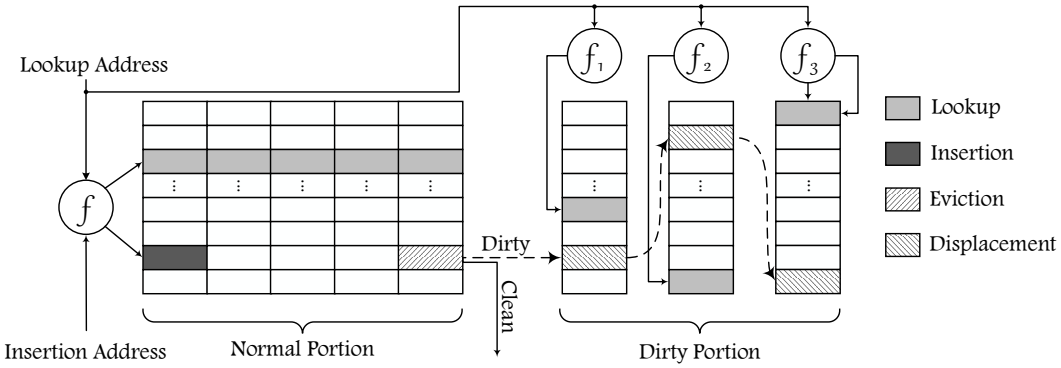

Fig. 2. Proposed design for managing the last-level cache. A portion of the LLC is dedicated to evicted dirty blocks, as a temporary storage before being sent to the main memory.

energy-efficient highly-associative structure. Every way in the dirty portion has its own hash function, *distinct* from others', enabling displacement in ways. In what follows, we describe the primary operations of the proposed design:

Lookup: The cache lookup is performed to determine whether the requested block is in the cache or not. In the proposed design, all of the physical ways (both normal and dirty) are searched in parallel. If the search results in a hit, the cache performs actions based on the location at which the block is found. If the block is found in the normal portion, the replacement status of the corresponding set is updated, quite similar to what happens in the baseline cache (e.g., the block is migrated to the head of the LRU stack). However, if the block is found in the dirty portion, it is *removed from the dirty portion* and is *inserted into the normal portion, as a new block*. Consequently, the location of the block in the dirty portion of the cache becomes *free*. If the search outcome is a miss, the block is simply requested from the lower level of the memory hierarchy.

Insertion: Whenever a new block is going to be stored in the cache, the insertion procedure is triggered. The incoming block is *always* inserted into the normal portion of the cache. The corresponding set of the incoming block is determined by the hash function of the normal portion, and then, the replacement policy (e.g., LRU) selects a victim to open a room for the new block. Finally, the block is inserted into the cache, and the replacement status is updated. Note that the replacement policy may decide to *bypass* the incoming block. In such cases, no further action is taken place, and both portions remain unchanged.

Replacement: Inserting a new block into the normal portion of the cache may result in evicting another block from the cache. For eviction, a candidate is determined based on the replacement policy of the baseline cache (e.g., LRU). If the eviction candidate is clean, it is safely removed from the cache. Otherwise, it is *inserted into the dirty portion*, in an attempt to maintain the block in the cache for further use.

The dirty portion iteratively displaces blocks in order to provide associativity as high as what is required to capture the rewrite distance of various applications. Whenever a dirty block is evicted from the normal portion of the cache, it should be inserted into the dirty

portion. The insertion procedure starts from one of the ways by determining the location at which the new block should be installed using the hash function of the corresponding way. Knowing the location, the content of that entry is evicted, and then the new block is written there. If the evicted block is a *valid* block, the insertion procedure repeats for it in a *neighboring* way (i.e., the evicted block is inserted into the adjacent way). This process repeats until an insertion discovers an empty location or a *stale block*.

By a stale block, we refer to a block that has remained in the dirty portion of the cache for a *long time*, and hence, is unlikely to be reused in the near future. We use the *number of displacements* of blocks as a heuristic for discovering stale blocks: a block is a stale block *if it has been displaced in the cache more than a predefined threshold*. At the hardware level, we augment each block in the dirty portion of the cache with a counter, named *displacement counter*, which indicates the number of times the block has been displaced in the dirty portion. At insertion time, blocks whose displacement counters are greater than a predefined threshold, named *displacement threshold*, are considered as stale blocks. That is, upon displacing a valid block in the dirty portion, its displacement counter is checked against the displacement threshold: if the displacement counter is greater than the displacement threshold, the victim is safely discarded from the cache and written back to the main memory, because it is a stale block. Otherwise, if the block is not stale, its displacement counter is incremented, and it is inserted into a neighboring way. Stale blocks are not likely to be reused as they have not been touched for a long period of time[6]. To have a regular distribution of blocks across the ways, the insertion operation begins at the way in which the last insertion has finished.

For avoiding livelock (i.e., infinite loops at insertion), a 4-bit counter, named *livelock counter*, is equipped to track the number of times an insertion attempt passes the first way. Whenever the counter overflows, the insertion is terminated, and the most recently displaced block is discarded. However, we find that this event rarely occurs, as most of the times, a stale or an empty entry is found in a few iterations. Figure 3 epitomizes the operations in the dirty portion of the cache for various cases.

The proposed design adapts the (actual) associativity of the dirty portion based on an application's demand and is different from what has been proposed for ZCache [66], which considers a *fixed* associativity. Moreover, the proposed design does not require a large timestamp counter for each entry to implement complete LRU stack (i.e., Full LRU [66]) and does not lose accuracy because of shortening the timestamp counter to save area (i.e., Bucketed LRU [66]). The proposed design just requires few (e.g., 2) bits for each entry to track displacement events, which grow drastically slower than the number of accesses.

4 EVALUATION METHODOLOGY

Table 1 summarizes key elements of our methodology, with the following sections detailing the evaluated designs, technology parameters, workloads, and simulation infrastructure.

4.1 Evaluation Parameters

We evaluate our proposal on both single and multicore platforms. Cores perform an in-order execution with fixed IPC = 1 for all instructions excluding memory operations. Cores are equipped with 32 KB instruction and data caches. The LLC capacity is 512 KB per core

[6]This phenomenon is known as Pareto distribution [5]: The longer a block remains idle (not written to) after a write, the longer it is expected to remain idle. Pareto distribution was extensively studied in computer science and is observed in system load [70], process lifetime [32], web traffic [23], and so on. Recently, Khan et al. [41] observed Pareto distribution in write requests of real-world applications.

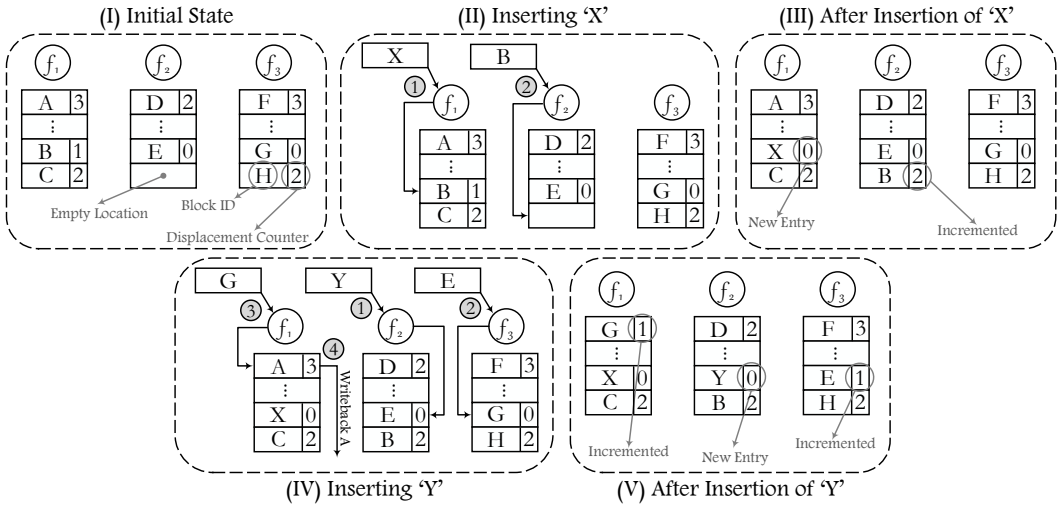

Fig. 3. Replacement process in the dirty portion of the cache. Considering that the *displacement threshold* is 3: (I) The initial state of the entries are shown. Some of the entries are empty while others are valid. Each entry is equipped with a *displacement counter*. (II) Block 'X' is dirty and now is evicted from the normal portion of the cache. As a result, it should be inserted into the dirty portion. Using f_1 hash function, it finds the location at which it should be installed. Since that location is neither empty nor contains a *stale* block, the insertion procedure should be continued after inserting 'X.' Therefore, 'X' is stored in that location, and the previous content of the location is used for indexing the neighboring way. As f_2 hash function maps 'B' to an empty location, 'B' is stored, and the insertion procedure is finished. (III) The state of the entries after inserting 'X' is shown. Since 'X' is a new entry, its *displacement counter* is set to 0. Moreover, as 'B' experienced displacement in the previous insertion procedure, its *displacement counter* is incremented. (IV) 'Y' is going to be inserted in the dirty portion. We start from the way in which the last insertion has finished (i.e., second dirty way). f_2 hash function maps 'Y' to a location in which 'E' exists. 'Y' is installed, and since 'E' is not a *stale* block, it should be displaced. 'E' resumes the insertion procedure in the next way, and uses f_3 to find the proper location. The location indicated by f_3 belongs to 'G' which again is not a *stale* block. Consequently, 'E' is installed, and the insertion procedure continues for 'G' in the next way (i.e., the first dirty way). f_1 maps 'G' to the location at which 'A' lasts. The *displacement counter* of 'A' is 3. If we want to place 'G' at the location of 'A' and then displace 'A,' the next *displacement counter* of 'A' will be 4, which is greater than the *displacement threshold*. As a result, we consider 'A' as a *stale* block and safely discard it. That is, we install 'G' in the cache and trigger a writeback operation for 'A.' (V) The state of the entries after inserting 'Y' is shown. The *displacement counter* of 'G' and 'E' is increased, and that of the 'Y' is set to 0.

Table 1. Evaluation parameters.

Parameter	Value
Processing Nodes	1 and 16 Cores, 2 GHz, UltraSPARC III ISA, In-Order, IPC = 1 for Non-Memory Operations
L1-I/D Caches	32 KB, 2-Way, 1-Cycle Load-to-Use
L2 NUCA Cache	Unified, 512 KB per Core, 1/4 Cycles Tag/Data Lookup Latency
Interconnect	4×4 2D Mesh, Three Cycles per Hop Latency
Main Memory	2-bit MLC PCM [14], 1/4 Memory Channels for 1/16 Cores, 8 KB Row Buffer

and is shared among all cores. We set the number of physical ways of the LLC to eight, as it is the lowest value in today's commercial processors [1, 43]. The cache line size is 64 bytes. The main memory is a 2-bit MLC PCM[7]. The delay and energy parameters of PCM is modeled based on a real design [14]. Read requests are prioritized over writes in the memory scheduling policy, as reads are more critical for performance.

4.2 Workloads

We include a wide variety of single- and multi-threaded workloads. We choose single-threaded workloads from SPEC2006 [33] and multi-threaded programs from PARSEC [15]. The simulated workloads are listed in Table 2.

Table 2. Application parameters.

SPEC2006	
`bzip2`	Compression and Decompression
`calculix`	Classical Theory of Finite Elements
`gromacs`	Simulation of Protein Lysozyme
`lbm`	Lattice Boltzmann Method
`mcf`	Vehicle Scheduling Problems
`milc`	Simulation 4D Lattice Gauge Theory
`xalancbmk`	XSLT Processor
PARSEC	
`blackscholes`	Portfolio Calculation Using PDE
`dedup`	Data Stream Compression Kernel
`facesim`	Human Face Modeling
`fluidanimate`	Incompressible Fluid Simulation
`freqmine`	Data Mining of the Frequent Pattern Growth Method
`raytrace`	Real-Time Animations (e.g., Computer Games)
`swaptions`	Portfolio Calculation Using Monte-Carlo Simulation
`vips`	VASARI Image Processing System
`x264`	H.264/AVC Video Encoder

4.3 Simulation Infrastructure

We use an in-house execution-driven simulator, which faithfully models cache hierarchy and interconnect's latency. Cores execute system- and user-level instructions, generated by Simics [2]. DRAMSim2 [65] is modified to model PCM and is used for simulating the main memory.

We simulate all applications with their maximum-size datasets (i.e., `reference`/`simlarge` input sets for SPEC/PARSEC workloads). For single-threaded workloads, we run 4 billion clock cycles to reach a steady-state, then collect measurements for the subsequent 2 billion clock cycles. We also launch simulations for multi-threaded workloads and run 8 billion clock cycles for warming the large last-level cache and other components before collecting measurements for the next 8 billion clock cycles.

[7]In this paper, to be in-line with most of the prior work in the literature, we evaluate our technique when the PCM is used as the main memory. However, all of the discussions are also true when another NVM technology is used as the main memory (e.g., STT-RAM as the main memory [44]).

4.4 Cache Management Schemes

We evaluate and compare four systems, as follows:

Baseline: Last-level cache does not consider read-write disparity and replaces blocks based on their recency (i.e., LRU).

Writeback-Aware Dynamic Cache Management: *WADE* [79] is the state-of-the-art NVM-aware cache management policy and is implemented on top of the baseline. It employs a predictor for identifying *frequently written* cache lines and attempts to minimize the eviction of such lines, as a way to coalesce writes and reduce the number of writebacks. We implement WADE including the dueling policies and the prediction configuration based on the original proposal. The total storage overhead of this method is 20 KB per core.

Hybrid-Memory–Aware Cache Partitioning: *HAP* [80] is a recently-proposed approach for managing the LLC in the context of hybrid memory systems (i.e., systems that use more than one technology for building the main memory, e.g., simultaneous use of DRAM and PCM). Meanwhile, with slight modifications, it can be evaluated in our simulated system. Similar to prior proposals [13, 28], HAP grants a *second chance* to dirty blocks, in an attempt to maintain them in the cache to increase write-coalescing. That is, the first time a dirty block is chosen as the victim for eviction, it survives: it is not evicted, and instead, the next block from the LRU stack is evaluated for eviction. However, if it is not used until the subsequent replacement evaluation, it is discarded from the cache. As each cache block is required to maintain only a single bit to indicate if the block has benefited from its second chance, the total area overhead of this method is low (1 KB per core).

In-Cache Displacement: Our proposal for managing the last-level cache is *In-Cache Displacement (ICD)*. We partition cache ways into normal and dirty portions. The dirty portion uses displacement to relocate blocks for offering associativity as high as necessary to capture the rewrite distance of different workloads. We set the configuration of ICD based on the sensitivity analysis (Section 5.1).

5 EVALUATION RESULTS

5.1 Sensitivity Analysis

Dirty Ways. The effectiveness of ICD at reducing the number of writebacks depends on the number of physical ways dedicated to the dirty portion. Figure 4 (left) shows the sensitivity of the number of writebacks to the number of dirty ways[8]. As the number of ways increases, the residence time of dirty blocks in the cache grows, and hence, the probability of coalescing increases. As such, a significant number of writes coalesce, which leads to a reduction in the number of costly writebacks. On the other hand, Figure 4 (right) shows the read hit ratio of the last-level cache as the number of dirty ways varies. Not surprisingly, dedicating more ways to the dirty portion lowers the total read hit ratio of the cache. As the total number of physical ways of the cache is constant, increasing the number of dirty ways implies decreasing the number of normal ways. By reducing the number of normal ways, the read miss ratio of the cache increases, as there are fewer possible locations for blocks on which

[8]In this experiment, to isolate the effect of the *displacement threshold*, we set its value to a large number. Moreover, for the same purpose, we disable the *livelock counter* of the proposed design.

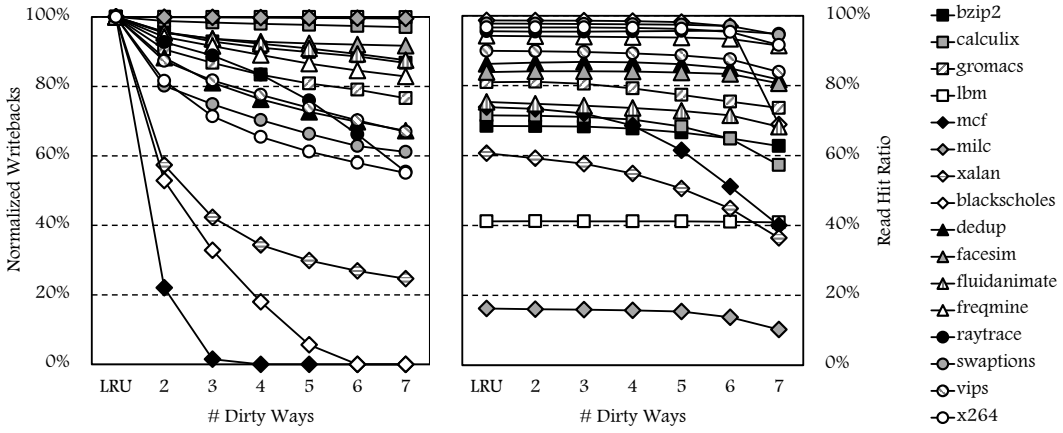

Fig. 4. The sensitivity of writebacks and read hit ratio of workloads to the number of ways dedicated to the dirty portion. Note that in the proposed technique, one dirty way is not possible, as displacement requires more than a single hash function. Dedicating all ways to the dirty portion is also impossible since we always insert the blocks into the normal portion of the cache.

the normal cache operations are performed. To take both writebacks and read hit ratio into consideration, we devote three ways to the dirty portion and use the rest of the ways for the normal operations of the cache[9]. Beyond three dirty ways, we observe diminishing writeback reduction, which indicates that merely three different hash functions are sufficient to provide a highly-associative structure for dirty blocks, capturing their various rewrite distances.

Displacement Threshold. Another parameter of ICD that can affect its efficacy is the displacement threshold. Displacement threshold is used to bound the number of displacements that each block experiences in the dirty portion. A small value for the displacement threshold may result in early eviction of useful blocks, while a large value can result in unnecessary displacements of stale blocks. Figure 5 shows the sensitivity of the number of writebacks (left) and the average number of block relocations (right) to the displacement threshold. Beyond three, there is a negligible reduction in the number of writebacks, at the same time, the average number of displacements grows linearly with the increase in the displacement threshold, causing unnecessary energy usage. Therefore, we limit the displacement threshold of each block in the dirty portion to three.

5.2 Writeback Reduction

To demonstrate the effectiveness of the proposed technique, Figure 6 shows the number of writebacks of various methods, normalized to that of the baseline LRU. As the number of writebacks directly influences the endurance and energy usage of PCM, its reduction is crucial for achieving efficiency. As clearly shown, except for few workloads, ICD significantly decreases the number of writebacks through maximizing write-coalescing. Compared to the baseline, ICD reduces the number of writebacks up to 87% and 26% on average. ICD outperforms

[9]Note that, dedicating three of the physical ways to the dirty portion of the cache does *not* mean that our design divests read requests of three cache ways. As discussed in Section 3, upon each request, all cache ways, either normal or dirty, are searched in parallel. That is, a read request may hit in the dirty portion of the cache, as well as the normal of the cache.

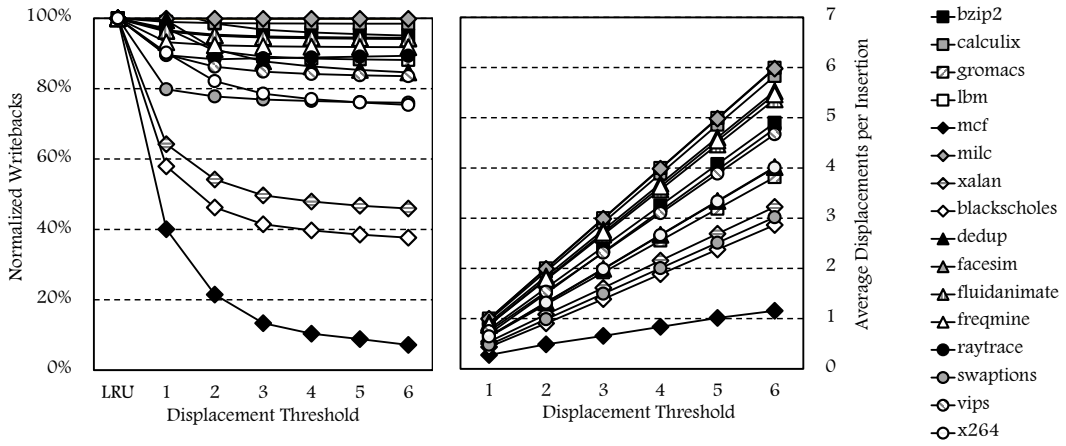

Fig. 5. The sensitivity of writebacks and movements of blocks to the upper-bound displacement limit.

WADE, the second best-performing method for single-threaded workloads, by 23%/19% in single-/multi-threaded workloads, respectively. ICD also outperforms HAP, the second best-performing method for multi-threaded workloads, by 31%/13% in single-/multi-threaded applications, respectively.

The efficiency of WADE is significantly restricted by the *accuracy of its predictions*. Whenever a wrong prediction is made, a costly write is imposed on the PCM, or a useless block remains in the cache for a long time. The problem exacerbates as WADE favors area-saving (i.e., reducing its area overhead) over prediction accuracy. For example, instead of allocating an entry in the predictor for each block, WADE offers the same prediction for

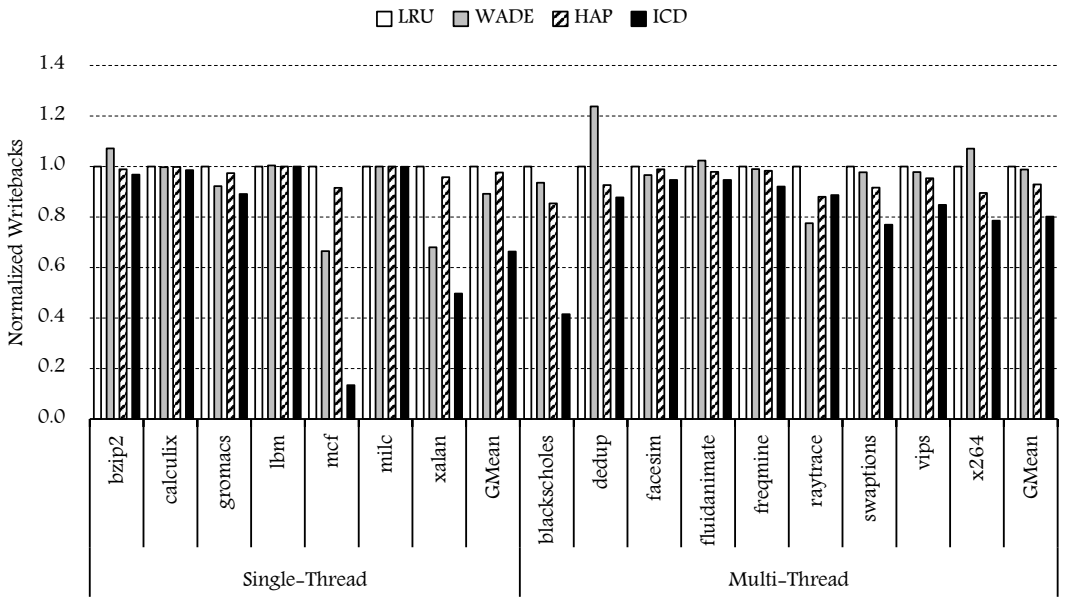

Fig. 6. The number of writebacks normalized to the baseline LRU.

many blocks, with the justification that they share an identical spatial region and exhibit the same behavior with respect to the write frequency, which is not always the case. Moreover, our investigations reveal that the *high start-up latency* of WADE is its another shortcoming. WADE requires observing multiple writebacks for a block to classify it as a frequently-written block, and filter its future writebacks. However, in some workloads, a significant number of writebacks repeat just a few times. For instance, in `calculix`, 72% of blocks are written back only twice over the course of the execution of the application, which WADE can do nothing for.

HAP, like WADE, does not consider the various rewrite distance of cache blocks. If the rewrite distance of a cache block is larger than the access frequency of the corresponding cache set, it is inevitably evicted before observing the subsequent access. Therefore, HAP (and other similar approaches [13, 28]) is able to coalesce only writes that happen close to each other in time. Meanwhile, ICD provides exclusive and highly-associative storage for dirty blocks that enables it to coalesce writes with various distances in time (cf. Figure 1).

5.3 Main Memory Energy

Figure 7 shows the energy usage of the PCM main memory under various techniques for managing LLC, normalized to that of the baseline system. As writes to the PCM are more energy-consuming than reads, reducing writebacks may lead to consuming less energy in the main memory. ICD outperforms both WADE and HAP, reducing the main memory energy usage by up to 15%. In some of the workloads, even though the number of writebacks has been reduced by an LLC management scheme, the memory energy usage has been increased (e.g., `raytrace` with WADE). The main reason is the increased number of read requests to the PCM because of the reduction in the LLC read hit ratio. Usually, the evaluated

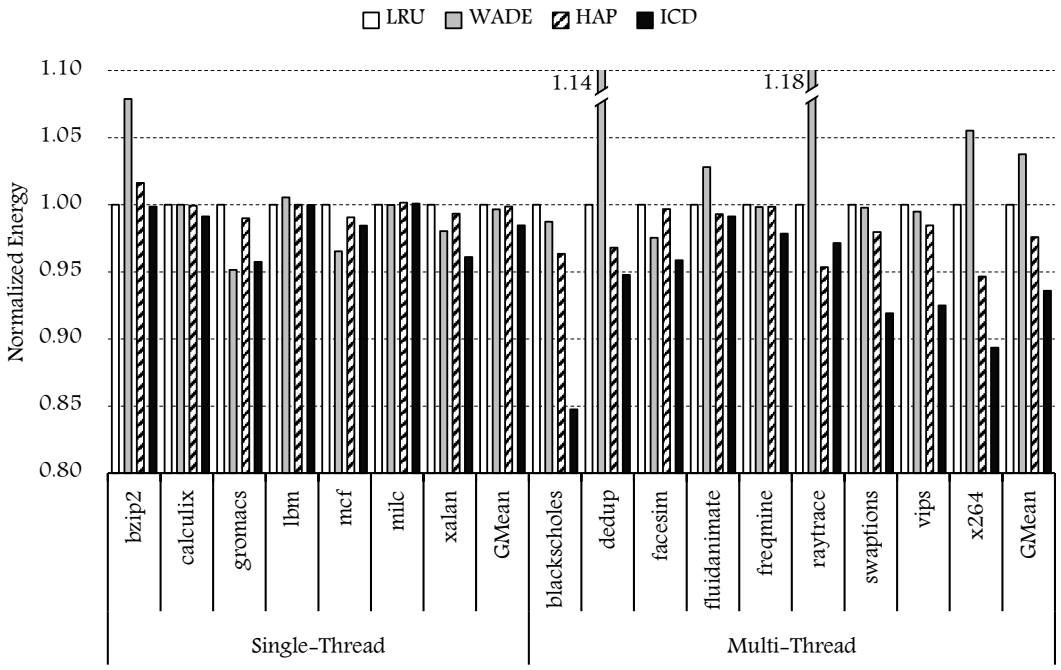

Fig. 7. The energy of PCM normalized to the baseline system.

methods sacrifice the read hit ratio of the LLC, to a certain extent, to further keep the dirty blocks in the cache. However, if the read hit ratio reduction is significant, there will be massive pressure on the main memory for serving read misses, which can potentially offset the benefits of writeback reduction (cf. Section 5.1). On average, ICD decreases the main memory energy usage by 6%/3% more than WADE/HAP.

5.4 Performance Analysis

Writebacks are not latency-sensitive operations by themselves, as they are usually served off the critical path of the execution [8]. Therefore, reducing or accelerating writebacks would not usually affect the performance of processors. However, frequent memory writes can delay read requests that are crucial for performance, resulting in performance degradation [4].

Figure 8 shows the performance of various cache management techniques, normalized to that of the baseline system. ICD either outperforms or matches the performance of the best of the baseline, WADE, and HAP across all workloads. On average and for single-threaded workloads, ICD improves the performance by 2%. In comparison, the average performance is almost unchanged with WADE and HAP. For multi-threaded workloads, the performance improvement of ICD is 12% on average (up to 41%), which is 13%/7% higher than performance improvement of WADE/HAP. For multi-threaded workloads, the performance improvement of ICD is more meaningful because of higher memory bandwidth utilization. Multicore processors are commonly bandwidth-limited [8, 12, 49], due to the insufficient number of memory channels, which itself is a result of poor pin count scalability of chips [36, 64]. In such systems, when PCM is used as the main memory, a long latency writeback caused by a core can induce significant contention with read accesses of other cores, leading to a notable performance penalty. Fortunately, ICD decreases such contentions by

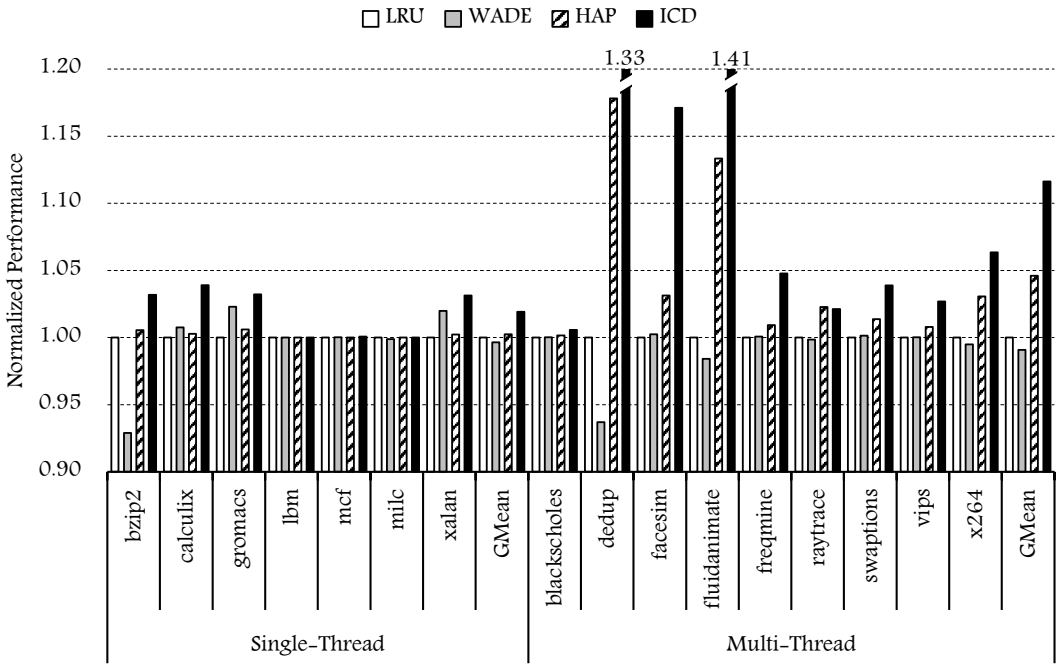

Fig. 8. Performance of methods normalized to the baseline system.

significantly reducing the number of writebacks, which results in a considerable performance improvement.

5.5 Overheads

As the displacement threshold in the proposed technique is three, ICD requires only two bits per each block in the dirty portion to track the number of displacements. Nevertheless, as the notion of *set* does not exist in the dirty portion, blocks do not carry their LRU-recency bits when they are transferred from the normal to the dirty portion. It means that cache lines of the dirty portion (without their displacement counter) are three bits[10] shorter than that of the baseline. Consequently, 2-bits-per-line storage requirement of ICD is offset by its 3-bits-per-line storage reduction. Furthermore, the hash functions that ICD practices are extremely low-cost and can be implemented with several XOR gates [69].

Displacing cache blocks in the dirty portion incurs energy overhead for the proposed design. Still, the energy usage of relocating blocks in the on-chip SRAM is far less than the energy of writes to the off-chip PCM. At 32 nm technology, a displacement consumes about 115^{pJ} (based on CACTI [55]), while a single writeback imposes 5.12^{nJ}–140^{nJ} depending on the written value [14]. On average across all workloads, the on-chip energy overhead of ICD accounts for 8% of its off-chip energy reduction.

Finally, as displacements are performed off the critical path of execution, they do not negatively affect the performance of the system. Displacements occur (1) when a request hits in the dirty portion of the cache, or (2) at a cache miss which triggers a dirty eviction from the normal portion of the cache. In both cases, the latency of displacements is not exposed to the performance-critical cache access. Whenever a read request hits in the dirty portion, first, the demanded data is sent to the corresponding L1, and then, the block is inserted into the normal portion of the cache, which may trigger several displacements. Displacements also do not stall the subsequent read requests, as reads are prioritized over displacements (and writes) in cache scheduling policies. Furthermore, cache misses experience delays much larger than the delay of displacements. Upon each LLC miss, the eviction candidate is determined based on the replacement policy of the normal portion. Concurrent with serving the miss, displacements are done, if the eviction candidate is a dirty line. As off-chip accesses impose long latency (120^{ns} at zero-load [14]), the latency of displacements (0.4^{ns} per single displacement) is completely hidden.

5.6 Comparison Against ZCache

The dirty portion of ICD has similarities with ZCache design from Sanchez and Kozyrakis [66]. ZCache, like ICD, uses different hash functions for physical ways, in order to provide high associativity via displacement. However, ZCache does not consider the read-write discrepancy in cache replacement and replaces the blocks solely based on their recency. Moreover, its *replacement procedure* (e.g., insertion of a new block, displacing blocks, manipulating metadata) is entirely different from that of ICD, as we discussed in Section 3.

Figure 9 shows the number of writebacks of ICD and ZCache, normalized to that of the baseline system. For ZCache, the actual associativity of the cache is 52, in which, each block contains an 8-bit timestamp counter for maintaining recency information, based on the original proposal [66]. The storage overhead of ZCache over the baseline cache is 5 KB and 80 KB in single and multicore systems, respectively.

[10] As the baseline cache is 8-way set-associative, each cache line keeps three bits for maintaining the LRU stack.

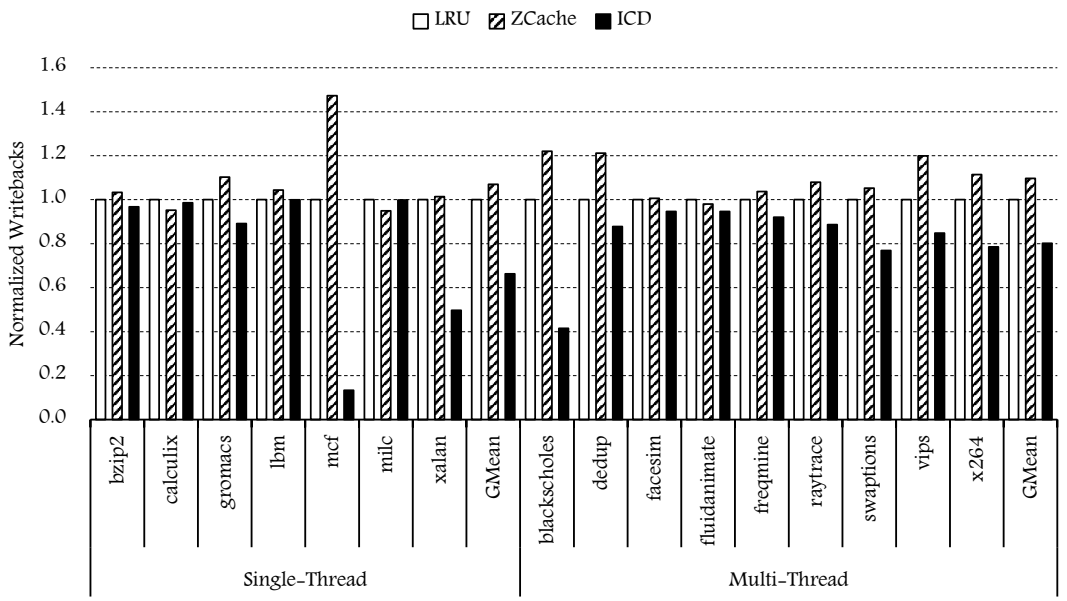

Fig. 9. The comparison of ICD and ZCache w.r.t. the number of writebacks. All numbers are normalized to the number of writebacks with the baseline system.

As shown in the figure, ZCache is unable to reduce the number of dirty evictions. This is because of the fact that ZCache is apathetic about clean-/dirtiness of blocks and replaces them only based on their recency. For several workloads, ZCache, not only does not decrease writebacks but also increases them, mainly because of trading early eviction of dirty blocks for further keeping recent clean blocks. On average and across all workloads, ZCache increases the number of writebacks by 9%, as compared to the baseline, which is 35% higher than the writeback traffic of ICD.

6 RELATED WORK

The discrepancy in the read-write cost of NVM has motivated a large body of research to overcome the performance, energy, and endurance limitations of writes. Wear leveling techniques attempt to improve the lifetime of memory by adjusting the distribution of writes. These techniques remap the frequently written cells to seldom written ones, and hence, alleviate the wear-out in hotspot cells of the memory. Start-Gap [60] uniforms physical layout of memory by regularly shifting the address-to-frame mapping. Segment swapping [46] performs the wear leveling at the granularity of large memory segments (e.g., 1 MB). In the proposed method, memory controllers maintain the record of writes of each segment, and periodically, swap segments with high and low number of writes. Zhou et al. [85] proposed to perform wear leveling in various layers for the sake of architecting a high-endurance memory. The scheme transfers bits in a line, changes lines in a segment, and swap segments in the memory for the purpose of maximizing write balancing.

Wear limiting approaches attempt to lessen the number of writes through mixing and/or dropping the writes in the memory. Joo et al. [40] proposed read-before-write to eliminate the expenses of redundant writes. Before each write, the current value of the cell is determined by a read operation, and the new value is compared against it. Finally, the write operation

(i.e., SET/RESET pulse) is performed only for cells with different values. Flip-N-Write [21] leverages the idea of bus-inverting [76] to further reduce the number of writes. Upon writing a word, the Hamming distance (i.e., the number of bits that differ) of the new value is checked against the previous value. If the Hamming distance is greater than half of the word size, the data is inverted and then is written to the NVM. Otherwise, the new value is stored without changes. Moreover, a single bit that indicates if the word is inverted is stored beside the data.

Mellow Write [83] is a novel approach to reduce the harmful effects of NVM writes. Based on the observation that slow writes result in less power dissipation and higher endurance, writes that are not likely to affect performance are written at a slow rate. Data compression, as a technique for mitigating the negative effect of NVM writes, was also studied in the literature [39]. Our method seeks to reduce the number of writebacks originated from the LLC and is orthogonal to the above approaches that target memory-side optimizations.

Alternatively, some other pieces of prior work attempt to reduce the adverse effect of off-chip NVM writes by managing on-chip SRAM caches. WCP [84] propose to partition the LLC capacity among several applications that are running on a multicore processor based on their generated writebacks. WCP favors producing balanced traffic on off-chip NVM and minimizing write-caused interference among applications. This method requires about 34 KB of storage per core to adjust its run-time decisions and applies only to multicore systems with LRU replacement policy. In this paper, we proposed a metadata-free LLC management that reduces the number of writebacks in both single and multicore processors, regardless of the number of different applications that are executed on cores, while supporting any cache management policy.

Managing on-chip caches in the presence of hybrid memories (i.e., leveraging both DRAM and NVM in a single system) [18, 80, 82] has been studied in the literature. These methods consider the disparity of DRAM and NVM when managing the blocks of each type in the LLC. ICD can be incorporated into such schemes for efficiently managing writeback-sensitive cache blocks (i.e., blocks that belong to NVM). Operating and controlling DRAM as a cache of NVM [47, 53, 61] was proposed for exploiting benefits of both classes of memories. Our proposal can be used with such methods to increase efficiency, as it aims at managing writebacks at higher levels of the memory hierarchy.

Cuckoo hashing was proposed by Pagh and Rodler [56] for building space-efficient hash tables. Though Cuckoo hashing has been mostly studied as a technique for software hash tables [54], hardware variants have been proposed to implement lookup tables in IP router [24], highly-associative cache [66], and scalable directory [27, 68]. Various proposals have pointed out the need for low-conflict and storage-efficient structures. Structures that rely on displacement were proposed for increasing area efficiency [31] and reducing conflicts [74]. Fotakis et al. [29] generalized Cuckoo hash for enhancing storage utilization. Panigrahy proposed to store multiple elements per bucket to raise Cuckoo hash space utilization [58]. ICD leverages Cuckoo hashing for managing a part of the LLC in order to reduce the number of writebacks by providing a highly-associative storage for dirty blocks.

7 CONCLUSION

Poor write performance of NVMs has been recognized as a major drawback. Therefore, a vast body of research targeted reducing the negative effect of writes on latency, energy, and endurance of NVMs, enabling them to replace unscalable DRAM in the main memory. Among various techniques, cache management methods can significantly reduce the number

of main memory writes because caches can exploit the high locality which exists in write access patterns.

In this paper, we proposed a low-cost cache management policy that attempts to maximize write-coalescing for the purpose of reducing costly writebacks. We showed that our proposal reduces the number of writebacks by up to 87% and outperforms the state-of-the-art designs.

ACKNOWLEDGMENT

We thank the members of HPCAN lab at Sharif University of Technology, in particular, Majid Jalili and Saeed Rashidi, and the anonymous reviewers for their valuable comments. We appreciate the member of the HPC center of IPM for preparing and managing the cluster that is employed to conduct the experiments of this work. This work is supported in part by a grant from Iran National Science Foundation (INSF).

REFERENCES

[1] Cortex-A7 MPCore, Technical Reference Manual. Available at <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464f/DDI0464F_cortex_a7_mpcore_r0p5_trm.pdf>.

[2] Wind River Simics Full System Simulator. Available at <http://www.windriver.com/products/simics>.

[3] Arjomand, M., Jadidi, A., Shafiee, A., and Sarbazi-Azad, H. A Morphable Phase Change Memory Architecture Considering Frequent Zero Values. In *International Conference on Computer Design* (2011).

[4] Arjomand, M., Kandemir, M. T., Sivasubramaniam, A., and Das, C. R. Boosting Access Parallelism to PCM-Based Main Memory. In *International Symposium on Computer Architecture* (2016).

[5] Arnold, B. C. *Pareto Distribution*. Wiley Online Library, 2015.

[6] Asadinia, M., Arjomand, M., and Sarbazi-Azad, H. OD3P: On-Demand Page Paired PCM. In *Design Automation Conference* (2014).

[7] Atwood, G., and Bez, R. 90nm Phase Change Technology with μTrench and Lance Cell Elements. In *VLSI Technology, Systems and Applications* (2007).

[8] Bakhshalipour, M., Lotfi-Kamran, P., Mazloumi, A., Samandi, F., Naderan, M., Modarressi, M., and Sarbazi-Azad, H. Fast Data Delivery for Many-Core Processors. *IEEE Transactions on Computers* (2018).

[9] Bakhshalipour, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. An Efficient Temporal Data Prefetcher for L1 Caches. *IEEE Computer Architecture Letters* (2017).

[10] Bakhshalipour, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. Domino Temporal Data Prefetcher. In *High-Performance Computer Architecture* (2018).

[11] Bakhshalipour, M., Shakerinava, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. Bingo Spatial Data Prefetcher. In *High-Performance Computer Architecture* (2019).

[12] Bakhshalipour, M., Zare, H., Lotfi-Kamran, P., and Sarbazi-Azad, H. Die-Stacked DRAM: Memory, Cache, or MemCache? *arXiv preprint arXiv:1809.08828* (2018).

[13] Barcelo, N., Zhou, M., Cole, D., Nugent, M., and Pruhs, K. Energy Efficient Caching for Phase-Change Memory. In *Mediterranean Conference on Design and Analysis of Algorithms* (2012).

[14] Bedeschi, F., Fackenthal, R., Resta, C., Donze, E. M., Jagasivamani, M., Buda, E. C., Pellizzer, F., Chow, D. W., Cabrini, A., Calvi, G. M. A., et al. A Bipolar-Selected Phase Change Memory Featuring Multi-Level Cell Storage. *IEEE Journal of Solid-State Circuits* (2009).

[15] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Parallel Architectures and Compilation Techniques* (2008).

[16] Bitirgen, R., Ipek, E., and Martinez, J. F. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *International Symposium on Microarchitecture* (2008).

[17] Burr, G. W., Breitwisch, M. J., Franceschini, M., Garetto, D., Gopalakrishnan, K., Jackson, B., Kurdi, B., Lam, C., Lastras, L. A., Padilla, A., et al. Phase Change Memory Technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena* (2010).

[18] Chen, D., Jin, H., Liao, X., Liu, H., Guo, R., and Liu, D. MALRU: Miss-Penalty Aware LRU-Based Cache Replacement for Hybrid Memory Systems. In *Design, Automation and Test in Europe* (2017).

[19] Chen, E., Apalkov, D., Diao, Z., Driskill-Smith, A., Druist, D., Lottis, D., Nikitin, V., Tang, X., Watts, S., Wang, S., et al. Advances and Future Prospects of Spin-Transfer Torque Random Access Memory. *IEEE Transactions on Magnetics* (2010).
[20] Chiou, D., Jain, P., Devadas, S., and Rudolph, L. Dynamic Cache Partitioning via Columnization. In *Proceedings of Design Automation Conference* (2000).
[21] Cho, S., and Lee, H. Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance. In *International Symposium on Microarchitecture* (2009).
[22] Cook, H., Moreto, M., Bird, S., Dao, K., Patterson, D. A., and Asanovic, K. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency While Preserving Responsiveness. In *International Symposium on Computer Architecture* (2013).
[23] Crovella, M. E., and Bestavros, A. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking* (1997).
[24] Demetriades, S., Hanna, M., Cho, S., and Melhem, R. An Efficient Hardware-Based Multi-Hash Scheme for High Speed IP Lookup. In *Symposium on High Performance Interconnects* (2008).
[25] Dybdahl, H., and Stenstrom, P. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *High-Performance Computer Architecture* (2007).
[26] Esmaili-Dokht, P., Bakhshalipour, M., Khodabandeloo, B., Lotfi-Kamran, P., and Sarbazi-Azad, H. Scale-Out Processors & Energy Efficiency. *arXiv preprint arXiv:1808.04864* (2018).
[27] Ferdman, M., Lotfi-Kamran, P., Balet, K., and Falsafi, B. Cuckoo Directory: A Scalable Directory for Many-Core Systems. In *High-Performance Computer Architecture* (2011).
[28] Ferreira, A. P., Zhou, M., Bock, S., Childers, B., Melhem, R., and Mossé, D. Increasing PCM Main Memory Lifetime. In *Design, Automation and Test in Europe* (2010).
[29] Fotakis, D., Pagh, R., Sanders, P., and Spirakis, P. G. Space Efficient Hash Tables with Worst Case Constant Access Time. In *Theoretical Aspects of Computer Science* (2003).
[30] Ghahani, S. A. V., Shahri, S. M., Bakhshalipour, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. Making Belady-Inspired Replacement Policies More Effective Using Expected Hit Count. *arXiv preprint arXiv:1808.05024* (2018).
[31] Hagersten, E. E., and Hill, M. D. Skewed Finite Hashing Function. US Patent 6,308,246.
[32] Harchol-Balter, M., and Downey, A. B. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. In *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (1996).
[33] Henning, J. L. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Compututer Architecture News* (2006).
[34] Hong, S. Memory Technology Trend and Future Challenges. In *International Electron Devices Meeting* (2010).
[35] Hoseinzadeh, M., Arjomand, M., and Sarbazi-Azad, H. Reducing Access Latency of MLC PCMs Through Line Striping. In *International Symposium on Computer Architecuture* (2014).
[36] Huh, J., Burger, D., and Keckler, S. W. Exploring the Design Space of Future CMPs. In *Parallel Architectures and Compilation Techniques* (2001).
[37] Jalili, M., Arjomand, M., and Azad, H. S. A Reliable 3D MLC PCM Architecture with Resistance Drift Predictor. In *Dependable Systems and Networks* (2014).
[38] Jalili, M., and Sarbazi-Azad, H. Captopril: Reducing the Pressure of Bit Flips on Hot Locations in Non-Volatile Main Memories. In *Design, Automation and Test in Europe* (2016).
[39] Jalili, M., and Sarbazi-Azad, H. Tolerating More Hard Errors in MLC PCMs Using Compression. In *International Conference on Computer Design* (2016).
[40] Joo, Y., Niu, D., Dong, X., Sun, G., Chang, N., and Xie, Y. Energy- and Endurance-Aware Design of Phase Change Memory Caches. In *Design, Automation and Test in Europe* (2010).
[41] Khan, S., Wilkerson, C., Wang, Z., Alameldeen, A. R., Lee, D., and Mutlu, O. Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content. In *International Symposium on Microarchitecture* (2017).
[42] Kim, K. Future Memory Technology: Challenges and Opportunities. In *VLSI Technology* (2008).
[43] Klug, B., and Shimpi, A. Qualcomm's New Snapdragon S4. Available at <http://www.anandtech.com/show/4940/qualcomm-new-snapdragon-s4-msm8960-krait-architecture>.
[44] Kültürsay, E., Kandemir, M., Sivasubramaniam, A., and Mutlu, O. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *International Symposium on Performance Analysis of Systems and Software* (2013).
[45] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. Architecting Phase Change Memory As a Scalable

Dram Alternative. In *International Symposium on Computer Architecture* (2009).
[46] Lee, B. C., Zhou, P., Yang, J., Zhang, Y., Zhao, B., Ipek, E., Mutlu, O., and Burger, D. Phase-change technology and the future of main memory. *IEEE Micro 30*, 1 (2010), 143–143.
[47] Lee, H. G., Baek, S., Nicopoulos, C., and Kim, J. An Energy- and Performance-Aware DRAM Cache Architecture for Hybrid DRAM/PCM Main Memory Systems. In *International Conference on Computer Design* (2011).
[48] Lefurgy, C., Rajamani, K., Rawson, F., Felter, W., Kistler, M., and Keller, T. W. Energy Management for Commercial Servers. *IEEE Computer* (2003).
[49] Lotfi-Kamran, P., Grot, B., Ferdman, M., Volos, S., Kocberber, O., Picorel, J., Adileh, A., Jevdjic, D., Idgunji, S., Ozer, E., and Falsafi, B. Scale-Out Processors. In *International Symposium on Computer Architecture* (2012).
[50] Mandelman, J. A., Dennard, R. H., Bronner, G. B., DeBrosse, J. K., Divakaruni, R., Li, Y., and Radens, C. J. Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM). *IBM Journal of Research and Development* (2002).
[51] Meena, J. S., Sze, S. M., Chand, U., and Tseng, T.-Y. Overview of Emerging Nonvolatile Memory Technologies. *Nanoscale Research Letters* (2014).
[52] Meisner, D., Gold, B. T., and Wenisch, T. F. PowerNap: Eliminating Server Idle Power. In *Architectural Support for Programming Languages and Operating Systems* (2009).
[53] Mirhosseini, A., Agrawal, A., and Torrellas, J. Survive: Pointer-based In-DRAM Incremental Checkpointing for Low-Cost Data Persistence and Rollback-Recovery. *IEEE Computer Architecture Letters* (2017).
[54] Mitzenmacher, M. Some Open Questions Related to Cuckoo Hashing. *ESA* (2009).
[55] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *International Symposium on Microarchitecture* (2007).
[56] Pagh, R., and Rodler, F. F. Cuckoo Hashing. *Journal of Algorithms* (2004).
[57] Pan, A., and Pai, V. S. Imbalanced Cache Partitioning for Balanced Data-Parallel Programs. In *International Symposium on Microarchitecture* (2013).
[58] Panigrahy, R. Efficient Hashing with Lookups in Two Memory Accesses. In *Symposium on Discrete Algorithms* (2005).
[59] Qureshi, M. K., Franceschini, M. M., Lastras-Montaño, L. A., and Karidis, J. P. Morphable Memory System: A Robust Architecture for Exploiting Multi-Level Phase Change Memories. In *International Symposium on Computer Architecture* (2010).
[60] Qureshi, M. K., Karidis, J., Franceschini, M., Srinivasan, V., Lastras, L., and Abali, B. Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling. In *International Symposium on Microarchitecture* (2009).
[61] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *International Symposium on Computer Architecture* (2009).
[62] Raoux, S., Burr, G. W., Breitwisch, M. J., Rettner, C. T., Chen, Y.-C., Shelby, R. M., Salinga, M., Krebs, D., Chen, S.-H., Lung, H.-L., et al. Phase-Change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development* (2008).
[63] Rashidi, S., Jalili, M., and Sarbazi-Azad, H. Improving MLC PCM Performance Through Relaxed Write and Read for Intermediate Resistance Levels. *ACM Transaction on Architecture and Code Optimization* (2018).
[64] Rogers, B. M., Krishna, A., Bell, G. B., Vu, K., Jiang, X., and Solihin, Y. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *International Symposium on Computer Architecture* (2009).
[65] Rosenfeld, P., Cooper-Balis, E., and Jacob, B. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* (2011).
[66] Sanchez, D., and Kozyrakis, C. The ZCache: Decoupling Ways and Associativity. In *International Symposium on Microarchitecture* (2010).
[67] Sanchez, D., and Kozyrakis, C. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of International Symposium on Computer Architecture* (2011).
[68] Sanchez, D., and Kozyrakis, C. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *High-Performance Computer Architecture* (2012).
[69] Sanchez, D., Yen, L., Hill, M. D., and Sankaralingam, K. Implementing Signatures for Transactional

Memory. In *International Symposium on Microarchitecture* (2007).
[70] Schroeder, B., and Harchol-Balter, M. Evaluation of Task Assignment Policies for Supercomputing Servers: The Case for Load Unbalancing and Fairness. *Cluster Computing* (2004).
[71] Servalli, G. A 45nm Generation Phase Change Memory Technology. In *International Electron Devices Meeting* (2009).
[72] Seznec, A. A Case for Two-Way Skewed-Associative Caches. In *International Symposium on Computer Architecture* (1993).
[73] Shao, Z., and Chang, Y.-H. Non-Volatile Memory (NVM) Technologies. *Journal of Systems Architecture* (2016).
[74] Spjuth, M., Karlsson, M., and Hagersten, E. Skewed Caches from a Low-Power Perspective. In *Conference on Computing Frontiers* (2005).
[75] Srikantaiah, S., Kandemir, M., and Wang, Q. SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors. In *International Symposium on Microarchitecture* (2009).
[76] Stan, M. R., and Burleson, W. P. Bus-Invert Coding for Low-Power I/O. *IEEE Transactions on Very Large Scale Integration 3*, 1 (1995).
[77] Sundararajan, K. T., Porpodas, V., Jones, T. M., Topham, N. P., and Franke, B. Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMPs. In *High-Performance Computer Architecture* (2012).
[78] Vakil-Ghahani, A., Mahdizadeh-Shahri, S., Lotfi-Namin, M.-R., Bakhshalipour, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. Cache Replacement Policy Based on Expected Hit Count. *IEEE Computer Architecture Letters* (2018).
[79] Wang, Z., Shan, S., Cao, T., Gu, J., Xu, Y., Mu, S., Xie, Y., and Jiménez, D. A. WADE: Writeback-Aware Dynamic Cache Management for NVM-Based Main Memory System. *ACM Transactions on Architecture and Code Optimization* (2013).
[80] Wei, W., Jiang, D., Xiong, J., and Chen, M. HAP: Hybrid-Memory-Aware Partition in Shared Last-Level Cache. *ACM Transactions on Architecture and Code Optimization* (2017).
[81] Xie, Y., and Loh, G. H. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *International Symposium on Computer Architecture* (2009).
[82] Zhang, D., Ju, L., Zhao, M., Gao, X., and Jia, Z. Write-Back Aware Shared Last-Level Cache Management for Hybrid Main Memory. In *Design Automation Conference* (2016).
[83] Zhang, L., Neely, B., Franklin, D., Strukov, D., Xie, Y., and Chong, F. T. Mellow Writes: Extending Lifetime in Resistive Memories Through Selective Slow Write Backs. In *International Symposium on Computer Architecture* (2016).
[84] Zhou, M., Du, Y., Childers, B., Melhem, R., and Mossé, D. Writeback-Aware Partitioning and Replacement for Last-level Caches in Phase Change Main Memory Systems. *ACM Transactions on Architecture and Code Optimization* (2012).
[85] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *International Symposium on Computer Architecture* (2009).