공학박사학위논문

Write Avoidance Schemes for Non-Volatile Memory based Last-Level Cache

비휘발성 메모리 기반의 최종 레벨 캐시를 위한 쓰기 회피 기법

2016년 2월

서울대학교 대학원
전기·컴퓨터공학부
최주희

ABSTRACT

Non-volatile memory (NVM) is considered to be a promising memory technology for last-level caches (LLC) due to its low leakage of power and high storage density. However, NVM has some drawbacks including high dynamic energy when modifying NVM cells, long latency for write operations, and limited write endurance. To overcome these problems, the thesis focuses on two approaches: cache coherence and NVM capacity management policy for hybrid cache architecture (HCA).

First, we review existing cache coherence protocols under the condition of NVM-based LLCs. Our analysis reveals that the LLCs perform unnecessary write operations because legacy protocols have very pay little attention to reducing the number of write accesses to the LLC. Therefore, a write avoidance cache coherence protocol (WACC) is proposed to reduce the number of write operations to the LLC.

In addition, novel HCA schemes are proposed to efficiently utilize SRAM in the thesis. Previous studies on HCA have concentrated on detecting write-intensive blocks and placing them into the SRAM ways. However, unlike other studies, a dynamic way adjusting algorithm (DWA) and a linefill-aware cache partitioning (LCP) calculate the optimal size of NVM ways and SRAM ways in order to minimize the NVM write counts and assigning the corresponding number of NVM ways and SRAM ways to cores.

The simulation results show that WACC achieves a 13.2% reduction in the dynamic energy consumption. For HCA schemes, the dynamic energy consumption of DWA and LCP is reduced by 26.9% and 37.2%, respectively.

Index Terms : Cache memories, Emerging technologies, Heterogeneous (hybrid) memory systems , Low-power design, Cache coherence, Cache partitioning

Student Number : 2012-30234

CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose of the thesis

The purpose of the thesis is to reduce the write counts of LLC to overcome drawbacks of NVM. To this end, three schemes are proposed in the thesis: write avoidance cache coherence protocol (WACC), dynamic way adjusting scheme (DWA), and linefill-aware cache partitioning (LCP).

Non-volatile memory (NVM) has been investigated as a resource to replace volatile memories such as SRAM or DRAM since their tendency to waste energy has grown to a substantial portion of total energy consumption [1, 2, 3, 4, 5, 6]. With conventional memory, static power is dissipated by transistors even when they make no switching. On the contrary, NVM adopts its own material as memory storage, instead of an electric charge, which limits leakage power dissipation.

However, there are some drawbacks to be considered when employing NVM as last level cache (LLC) directly: inefficient write operations and limited write endurance. Changing values in NVM requires long operating time and high level current. Thus, write operations generate long latency and

high dynamic energy consumption in the NVM cache system. Moreover, an NVM cell is worn out after a limited number of writing. Therefore, the lifetime of the NVM based cache is shorter than that of the SRAM cache due to the write limitation.

To overcome these drawbacks, the thesis introduces a new cache coherence protocol to reduce the write operations of the LLC [7]. The block data of the LLC is updated only if the cache block is written-back from a private cache, which leads to avoiding useless write operations in the LLC.

In addition, it is found that the previous researchers have overlooked that the capacity of NVM is also one of important factors affecting the number of write accesses to NVM. This discovery leads to the necessity of NVM capacity management policy such that the size of NVM is dynamically adjusted according to the demand of applications. To implement the idea, we propose a dynamic way adjusting (DWA) algorithm which dynamically monitors the optimal number of NVM ways using the stack property and disabling the unnecessary NVM ways [8].

Finally, the thesis proposes a cache partitioning scheme called linefill-aware cache partitioning (LCP) mechanism, taking into account the NVM linefill counts as well as the NVM write hit counts during cache partitioning. Most previous works have concentrated on managing write-intensive blocks by allocation these blocks to SRAM to reduce the number of the write operations to NVM. However, those schemes have not considered that reducing the number of linefill operations to NVM is important to reduce the

total number of write operations to NVM. To overcome this weakness, an algorithm for cache partitioning of LCP considers the NVM linefill counts.

The proposed schemes are simulated with the gem5 simulator [9] for WACC and macsim [10] for DWA and LCP. We used the PARSEC benchmark suite [11] for evaluating WACC and SPEC CINT2006 and SPEC CFP2006 of the SPEC CPU2006 benchmark suite [12] for DWA and LCP. The experitmental results show that WACC achieves a 13.2% reduction in the dynamic energy consumption. For HCA schemes, the dynamic energy consumption of DWA and LCP are reduced by 26.9% and 37.2%, respectively.

1.2 Background

According to the material used in NVM, several kinds of NVM [1, 2, 3, 4, 5, 6] have been introduced such as spin-torque transfer RAM (STT-RAM), phase change memory (PCM), and ferroelectric RAM (FeRAM). Even though their compositions are different, all NVM can be considered similar in terms of cache architecture. First, they sustain their information without electric power; this is the reason why they called non-volatile memory. Their main advantage comes from their characteristics of extremely low leakage power consumption. In addition, their density is much higher than that of SRAM even that of DRAM for some kinds of NVM. Table 1 shows comparison of parameters of SRAM and STT-RAM obtained from the modified CACTI [13, 14] in previous work [15].

Table 1: Comparison of area, latency, and energy [15].

Parameters	SRAM	STT-RAM	PCM
Cache Size	128KB	512KB	2MB
Area(mm^2)	3.262	3.30	3.85
Read Latency(ns)	2.252	2.318	4.636
Write Latency(ns)	2.264	11.024	23.180
Read Energy(nJ)	0.895	0.858	1.732
Write Energy(nJ)	0.797	4.997	3.475
Static power(80 °C)(W)	1.131	0.016	0.031
Write Endurance	10^{16}	$4 * 10^{12}$	10^{9}

1.3 Motivation

The thesis focuses on two approaches such as cache coherence protocol and NVM capacity management policy for hybrid cache architecture (HCA). For cache coherence protocol, the existing studies have not concentrated on reducing the write operations because it does not matter in the SRAM-based LLC. Since there is no drawback of write operation compared to read operation, the number of write access is not taken into account. However, reducing the write operations is an important issue in NVM-based LLC. The dynamic energy consumption largely depends on the write operations, because the dynamic energy of write operation is greater than that of read operation. Moreover, the lifetime is inversely proportional to the number of write access. Therefore, a new protocol for NVM to minimize the write operations is needed.

In addition, it is found that there is a relationship between the capacity of NVM in HCA and the write counts of NVM. The analysis implies the necessity of efficient NVM capacity management policy: the HCA dynamically manages the capacity of NVM according to the demand of applications. As the first step of realizing this idea, we use the number of active NVM ways in a set as the measure of the capacity of NVM. The capacity of NVM is expressed by the number of currently available NVM ways and the demand of NVM is converted to the requested number of NVM ways.

1.4 Contributions

Firstly, the thesis introduces a new cache coherence protocol for NVM to decrease the number of write access to the LLC [7]. In our protocol, the data array of the LLC is not updated during the linefill operation, while the tag array is changed to maintain the inclusion property. The data array is modified only when the cache block is written-back from the private cache. Our protocol reduces the number of write access to the LLC; thus, the dynamic energy consumption is reduced and the lifetime is enhanced in our protocol.

- We investigate the existing cache coherence protocol for NVM and reveal the drawback of them.
- We propose a cache coherence protocol for NVM, which avoids unnecessary write operation in the LLC based on the analysis.

- We present experimental results of a write avoidance coherence protocol with number of write accesses to LLC, dynamic energy consumption, and lifetime.

In addition, hybrid cache architecture (HCA) has been proposed to overcome these limitations of NVM [16, 17, 18, 19, 20]. Most previous works have concentrated on managing write-intensive blocks by storing these blocks to SRAM to reduce the number of the write operations to NVM. However, we show the concept of NVM capacity management policy for reducing the number of write accesses to NVM and propose a dynamic way adjusting algorithm [8]. It dynamically resizes the number of active NVM ways to improve the dynamic energy consumption and the lifetime. To adjust the number of NVM ways, the maximum stack distance is monitored and rearranging the replaceable NVM ways is regularly performed.

- We investigate the relationship between the number of write operations and the capacity of NVM in HCA by performing both analysis based on the devised analytical model and experiments.
- We find out that decreasing the number of active NVM ways can be beneficial to reduce the number of write accesses to NVM ways, only if it does not increase the miss rate significantly.
- We propose a dynamic way adjusting algorithm (DWA) to find the optimal number of NVM ways and dynamically adjust active NVM ways without physical change of the cache.

- We conduct a simulation to evaluate the effectiveness of the proposed policy in terms of the reduction in the write counts of NVM, the decrement of the dynamic energy consumption, the lifetime extension, and the variation of the miss rate.

While previous studies focus on reducing NVM write counts due to the write-intensive blocks, they have not considered the NVM write operation is also occurred by linefill operation to NVM. Reducing the NVM write counts due to linefill operations are also very important for minimizing overall NVM write counts in chip-multiprocessor (CMP) environments. The thesis proposes a cache partitioning scheme called a linefill-aware cache partitioning (LCP) mechanism, taking into account the NVM linefill counts as well as the NVM write hit counts during cache partitioning.

- We propose a linefill-aware cache partitioning scheme (LCP) for HCA, which takes into account the reduction in the number of linefill operations to NVM to minimize the NVM write counts.
- We devise new metrics for LCP: write counts change (ΔW) and NVM write counts change ($\Delta NVMW$), which are based on the miss counts change (ΔM).
- We propose an algorithm to make partitions by predicting metrics according to the change of the number of allocated ways for each core.

Table 2: Summary of proposed schemes.

Scheme	Aim	Description
Write avoidance cache coherence protocol (WACC)	Reduction in the number of write access to LLC	The data array is modified only when the cache block is written-back from the private cache.
Dynamic way adjusting algorithm (DWA)	Reduction in the number of write access to NVM	The number of active NVM ways is dynamically resized.
Linefill-aware cache partitioning (LCP)	Reduction in the number of write access to NVM and increase in the hit rate of LLC	The NVM linefill counts is taken into account as well as the NVM write hit counts during cache partitioning.

- We present experimental results of LCP with the prediction accuracy, number of write accesses to NVM, miss rates, performance for multicore workloads, and dynamic energy consumption.

The schemes in the thesis are summarized in Table 2.

1.5 Organization of the thesis

The rest of the thesis is organized as follows. Chapter 2 provides related work about NVM. In Chapter 3, a new cache coherence protocol for NVM called a write avoidance cache coherence protocol is proposed. Chapter 4 describes NVM capacity management policy for HCA. The conclusion is given in Chapter 5.

Chapter 2

Related work

2.1 Hybrid cache architecture

Researchers have merged two types of memory into a single cache system, which is called HCA, to reduce the number of write access to NVM to alleviate the shortcomings of it especially related to a write operation [16, 17, 18, 19, 21]. As described in above section, the shortcomings of NVM come from write operation of NVM. In other terms, the number of write access to NVM is the most important factor for both the dynamic energy consumption and the lifetime. Since the write energy consumption of NVM is much larger than read energy of NVM or dynamic energy of SRAM, the write energy consumption of NVM is dominant for the total dynamic energy consumption. Furthermore, the lifetime is proportional to the number of write access to NVM cells. Therefore, reducing the number of write access to NVM is one of the most important methods to mitigate the drawbacks of NVM. For this reason, a small number of SRAM ways are used to accommodate heavily written blocks in the hybrid cache system as depicted in Figure 1.

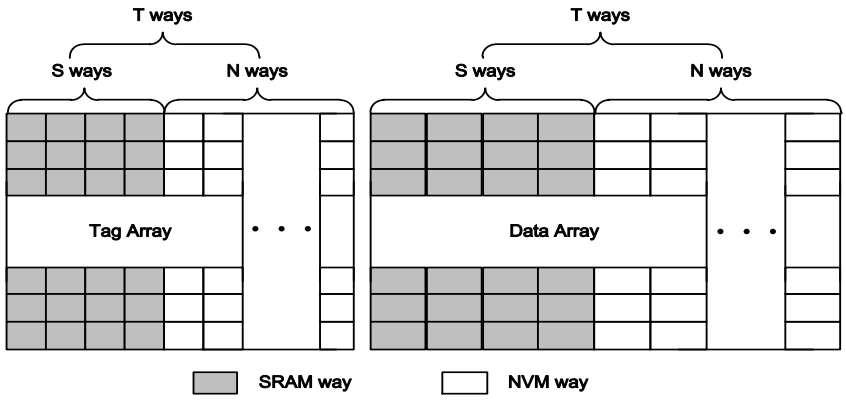


Figure 1: Basic structure of hybrid cache architecture (HCA).

First, swapping or migration schemes between SRAM and NVM in a hybrid cache system were proposed. Wi et al. introduced the region based cache architecture in [16]. They divided a single level of cache into two regions: read region which consists of STT-RAM and write region which consists of SRAM. If a block is predicted as write-intensive, the block is placed or swapped to the write region. Besides the schemes, merging set schemes were proposed [17] and [18]. The authors noticed that non-uniformity of write operations among sets. While some sets are frequently utilized, other sets receive relatively small requests. Therefore, write-intensive blocks in the highly utilized sets are forwarded to the idle sets. In addition, a predictor was equipped to find the correlation between write intensive blocks and addresses of trigger instructions [19]. In summary, existing policies focused on placing write-intensive blocks into the SRAM.

2.1.1 Write intensity prediction studies

Almost all papers on HCA have focused on devising methods to identify write-intensive blocks and place them to SRAM ways. Wi et al. suggested the region based cache architecture in [16]. They separated a single level of cache into two regions: read and write regions. The read region is prepared for non-write-intensive blocks composed of NVM, while the write region is composed of SRAM for write-intensive blocks. When a block is considered as write-intensive, the block is migrated or placed to the write region. On top of these schemes, combining set schemes were proposed [17, 22, 23]. This insight came from the fact that the write operations among sets are not uniformly distributed. While some sets receive relatively small write requests, other sets are highly utilized. To take advantage of these characteristics, some blocks in the frequently utilized sets are moved to the other sets. To elaborate the prediction algorithm, Quan et al. introduced a prediction table [18] containing the history of the write requests of the LLC. Another prediction table is proposed to store the value of combining addresses of the blocks and program counter of instructions [19]. What distinguishes these works from our scheme is that they have not focused on the CMP environment.

2.1.2 Static approaches

Various methods utilizing the compiler have been proposed. Chen et al. [24] proposed a scheme in which the compiler provides hints to find the write-

intensive block and the hardware is modified to correct the hints. Software dispatch was presented to detect write reuse patterns in [25]. In addition, the migration-intensive blocks are loaded into the SRAM region with the compiler assistance in [26] to mitigate the burden of migration blocks. Moreover, a loop retiming framework was proposed for loops with intensive data array operations to relieve the migration overhead [27]. Another study improves the read performance and energy efficiency guided by the analysis of read bottlenecks [28]. They focused on the recompilation or profiling schemes, while our proposed mechanism modifies the hardware structure and logics.

2.1.3 Hybrid cache architecture for main memory

As the write endurance problem has become important for the main memory, which is based on NVM, many methods have been proposed to prolong its lifetime. They have employed DRAM as a cache for NVM. Qureshi et al. firstly suggested the concept of a small DRAM cache to overcome the latency gap between DRAM and PCM [29]. The mechanism exploits both the short latency of DRAM and the large capacity of PCM by preventing unnecessary access to PCM. They also have shown advanced approaches such as write cancellation and write pausing policies [30] to mitigate the long read access time due to the long write latency. Meanwhile, a scheme proposed in Meza et al. [31] stores the metadata for the last accessed rows into a small buffer to manage the difficulty of fine-granularity DRAM caches. It is found that row buffer misses generate long latencies, and a policy is devised to exploit this observation [32]. They predict the data incurring a row

buffer miss and store it into a DRAM buffer by investigating the row buffer miss counts in PCM. Writeback-aware partitioning offers a new perspective on cache partitioning, taking into account the writeback information [33]. It is innovative in regard to reducing the amount of write access to the PCM main memory by managing the cache partition.

Another approach for the hybrid cache architecture is based on OS support. For PDRAM [34], the researchers introduced a hybrid solution related to software as well as hardware to extend the lifetime of the PCM pages. They modified the OS-level page manager and added a small device to contain the number of write requests for PCM at a page level granularity. Ferreira et al. [35] also inserted a DRAM buffer to decrease the number of read and write requests to PCM via page partitioning. Zhang and Li [36] improved the write endurance and reduced write latency of PCM by exploiting the workload characteristics as an aspect of an OS level paging. New page migration schemes were proposed to track read-bound access NVM pages [37].

All schemes described above are based on the physical features of DRAM or characteristics of OS, thus they are inadequate applied to the SRAM and NVM based LLC, which is the target of the thesis.

2.2 Cache partitioning schemes

To improve the cache efficiency, several methods using stack property have been proposed. The number of cache hit counts of LRU position is monitored to calculate the cache utility of each application or core. Based on the information, the cache is partitioned to minimize the number of total cache misses. Suh et al. [38] dynamically partitioned the LLC and assigned the guided number of cache ways to each application. Even though it successfully raised the cache utility, there was a problem in that the utility information of an application was affected by other applications. To avoid this drawback, Qureshi and Patt [39] introduced a separate utility monitor, which counts the number of hits without interference by other applications. An adaptive placement policy [40] was proposed to load a new block into the local or remote cache for enhancing the efficiency of cache based on stack distance profiling. In addition, compliers used the information to predict the memory behavior of the application [41]. For a real-time system, Liu and Zhang [42] suggested the compilation technique, which improves the worst case data cache performance using the stack distance approach. Most papers on cache partitioning assumed that the LLC consists of SRAM only, hence they do not consider the NVM write counts in their schemes.

Chapter 3

Write avoidance cache coherence protocol

3.1 Limitation of existing cache coherence protocol

We review the legacy cache coherence protocols to get a new insight to reduce the write operations. There are useless write operations in the existing protocol. Generally, memory systems of CMPs are composed of a shared LLC and several private caches which are dedicated to cores [43]. In addition, the cache block is divided into two arrays: tag array and data array. Tag array stores tag bits and cache coherence state, while data array stores block data. When a linefill operation occurs, the requested block data is written to the data array, and the tag bits and cache coherence state are updated to the tag array. Then, the cache block is forwarded and linefilled to the private cache. When a core tries to modify the cache block in the private cache, an invalidation signal is sent to the shared LLC and other private caches to maintain the cache coherence. Thus, the previous write access to the LLC during the linefill operation is considered as the useless write operation, if the cache block in the LLC has been never used until it is invalidated.

Figure 2 illustrates an example of write inefficiency in widely used cache coherence protocols such as MESI or MOESI [44]. In the example, we assume that a core reads and writes a block data of the PC (Private Cache) 1. Table 3 lists the cache states in the figure and their descriptions. When the core tries to read the block data, since the PC1 has no valid block data, the cache controller sends the request for the block data to the LLC.

However, the LLC also has no valid copy; thus, the request is sent to the external sources such as the main memory or other chipsets. When the block data "ABCD" is arrived at the LLC, it is written into the LLC and the state of the LLC is changed to S state, which means the cache block is valid and other private caches may have the same cache block. Then, the block data "ABCD" is forwarded to the PC1.

When the block data is received in the PC1, it is written into the PC1 and the state of the PC1 is changed to E state. After the linefill operation is completed, if the core tries to modify the block data "ABCD" to "EFEF", an invalidation request is sent to the LLC to maintain cache coherence. The purpose of the invalidation request is indicating that the block data of the PC1 is modified and the cache block in the LLC should be invalidated. If the block data "ABCD" in the LLC has not been used until it is invalidated, writing the block data "ABCD" to the LLC during the linefill operation was a useless write operation.

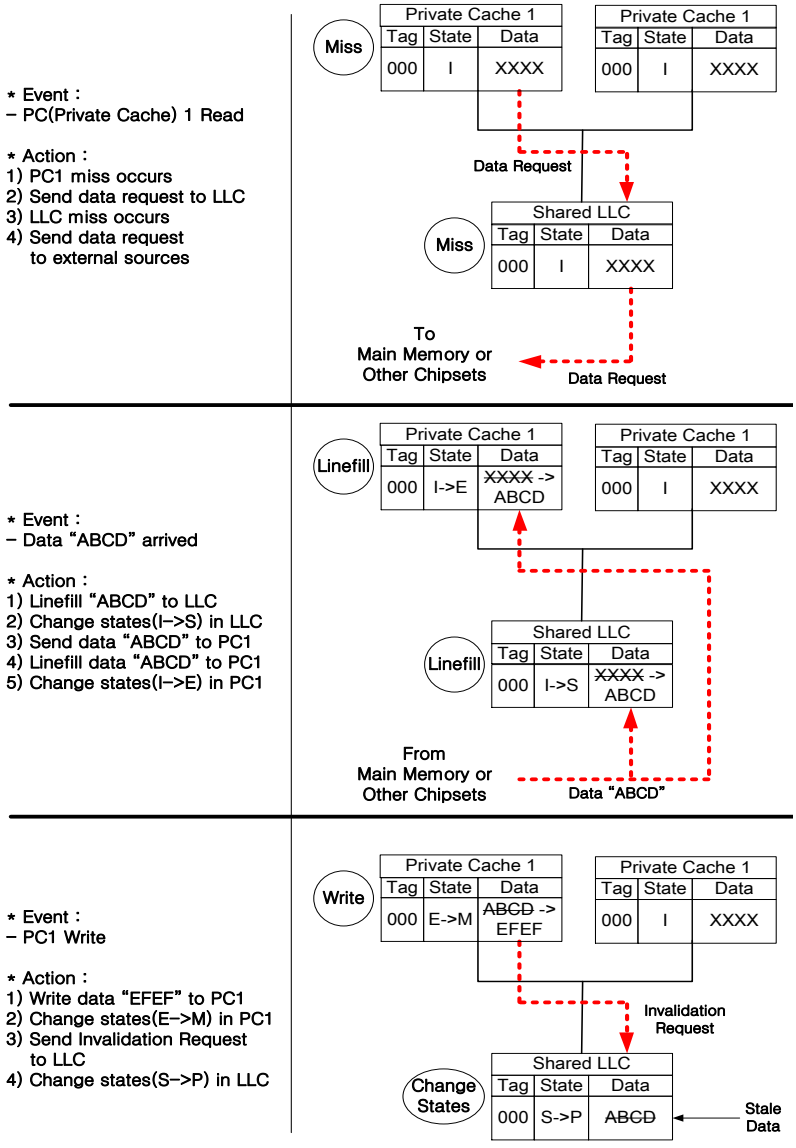


Figure 2: Conventional cache coherence protocol.

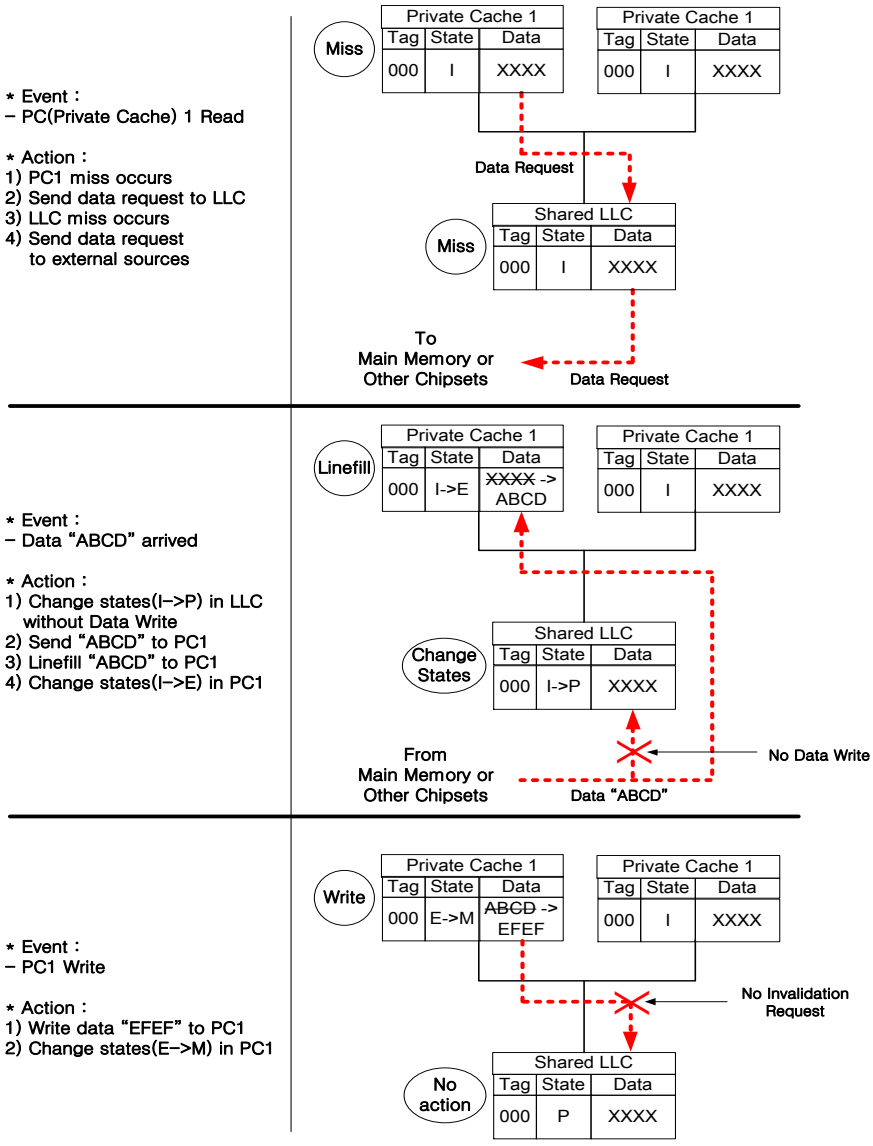


Figure 3: Write avoidance cache coherence protocol (WACC).

Table 3: States and descriptions for write avoidance cache coherence protocol (WACC).

State	Description
I(nvalid)	The cache block is invalid
S(hared)	The cache block has valid block data and other private caches may have valid copy.
E(xclusive)	The cache block has valid block data with exclusive permission and other caches have no valid copy.
M(odified)	The cache block has valid and modified block data. Other caches have no valid copy. This state appears in the private cache only.
P(rivate cache)	The cache block in the LLC has no valid block data, but more than one of the private caches has valid block data. This state appears in the LLC only.

* P state is introduced due to keeping the inclusion property. Modern multiprocessors have employed the inclusive LLC to filter the cache coherence traffic from other chipset or the main memory. Thus, it is needed that a state represents one of the private caches has valid data even the LLC has no valid data.

3.2 Write avoidance cache coherence protocol

To deal with this problem, we suggest a new cache coherence protocol which is called Write avoidance cache coherence (WACC) protocol. In our protocol, the block data of the cache block is not written into the LLC during the linefill operation, while the tag bits and the cache coherence state are updated. Since the block data is not placed in the LLC, one of the private caches has responsibility to provide the valid block data. The block data in the LLC is only updated when it is written-back from the private cache. The writeback is initiated only when no other private cache has the block data in WACC protocol. Therefore, we avoid useless write operation due to modifications of the block data in the private cache.

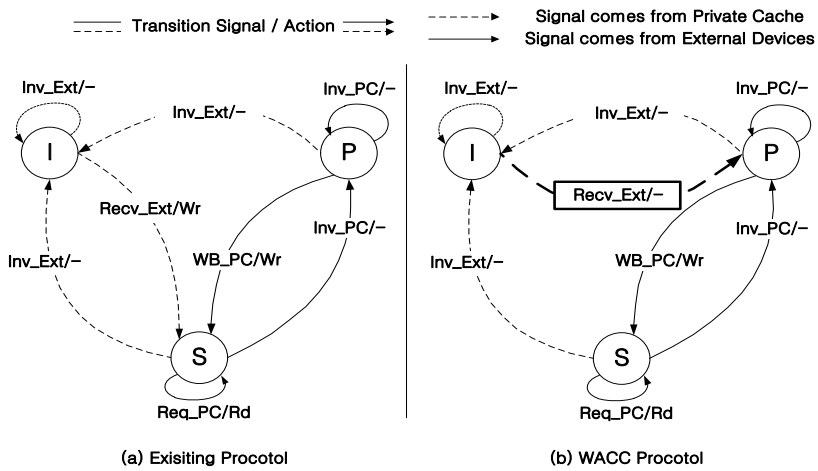


Figure 4: State transition diagrams for WACC.

Figure 3 shows an example of WACC protocol. Unlike the conventional protocols, when the block data ABCD is arrived at the LLC, it is not written to the LLC. Instead, the state is changed to P state and the block data is forwarded to the PC1. When the PC1 is modified to EFEF, there is no need to send an invalidation request to the LLC for the block data ABCD is not written to the LLC. Therefore, one write operation of the LLC and one request for cache coherence is decreased compared to the baseline protocols.

We compare a simple version of the existing MOESI protocol with its modified protocol in Figure 4. Table 4 shows the coherence signals and actions. The transition signal is divided into two parts: {signal}_{source} and the action indicates the operation of the data array. For example, WB_PC/Wr means that if the block is P state and receives the WB signal from a private cache, the block data is written to the data array.

Table 4: Signals/actions and descriptions.

Signal	Description
Inv	Invalidate the cache block if it is valid. This signal is generated when another device tries to modify the block data.
Recv	Provide the block data in the cache block. This signal is generated when a cache hit occurs.
Req	Request the block data for read operation. This signal is generated when a cache miss occurs.
WB	Writeback the block data to the LLC. This signal is generated when a private cache evicts the cache block.
Action	Description
Wr	Write the block data of the received cache block into the data array.
Rd	Read the block data and provide it with the requestor.

As shown in Figure 4(a), when a new cache block is received in the LLC, the state of the cache is transition to S state and the block data is written to the data array in the existing protocol. On the contrary, the state is transition to P state instead of S state in our protocol under the same condition. Furthermore, the write operation is omitted as shown in Figure 4(b). This is because the block data is forwarded without write access to the data array in WACC protocol.

Another point to be considered is that the protocol of the private cache should be changed. The writeback operation is initiated if the cache block in the private cache is modified and evicted in the existing protocols. However, the cache block should be written-back to the LLC in WACC protocol when it is evicted in the private cache regardless of whether the cache block is dirty or not.

Chapter 4

NVM capacity management policy for hybrid cache architecture

4.1 NVM capacity management policy

In this section, we propose two schemes for NVM capacity management policy. First, we introduce a dynamic way adjusting algorithm (DWA) that monitors the optimal number of NVM ways and dynamically adjust the number of active NVM ways [8]. In addition, we also propose a linefill-aware cache partitioning scheme (LCP) to save the dynamic energy consumption by efficiently allocating SRAM ways and NVM ways to cores.

The DWA keeps track of maximum stack distance (MSD), which means the minimum number of ways to maintain the miss rate. If the number of the current active NVM ways is not the optimal value, it is adjusted according to the MSD. In addition, an efficient method to disable NVM ways is required because it is impossible that NVM ways are physically added or removed during execution. Thus, the DWA prevents deactivated NVM ways from victim selection. A newly fetched block is prohibited to be loaded into the disabled NVM ways, which has the effect of virtually deactivating them.

The basic idea of LCP comes from cache partitioning [38, 39, 40], which has been a well-known scheme to improve the performance in CMP systems. The key idea of the cache partitioning is that all cache ways should be efficiently allocated for each application to maximize the hit rate of the LLC. They have contributed the studies of the LLC. However, it is inefficient to apply them directly into HCA because their models assume that all cache ways consist of the same memory type. Even though the cache misses are minimized by the previous cache partitioning schemes, if the linefill operations heavily occur in NVM ways, it fails to reduce the linefill counts of NVM. Therefore, LCP assigns the SRAM ways and the NVM ways to each core based on the change of the NVM linefill counts as well as the NVM write hit counts according to partitioning.

4.1.1 Concept of NVM capacity management policy

This section presents an NVM capacity management policy that resizes the number of NVM ways to fit the demand of applications. This policy comes from the observation that reducing the size of NVM usually decreases the write counts of NVM if the miss rate does not grow. The thesis will propose an analytical model and perform a simulation to verify this observation.

Cache researchers have been investigating the relationship between the size of cache and the miss rate [39]. For many programs, as the cache size grows, the miss rate becomes small. On the contrary, the miss rates of some programs are saturated or remain despite incremental growth of the cache

size. In addition, even the same program always does not require the fixed size of cache. Therefore, the number of requested ways of the cache varies during execution, and the unnecessary ways are disabled without performance degradation.

The number of write accesses to the cache is strongly coupled with the miss rate. Generally, the cache operations are divided into three categories: read hit, write hit, and linefill. Among these operations, write hits and linefill operations compose the write requests. If some read hits are changed to cache misses due to the increasing miss rate, new linefill operations occur as much as the removed read hits. This implies that the total number of write operations are increased. Alternately, if the number of cache misses is not increased, the number of write accesses to the cache remains because the hit counts and miss counts is not changed.

Assume that we minimize the number of NVM ways without generating significant extra cache misses. In that case, the write operations which originally occurred in the deactivated NVM ways are forwarded to SRAM ways or other NVM ways. If a part of write accesses is sent to SRAM ways, the number of write accesses to NVM ways is reduced. Therefore, partial deactivating NVM ways with the stable miss rate highly tends to decrease the write counts of NVM ways.

An illustration is provided in Figure 5 to aid in the understanding of this concept. There are two caches in the example. One of the caches consists of one SRAM and three NVM ways, and another cache is composed of one

SRAM and two NVM ways. The program in our example needs only three ways. For the sake of convenience, suppose that all memory references are write requests.

When the program starts, cache accesses are performed according to the sequence in Figure 5. There is no difference between the two caches in the first three accesses. However, when ”d” miss is encountered, two caches behave differently. While ”d” is placed in the fourth way in cache A, ”a” is replaced with ”d” in cache B. Writing ”d” in the second iteration, SRAM access is made instead of NVM access in cache B. As a result, the number of write to NVM ways is reduced in cache B. The linefill operation of ”d” is forwarded to a SRAM way, and thus one linefill operation and one write hit of NVM ways is reduced.

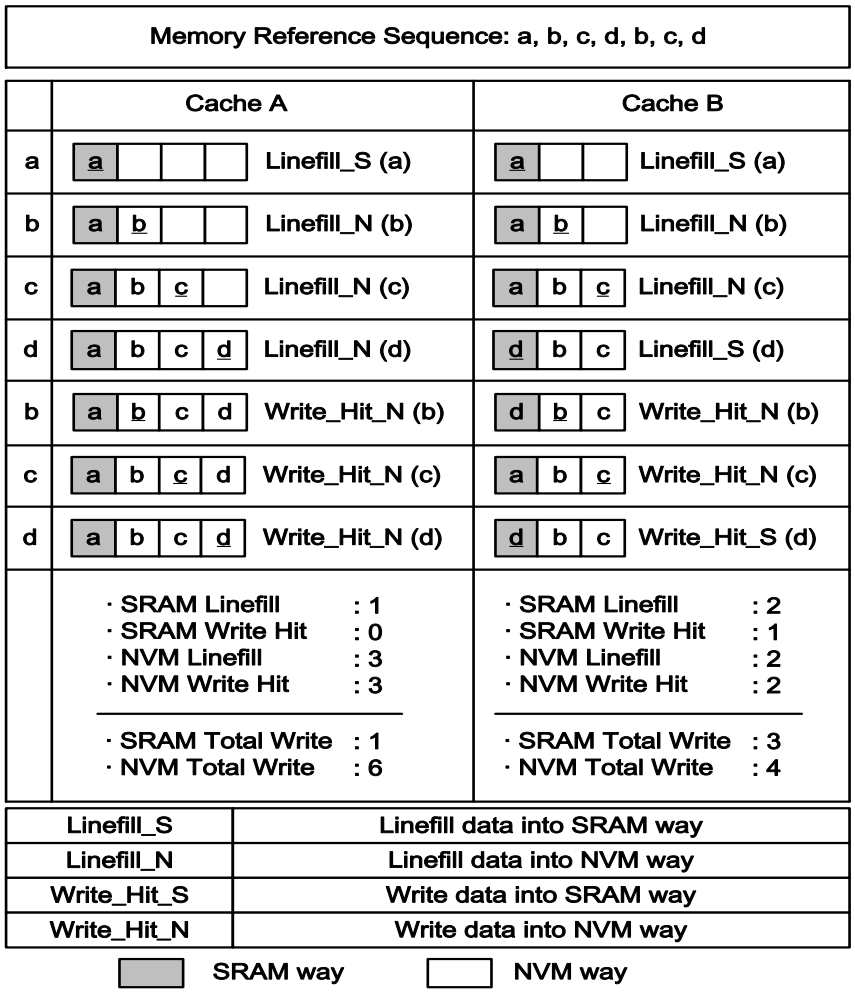


Figure 5: Example for NVM capacity management policy.

4.1.2 Feasibility of NVM capacity management policy

A metric, write intensity of a way (WI), is defined as the portion of write accesses to the way over the write accesses to all ways. It is given by

$$WI_i = \frac{W_i}{W_{total}} \ (1 \leq i \leq T) \tag{4.1}$$

where W_i is the number of write accesses to ith way and W_{total} means the number of total write accesses to the cache, while T is the number of all cache ways. This metric indicates the distribution of write requests among the ways. If all ways have the same write intensity, the write requests are evenly distributed. Unless, write operations occur more frequently in some ways which have higher value than other ways.

Since the total number of write counts is calculated by summation of write counts of each way, it is expressed as

$$W_{total} = \sum_{i=1}^{T} W_i \tag{4.2}$$

The above equation is expressed as form of WI as follows

$$W_{total} = \sum_{i=1}^{T} (WI_i * W_{total})$$

$$= W_{total} * \sum_{i=1}^{T} WI_i \tag{4.3}$$

We rewrite the above equation as form of SRAM ways and NVM ways, and it is given by

$$W_{total} = W_{sram} + W_{nvm}$$

$$= W_{total} * \sum_{i=1}^{S} WI_i + W_{total} * \sum_{i=S+1}^{T} WI_i \tag{4.4}$$

$$W_{sram} = W_{total} * \sum_{i=1}^{S} WI_i \tag{4.5}$$

$$W_{nvm} = W_{total} * \sum_{i=S+1}^{S+N} WI_i = W_{total} * \sum_{i=S+1}^{T} WI_i \tag{4.6}$$

where S is the number of SRAM ways and N is the number of NVM ways, while W_{sram} means the number of write accesses to SRAM ways and W_{nvm} is the number of write accesses to NVM ways. We found that there are three factors that influence the write counts of NVM ways: the number of total counts (W_{total}), the write intensity per way (WI), and the number of NVM ways ($N = T - S$).

So far, the main strategy for reducing the number of write counts of NVM ways has been keeping average WI of NVM ways lower than that of SRAM ways. Throughout previous HCA research, WI is thought as the only important factor among the three factors. It is assumed that N is fixed and W_{total} is not significantly changed. Therefore, they have focused on minimizing WI of NVM ways by detecting write intensive blocks and placing them into SRAM ways. These approaches are successful to reduce write accesses to NVM.

Different from previous approach, we consider N as a variable instead of a constant value. When the number of NVM ways is reduced to N' ($N' < N$), W'_{total}, W'_{sram}, and W'_{nvm} are defined as the number of write accesses to the cache, SRAM ways, and NVM ways:

$$W'_{total} = W'_{sram} + W'_{nvm} \tag{4.7}$$

In addition, we define the altered number of all ways as T' ($T' = S + N' < T$), and Eq. 4.6 is transformed below:

$$W_{nvm} = W_{total} * (\sum_{i=S+1}^{T'} WI_i + \sum_{i=T'+1}^{T} WI_i)$$

$$= \sum_{i=S+1}^{T'} WI_i * W_{total} + \sum_{i=T'+1}^{T} WI_i * W_{total} \tag{4.8}$$

The second term indicates the number of write accesses to the NVM ways that will be removed. If we adjust the number of NVM ways to N', the remaining ways should absorb the write requests of the amount of second term. For simplicity, this term substitute for X and Eq. 4.6 is expressed as follows:

$$X = \sum_{i=T'+1}^{T} WI_i * W_{total} \tag{4.9}$$

$$W_{total} = W_{sram} + (W_{nvm} - X) + X \tag{4.10}$$

Hereby, we introduce a condition that the total write counts are not changed ($W'_{total} = W_{total}$). Under the condition, W'_{total} is given by

$$W'_{total} = W_{sram} + (W_{nvm} - X) + X \tag{4.11}$$

If we divide X into X_{sram} and X_{nvm} that are the write requests of the amount of forwarded to SRAM ways and NVM ways, we obtain

$$W'_{total} = W_{sram} + (W_{nvm} - X) + X_{sram} + X_{nvm}$$

$$= (W_{sram} + X_{sram}) + ((W_{nvm} - X) + X_{nvm}) \tag{4.12}$$

Because W'_{sram} and W'_{nvm} are defined as the number of write accesses to SRAM and NVM in the resized cache, they can be expressed by as following equation:

$$W'_{sram} = W_{sram} + X_{sram} \tag{4.13}$$

$$W'_{nvm} = W_{nvm} - X + X_{nvm} \tag{4.14}$$

Before advancing the discussion, we state that it is assumed that X_{sram} is greater than zero for the simplicity of the model. When the number of ways is changed, the blocks are placed differently than they were. There is a possibility that some write intensive blocks that were originally located in SRAM ways are inserted into NVM ways. In that case, X_{sram} could be zero or minus value. To avoid this problem, we adopt a policy for placing write intensive blocks into SRAM ways as presented [16] to our scheme.

Since X is summation of X_{sram} and X_{nvm}, if X_{sram} is greater than zero, X_{nvm} is given by

$$X_{nvm} < X \tag{4.15}$$

By transforming Eq. 4.14 and substitution W_{nvm} into Eq. 4.15, we obtain

$$W'_{nvm} - W_{nvm} + X < X \tag{4.16}$$

$$W'_{nvm} < W_{nvm} \tag{4.17}$$

Thus, we conclude that fewer NVM ways causes lower write requests to NVM if the miss rate does not grow.

We examined the impact of NVM capacity management on the miss rate, the total write counts, and the write accesses to NVM ways. We assume that the hybrid cache has 4 SRAM ways and 12 NVM ways and that the number of NVM ways varies from 12 to 0. The results are sorted in decreasing order by the number of NVM ways among each application. To improve the readability, we abbreviate SRAM ways to ”S” and NVM ways to ”N”. For example, 4S_2N in the figure means that 4 SRAM ways and 2 NVM ways are used during the simulation.

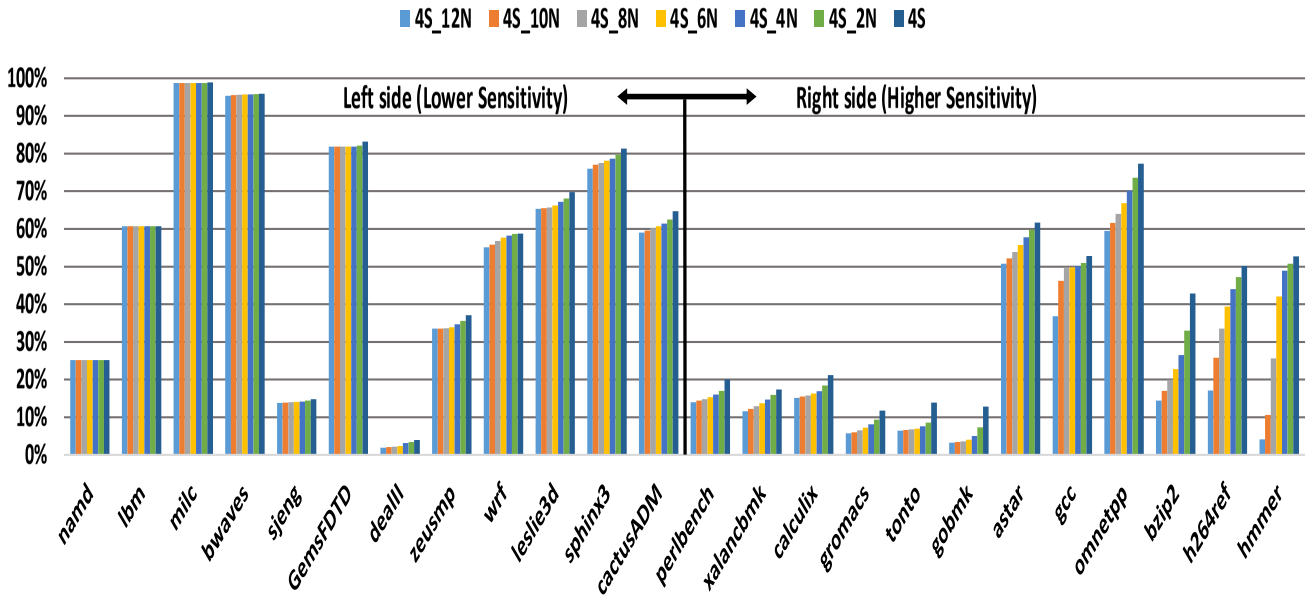


Figure 6: Miss rates with various number of NVM ways.

Figure 6 represents the miss rates with various number of NVM ways to show sensitivity of the miss rate to the size of NVM. We sort all applications by geometric standard deviation (GSD), which represents the amount of dispersion from the geometric mean. In Figure 6, the miss rates of the left applications are not less influenced by the number of NVM ways, while the right side applications are more sensitive to the number of NVM ways. The miss rates of two left most applications such as *namd* and *lbm* remain even when all NVM ways are removed. Part of NVM ways are unnecessary for some left side applications: *milc, bwaves, sjeng, GemsFDTD, dealII,* and *zeusmp*. On the contrary, the growth of the miss rates of the higher sensitive applications is large. Especially, the miss rates of *bzip2* and *h264ref* is multiplied about three times and the miss rate of *hmmer* soars to 12.8 times.

Figure 7 shows normalized write accesses to the HCA with various sizes of NVM. We find that the total write counts of the lower sensitive applications are not greatly increased, while many higher sensitive applications show rapid growth. For the left side applications, only 2.8% of average extra write operations occur. Especially, no change is detected through all sizes of NVM in *namd,lbm,* and *milc*. The number of NVM ways can be decreased to 2 without increasing write counts in *bwaves* and *GemsFDTD*. Other benchmarks such as *sjeng* and *zeusmp* have the same values when NVM ways varies from 12 to 8. On the other hand, the total write counts of the right side applications increase by 29.4% on average.

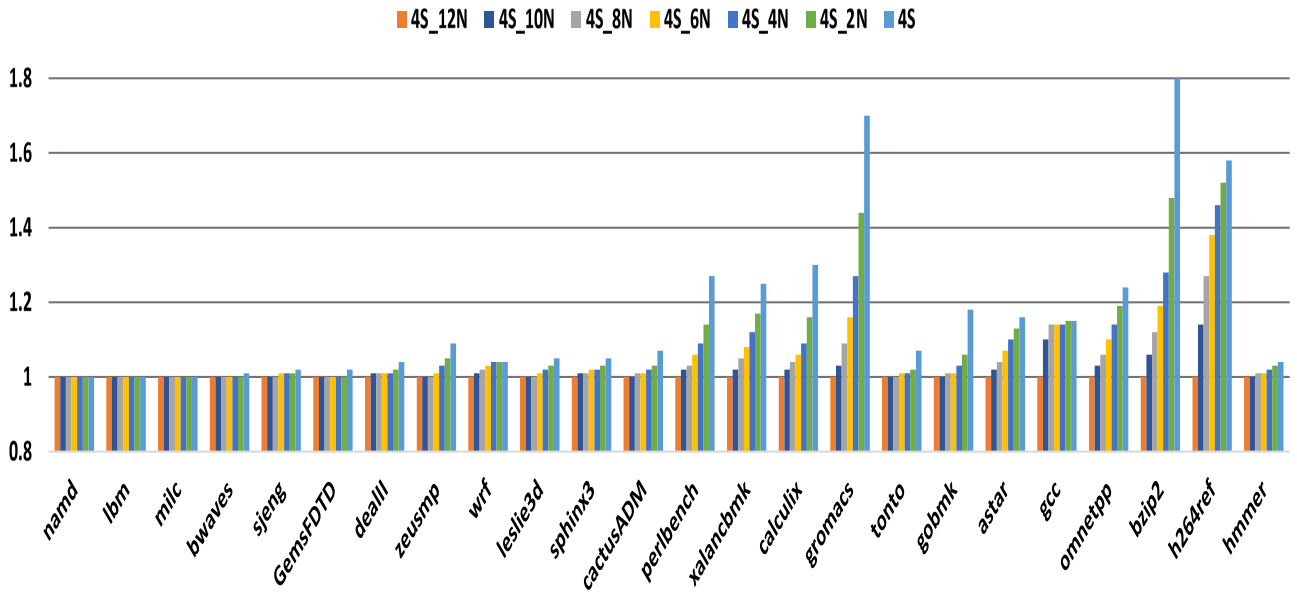


Figure 7: Normalized total write counts of HCA various number of NVM ways. 4S_12N is the standard of normalization.

The normalized write accesses to NVM ways with various number of NVM ways is depicted in Figure 8. As we expected, reducing the number of NVM ways decreases the write accesses to NVM ways in lower sensitive applications. On the other hand, the reduction in the write counts of NVM ways is not guaranteed by resizing the number of active NVM ways in higher sensitive applications. Adjusting NVM ways even results in increasing the write operations of NVM ways in *gobmk, gcc,* and *h264ref*. Some applications such as *gromacs, tonto, bzip2,* and *hmmer* show the similar pattern of the left applications, but their reduction ratios are small.

In summary, we find out that the number of write accesses to NVM ways is usually reduced if resizing the number of active NVM ways does not significantly increase the miss rate by adopting efficient NVM capacity management policy.

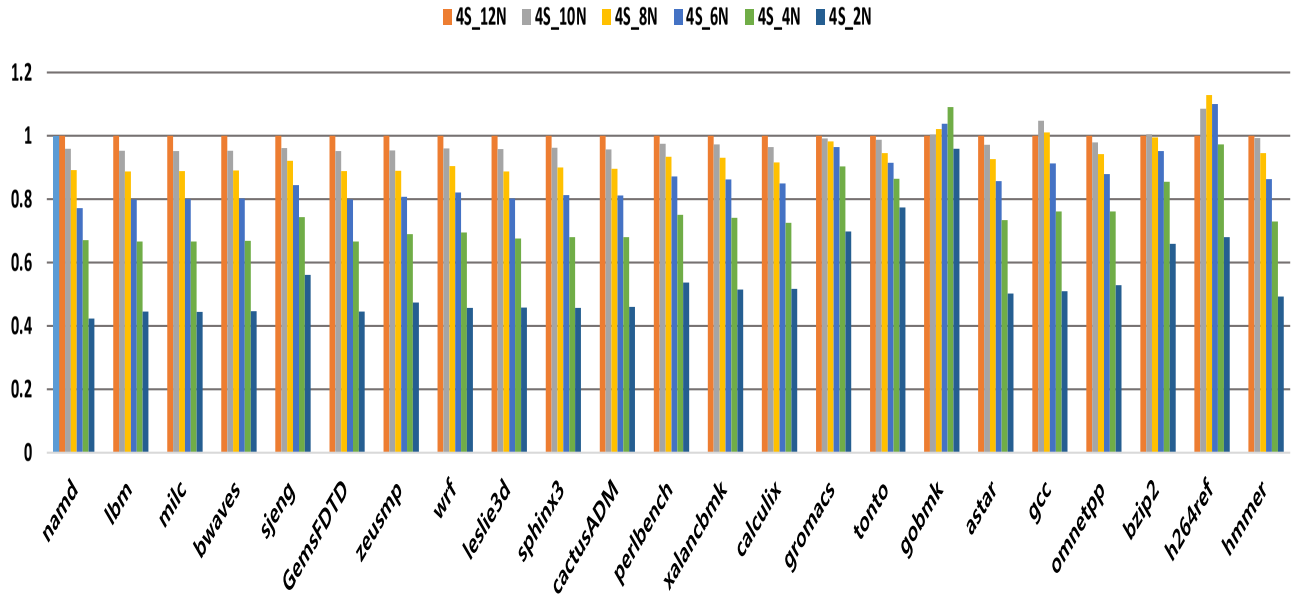


Figure 8: Normalized write counts of NVM with various number of NVM ways. 4S_12N is the standard of normalization.

4.2 Dynamic way adjusting

We propose a dynamic way adjusting algorithm (DWA) to implement NVM capacity management policy. To discover the optimal size of NVM, the maximum stack distance (MSD) is dynamically monitored. Using the MSD, the DWA marks all NVM ways either as ”replaceable way” or ”non-replaceable way” to realize adjusting the number of NVM ways. Replaceable ways are regularly changed to prevent write requests from concentrating on a few NVM ways. This section explains these key ideas and the operations of the DWA.

4.2.1 Maximum stack distance

In order to find the minimum number of ways which sustain the miss rate, we introduce the MSD based on the stack property [39]. It is well known that the LRU replacement policy follows the stack property [45], which means that a cache of a size C always contains all blocks of the cache of size less than C. Assume that the number of sets is a constant value. If a cache block is in an N way cache, it is guaranteed that the block is in the cache, which has more than N ways. A metric related to stack property is the ”stack distance”. When a cache hit regardless of a read hit or a write hit, the stack distance is defined as the LRU order of the hit block. For example, the stack distance of the block at MRU position is one, and that of the LRU position is N in the N way cache. Figure 9 presents the stack distance histogram of a hypothetical application. If the number of the ways is reduced to 3 from 8, the number of

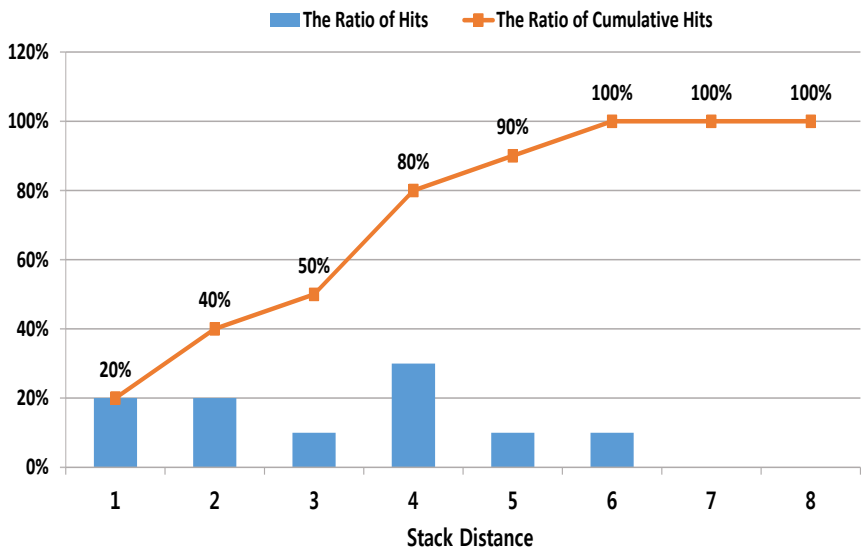


Figure 9: Stack distance histogram.

hits will be halved because the cumulative hits for stack distance 3 is 50%. This means that the miss rate of three-way cache will be increased to 50% in this case. However, if we use 6 ways instead of 8 ways, no additional cache miss occurs. Therefore, the maximum value of the stack distance indicates the minimum number of ways to maintain the hit rate.

We employ an auxiliary tag directory (ATD), a maximum stack distance register (MSDR), and a replaceable way size register (RWSR) to monitor the MSD as shown in Figure 10. The ATD is a separate storage constructed with the same associativity as the main tag array of the cache. It keeps track of the LRU order information and tag bits. When an ATD hit occurs, the MSDR is updated if LRU of the hit block is larger than the current value of the MSDR. The RWSR is updated in two cases. First, if

the MSDR exceeds the RWSR, the RWSR is increased to the MSDR. The condition that the RWSR is smaller than the MSDR means that the current working set needs more cache capacity. Thus additional NVM ways should be replaceable ways by increasing the RWSR. Second, when the value of the RWSR has been larger than that of the MSDR for a while, it is decreased to the value of the MSDR. Keeping the situation in which the RWSR is larger than the MSDR means that unnecessary NVM ways have been used. Therefore, some NVM ways should be deactivated by decreasing the RWSR. To detect this situation and initiate resizing the number of NVM ways, a resizing counter register (RCR) is added. The RCR is increased by 1 when the RWSR is larger than RWSR during the ATD hit operation. Whenever the RWSR is updated to the MSDR, the RCR is reset to 0.

Another consideration in adopting the ATD is the storage overhead. If the ATD has tag information of all sets, the size of the tag array will be doubled. Therefore, to reduce the storage overhead, we use a set sampling policy [46]. The ATD is designed to have only a part of sets which is sampled every 32nd in the proposed algorithm. It is verified that the sampled sets are enough to correctly capture the stack distance value in [46] instead of using all sets.

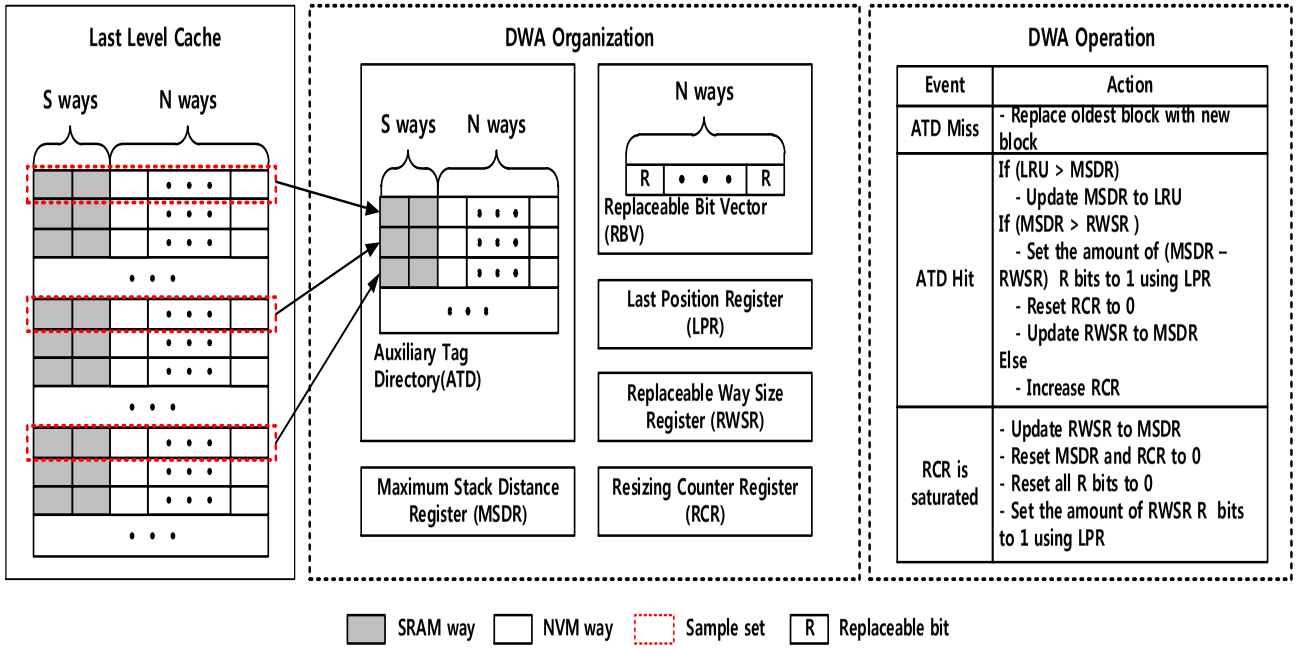


Figure 10: Overall structure of dynamic way adjusting (DWA).

4.2.2 Adjusting the number of NVM ways

Since physical NVM cells are not inserted or deleted according to the change of the MSD, we devise a method to dynamically activate or deactivate NVM ways. To disable unnecessary NVM ways, we introduce the concept of ”replaceable way” and ”non-replaceable way” The replaceable way implies the normal way that participates in all kinds of cache operations, such as read access, write access, and replacement. The non-replaceable way means that it is excluded from block replacement; thus, a new block is not placed into the way. However, when a cache hit occurs, read access and write access are performed, same as the replaceable way. All NVM ways in the DWA are divided into replaceable ways and non-replaceable ways.

The role of the replaceable bit vector (RBV) in Figure 10 is indicating that each way is non-replaceable or not by controlling replaceable (R) bits. Since each R bit is corresponded to each NVM way, the size of R bits is identical to the number of NVM ways. The RBV is altered when the RWSR is changed. If the RWSR is increased, additional R bits are set to 1. Unless, all R bits are updated to rearrange non-replaceable ways.

The cache operation for non-replaceable ways should be different from that for replaceable ways. When a cache hit is occurred to a non-replaceable way, the LRU information is not updated. In the case of a cache miss, the non-replaceable ways are not involved in the victim selection. A detailed description of the management policy is as follows:

1. Cache hit in the replaceable way: If a requested block is in the replaceable ways, the cache operations do not differ from the conventional cache. When a read hit occurs in the replaceable ways, the data is sent to the requestor. In case of a write hit, the data is modified. LRU information is updated in both cases.

2. Cache hit in the non-replaceable way: When the block is in the non-replaceable way, the data is sent to the requestor or the data is written the same as the replaceable way. However, no operation for updating LRU bits occurs because the LRU information of the non-replaceable way is useless in the DWA.

3. Cache Miss: A new block is only placed into the replaceable way. When a cache miss occurs and a requested block arrives, the LRU block in the replaceable ways is selected to load the requested block.

4.2.3 Algorithm of dynamic way adjusting

We rearrange the replaceable ways to avoid lifetime shortening when the replaceable NVM ways are reduced. If some NVM ways are frequently selected as replaceable way during execution, these ways will be worn out earlier than other NVM ways. Thus, we shift the start point of replaceable ways to allow write operations be performed as evenly as possible through the ways. The basic concept is similar to the round robin policy. At the time of selecting the replaceable ways, the NVM way next to the current replaceable ways is chosen for the first replaceable way. The last position register

(LPR) remembers the current last replaceable way to support way shifting. This policy is initiated when RCR is saturated.

Figure 11 shows an example of how this policy works. Assume that the number of the replaceable ways is five and the first three NVM ways are assigned to the replaceable ways. Note that two SRAM ways are always considered the replaceable ways. If the number of the replaceable ways is increased to six, from the fourth NVM way to the sixth NVM way, then the first NVM way is chosen as the replaceable ways.

Figure 12 presents the DWA in detail. When a cache access is confirmed to an ATD hit (line 1), the MSDR is updated if it is not the maximum LRU value (line 2-4). Then, we compare the RWSR with the MSDR to check whether the current size of NVM ways is less than the minimum size of NVM ways (line 5). If the MSDR exceeds the RWSR, some non-replaceable NVM ways are changed to be replaceable from the last NVM way of the current replaceable NVM ways. The amount of activated NVM ways is the difference between the MSDR and the RWSR. The LPR is automatically updated during way adjusting within range from 0 to W_{nvm} (line 6-9). After this adjustment, the RWSR is updated to the MSDR and the RCR is reset to 0 (line 10-11). The replaceable NVM ways are rearranged when the MSDR does not exceed RWSR when RCR is saturated (line 13). If the MSDR is larger than the number of SRAM ways, the RWSR is updated to MSDR (line 14-15). Unless, the RWSR is set to the number of SRAM ways because all SRAM ways are replaceable (line 16-17). As a first step of shifting replaceable ways, all R bits are set to 0 (line 19). Then, from the last

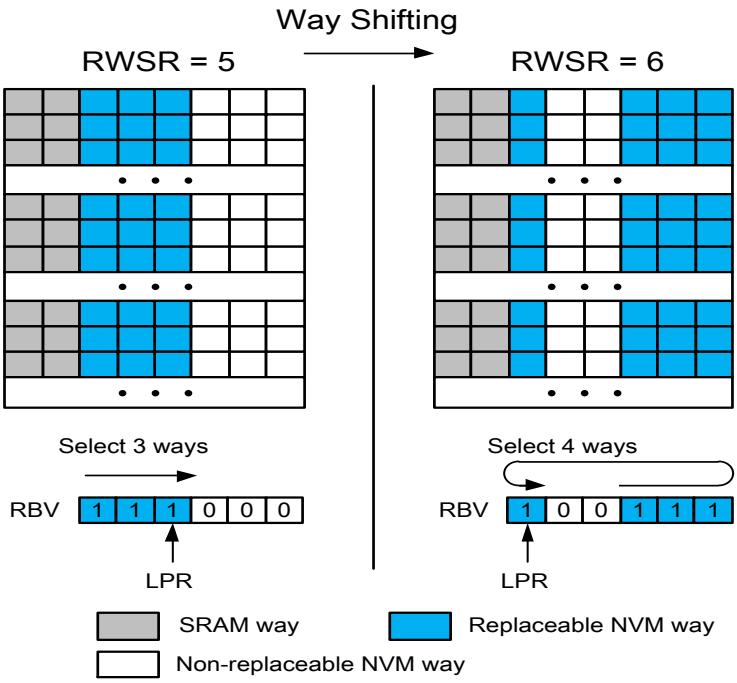


Figure 11: Example of way shifting.

replaceable NVM way, NVM ways of the amount of RWSR are assigned to be replaceable (line 20-23). To keep track of the maximum stack distance again, the MSDR is initialized to 0 and RCR is reset to 0 (line 24-25). If RCR is smaller than the threshold, RCR is increased by 1 (line 27).

Algorithm : Adjust_Replaceable_Ways

Parameters:

$RWSR$: Replaceable way size register
$MSDR$: Maximum stack distance register
LPR: Last position register $(1 \leq LPR \leq W_{nvm})$
RCR: Resizing counter register
$R[x]$: Replaceable bit at xth NVM way

Initial conditions:

$RWSR \leftarrow W_{nvm} + W_{sram}$
$MSDR \leftarrow 1$
$LPR \leftarrow W_{nvm} - 1$
$RCR \leftarrow 0$
All $R[x] \leftarrow 1$

During execution:

```
1 : if ATD hit then
2 :   if hit_block.LRU > MSDR then
3 :     MSDR ← hit_block.LRU
4 :   end if
5 :   if MSDR > RWSR then
6 :     for i ← 1 to (MSDR − RWSR) do
7 :       LPR ← (LPR + 1) % W_nvm
8 :       R[LPR] ← 1
9 :     end for
10:     RWSR ← MSDR
11:     RCR ← 0
12:   else
13:     if RCR is saturated then
14:       if MSDR > W_sram then
15:         RWSR ← MSDR
16:       else
17:         RWSR ← W_sram
18:       end if
19:       All R[x] ← 0
20:       for i ← 1 to (RWSR − W_nvm) do
21:         LPR ← (LPR + 1) % W_nvm
22:         R[LPR] ← 1
23:       end for
24:       MSDR ← 0
25:       RCR ← 0
26:     else
27:       RCR ← RCR + 1
28:     end if
29:   end if
30: end if
```

Figure 12: Algorithm for DWA.

4.3 Cache partitioning for hybrid cache architecture

Modern chip-multiprocessors (CMP) have employed multi-level on-chip caches to address the memory wall problem that is caused by the difference between access latencies of the memory and the processor. Generally, the last-level cache (LLC) occupies the largest area in the cache system and consumes a significant static energy in the CMP. To reduce the area and the leakage power, researchers have considered using non-volatile memory (NVM) [1, 3, 5] as LLC. Unlike the SRAM-based LLC, the NVM-based LLC consumes little leakage power and requires less area with higher density than SRAM. While NVM has these advantages, they also suffer from shortcomings such as longer latency to complete a write operation and higher dynamic energy consumption for a write operation compared to SRAM. Most researchers have focused on minimizing the write counts of NVM because the number of write operations strongly affects the dynamic energy consumption as well as performance.

Hybrid cache architectures (HCA) have been proposed [16, 17, 18, 19, 47] to overcome these limitations of NVM. HCA mainly consists of NVM, but some of them are replaced with SRAM to reduce the number of write requests on NVM. Previous studies concerning HCA have attempted to detect the write-intensive blocks, sets, or ways to allocate these to the SRAM. However, their schemes have not usually focused on reducing the NVM linefill counts, while the portion of NVM linefill operations is larger than

that of NVM write hit operations over the total of write operations to NVM for many applications. In addition, there is no accurate prediction model to estimate the change of the write counts of NVM when the number of SRAM and NVM ways allocated to each core are changed in CMP environments. Since the number of cache ways is closely related to the cache misses, assigning cache ways or releasing cache ways influences the miss rate of the LLC. Even though the write intensity of NVM ways of a core is larger than other cores, providing more SRAM ways with the core does not guarantee reducing the NVM write counts. If a core which hands over SRAM ways to other core generates much more cache misses with the reduced cache capacity, the write counts can be increased due to the extra linefill operations. However, they have not considered this kind of side effects in their schemes.

We propose a novel cache partitioning that is called a linefill-aware cache partitioning scheme (LCP) to reduce the dynamic energy consumption by efficiently allocating SRAM ways and NVM ways to cores. To this end, the thesis presents appropriate metrics and an algorithm for partitioning to realize LCP. We introduce three metrics that represent change of miss counts (ΔM), write counts (ΔW), and NVM write counts ($\Delta NVMW$), respectively. An algorithm for cache partitioning of LCP consists of two steps. First, the number of cache ways for each core is determined in order to reduce the miss counts. Next, the SRAM ways and the NVM ways are allocated to cores to minimize write counts of NVM.

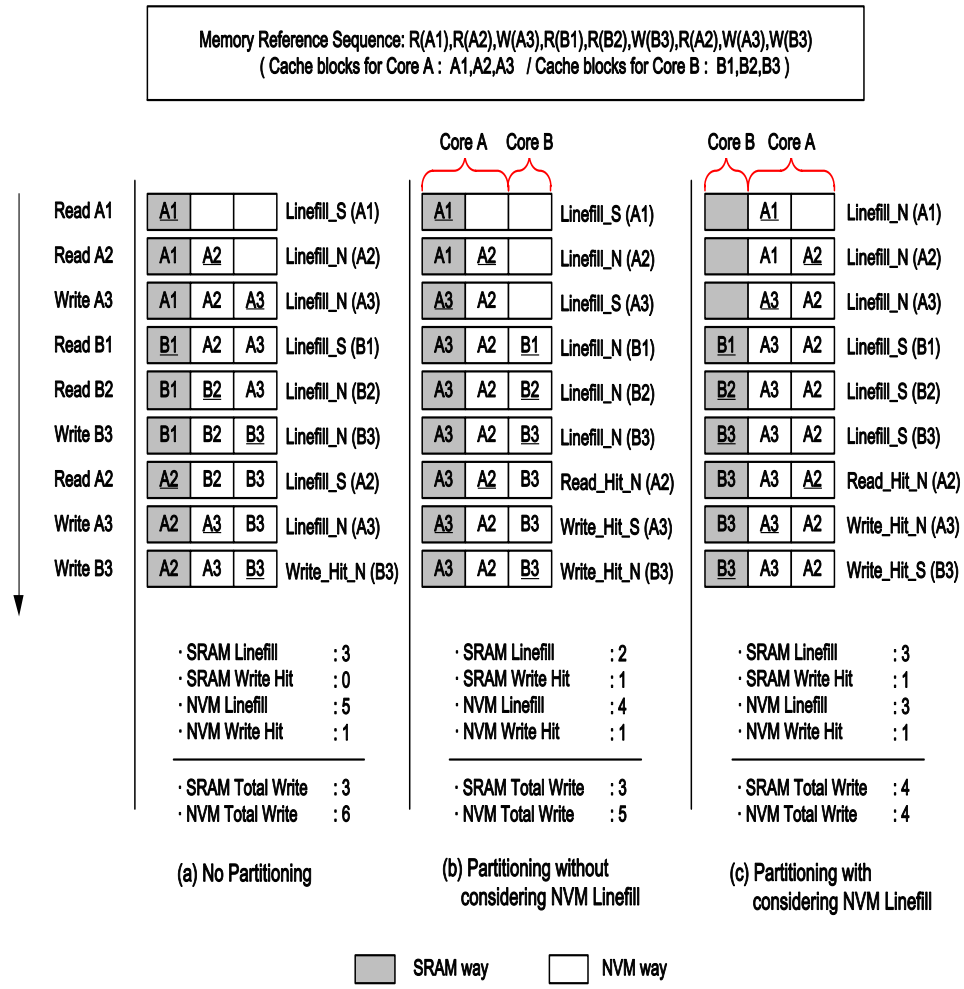


Figure 13: (a) No partitioning is applied. (b) Partitioning without NVM linefill. (c) Partitioning with NVM linefill.

4.3.1 Linefill-aware cache partitioning

To optimize the NVM write counts in HCA, SRAM ways and NVM ways should be efficiently allocated to cores. To help the understanding, we provide an illustration in Figure 13. The cache in this example consists of one SRAM way and two NVM ways. We assumed that there are two cores: core A and core B. A1, A2, and A3 are cache blocks for core A, and B1, B2, and B3 are cache blocks for core B. The cache accesses occur as the memory reference sequence shown in the box of the top in Figure 13.

When there is no special care for the LLC, the total write for the SRAM way is 3 (3 for SRAM linefill) and the NVM total write is 6 (5 for NVM linefill and 1 for NVM write hit), as shown in Figure 13(a). If the cache partitioning only considering the cache misses is applied [39], core A can occupy two cache ways and only one cache way can be assigned to core B (Figure 13(b)). Even though this partitioning decreases two cache misses and one NVM total write, the NVM write counts are not optimized. If a partitioning algorithm can predict the NVM linefill counts as well as the NVM write hit counts for every possible partitioning, the SRAM way should be allocated to core B to minimize the NVM write counts, as shown in Figure 13(c).

Therefore, a new scheme is required to reduce both the NVM write hit counts and the NVM linefill counts, which saves dynamic energy consumption of HCA. This paper devises new metrics to evaluate the effectiveness of cache partitioning schemes and proposes a linefill-aware cache partition-

Table 5: Notation descriptions for metrics.

Notation	Description
$H[i]$	Hit counts of *i*th recency position
$WH[i]$	Write hit counts of *i*th recency position
M_{CONF}	Conflict misses which are the number of cache misses due to partitioning
M_{NON_CONF}	Non-conflict misses which are the number of cache misses regardless of partitioning
$H(N)$	Total cache hit counts when the number of allocated ways is N
$M(N)$	Total cache misses when the number of allocated ways is N
$W(N)$	Total write counts when the number of allocated ways is N
$WH(N)$	Total write hit counts when the number of allocated ways is N
$\Delta M(N,N')$	Miss counts change when the number of allocated ways is changed from N to N'
$\Delta W(N,N')$	Write counts change when the number of allocated ways is changed from N to N'
$\Delta NVMW(N_{SRAM},N'_{SRAM},N_{NVM},N'_{NVM})$	NVM write counts change when the number of allocated SRAM ways is changed from N_{SRAM} to N'_{SRAM} and the number of allocated NVM ways is changed from N_{NVM} to N'_{NVM}

ing scheme (LCP) based on these metrics. Table 5 provides a description of notation we define in this section.

4.3.2 Metrics for cache partitioning

This section describes three metrics for a partitioning decision: Miss counts change (ΔM), write counts change (ΔW), and NVM write counts change ($\Delta NVMW$). We newly devise ΔW and $\Delta NVMW$ and redefine ΔM by revisiting the concept of "the utility" in the previous work [39].

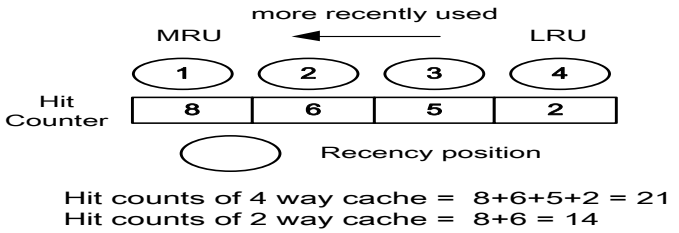


Figure 14: Example of stack property.

- ΔM: the change of the number of cache misses according to adjusting cache capacity for each core. This metric has been usually adopted to decide the cache partition to improve the performance in previous studies.

- ΔW: the change of the write counts for both the SRAM write counts and the NVM write counts according to the change of cache capacity.

- $\Delta NVMW$: the change of the NVM write counts. It is used for the HCA when memory elements are heterogeneous, while two other metrics can be applied in the cache consisting of the same memory element.

The concept of the utility [39] is used to estimate the reduction in the number of cache misses when a core has extra cache ways. The original paper named this concept as ”the utility”, but we redefine it ”ΔM” to clarify its meaning. They noticed that LRU replacement policy followed the stack property [45]. If a cache has the stack property, the cache having more cache ways always contains all blocks of the cache having smaller cache ways when the number of sets remains. Figure 14 presents the hit counts of each

recency position in a 4-way cache. In general, the recency position of the block at MRU position is called position 1, and that of the LRU position is called position 4. In this example, if the number of cache ways is reduced to 2 from 4, we expect that the hit counts of the cache will decrease by one-thirds without performing the experiments for a 2-way cache.

ΔM indicates the change of the miss counts with the change of the number of allocated ways[1]. Let $H[i]$ denote the hit counts of ith recency position of a core and $H(N)$ be the total hit counts when the number of allocated ways is N of the core. A relationship is established between two metrics.

$$H(N) = \sum_{i=1}^{N} H[i] \tag{4.18}$$

Since the increase in the miss counts is the same as the reduction in the hit counts, when the number of allocated ways is changed from N to N' of a core, $\Delta M(N,N')$ is given by

$$\Delta M(N,N') = -(H(N') - H(N)) = \sum_{i=1}^{N} H[i] - \sum_{i=1}^{N'} H[i] \tag{4.19}$$

A new model is built to estimate the change of the number of write operations with the change of the capacity in the cache. Since improving the hit rate is the most important goal in previous studies, ΔM is the only

[1] To clear the meaning of the terminology, the number of cache ways assigned for a core are called "the number of allocated cache ways of the core"

metric for cache partitioning in SRAM-based LLC in CMP environment. However, minimizing the write counts should be considered as well as maximizing the overall hit counts in HCA. Thus, we define a new metric (ΔW) for representing the change of the number of write accesses caused by the change of partitioning.

The change of write counts over the change of the amount of allocated ways is not easily determined, while ΔM is obtained by just accumulating $H[i]$. A cache block of the LLC is updated by two cases. First, when a write hit occurs in the LLC, the corresponding block is overwritten. In addition, if a new block is loaded due to a cache miss, the contents of the block are updated. Therefore, the write counts change (ΔW) is the sum of the write hit counts and the linefill counts.

To find the total write hit counts, we define $WH[i]$ as the write hit counts for ith recency position. The write hit counts $WH(N)$ is expressed in a similar form as the hit counts.

$$WH(N) = \sum_{i=1}^{N} WH[i] \tag{4.20}$$

Calculating the total linefill operations is more complicated than obtaining the total write hit counts because there are two kinds of cache misses to be considered. The first category of the cache miss is called a conflict miss (M_{CONF}), which occurs when a core partially uses the LLC due to cache partitioning. If all cache ways are allocated to the core, the amount of the

conflict miss becomes zero; thus, it varies across resizing the number of allocated ways. On the other hand, there is another kind of cache miss, called a non-conflict miss (M_{NON_CONF}), which occurs regardless of partitioning. In other words, when a core utilizes all cache ways, there is no M_{CONF} in the core, while M_{NON_CONF} can occur. Note that the non-conflict miss is composed of two kinds of misses, usually referred to as capacity and compulsory misses [48]. In our proposal, we use a single term as a non-conflict miss because there is no need to distinguish these misses.

Combining the two cache misses, the miss counts ($M(N)$) can be written as follows:

$$\begin{aligned} M(N) &= M_{CONF} + M_{NON\_CONF} \\ &= H(N_{ALL}) - H(N) + M_{NON\_CONF} \\ &= \sum_{i=1}^{N_{ALL}} H[i] - \sum_{i=1}^{N} H[i] + M_{NON\_CONF} \end{aligned} \tag{4.21}$$

where N_{ALL} is the number of total cache ways in the LLC.

To put it all together, $W(N)$ is expressed as

$$W(N) = WH(N) + M(N) \tag{4.22}$$

Since $\Delta W(N,N')$ means the change of the write counts, we reach the following equation:

$$\Delta W(N,N') = (WH(N') + M(N')) - (WH(N) + M(N)) \tag{4.23}$$

From Eq. 4.20 and Eq. 4.21, we transform Eq. 4.23 into the following:

$$\begin{aligned}\Delta W(N,N') = &(\sum_{i=1}^{N'} WH[i] + \sum_{i=1}^{N_{ALL}} H[i] - \sum_{i=1}^{N'} H[i] + M_{NON\_CONF}) \\ &-(\sum_{i=1}^{N} WH[i] + \sum_{i=1}^{N_{ALL}} H[i] - \sum_{i=1}^{N} H[i] + M_{NON\_CONF})\end{aligned} \tag{4.24}$$

This can be written in this form:

$$\begin{aligned}\Delta W(N,N') = &\sum_{i=1}^{N'} WH[i] - \sum_{i=1}^{N'} H[i] - \sum_{i=1}^{N} WH[i] + \sum_{i=1}^{N} H[i] \\ &+ (\sum_{i=1}^{N_{ALL}} H[i] - \sum_{i=1}^{N_{ALL}} H[i]) + (M_{NON\_CONF} - M_{NON\_CONF})\end{aligned} \tag{4.25}$$

$H(N_{ALL})$ and M_{NON_CONF} in the above equation are removed because they do not change with the number of allocated ways. Therefore, after simplifying Eq. 4.25, this becomes

$$\Delta W(N,N') = \sum_{i=1}^{N'} (WH[i] - H[i]) - \sum_{i=1}^{N} (WH[i] - H[i]) \tag{4.26}$$

To aid the understanding of the equation, we provide illustrations in Figure 15. In this figure, Eq. 4.26 is applied to find the write counts change, while Eq. 4.19 is used to calculate the miss counts change. When the amount of allocated ways is increased to 3 from 2 ($N = 2$ and $N' = 3$), $\Delta M(2,3)$ is -5 and $\Delta W(2,3)$ is -3.

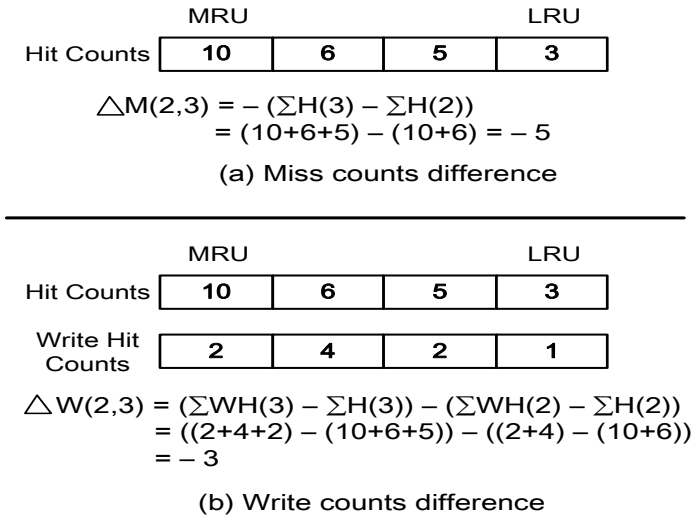


Figure 15: Examples of (a) miss counts change (ΔM) and (b) write counts change (ΔW).

This section describes the NVM write counts change ($\Delta NVMW$) used for calculating the variation of the write accesses to NVM in HCA. In the above section, we showed that the write counts are changed, but it is only applied in the LLC, which has one memory type. Thus, another metric is required to measure the change of NVM write counts. Note that $\Delta NVMW$ has four kinds of parameters because two types of memory elements are considered in this model. N is divided into N_{SRAM} and N_{NVM}, which are the number of allocated SRAM ways and NVM ways before new partitioning is initiated, respectively. Instead of N', N'_{SRAM} and N'_{NVM} are used to indicate how many SRAM ways and NVM ways are allocated to a specific core based on the new partitioning. Therefore, this metric is expressed as $\Delta NVMW(N_{SRAM},N'_{SRAM},N_{NVM},N'_{NVM})$.

We propose a new method to measure the variation of the write counts of NVM because the methods on the stack property cannot calculate the exact change of the write counts of NVM. For example, when a certain NVM way receives five write requests, removing the NVM way does not decrease the write counts of NVM by five. Since the concept of recency position is independent to the order of way, every way can have any recency position and the position usually changes after every cache access. When the number of allocated ways is changed, the blocks are stored into different ways from they were, and the hit counts of each way are not reserved. Therefore, it is impossible to exactly predict the change of the write counts of NVM or SRAM when the number of the allocated cache ways is changed.

Instead, we use a statistical approach to find the NVM write counts. In general, every way has the same probability of receiving write requests, which means write requests are statistically evenly distributed among the ways. Therefore, the portion of the NVM write counts over the all write counts is assumed to be proportional to the ratio of the number of NVM ways over the total number of cache ways.

$$\begin{aligned} &NVMW(N_{SRAM}, N_{NVM}) \approx \\ &W(N_{SRAM} + N_{NVM}) * \frac{N_{NVM}}{N_{SRAM} + N_{NVM}} \end{aligned} \tag{4.27}$$

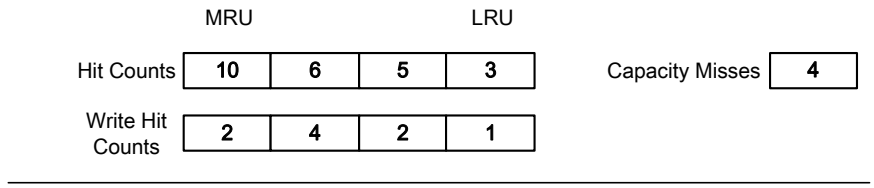
Therefore, $\Delta NVMW$ is calculated as follows:

$$\begin{aligned}
&\Delta NVMW(N_{SRAM}, N'_{SRAM}, N_{NVM}, N'_{NVM}) \\
&= NVMW(N'_{SRAM}, N'_{NVM}) - NVMW(N_{SRAM}, N_{NVM}) \\
&= W(N') * \frac{N'_{NVM}}{N'} - W(N) * \frac{N_{NVM}}{N}
\end{aligned} \tag{4.28}$$

$$\begin{aligned}
&= (WH(N') + M(N') + M_{NON_CONF}) * \frac{N'_{NVM}}{N'} \\
&-(WH(N) + M(N) + M_{NON_CONF}) * \frac{N_{NVM}}{N}
\end{aligned} \tag{4.29}$$

$$\begin{aligned}
&= (\sum_{i=1}^{N'} WH[i] + \sum_{i=N'+1}^{N_{ALL}} H[i] + M_{NON_CONF}) * \frac{N'_{NVM}}{N'} \\
&-(\sum_{i=1}^{N} WH[i] + \sum_{i=N+1}^{N_{ALL}} H[i] + M_{NON_CONF}) * \frac{N_{NVM}}{N}
\end{aligned} \tag{4.30}$$

Figure 16 shows the procedure of calculation of the equation. On top of the write hit counters, a non-conflict miss counter is inserted. A cache in the example is composed of two SRAM ways and two NVM ways. We assume that a core takes one SRAM way and one NVM way at first. If one more way is assigned to the core, there are two options; the core gets either an extra NVM way or SRAM way. For former case, we add an NVM way to the core, $\Delta NVMW$ is increased by 1. On the contrary, the latter case shows that $\Delta NVMW$ becomes -4.



△NVMW(1,1,1,2) = ∑NVMW(1,1) – ∑NVMW(1,2)
= (∑WH(2) + ∑M(2)) * (1 / 2) – (∑WH(3) + ∑M(3)) * (2 / 3)
= ((2+4) + (5+3+4)) * (1 / 2) – ((2+4+2) + 3 + 4) * (2 / 3) = – 1

(a) An NVM way is added (1S1N -> 1S2N)

△NVMW(1,1,1,2) = ∑NVMW(1,1) – ∑NVMW(2,1)
= (∑WH(2) + ∑M(2)) * (1 / 2) – (∑WH(3) + ∑M(3)) * (1 / 3)
= ((2+4) + (10+6+4)) * (1 / 2) – ((2+4+2) + 6 + 4) * (1 / 3) = – 4

(b) An SRAM way is added (1S1N -> 2S1N)

Figure 16: Examples of NVM write counts change ($\Delta NVMW$). Initially, a core owns an SRAM way and an NVM way (1S1N). (a) The core acquires one more NVM way (1S2N). (b) The core acquires one more SRAM way (2S1N).

4.3.3 Algorithm for cache partitioning

The algorithm for LCP consists of two steps to optimize the NVM write counts without increasing cache misses, as shown in Figure 17. The first step is finding the best partitions for optimizing the linefill counts. LCP utilizes ΔM to search for the optimal size of partition in this step. After that, the SRAM partition and NVM partition of each core are determined within its budget determined by the first step, based on ΔW and $\Delta NVMW$. Table 6 lists the description of notation we define in this section.

To make our algorithms more efficient, we employ the concept of the marginal utility approach introduced in UCP [39]. Since prior studies of

Algorithm 1 : Linefill-aware Cache Partitioning

Step 1 : finding the number of allocated cache ways

1 : $U_{ALL} \leftarrow T_{ALL}$ - T_{CORE}
2 : **foreach** *i* ← *all cores* **do**
3 : $A_{ALL}[i] \leftarrow 1$
4 : **end if**
5 : **while** $U_{ALL} > 0$ **do**
6 : min_MU ← ∞
7 : **foreach** *i* ← *all cores* **do**
8 : **for** *w* ← 1 **to** U_{ALL} **do**
9 : MU ← Δ*M* ($A_{ALL}[i]$,$A_{ALL}[i]$+w) / w
10: **if** *MU* < *min_MU* **do**
11: min_MU ← MU
12: $C_{CORE} \leftarrow i$
13: Req ← w
14: **end if**
15: **end for**
16: **end foreach**
17: $A_{ALL}[C_{CORE}] \leftarrow A_{ALL}[C_{CORE}]$ + Req
18: $U_{ALL} \leftarrow U_{ALL}$ - Req
19: **end while**

Step 2 : finding the number of allocated NVM ways

20: $U_{SRAM} \leftarrow T_{SRAM}$
21: **foreach** *i* ← *all cores* **do**
22: $A_{NVM}[i] \leftarrow A_{ALL}[i]$
23: **end foreach**
24: **while** $U_{SRAM} > 0$ **do**
25: **foreach** *i* ← *all cores* **do**
26: min_MU ← ∞
27: **if** $U_{SRAM} > A_{ALL}[i]$ **then**
28: w' ← $A_{ALL}[i]$
29: **else**
30: w' ← U_{SRAM}
31: **end if**
32: **for** *w* ← 1 **to** *w'* **do**
33: **if** $U_{SRAM} == 0$ and $A_{SRAM}[i] == 0$ **do**
34: MU ← Δ*W*($A_{NVM}[i]$,$A_{NVM}[i]$ + w) / w
35: **else**
36: MU ← Δ*NVMW*($A_{SRAM}[i]$,$A_{SRAM}[i]$ + w, $A_{NVM}[i]$,$A_{NVM}[i]$ - w) / w
37: **end if**
38: **if** *MU* < *min_MU* **do**
39: min_MU ← MU
40: $C_{CORE} \leftarrow i$
41: Req ← w
42: **end if**
43: **end for**
44: **end foreach**
45: $A_{SRAM}[C_{CORE}] \leftarrow A_{SRAM}[C_{CORE}]$ + Req
46: $A_{NVM}[C_{CORE}] \leftarrow A_{ALL}[C_{CORE}]$ - $A_{SRAM}[C_{CORE}]$
47: $U_{SRAM} \leftarrow U_{SRAM}$ - Req
48: **end while**

Figure 17: Algorithm of linefill-aware cache partitioning (LCP).

Table 6: Notation descriptions for algorithms.

Notation	Description
T_{ALL}	Number of total cache ways in the LLC
T_{SRAM}	Number of total SRAM ways in the LLC
T_{NVM}	Number of total NVM ways in the LLC
T_{CORE}	Number of total cores
U_{ALL}	Number of unallocated ways
U_{SRAM}	Number of unallocated SRAM ways
U_{NVM}	Number of unallocated NVM ways
$A_{ALL}[i]$	Number of allocated ways per *i*th core
$A_{SRAM}[i]$	Number of allocated SRAM ways for *i*th core
$A_{NVM}[i]$	Number of allocated NVM ways for *i*th core
MU	Marginal utility of metrics
min_MU	Minimum value of marginal utility
Req	Number of requesetd ways to get min_MU
C_{CORE}	A specific core gaining extra cache ways

NVM-based CMP used the greedy algorithm [49, 50], there is a risk of reaching to a suboptimal partitioning, which commonly occurs in greedy algorithms. To avoid this problem, LCP uses the marginal utility. Therefore, our algorithm uses a value which is divided by the number of allocated ways instead of the value directly obtaining from the calculation. For example, if ΔW is -4 and the number of allocated ways is 2, the marginal utility (MU) of ΔW is -2 (= -4 / 2). In addition, the partitioning algorithm is designed to perform the cache repartitioning every 1M cycles because it shows the best efficiency compared with other periods.

Step 1 starts initializing U_{ALL}, which is a key variable of the first loop (line 1). Since each core has at least one way, U_{ALL} has the difference be-

tween the number of total cache ways in the LLC and the number of cores (line 2-4). Step 1 is executed until all ways are assigned to cores (line 5). When each iteration begins, *min_MU* is initialized to infinity; in reality, it has the maximum integer value that a system allows (line 6). For every core, ΔM per way are calculated by varying the number of allocated cache ways (line 7-9). If *MU* is smaller than the currently minimum value of *MU* (line 10), *min_MU* is updated (line 11), and the current core is tentatively indicated as the target core to be allocated more cache ways (line 12). *Req* has the current number of allocated ways (line 13). When the loop ends, the requested ways are allocated to the target core (line 17) and U_{ALL} is updated as well (line 18). Note that this step is performed based on the UCP [39], which is known as one of the best partitioning schemes. Because this step is orthogonal to second step, other partitioning schemes can be used if they provide the better partitioning efficiency.

Step 2 works similar to step 1, but a key variable of the loop becomes U_{SRAM} substituting U_{ALL} and $\Delta NVMW$ and ΔW are used instead of ΔM because SRAM ways are distributed among cores in this step. At first, U_{SRAM} has the number of SRAM cache ways (line 20). The number of the allocated NVM ways for each core is temporarily the number of allocated cache ways, which is determined by the previous step (line 22-24). Another difference from step 1 is that a loop for finding the *min_MU* is iterated when the candidate number of cache ways is from 1 to the maximum value between $A_{ALL}[i]$ and U_{SRAM} (line 27-31). This is because each core cannot have more ways than $A_{ALL}[i]$. $\Delta NVMW$ is basically used to find the value of *MU* (line 36),

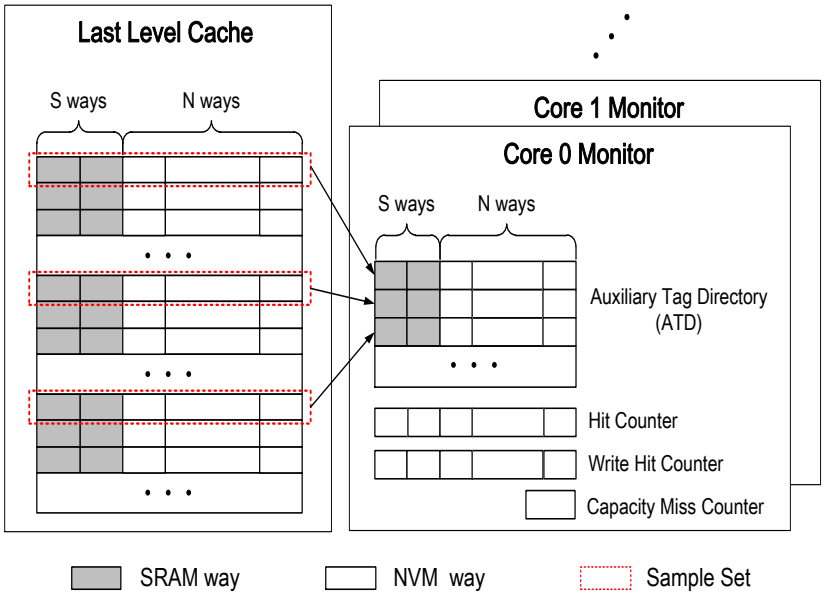


Figure 18: Overall structure of LCP.

however ΔW is applied for simplicity if it is guaranteed that no SRAM way involves calculation (line 34). In this algorithm, the number of NVM ways are simply calculated; we obtain it by subtracting $A_{ALL}[i]$ to $A_{SRAM}[i]$ (line 46).

We extend the conventional utility monitor [39] and utilize a cache partitioning logic of UCP to implement our proposal. Therefore, storage overhead is estimated as less than 1%. The traditional utility monitor contains an auxiliary tag directory (ATD) and hit counters. On top of that, two additional counters are added which are a write hit counter and a non-conflict miss counter, as depicted in Figure 18. As many write hit counters as the number of cache ways are needed, and only a single counter is required for

accumulating the non-conflict misses.

The role of the ATD is keeping track of the recency positions of blocks for each core. Using the ATD, the hit counter indicates the hit counts of each recency position. Similar to the hit counter, the write hit counters store the number of write hit for the corresponding position. The associativity of the hit counter and the write hit counter is the same as the LLC. The non-conflict miss counter is inserted to obtain the total non-conflict miss counts. If a cache miss occurs in the ATD, the non-conflict miss counter is increased by one, while the hit counter is increased when a cache hit occurs in the corresponding recency position.

Assuming that the LLC has 16-way associativity and the size of each counter is 32 bits, the total storage overhead of the LCP is (16 + 1) * 32 bits = 68 bytes. Considering the capacity of the LLC is 2MB in our system, it is obvious that the storage overhead is not significant.

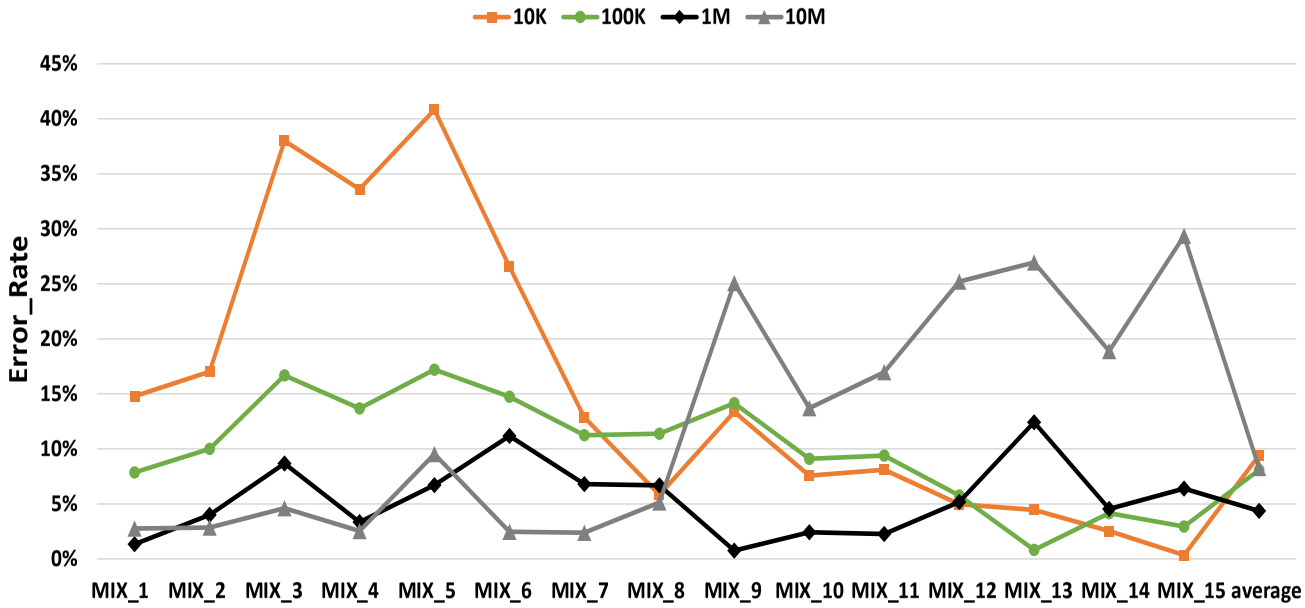


Figure 19: Error rates with various repartitioning period from 10K to 10M.

We start by analyzing how accurate the proposed algorithm predicts the NVM write counts. Whenever the cache partitioning is done, the expected NVM write counts during the execution period is accumulated. At the end of the program execution, the difference between the predicted value and the measured value is used to calculate the error value of the algorithm. In this way, we estimate the error rate of our algorithms as follows:

$$ErrorRate = \frac{|Predicted\ NVM\ Writes - Measured\ NVM\ Writes|}{Predicted\ NVM\ Writes} * 100 \tag{4.31}$$

Figure 19 summarizes error rates of our algorithm with various sizes of repartitioning periods from 10K to 10M. LCP utilizes the statistics of each period to predict the behavior of the next. If a previous period has a similar access pattern of the following period, this approach will be effective. Unfortunately, if partitioning occurs in the middle of transition of working sets in the program, the information gathered by the ATD during the current period does not represent the next period. In this case, the accuracy of hit counts, write hit counts, and cache misses will decrease. Thus, we have experimented with several repartitioning periods and the consequential change of the accuracy. The proposed LCP with the 1M period cycle shows that the error rate is 4.3%, which is meaningfully lower than the error rate of other period sizes. Therefore, we choose 1M as the repartitioning period for our proposal.

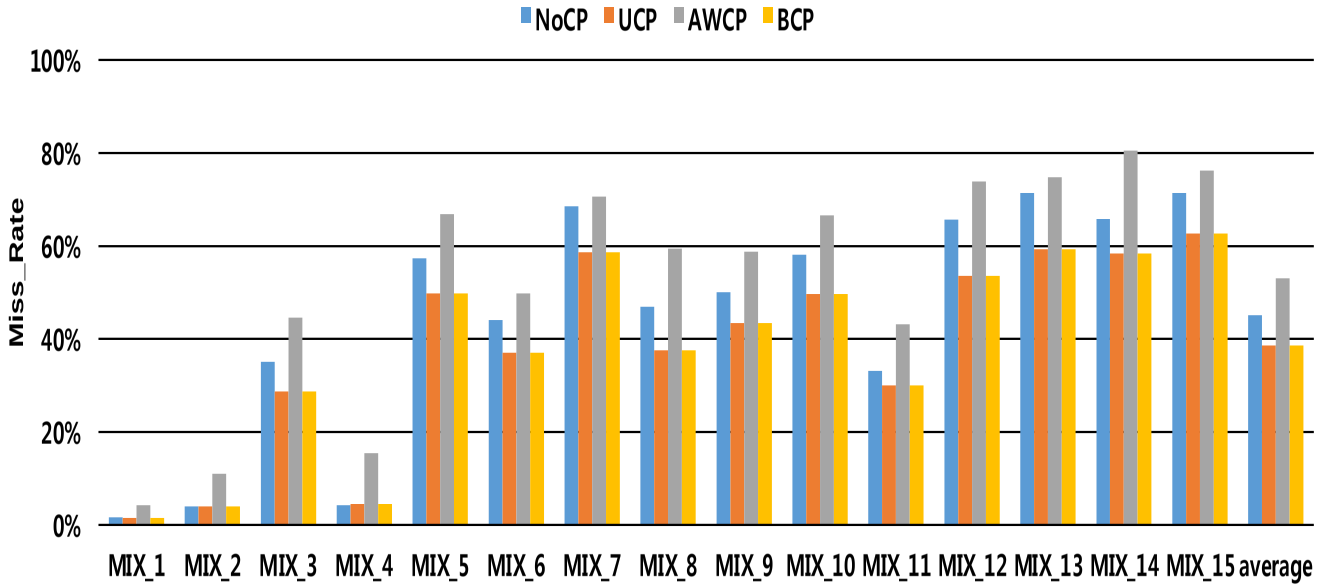


Figure 20: Miss rates with four schemes.

The miss rates for all workloads are given in Figure 20 for NoCP, BSABM, AWCP, and LCP. AWCP shows the worst miss rate for all benchmark programs because the number of cache ways for each core is adjusted according to its NVM write intensity. Even though this approach is beneficial to reducing the number of write counts, it is not helpful to improve the total hit counts. The miss rate of BSABM is the nearly same as NoCP because they use a similar replacement policy. The miss rate of LCP is decreased by 4.3% over NoCP, and the difference between average miss rate of AWCP and LCP is 13.7%. While the efficiency of LCP varies significantly depending on characteristics of workload, the miss rates of all applications are decreased. For MIX_4, the miss rate of LCP is lower than that of AWCP by 21.9%.

4.4 Overhead of NVM capacity management policy

Table 7 shows the storage overhead of the DWA. We assume that the system uses a 40-bit physical address space. To keep track of the MSD, an entry of the ATD has a separate tag and LRU bits. The each ATD has 64 entries and 256 entries because the number of sample sets is 64 and 256 respectively. The size of R bits is 12 as the number of NVM ways is 12. The DWA also needs three kinds of 4-bit registers and a 2-bit resizing counter register. Both HCAs have about less than 1% extra area. With a low hardware overhead, our proposal achieved the dynamic energy saving and write endurance en-

Table 7: Storage overhead.

Component	HCA with STT-RAM	HCA with PCM
ATD entry	LRU + Tag + Valid = 4 + 22 + 1 = 27 bits 27 bits * 16 way = 54 bytes	LRU + Tag + Valid = 4 + 20 + 1 = 25 bits 25 bits * 16 way = 50 bytes
ATD	54 bytes * 64 sets = 3.8KB	50 bytes * 256 sets = 12.5KB
R bits	12 bits	12 bits
LPR	4 bits	4 bits
MSDR	4 bits	4 bits
RWSR	4 bits	4 bits
RCR	2 bits	2 bits
Overhead for LCP	(16 + 1) * 32 bits = 68 bytes	(16 + 1) * 32 bits = 68 bytes
Total	about 4KB (0.1%)	about 13KB (0.31%)

hancement. For the LCP, as we discussed earlier, the total storage overhead of the LCP is (16 + 1) * 32 bits = 68 bytes on top of the extra storage of the DWA. Therefore, the storage overhead of both schemes is not significant.

Another consideration for cache partitioning is the timing overhead of obtaining the optimal value. To investigate the timing overhead, we calculated the latencies of the algorithm in detail as shown in Table 8. According to Eq. 4.19 one iteration of the main loop of step 1 requires one addition, one subtraction, one division, one comparison, and one assignment. The latencies of an adder and a comparator are one cycle and the latency of a divider is thirteen cycles in modern processors [43], thus one iteration takes 17 cycles (we assume that each register captures the value in a cycle). Ac-

Table 8: Timing overhead.

Component	Cycles
Step1 Initialization (line 1-4)	2 cycles
Step1 Main loop (line 6-16)	17 cycles
Step1 Result assigning (line 17-18)	2 cycles
Step2 Initialization (line 20-23)	3 cycles
Step2 Main loop preparation (line 24-31)	2 cycles
Step2 Main loop (line 32-44)	36 cycles
Step2 Result assigning (line 45-47)	3 cycles
Total	851 cycles (0.9%)

cording to Eq. 4.30, one iteration of the main loop of the step 2 requires three additions, one multiplication, two divisions, one comparison, and one assignment. The latency of a multiplier is five cycles in modern processors [43], thus one iteration takes 36 cycles.

The initialization steps are executed once for every partitioning. The main loop in step one of LCP is iterated 24.95 times and the main loop in step two is iterated 10.21 times. The other parts of the algorithm are executed 4.57 times and 2.31 times for each step respectively. Therefore, the algorithm takes 851 cycles to identify the average of the partitioning (2+17*25+2*5+3+2*3+36*11+3*3 = 851). Considering that the period of partitioning is 1M, the latency of the algorithm does not have an influence on the overall performance.

Chapter 5

Experimental results

5.1 Experimental environment

We simulated our approach with PARSEC benchmark suite [11] for evaluating WACC. The gem5 simulator is used to evaluate the normalized energy and normalized lifetime of our protocol [9]. The overall simulation parameters are shown in Table 9. We assume that the cache coherence protocol is a MOESI protocol. In addition, LLC is composed of STT-RAM because STT-RAM is considered as the right alternative among several types of NVM [51]. The power value of STT-RAM is derived from the previous work [52].

For DWA, a simulation was performed using Macsim [10] which is a trace-driven and cycle level simulator. It is designed to thoroughly model the detailed microarchitectural behavior, including pipeline stages and memory systems. Our baseline system has a three level cache hierarchy. The L1 and L2 caches are composed of the SRAM memory. Table 9 shows our baseline processor configurations in detail. Since STT-RAM and PCM are widely studied among several kinds of NVM, the LLC has two hybrid cache configurations: STT-RAM with SRAM, PCM with SRAM. We examined our proposal on multi core configuration which has 4 cores as well. We used

SPEC CINT2006 and SPEC CFP2006 of the SPEC CPU2006 benchmark suite [12]. Because the benchmark programs with the reference input set take a very long time to run, we simulated 500M instructions of the region selected by Pinpoints [53, 54] which is a well-known tool to find the representative regions. To compare our proposal with previous studies, we also conducted the experiments with prediction table based cache line replacement and management policy (PTHCM) [18]. For multi core system simulation, we generated ten workloads by mixing six applications as listed in Table 12.

In addition, the standard of normalization in our results is the baseline hybrid cache, which is operated as a conventional cache except that it consists of both SRAM and STT-RAM cells. Thus, the baseline hybrid cache has no special policy such as the DWA or the PTHCM. For DWA, note that write intensity block migration policy is always applied. Finally, we assume that cache hierarchy maintains inclusion property in our proposal as like many modern processors such as the Intel i7 processor [43] or ARM CORTEX-A57 processor [55].

We have performed experiments to evaluate the proposed cache partitioning scheme with Macsim [10] for LCP. Table 9 presents the system parameters used for the simulation. It has four cores and a two-level cache hierarchy. The capacity of the L1 instruction and data caches are 32KB, and they are 4-way associative caches. The LLC (L2) cache is a 2MB 16-way cache, which is composed of 4-way SRAM and 12-way NVM. The line size of all caches is 64B.

Table 9: Processor configurations.

WACC	
Cores	4
L1 Inst / Data Cache	64KB, 2-way, 64B line
L2 Unified Cache	2MB, 16-way, 64B line
Memory	64bit bus width , 4 read/write ports
Function Units	6 IALU, 2 IMULT, 4 FPALU, 2 FPMULT

DWA	
Core Type	x86, out-of-order, 2GHz
Core Count	1 / 4
INT / MEM / FP	4 / 4 / 4
Branch Predictor	gshare predictor, 16 history length
ROB Size	256
I/D Cache	16KB, 4-way, 64B blocks, 1-cycle latency
L2 Cache	512KB, 8-way, 64B blocks, 5-cycle latency
Hybrid LLC with STT-RAM	4MB(4-way SRAM and 12-way STT-RAM), 64B blocks SRAM: 10-cycle latency STT-RAM: 10-cycle (read) and 45-cycle (write) latency
Hybrid LLC with PCM	16MB(4-way SRAM and 12-way PCM), 64B blocks SRAM: 10-cycle latency PCM: 19-cycle (read) and 93-cycle (write) latency
Memory Latency	200 cycles

LCP	
Core Type	x86, out-of-order, 2GHz
Core Count	4
INT / MEM / FP	4 / 4 / 4
Branch Predictor	gshare predictor, 16 history length
ROB Size	256
I/D Cache	32KB, 4-way, 64B blocks, 2-cycle latency
Hybrid LLC	2MB(4-way SRAM and 12-way STT-RAM), 64B blocks
Memory Latency	200 cycles

We used SPEC CINT2006 and SPEC CFP2006 of the SPEC CPU2006 benchmark suite for the simulation [12] for LCP. To evaluate the efficiency of our proposal across write intensive and non-write intensive applications, workloads are created based on write counts per kilo-instructions (WBKI). At first, we sorted the applications by increasing the order based on WBKI as shown in Table 10 and divided them into three categories: such as low, mid, and high. Mixing four benchmarks from the three categories, we generated 15 workloads as listed in Table 11 (The number of combination of selecting 4 applications from 3 categories with repetitions is 15 and applications in each category are randomly selected.) Each trace is collected by Pinpoints [53], which is widely used to extract the representative regions.

There are four schemes tested in our simulation: the baseline which uses no partitioning scheme (NoCP), block swapping and active block migration (BSABM) [49], access-aware cache partitioning policy (AWCP) [50], and LCP proposed in the thesis. NoCP has no partitioning scheme and follows the LRU replacement. To compare the previous studies with our proposal, BSABM and AWCP, which are available for the HCA-based LLC in CMP, are included for the experiment.

To fairly compare the results of our proposal and previous studies, we used the same parameters of STT-RAM that were used in the previous study [50]; the dynamic energy consumption of cache operation for an SRAM cache bank 0.609nJ, while the read energy for an STT-RAM cache bank is 0.598nJ and the write energy is 4.375nJ.

Table 10: Write counts per kilo-instructions for LCP.

Type	Benchmark	WPKI	Type	Benchmark	WPKI
Low	dealII	0.90	Mid	zeusmp	30.92
	gamess	1.04		cactusADM	41.78
	gromacs	1.79		gcc	51.96
	povray	2.31		omnetpp	65.46
	perlbench	2.38	High	milc	75.94
	h264ref	4.13		wrf	92.29
	calculix	7.56		libquantum	114.29
	xalancbmk	8.10		GemsFDTD	133.44
Mid	gobmk	11.20		leslie3d	138.10
	hmmer	12.99		soplex	145.47
	tonto	13.53		lbm	221.45
	bzip2	15.75		mcf	228.77

Table 11: Multi-core workloads for LCP.

Workload	Benchmarks
MIX_1	dealII(L), gamess(L), calculix(L), xalancbmk(L)
MIX_2	gamess(L), gromacs(L), h264ref(L), cactusADM(M)
MIX_3	dealII(L), povray(L), xalancbmk(L), lbm(H)
MIX_4	gromacs(L), povray(L), gcc(M), omnetpp(M)
MIX_5	povray(L), perlbench(L), cactusADM(M), libquantum(H)
MIX_6	dealII(L), gamess(L), soplex(H), lbm(H)
MIX_7	xalancbmk(L), gobmk(M), cactusADM(M), omnetpp(M)
MIX_8	dealII(L), gcc(M), omnetpp(M), mcf(H)
MIX_9	povray(L), zeusmp(M), wrf(H), lbm(H)
MIX_10	povray(L), libquantum(H), lbm(H), mcf(H)
MIX_11	gobmk(M), hmmer(M), gcc(M), omnetpp(M)
MIX_12	gobmk(M), tonto(M), omnetpp(M), lbm(H)
MIX_13	hmmer(M), bzip2(M), leslie3d(H), lbm(H)
MIX_14	hmmer(M), GemsFDTD(H), leslie3d(H), mcf(H)
MIX_15	milc(H), wrf(H), lbm(H), mcf(H)

Table 12: Multi-core workloads for DWA.

Workload	Benchmarks
MIX_1	bwaves, calculix, wrf, gromacs
MIX_2	bwaves, calculix, wrf, hmmer
MIX_3	bwaves, calculix, wrf, h264ref
MIX_4	bwaves, calculix, gromacs, hmmer
MIX_5	bwaves, calculix, gromacs, h264ref
MIX_6	bwaves, calculix, hmmer, h264ref
MIX_7	bwaves, wrf, gromacs, hmmer
MIX_8	bwaves, wrf, gromacs, h264ref
MIX_9	bwaves, wrf, hmmer, h264ref
MIX_10	bwaves, gromacs, hmmer, h264ref
MIX_11	calculix, wrf, gromacs, hmmer
MIX_12	calculix, wrf, gromacs, h264ref
MIX_13	calculix, wrf, hmmer, h264ref
MIX_14	calculix, gromacs, hmmer, h264ref
MIX_15	wrf, gromacs, hmmer, h264ref

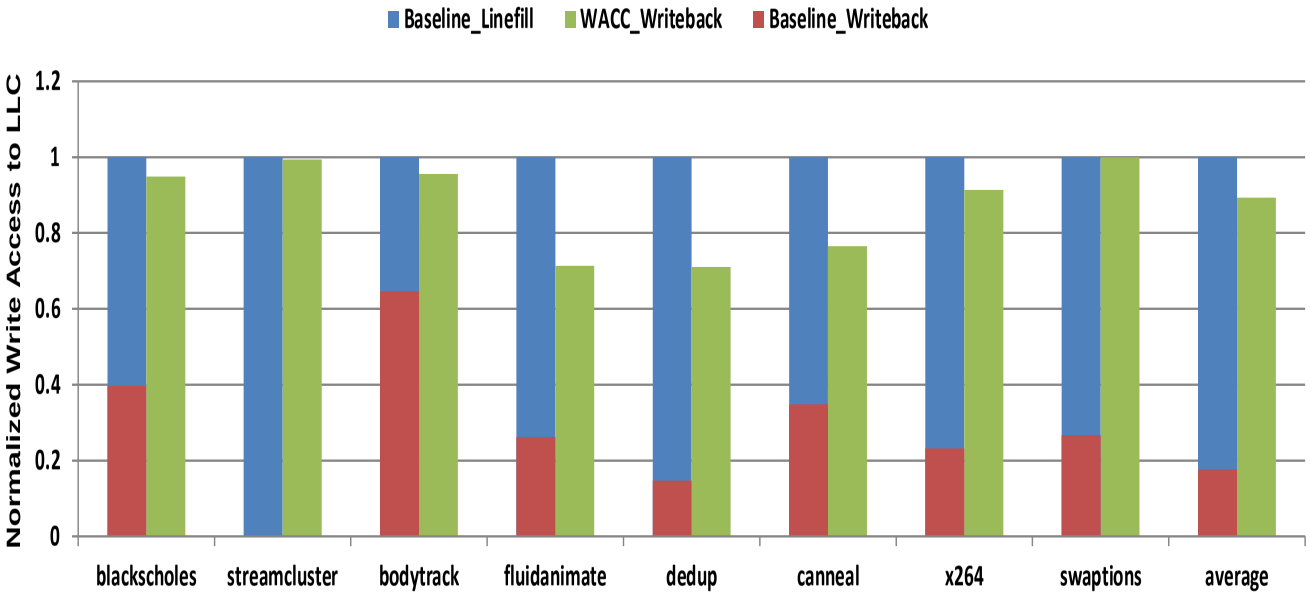


Figure 21: Normalized number of the access to LLC of WACC protocol compared to the MOESI protocol.

5.2 Write access to NVM

Figure 21 presents the normalized number of the read and write access to LLC in our protocol compared to the baseline MOESI protocol. Note that write access is divided into writeback access and linefill access. As a result, 13.2% of the write operations were decreased on average. The noticeable result is that the number of the writeback access was increased, while there were no linefill operation. When a cache block is evicted in a private cache, the writeback operation is not required in the existing protocols if the cache block is not modified. This is because the LLC already has the valid block data if the cache block is clean. On the contrary, the writeback operation should be initiated if no other private cache has the valid copy during cache replacement in WACC protocol. This difference generates the extra write-back operations. However, the total number of the write access in WACC protocol is smaller than that of other protocols because the reduction in the linefill operation is much larger than the increment in the writeback operation.

We first examined the write counts of NVM ways as depicted in Figure 22 and Figure 23. About 75.4% reduction and 77.2% reduction in the number of write accesses is achieved on average in the DWA for HCAs with STT-RAM and PCM, respectively, while the decrement on the number of write accesses to NVM ways of PTHCM are about 5.7% and 11.0%.

From the two figures, we discover that the write access reduction ratio of the DWA follows the sensitivity of the miss rate to the number of NVM

ways. First, low sensitive applications require a small number of NVM ways; therefore, the number of write accesses to NVM is largely reduced. On the contrary, highly sensitive applications show only a little change of write access because they have very little room for the DWA. To show this trend clearly, we calculate the reduction ratio of each category. For the left side applications, 92.2% reduction and 88.3% reduction in the write counts of STT-RAM and PCM ways is achieved on average, while 22.6% reduction and 55.6% reduction in the number of write accesses is achieved on average for the right side applications.

Furthermore, we combined the PTHCM with the DWA to check that it is orthogonally effective with other HCA algorithms. Since our proposal does not affect the fundamentals of operation of other HCA algorithms, the DWA can create a synergy effect. The results show that the PTHCM with the DWA (PTHCM_DWA) achieved the best results among four HCA algorithms as it showed 77.6% reduction and 80.0% reduction in write counts of NVM ways. Combining PTHCM with DWA reduces the write access to NVM more 8.9% when only DWA is applied for STT-RAM. In addition, PTHCM_DWA shows the lower NVM write counts by 11.0%. Therefore, we conclude that merging two algorithms takes advantage of both algorithms successfully.

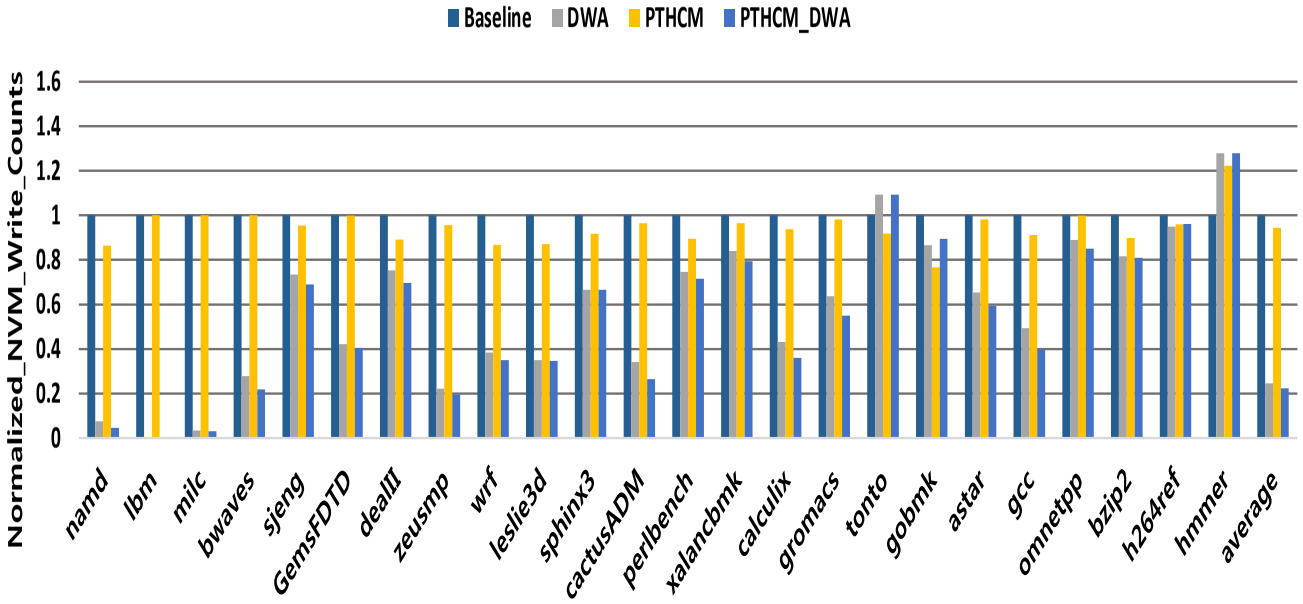


Figure 22: Normalized NVM write counts of DWA with STT-RAM.

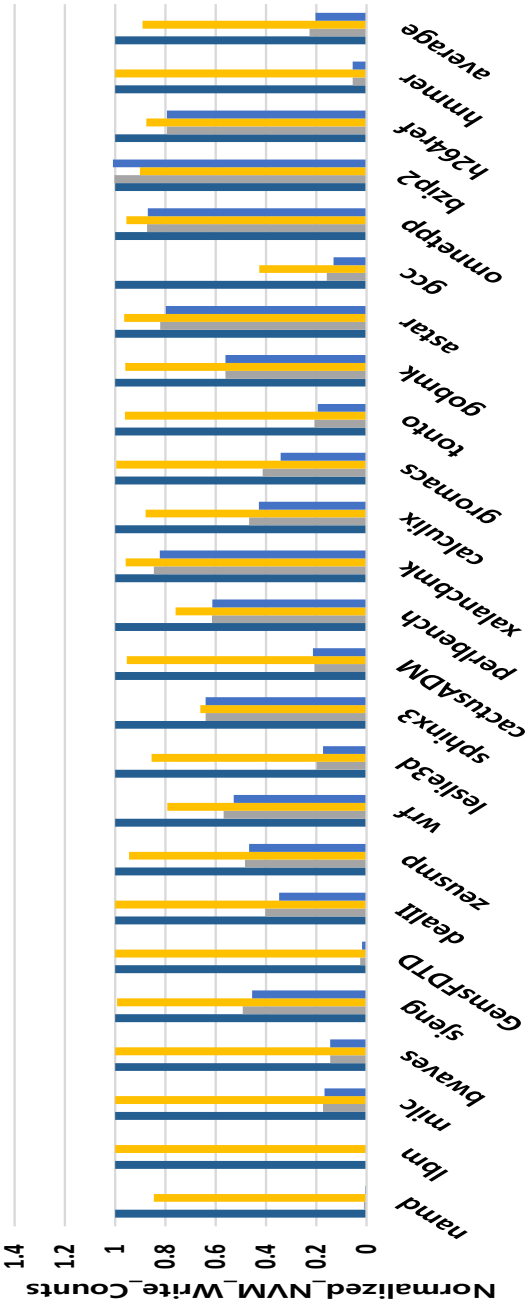


Figure 23: Normalized NVM write counts of DWA with PCM.

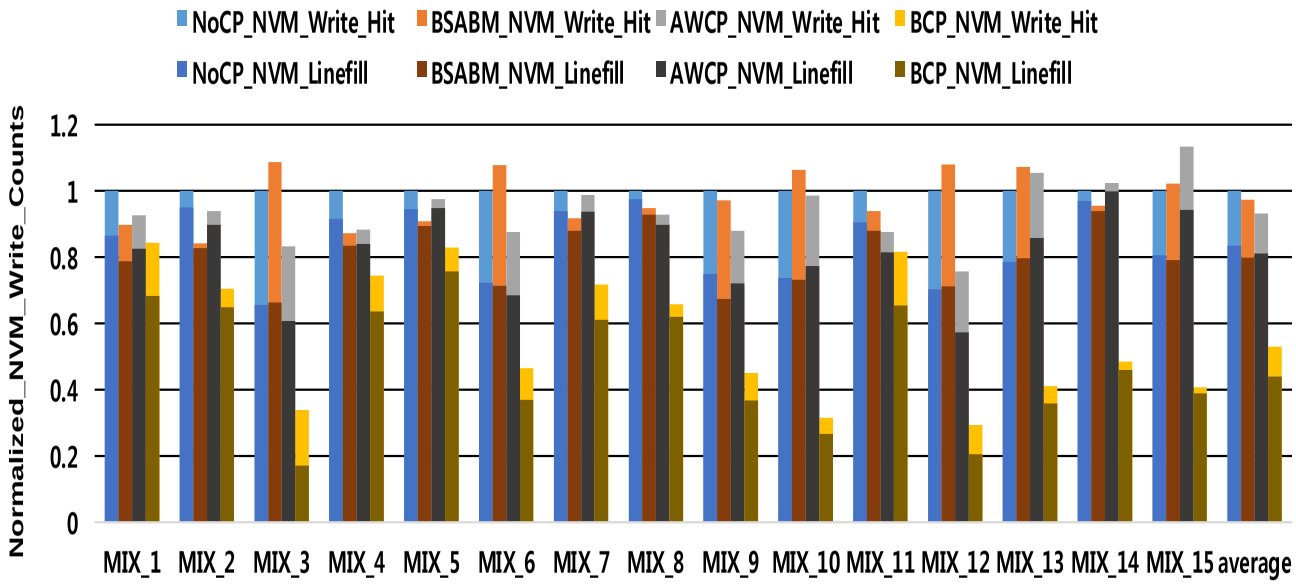


Figure 24: Normalized NVM write counts with four schemes.

Next, we analyze the NVM write counts of BSABM, AWCP, and LCP normalized to NoCP as depicted in Figure 24. The average value in the figure indicates the geometric mean of all workloads. BSABM and AWCP decreased the NVM write counts by 2.6% and 6.7%, respectively. LCP achieved a 46.9% reduction in the NVM write counts, which is the much better than previous studies. To investigate these results further, we divide the total NVM write counts into the NVM write hit counts and the NVM write linefill counts. At first, we found that the linefill operation occupies a significant portion of the NVM write counts. While the portion of the write hit counts is 16.5% on average, the portion of the NVM linefill counts is 83.5%. BSABM, AWCP, and LCP reduced the NVM write hit counts by 21.7%, 26.4%, and 39.2%, respectively. LCP shows the best results, and the previous schemes for HCA also achieved the meaningful reduction in the NVM write hit counts. On the contrary, the reduction ratio of the NVM linefill counts of BSABM and AWCP are only 4.3% and 2.8%, while LCP reduced the NVM linefill counts by 47.4%. These results confirm that LCP accomplishes the reduction in the NVM write counts by reducing the NVM linefill counts significantly as we intended.

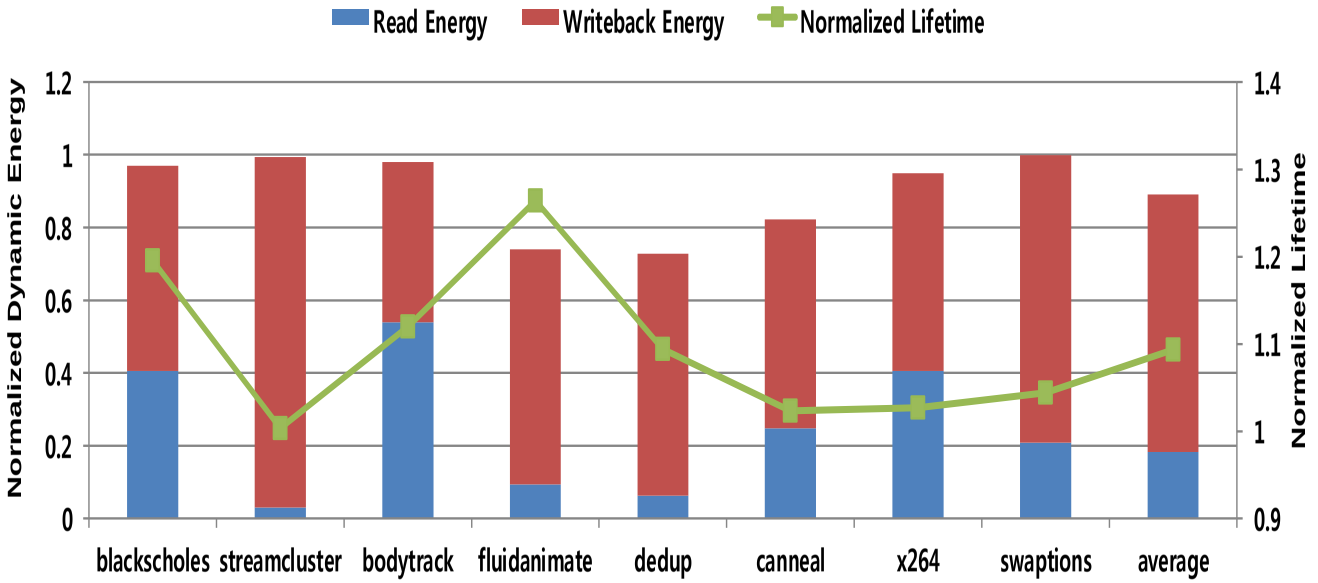


Figure 25: Normalized dynamic energy consumption and lifetime of WACC compared to the baseline MOESI protocol.

5.3 Dynamic energy consumption

We show the normalized dynamic energy consumption and lifetime in Figure 25. Since the dynamic energy in write operation dominates the dynamic energy consumption in read operation, the reduction of the write operations leads to reducing the total dynamic energy consumption. Our protocol achieves 27.1% energy savings at maximum and 10.8% energy savings on average. In addition, WACC protocol also extends the lifetime of the LLC because the lifetime of STT-RAM is inversely proportional to the number of write access to the LLC. The improvement of average write endurance in WACC protocol is 26.3% at maximum and 9.3% on average.

We investigated the normalized dynamic energy consumption compared to the baseline hybrid cache as shown in Figure 26 and Figure 27, which also present the portion of the write energy consumption of NVM over the total dynamic energy consumption. The results of HCA with STT-RAM show that the DWA achieved 26.4% reduction in the total dynamic energy consumption. The dynamic energy consumption of the PTHCM and the PTHCM_DWA was saved 2.3% and 28.4% over the baseline hybrid cache, respectively. For HCA with PCM, the DWA saved 27.4% of dynamic energy consumption, while the PTHCM and the PTHCM_DWA reduced the dynamic energy consumption by 2.7% and 30.0%. The trend of reduction is similar to that of reduction in the write accesses. This is because the dynamic energy consumption is mainly affected by the write accesses to STT-RAM.

Based on the observation of these figures, the write energy consumption of NVM occupies a significant portion of the total dynamic energy consumption. In the baseline hybrid cache, 78.6% and 56.0% of the dynamic energy was consumed due to the write accesses to STT-RAM and PCM ways. Therefore, we conclude that the number of write accesses to NVM ways is the most important factor for dynamic energy consumption. The results show that the portion of write dynamic energy of NVM ways was reduced to 32.8% and 14.7% in the DWA. The dynamic energy consumption of NVM write operations of the PTHCM occupies 74.3% and 48.8% of the total dynamic energy consumption. For the PTHCM_DWA, the portion is reduced to 30.0% and 14.1%. The reduction trend is also similar to that of the write access reduction. Therefore, the reduction in the dynamic energy consumption mainly comes from the reduction of the write energy consumption of NVM.

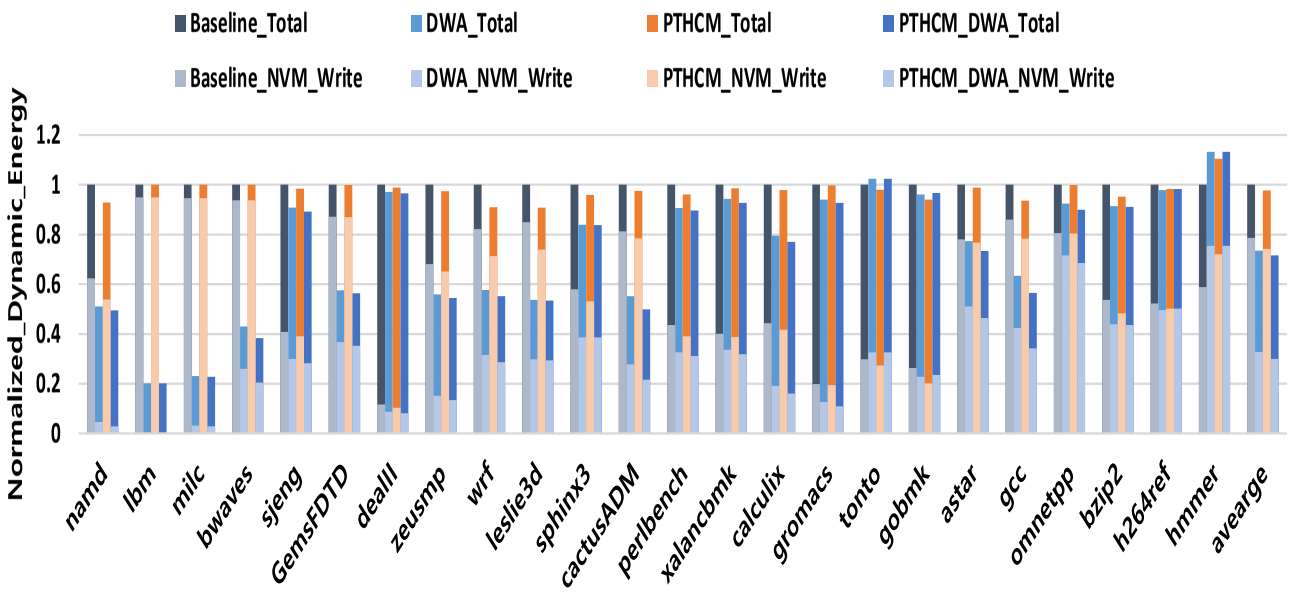


Figure 26: Normalized dynamic energy consumption of DWA with STT-RAM.

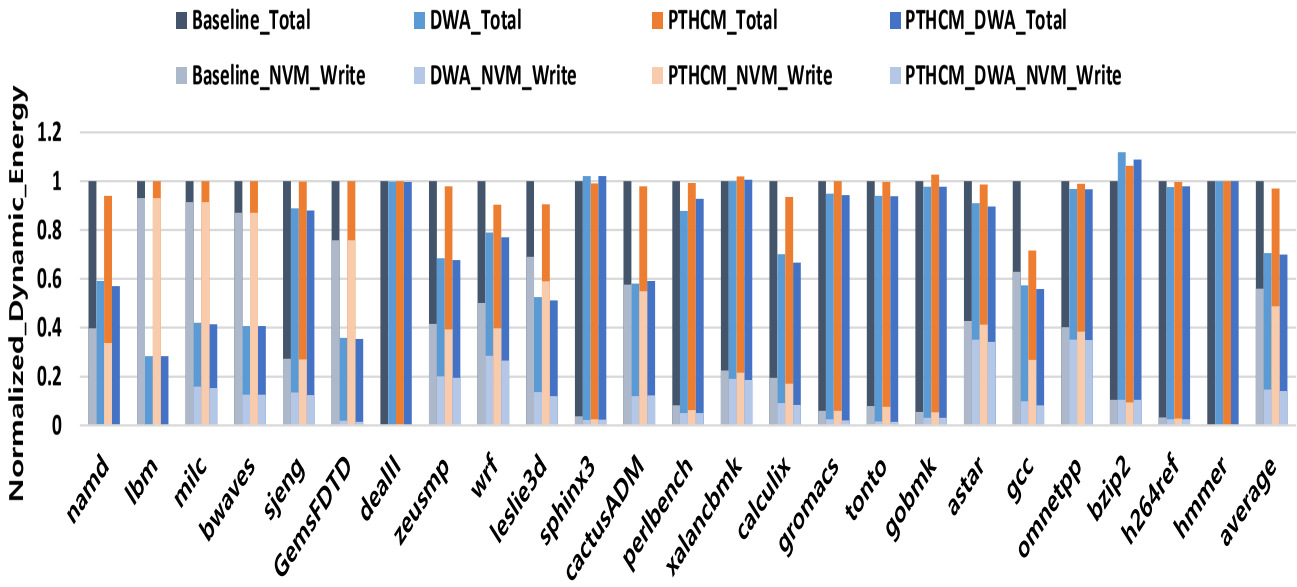


Figure 27: Normalized dynamic energy consumption of DWA with PCM.

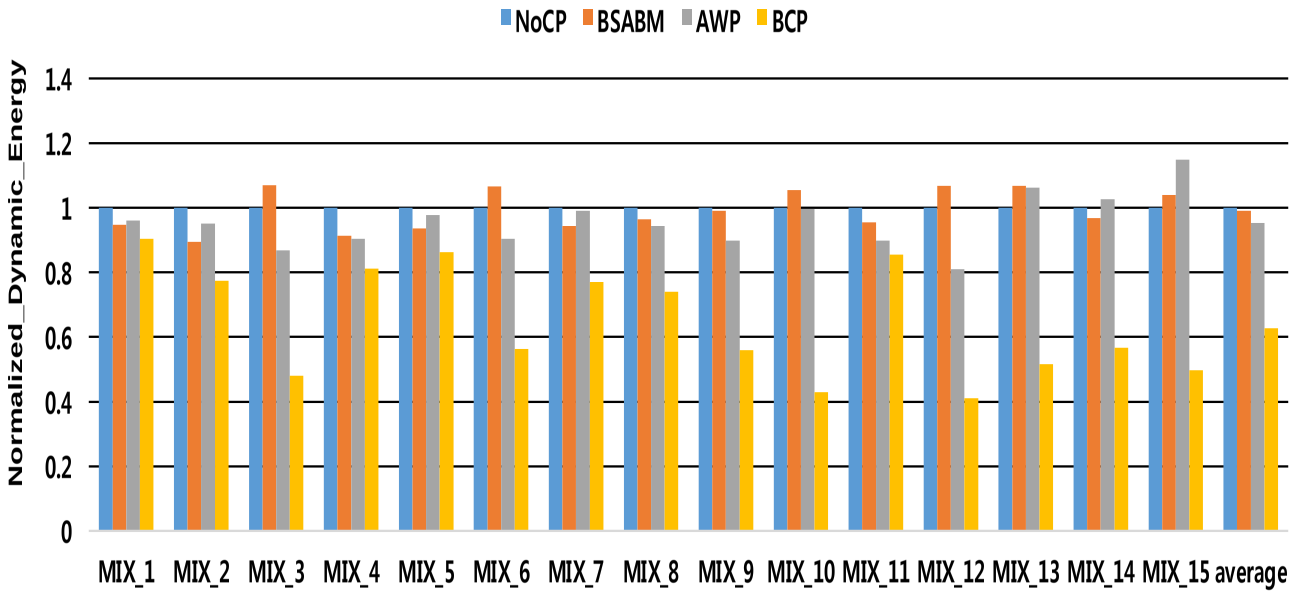


Figure 28: Normalized dynamic energy consumption compared to NoCP.

The normalized dynamic energy consumption of four schemes are presented in Figure 28. LCP saved 37.2%, 36.6%, and 34.1% of dynamic energy consumption over NoCP, BSABM, and AWCP, respectively. The trends of the dynamic energy reduction are similar to those of the normalized NVM write counts, while the variation is small. For MIX_12, the dynamic energy consumption is reduced by nearly 60% compared to AWCP at maximum, while the difference between AWCP and LCP is less than 1% for MIX_1. The reason for this similarity is that the NVM write counts is a main contributor to the total energy consumption; thus, reducing the number of NVM write accesses to the LLC highly influenced the total dynamic energy consumption.

5.4 Lifetime

We estimated the normalized lifetime as shown in Figure 29 and Figure 30. There is a general consensus among researchers that PCM has a limited lifetime. However, opinions are different about the write endurance of STT-RAM. Many studies assume that its write endurance is high enough, and thus they set aside the lifetime problem. On the other hand, another group argues that the assumption is unrealistic [19, 56]. Since determining the correctness of their claims is not the focus in the thesis, the results of both types of NVM are presented.

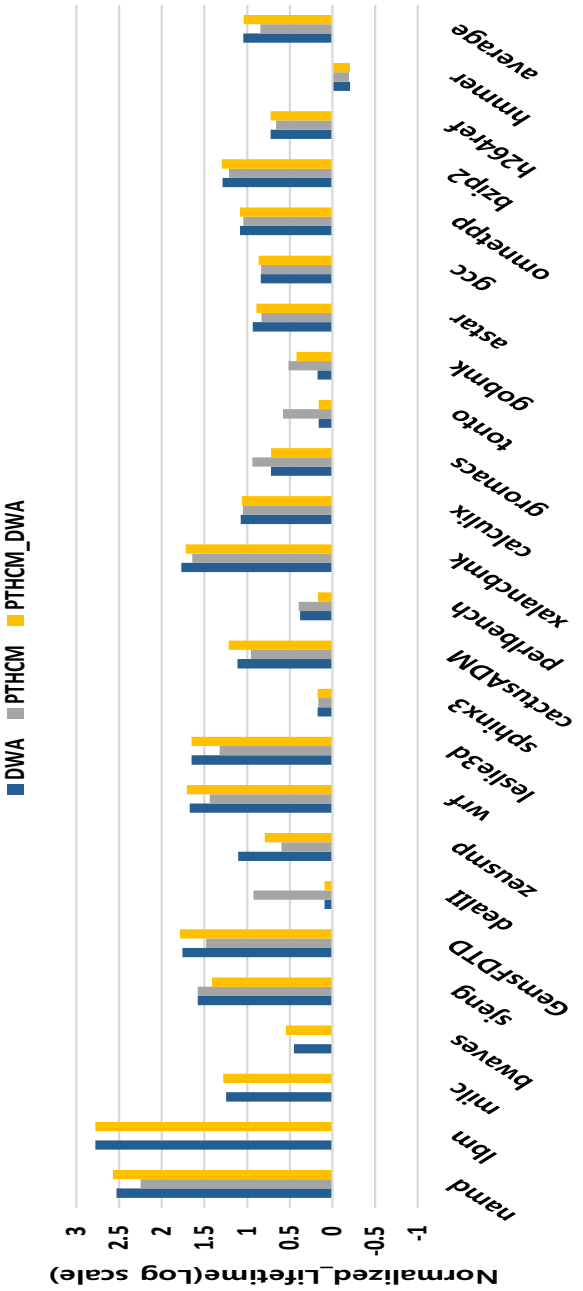


Figure 29: Normalized lifetime of DWA with STT-RAM.

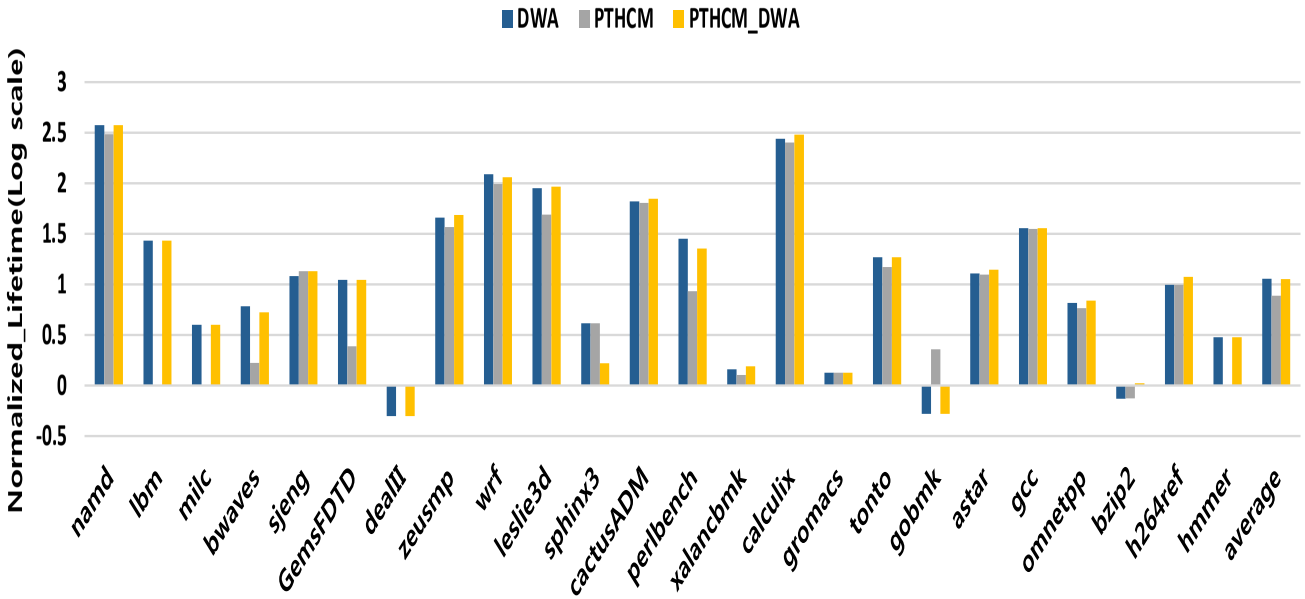


Figure 30: Normalized lifetime of DWA with PCM.

Notice that the results of two figures are presented in log scale because the lifetime of some applications were extended significantly. Especially, the write endurance of *namd* and *lbm* was increased by more than 300 times. For these applications, the number of replaceable ways was almost always less than the number of SRAM ways. Since NVM ways were rarely used in the DWA, the lifetime soared up. The PTHCM_DWA extended the lifetime by 10.9 times and 11.3 times for HCAs with STT-RAM and PCM, respectively.

To confirm that our proposal does not increase the miss rate significantly, we present the miss rates of each HCA configuration compared to the baseline hybrid cache in Figure 31 and Figure 32. The miss rate of the DWA was increased only by 1.8% and 1.9% for HCAs with STT-RAM and PCM, respectively, while the PTHCM decreased the miss rate by 1%. Since the PTHCM did not improve the miss rate meaningfully, the miss rate of the PTHCM_DWA followed the miss rate of the DWA. Therefore, the miss rates of the DWA and the PTHCM_DWA are very similar and the PTHCM_DWA increased the miss rate by 1.9% and 1.9% on average which are the same values of the DWA. As expected, this result confirms that our proposed algorithm does not significantly increase the miss rate.

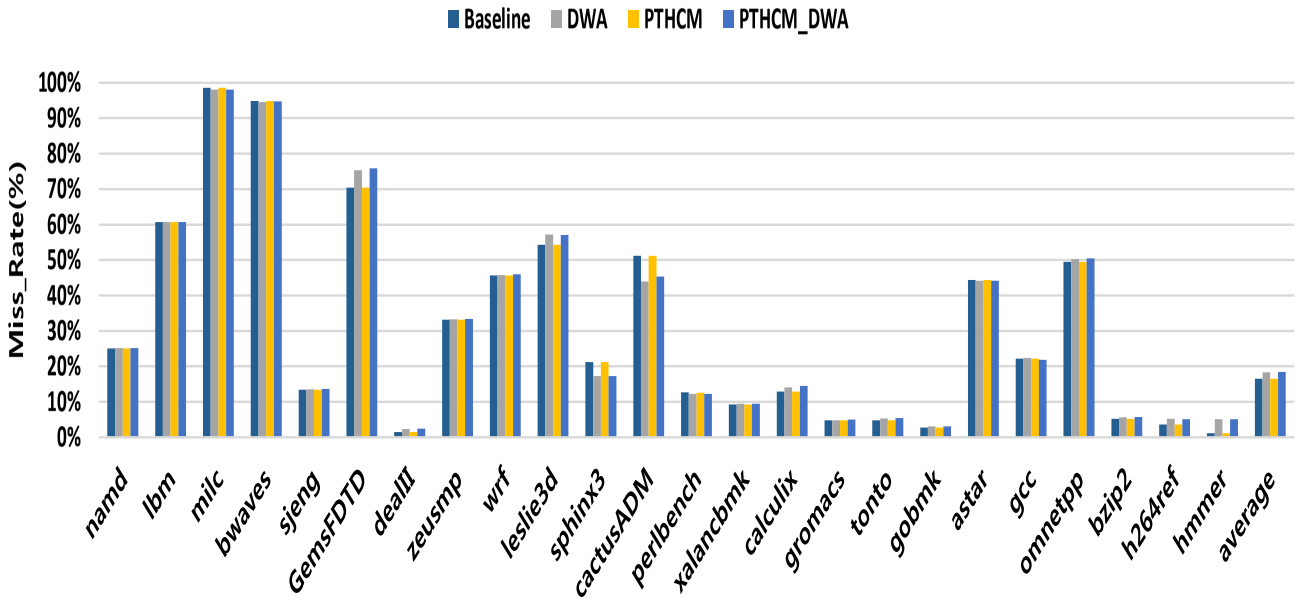


Figure 31: Miss rates with various HCA confiruations with STT-RAM.

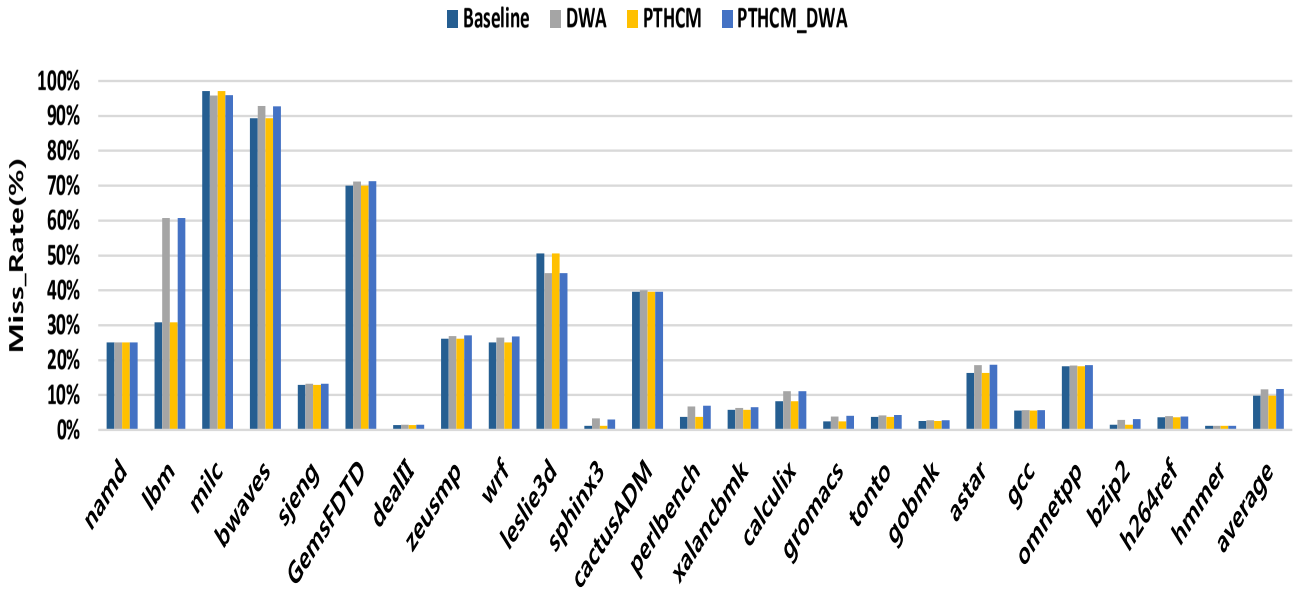


Figure 32: Miss rates with various DWA configurations with PCM.

5.5 Multi-core environment

We investigated several metrics for multi-core environments as shown in Figure 33 and Figure 34. For multi core system simulation, we generated ten workloads by mixing six applications as listed in Table 12. The two benchmarks for low sensitivity are *bwaves* and *calculix*, while *hmmer* and *h264ref* represent high sensitivity. Other two benchmarks such as *wrf* and *gromacs* are selected as the middle range of sensitive programs.

First of all, a significant reduction in the write accesses was achieved in both HCA configurations. The DWA removed 80.7% of write accesses on average, while the average write reduction ratio of six benchmarks is 61.3% for HCA with STT-RAM in single-core environments. This result means that our proposal has the extendibility for the multi-core system. In case of HCA with PCM, the average reduction ratio of multi-core results is 59.4%, while each application removed 76.3% of write accesses on average. Even though the results of HCA with PCM are less impressive compared to HCA with STT-RAM, our proposal still removed a great deal of unnecessary NVM write operations. The results of dynamic energy consumption are consistent with the trend of the write accesses to NVM. For HCAs with STT-RAM and PCM, 55.5% and 33.7% of dynamic energy consumption were saved, respectively. The lifetime was prolonged by 1.76 times and 1.35 times on average.

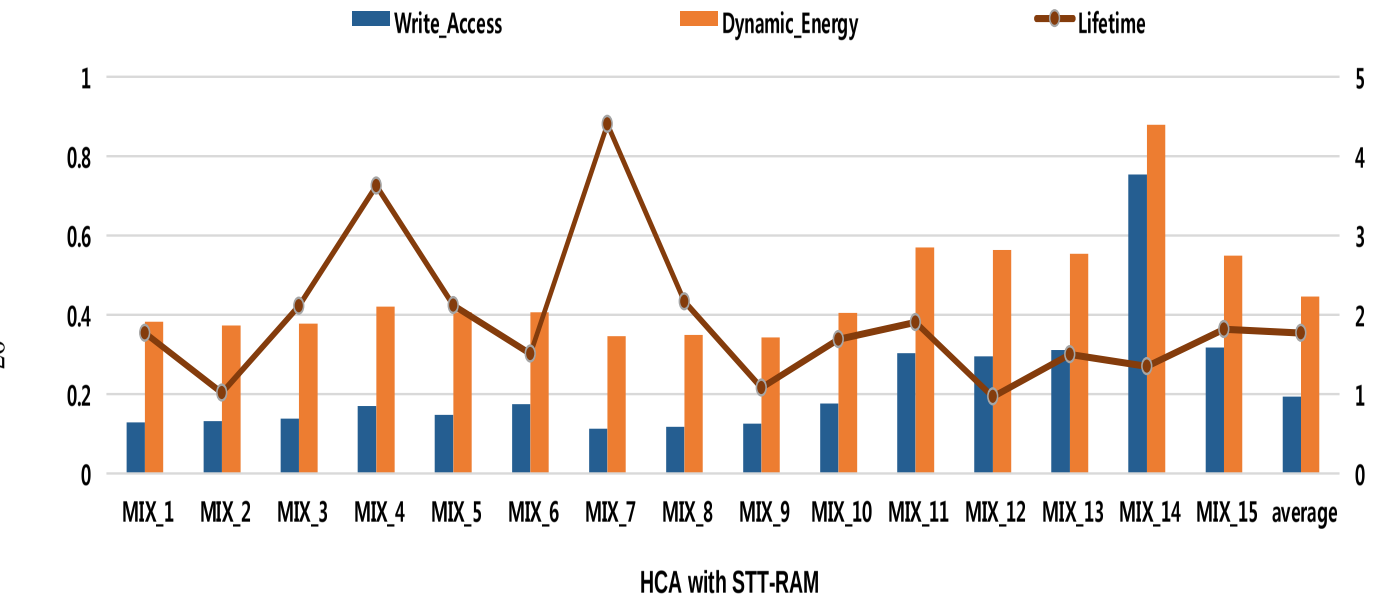


Figure 33: Normalized write access, dynamic energy consumption, and lifetime of HCA with STT-RAM with the multi-core workloads.

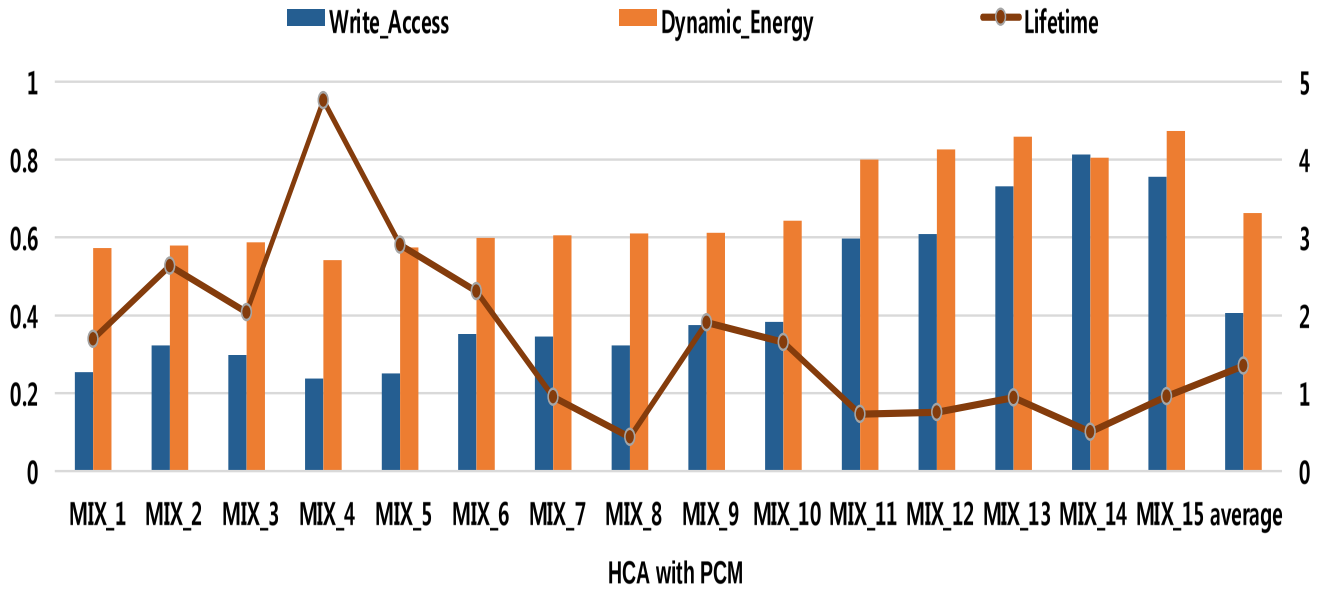


Figure 34: Normalized write access, dynamic energy consumption, and lifetime of HCA with PCM with the multi-core workloads.

To represent the performance improvement in a multi-core environment, three metrics usually are presented – nstruction per cycle (IPC) throughput, weighted speedup, and fairness – which have their own purposes [57]. They usually are defined as follows:

$$IPC\ throughput = \sum_{i=1}^{n} IPC_i \tag{5.1}$$

$$Weighted\ Speedup = \sum_{i=1}^{n} \frac{IPC_i^{MP}}{IPC_i^{SP}} \tag{5.2}$$

$$Fairness = \frac{n}{\sum_{i=1}^{n} \frac{IPC_i^{SP}}{IPC_i^{MP}}} \tag{5.3}$$

where IPC_i^{SP} is the IPC of ith program under single program mode (SP) and IPC_i^{MP} is the IPC under multi-program mode (MP). IPC throughput is simply and intuitively defined as the sum of the IPCs of the all applications. The weighted speedup is proposed to equalize the contribution of programs using normalized IPCs [58]. Luo et al. argued that harmonic mean is more suitable to represent the fairness than weighted speedup [59].

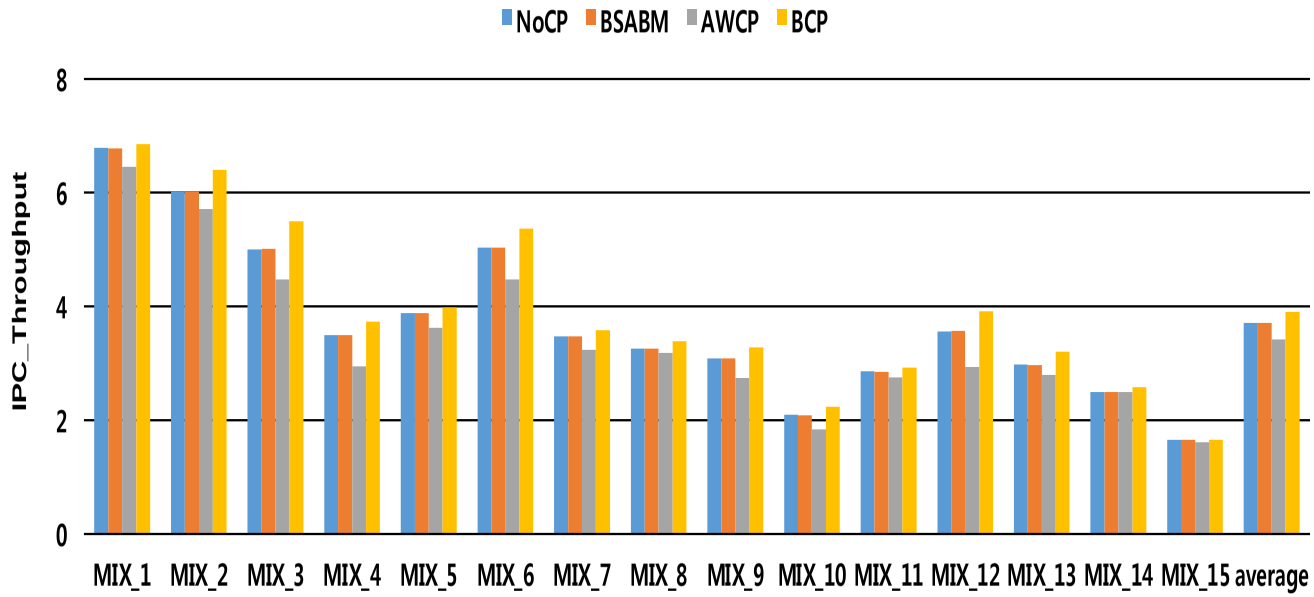


Figure 35: IPC throughput with four schemes.

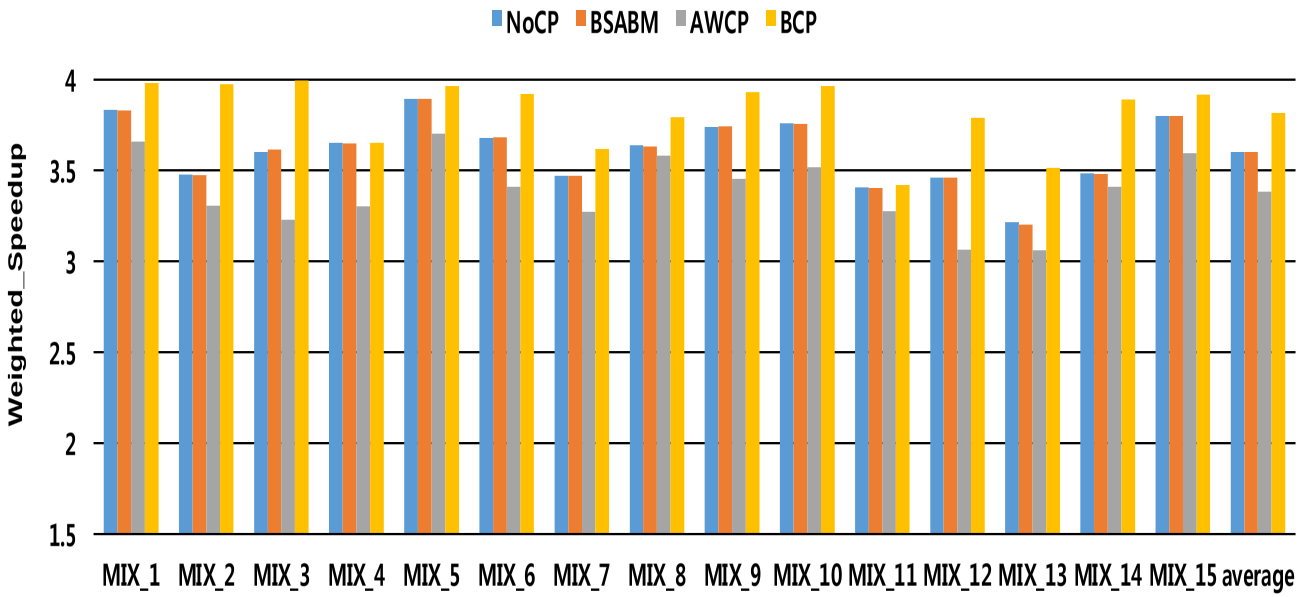


Figure 36: Weighted speedup with four schemes.

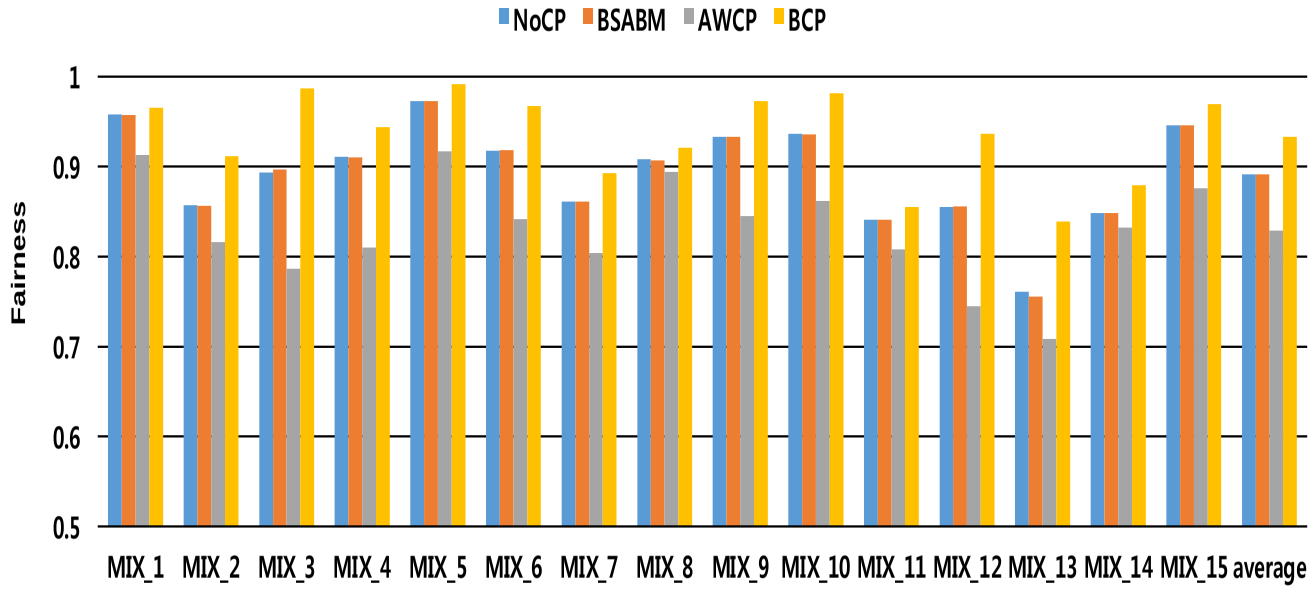


Figure 37: Fairness with four schemes.

Therefore, we plot three metrics in Figure 35, Figure 36, and Figure 37 for different schemes. LCP outperforms NoCP and AWCP by 5.0% and 14.3% in terms of IPC throughput as depicted in Figure 35. In addition, our scheme improved the weighted speedup by 5.6% and 11.4% for NoCP and AWCP as shown in Figure 36. Finally, Figure 37 compares the fairness improvement for four schemes; the fairness of LCP is improved to 0.93, while NoCP and AWCP have 0.89 and 0.83, respectively. The IPC throughput improvement is maximized for MIX_3, whereas MIX_2 shows the best weighted speedup improvement compared to AWCP. The fairness of the applications of MIX_12 is most increased.

Chapter 6

Conclusion

6.1 Conclusion

In the thesis, three proposals have been provided to compensate for identified weaknesses of NVM: write avoidance cache coherence protocol (WACC), dynamic way adjusting scheme (DWA), and linefill-aware cache partitioning (LCP).

We proposed a novel cache coherence protocol to eliminate useless write operations of LLC for a multi-core system. Based on the analysis of the existing protocols, it was found that they generated useless write accesses to the LLC during the linefill operation. Thus, our protocol,which is called WACC, modifies the cache states without storing the block data during linefill. This write policy reduced the number of write access attempts to the LLC, which led to improvements in the energy consumption and lifetime. The simulation result showed that the reduction of maximum energy consumption in WACC protocol is 27.1% and the lifetime extension is 26.3% at maximum in STT-RAM based LLC.

The thesis introduced the concept of an NVM capacity management policy for reducing the number of write accesses to NVM. This policy is implemented by two methods called dynamic way adjusting scheme (DWA) and linefill-aware cache partitioning (LCP). DWA dynamically resized the number of active NVM ways to improve the dynamic energy consumption and the lifetime of the components. To adjust the number of NVM ways, the maximum stack distance is dynamically monitored and rearranging of the replaceable NVM ways is regularly performed. The proposed policy reduced the number of write accesses to STT-RAM by about 77.6% and PCM by 79.6%. The results also showed that HCAs with STT-RAM and PCM achieves 30.0% reduction and 28.4% in dynamic energy consumption. The lifetime of the two HCAs was prolonged by 10.9 times and 11.3 times over a conventional hybrid cache system. Both HCAs can achieve these improvements without any meaningful miss rate increment. While the portion of the NVM linefill operations, over the write counts, is about 83.5% in our experimental results, previous studies have not considered the linefill operations to NVM in CMP environments during partitioning.

We also proposed LCP, to minimize the NVM write counts, in consideration of the NVM linefill counts, as well as the NVM write hit counts. In the thesis, three kinds of metrics were introduced to analyze the efficiency of adjusting the cache partitioning; if a core gets or loses ways, how many the miss counts, write counts, and NVM write counts are changed. A cache partitioning algorithm for LCP is proposed to provide the best partitioning through a two-step approach based on these metrics. We have shown that

the proposed LCP predicts the NVM write counts with less than a 5% error rate and reduces the dynamic energy consumption by 34.1% on average with improved performance.

6.2 Future work

We will extend the findings of thesis in two ways. First, we plan to combine our proposal with schemes for non-uniformity of write operations among sets which are inspired that the write varies across different cache sets. They separated the physical mapping and logical mapping of cache sets and stored data between sets. The key idea is decent, but there is a pitfall to simply merge LCP with the inter-set variation wear leveling scheme (ISWLs). Since the data is possible to be placed in a different set, they violate the stack property which our scheme is based on. Keeping track of all recency position of remapped blocks would not be a feasible method because it needs a significant area overhead and consumes a lot of dynamic energy. Hence, we are developing a new method to efficiently bond LCP and ISWLs.

In addition, we will consider combining data bypassing techniques to the proposed scheme. Even though cache bypassing techniques are apparently promising schemes for NVM, they cannot be directly applied to our mechanism because the inclusion property is not maintained in most of their schemes. We will investigate a new scheme that both keeps inclusion property and utilizes the bypass schemes.

Bibliography

[1] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, *et al.*, "A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram," in *Proceedings of IEEE International Electron Devices Meeting*, pp. 459–462, IEEE, 2005.

[2] H. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[3] N. Yamada, E. Ohno, K. Nishiuchi, N. Akahira, and M. Takao, "Rapid-phase transitions of gete-sb2te3 pseudobinary amorphous thin films for an optical disk memory," *Journal of Applied Physics*, vol. 69, no. 5, pp. 2849–2856, 1991.

[4] A. Driskill-Smith, S. Watts, D. Apalkov, D. Druist, X. Tang, Z. Diao, X. Luo, A. Ong, V. Nikitin, and E. Chen, "Non-volatile spin-transfer torque ram (stt-ram): An analysis of chip data, thermal stability and scalability," in *Proceedings of IEEE International Memory Workshop*, pp. 1–3, IEEE, 2010.

[5] T. Sumi, Y. Judai, K. Hirano, T. Ito, T. Mikawa, M. Takeo, M. Azuma, S.-i. Hayashi, Y. Uemoto, K. Arita, *et al.*, "Ferroelectric nonvolatile memory technology and its applications," *Japanese Journal of Applied Physics*, vol. 35, no. 2S, p. 1516, 1996.

[6] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[7] J. H. Choi, J. W. Kwak, and C. S. Jhon, "Write avoidance cache coherence protocol for non-volatile memory as last-level cache in

chip-multiprocessor," *IEICE Transactions on Information and Systems*, vol. 97, no. 8, pp. 2166–2169, 2014.

[8] J. H. Choi and G. H. Park, "Demand-aware nvm capacity management policy for hybrid cache architecture," *Computer Journal*, advance online publication, 2015, doi:10.1093/comjnl/bxv103.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[10] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pp. 72–81, ACM, 2008.

[12] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[13] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[14] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[15] S. Lee, K. Kang, and C.-M. Kyung, "Runtime thermal management for 3-d chip-multiprocessors with hybrid sram/mram l2 cache," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 23, no. 3, pp. 520–533, 2014.

[16] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie, "Power and performance of read-write aware hybrid caches with non-volatile memories," in *Proceedings of International Conference on Design, Automation and Test in Europe*, pp. 737–742, IEEE, 2009.

[17] J. Li, L. Shi, C. J. Xue, C. Yang, and Y. Xu, "Exploiting set-level write non-uniformity for energy-efficient nvm-based hybrid cache," in *Proceedings of International Symposium on Embedded Systems for Real-Time Multimedia*, pp. 19–28, IEEE, 2011.

[18] B. Quan, T. Zhang, T. Chen, and J. Wu, "Prediction table based management policy for stt-ram and sram hybrid cache," in *Proceedings of International Conference on Computing and Convergence Technology*, pp. 1092–1097, IEEE, 2012.

[19] J. Ahn, S. Yoo, and K. Choi, "Write intensity prediction for energy-efficient non-volatile caches," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 223–228, IEEE, 2013.

[20] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 34–45, ACM, 2009.

[21] J. H. Choi, J. W. Kwak, S. T. Jhang, and C. S. Jhon, "Adaptive cache compression for non-volatile memories in embedded system," in *Proceedings of International Conference on Research in Adaptive and Convergent Systems*, pp. 52–57, ACM, 2014.

[22] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad, "High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 79–84, IEEE, 2011.

[23] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi, "i 2 wap: Improving non-volatile cache lifetime by reducing inter-and intra-set write vari-

ations," in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 234–245, IEEE, 2013.

[24] Y.-T. Chen, J. Cong, H. Huang, C. Liu, R. Prabhakar, and G. Reinman, "Static and dynamic co-optimizations for blocks mapping in hybrid caches," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 237–242, ACM, 2012.

[25] Y. Li, Y. Chen, and A. K. Jones, "A software approach for combating asymmetries of non-volatile memories," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 191–196, ACM, 2012.

[26] Q. Li, M. Zhao, C. J. Xue, and Y. He, "Compiler-assisted preferred caching for embedded systems with stt-ram based hybrid cache," *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 109–118, 2012.

[27] K. Qiu, M. Zhao, C. Fu, L. Shi, and C. J. Xue, "Migration-aware loop retiming for stt-ram based hybrid cache for embedded systems," in *Proceedings of International Conference on Application-Specific Systems, Architectures and Processors*, pp. 83–86, IEEE, 2013.

[28] Y. Li, Y. Zhang, H. Li, Y. Chen, and A. K. Jones, "C1c: A configurable, compiler-guided stt-ram l1 cache," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 4, p. 52, 2013.

[29] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 24–33, 2009.

[30] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montaño, "Improving read performance of phase change memories via write cancellation and write pausing," in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 1–11, IEEE, 2010.

[31] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, 2012.

[32] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *Proceedings of International Conference on Computer Design*, pp. 337–344, IEEE, 2012.

[33] M. Zhou, Y. Du, B. Childers, R. Melhem, and D. Mossé, "Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, p. 53, 2012.

[34] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: a hybrid pram and dram main memory system," in *Proceedings of Internaional Conference on Design Automation Conference*, pp. 664–669, IEEE, 2009.

[35] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing pcm main memory lifetime," in *Proceedings of Internaional Conference on Design, Automation and Test in Europe*, pp. 914–919, IEEE, 2010.

[36] W. Zhang and T. Li, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pp. 101–112, IEEE, 2009.

[37] H. Seok, Y. Park, and K. H. Park, "Migration based page caching algorithm for a hybrid main memory of dram and pram," in *Applied Computing, International Symposium on*, pp. 595–599, ACM, 2011.

[38] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[39] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Microarchitecture, IEEE/ACM International Symposium on*, pp. 423–432, IEEE Computer Society, 2006.

[40] A. Samih, Y. Solihin, and A. Krishna, "Evaluating placement policies for managing capacity sharing in cmp architectures with private caches," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 3, p. 15, 2011.

[41] C. CaBcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of International Conference on Supercomputing*, pp. 150–159, ACM, 2003.

[42] Y. Liu and W. Zhang, "Exploiting stack distance to estimate worst-case data cache performance," in *Proceedings of International Symposium on Applied Computing*, pp. 1979–1983, ACM, 2009.

[43] "The intel 64 and ia-32 architectures software developer's manual." http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf. accessed 3-Mar-2014.

[44] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[45] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.

[46] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 167–178, 2006.

[47] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie, "Adaptive placement and migration policy for an stt-ram-based hybrid cache,"

[48] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2011.

[49] J. Li, C. J. Xue, and Y. Xu, "Stt-ram based energy-efficiency hybrid cache for cmps," in *Proceedings of International Conference on VLSI and System-on-Chip*, pp. 31–36, IEEE, 2011.

[50] S.-M. Syu, Y.-H. Shao, and I.-C. Lin, "High-endurance hybrid cache design in cmp architecture with cache partitioning and access-aware policy," in *Proceedings of International Conference on Great Lakes Symposium on VLSI*, pp. 19–24, ACM, 2013.

[51] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for stt-ram using early write termination," in *Proceedings of International Conference on Computer-Aided Design-Digest of Technical Papers*, pp. 264–268, IEEE, 2009.

[52] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3d stacked mram l2 cache for cmps," in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 239–249, IEEE, 2009.

[53] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation," in *Proceedings of International Symposium on Microarchitecture*, pp. 81–92, IEEE Computer Society, 2004.

[54] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, pp. 190–200, ACM, 2005.

[55] "Arm cortex-a57 processor." "`http://www.arm.com/products/processors/cortex-a/cortex-a57-processor.php`" (accessed 1-Sep-2015).

[56] J. Wang, Y. Tim, W.-F. Wong, Z.-L. Ong, Z. Sun, and H. H. Li, "A coherent hybrid sram and stt-ram l1 cache architecture for shared memory multicores.," in *Proceeding of Asia and South Pacific Design Automation Conference*, pp. 610–615, IEEE, 2014.

[57] L. Eeckhout, "Computer architecture performance evaluation methods," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–145, 2010.

[58] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous mutlithreading processor," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 234–244, 2000.

[59] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors.," in *Performance Analysis of Systems and Software, International Symposium on*, pp. 164–171, IEEE, 2001.

초록

비휘발성 메모리 기반의 최종 레벨 캐시를 위한 쓰기 회피 기법

비휘발성 메모리는 높은 집적성과 낮은 정적 전력 소모량이라는 특성으로 인해 최종 레벨 캐시로 사용되기에 유력한 기술로 떠오르고 있다. 그러나 비휘발성 메모리는 쓰기 작업을 위해 많은 전력과 시간을 소모하고, 제한된 수명을 가진다는 단점이 있기 때문에 이를 보완하기 위한 방법이 없다면 최종 레벨 캐시로 사용되기 어렵다. 본 논문에서 비휘발성 메모리의 단점을 보완하기 위해 쓰기 회피 기법들을 제시하였다. 먼저, 멀티 코어 환경에서 쓰기 횟수를 줄이기 위한 캐시 일관성 정책(Write avoidance cache coherence protocol)을 제시하였고, 이종 캐시 구조(Hybrid cache architecture) 에서 쓰기 회수를 최소화하기 위한 2가지 기법을 제안하였다. 첫번째 기법은 NVM way을 동적으로 조정하는 방식이며(Dynamic way adjusting), 다른 기법은 linefill을 고려한 캐시 분할 기법(Linefill-aware cache partitioning)이다.

우선 본 논문에서는 쓰기 횟수를 줄이기 위한 새로운 캐시 일관성 정책을 제안한다. 새로운 정책을 사용하는 시스템에서는 상위 레벨 캐시에 동일한 데이터가 있는 경우, 최종 레벨 캐시에서는 태그 정보만 저장하고 데이터 정보는 기록하지 않는다. 따라서 상위 레벨 캐시에서 쓰기 수정이 일어났을 때, 불필요한 쓰기를 줄일 수 있게 된다.

다음으로 이종 캐시 구조 환경하에서 비휘발성 메모리의 크기를 제한하여 쓰기 횟수를 줄이는 기법을 제안한다. 이종 캐시 구조는 비휘발성 메모리의 일부를 휘발성 메모리인 SRAM로 교체하여 두 가지 종류의 메모리가 하나의 캐시에 존재하는 구조이다. 통계적으로 비휘발성 메모리의 way의 비율이 많아질수록 전체 쓰기 작업에서 비휘발성 메모리의 쓰기 작업의 비율 또한 커지게 된다. 그런데 모든 프로그램이 항상 전체 메모리를 요구하는 것은 아니다. 프로그램에 따라서 또는 실행 시간에 따라서 메모리의 일부만을 요구할 때도 있다. 그러한 경우에는 필요한 만큼만 비휘발성 메모리를 사용하도록 메모리의 크기를 제한한다면 성능의 저하 없이 비휘발성 메모리의 쓰기 횟수를 줄일 수 있다.

또한, 본 논문에서는 이종 캐시 구조를 사용하는 멀티 코어 시스템에서 비휘발성 메모리의 쓰기 회수를 최소화하는 캐시 분할(Cache partitioning)을 제안한다. 기존의 캐시 분할 방식들은 휘발성 메모리를 사용한 동종 캐시 구조를 사용하기 때문에, 각 코어에 할당할 way의 수만 계산하였다. 그러나 이종 캐시 구조에서는 각 코어가 사용할 전체 way의 수뿐만 아니라 비휘발성 메모리 way의 수와 휘발성 메모리 way의 수를 따로 구해야 한다. 그렇지 않으면 휘발성 메모리 way가 비효율적으로 코어에 분배되어, 전체적인 비휘발성 메모리의 쓰기 회수가 최적화되지 않는다. 따라서, 본 논문에서는 일정한 주기마다 캐시 분할 방식을 바꾸어 가면서 비휘발성 메모리의 쓰기 회수를 최소화하는 캐시 분할 구성을 찾아낸다.

실험을 수행한 결과, Write avoidance cache coherence protocol을 적용하게 되면 전력 소모량은 13.2%가 감소하며, Dynamic way adjusting와 Linefill-aware cache partitioning을 적용하는 경우 각각 전력 소모량이 26.9%와 37.2% 감소하였다.

공학박사학위논문

Write Avoidance Schemes for Non-Volatile Memory based Last-Level Cache

비휘발성 메모리 기반의 최종 레벨 캐시를 위한 쓰기 회피 기법

2016년 2월

서울대학교 대학원
전기·컴퓨터공학부
최주희

ABSTRACT

Non-volatile memory (NVM) is considered to be a promising memory technology for last-level caches (LLC) due to its low leakage of power and high storage density. However, NVM has some drawbacks including high dynamic energy when modifying NVM cells, long latency for write operations, and limited write endurance. To overcome these problems, the thesis focuses on two approaches: cache coherence and NVM capacity management policy for hybrid cache architecture (HCA).

First, we review existing cache coherence protocols under the condition of NVM-based LLCs. Our analysis reveals that the LLCs perform unnecessary write operations because legacy protocols have very pay little attention to reducing the number of write accesses to the LLC. Therefore, a write avoidance cache coherence protocol (WACC) is proposed to reduce the number of write operations to the LLC.

In addition, novel HCA schemes are proposed to efficiently utilize SRAM in the thesis. Previous studies on HCA have concentrated on detecting write-intensive blocks and placing them into the SRAM ways. However, unlike other studies, a dynamic way adjusting algorithm (DWA) and a linefill-aware cache partitioning (LCP) calculate the optimal size of NVM ways and SRAM ways in order to minimize the NVM write counts and assigning the corresponding number of NVM ways and SRAM ways to cores.

The simulation results show that WACC achieves a 13.2% reduction in the dynamic energy consumption. For HCA schemes, the dynamic energy consumption of DWA and LCP is reduced by 26.9% and 37.2%, respectively.

Index Terms : Cache memories, Emerging technologies, Heterogeneous (hybrid) memory systems , Low-power design, Cache coherence, Cache partitioning

Student Number : 2012-30234

CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose of the thesis

The purpose of the thesis is to reduce the write counts of LLC to overcome drawbacks of NVM. To this end, three schemes are proposed in the thesis: write avoidance cache coherence protocol (WACC), dynamic way adjusting scheme (DWA), and linefill-aware cache partitioning (LCP).

Non-volatile memory (NVM) has been investigated as a resource to replace volatile memories such as SRAM or DRAM since their tendency to waste energy has grown to a substantial portion of total energy consumption [1, 2, 3, 4, 5, 6]. With conventional memory, static power is dissipated by transistors even when they make no switching. On the contrary, NVM adopts its own material as memory storage, instead of an electric charge, which limits leakage power dissipation.

However, there are some drawbacks to be considered when employing NVM as last level cache (LLC) directly: inefficient write operations and limited write endurance. Changing values in NVM requires long operating time and high level current. Thus, write operations generate long latency and

high dynamic energy consumption in the NVM cache system. Moreover, an NVM cell is worn out after a limited number of writing. Therefore, the lifetime of the NVM based cache is shorter than that of the SRAM cache due to the write limitation.

To overcome these drawbacks, the thesis introduces a new cache coherence protocol to reduce the write operations of the LLC [7]. The block data of the LLC is updated only if the cache block is written-back from a private cache, which leads to avoiding useless write operations in the LLC.

In addition, it is found that the previous researchers have overlooked that the capacity of NVM is also one of important factors affecting the number of write accesses to NVM. This discovery leads to the necessity of NVM capacity management policy such that the size of NVM is dynamically adjusted according to the demand of applications. To implement the idea, we propose a dynamic way adjusting (DWA) algorithm which dynamically monitors the optimal number of NVM ways using the stack property and disabling the unnecessary NVM ways [8].

Finally, the thesis proposes a cache partitioning scheme called linefill-aware cache partitioning (LCP) mechanism, taking into account the NVM linefill counts as well as the NVM write hit counts during cache partitioning. Most previous works have concentrated on managing write-intensive blocks by allocation these blocks to SRAM to reduce the number of the write operations to NVM. However, those schemes have not considered that reducing the number of linefill operations to NVM is important to reduce the

total number of write operations to NVM. To overcome this weakness, an algorithm for cache partitioning of LCP considers the NVM linefill counts.

The proposed schemes are simulated with the gem5 simulator [9] for WACC and macsim [10] for DWA and LCP. We used the PARSEC benchmark suite [11] for evaluating WACC and SPEC CINT2006 and SPEC CFP2006 of the SPEC CPU2006 benchmark suite [12] for DWA and LCP. The experitmental results show that WACC achieves a 13.2% reduction in the dynamic energy consumption. For HCA schemes, the dynamic energy consumption of DWA and LCP are reduced by 26.9% and 37.2%, respectively.

1.2 Background

According to the material used in NVM, several kinds of NVM [1, 2, 3, 4, 5, 6] have been introduced such as spin-torque transfer RAM (STT-RAM), phase change memory (PCM), and ferroelectric RAM (FeRAM). Even though their compositions are different, all NVM can be considered similar in terms of cache architecture. First, they sustain their information without electric power; this is the reason why they called non-volatile memory. Their main advantage comes from their characteristics of extremely low leakage power consumption. In addition, their density is much higher than that of SRAM even that of DRAM for some kinds of NVM. Table 1 shows comparison of parameters of SRAM and STT-RAM obtained from the modified CACTI [13, 14] in previous work [15].

Table 1: Comparison of area, latency, and energy [15].

Parameters	SRAM	STT-RAM	PCM
Cache Size	128KB	512KB	2MB
Area(mm^2)	3.262	3.30	3.85
Read Latency(ns)	2.252	2.318	4.636
Write Latency(ns)	2.264	11.024	23.180
Read Energy(nJ)	0.895	0.858	1.732
Write Energy(nJ)	0.797	4.997	3.475
Static power(80 °C)(W)	1.131	0.016	0.031
Write Endurance	10^{16}	$4 * 10^{12}$	10^{9}

1.3 Motivation

The thesis focuses on two approaches such as cache coherence protocol and NVM capacity management policy for hybrid cache architecture (HCA). For cache coherence protocol, the existing studies have not concentrated on reducing the write operations because it does not matter in the SRAM-based LLC. Since there is no drawback of write operation compared to read operation, the number of write access is not taken into account. However, reducing the write operations is an important issue in NVM-based LLC. The dynamic energy consumption largely depends on the write operations, because the dynamic energy of write operation is greater than that of read operation. Moreover, the lifetime is inversely proportional to the number of write access. Therefore, a new protocol for NVM to minimize the write operations is needed.

In addition, it is found that there is a relationship between the capacity of NVM in HCA and the write counts of NVM. The analysis implies the necessity of efficient NVM capacity management policy: the HCA dynamically manages the capacity of NVM according to the demand of applications. As the first step of realizing this idea, we use the number of active NVM ways in a set as the measure of the capacity of NVM. The capacity of NVM is expressed by the number of currently available NVM ways and the demand of NVM is converted to the requested number of NVM ways.

1.4 Contributions

Firstly, the thesis introduces a new cache coherence protocol for NVM to decrease the number of write access to the LLC [7]. In our protocol, the data array of the LLC is not updated during the linefill operation, while the tag array is changed to maintain the inclusion property. The data array is modified only when the cache block is written-back from the private cache. Our protocol reduces the number of write access to the LLC; thus, the dynamic energy consumption is reduced and the lifetime is enhanced in our protocol.

- We investigate the existing cache coherence protocol for NVM and reveal the drawback of them.
- We propose a cache coherence protocol for NVM, which avoids unnecessary write operation in the LLC based on the analysis.

- We present experimental results of a write avoidance coherence protocol with number of write accesses to LLC, dynamic energy consumption, and lifetime.

In addition, hybrid cache architecture (HCA) has been proposed to overcome these limitations of NVM [16, 17, 18, 19, 20]. Most previous works have concentrated on managing write-intensive blocks by storing these blocks to SRAM to reduce the number of the write operations to NVM. However, we show the concept of NVM capacity management policy for reducing the number of write accesses to NVM and propose a dynamic way adjusting algorithm [8]. It dynamically resizes the number of active NVM ways to improve the dynamic energy consumption and the lifetime. To adjust the number of NVM ways, the maximum stack distance is monitored and rearranging the replaceable NVM ways is regularly performed.

- We investigate the relationship between the number of write operations and the capacity of NVM in HCA by performing both analysis based on the devised analytical model and experiments.
- We find out that decreasing the number of active NVM ways can be beneficial to reduce the number of write accesses to NVM ways, only if it does not increase the miss rate significantly.
- We propose a dynamic way adjusting algorithm (DWA) to find the optimal number of NVM ways and dynamically adjust active NVM ways without physical change of the cache.

- We conduct a simulation to evaluate the effectiveness of the proposed policy in terms of the reduction in the write counts of NVM, the decrement of the dynamic energy consumption, the lifetime extension, and the variation of the miss rate.

While previous studies focus on reducing NVM write counts due to the write-intensive blocks, they have not considered the NVM write operation is also occurred by linefill operation to NVM. Reducing the NVM write counts due to linefill operations are also very important for minimizing overall NVM write counts in chip-multiprocessor (CMP) environments. The thesis proposes a cache partitioning scheme called a linefill-aware cache partitioning (LCP) mechanism, taking into account the NVM linefill counts as well as the NVM write hit counts during cache partitioning.

- We propose a linefill-aware cache partitioning scheme (LCP) for HCA, which takes into account the reduction in the number of linefill operations to NVM to minimize the NVM write counts.
- We devise new metrics for LCP: write counts change (ΔW) and NVM write counts change ($\Delta NVMW$), which are based on the miss counts change (ΔM).
- We propose an algorithm to make partitions by predicting metrics according to the change of the number of allocated ways for each core.

Table 2: Summary of proposed schemes.

Scheme	Aim	Description
Write avoidance cache coherence protocol (WACC)	Reduction in the number of write access to LLC	The data array is modified only when the cache block is written-back from the private cache.
Dynamic way adjusting algorithm (DWA)	Reduction in the number of write access to NVM	The number of active NVM ways is dynamically resized.
Linefill-aware cache partitioning (LCP)	Reduction in the number of write access to NVM and increase in the hit rate of LLC	The NVM linefill counts is taken into account as well as the NVM write hit counts during cache partitioning.

- We present experimental results of LCP with the prediction accuracy, number of write accesses to NVM, miss rates, performance for multicore workloads, and dynamic energy consumption.

The schemes in the thesis are summarized in Table 2.

1.5 Organization of the thesis

The rest of the thesis is organized as follows. Chapter 2 provides related work about NVM. In Chapter 3, a new cache coherence protocol for NVM called a write avoidance cache coherence protocol is proposed. Chapter 4 describes NVM capacity management policy for HCA. The conclusion is given in Chapter 5.

Chapter 2

Related work

2.1 Hybrid cache architecture

Researchers have merged two types of memory into a single cache system, which is called HCA, to reduce the number of write access to NVM to alleviate the shortcomings of it especially related to a write operation [16, 17, 18, 19, 21]. As described in above section, the shortcomings of NVM come from write operation of NVM. In other terms, the number of write access to NVM is the most important factor for both the dynamic energy consumption and the lifetime. Since the write energy consumption of NVM is much larger than read energy of NVM or dynamic energy of SRAM, the write energy consumption of NVM is dominant for the total dynamic energy consumption. Furthermore, the lifetime is proportional to the number of write access to NVM cells. Therefore, reducing the number of write access to NVM is one of the most important methods to mitigate the drawbacks of NVM. For this reason, a small number of SRAM ways are used to accommodate heavily written blocks in the hybrid cache system as depicted in Figure 1.

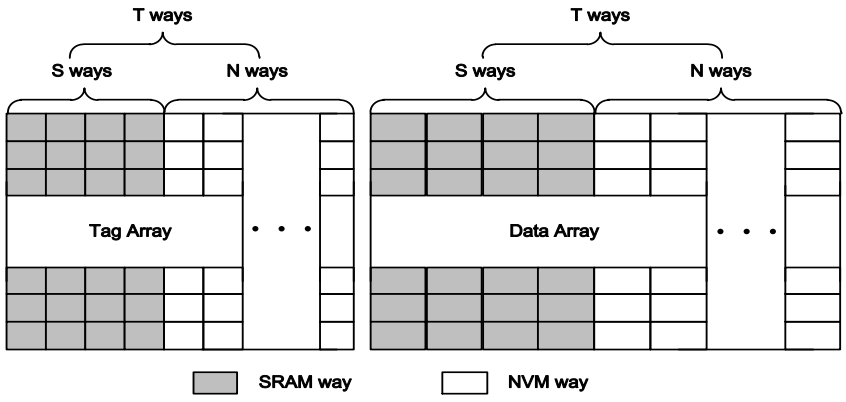


Figure 1: Basic structure of hybrid cache architecture (HCA).

First, swapping or migration schemes between SRAM and NVM in a hybrid cache system were proposed. Wi et al. introduced the region based cache architecture in [16]. They divided a single level of cache into two regions: read region which consists of STT-RAM and write region which consists of SRAM. If a block is predicted as write-intensive, the block is placed or swapped to the write region. Besides the schemes, merging set schemes were proposed [17] and [18]. The authors noticed that non-uniformity of write operations among sets. While some sets are frequently utilized, other sets receive relatively small requests. Therefore, write-intensive blocks in the highly utilized sets are forwarded to the idle sets. In addition, a predictor was equipped to find the correlation between write intensive blocks and addresses of trigger instructions [19]. In summary, existing policies focused on placing write-intensive blocks into the SRAM.

2.1.1 Write intensity prediction studies

Almost all papers on HCA have focused on devising methods to identify write-intensive blocks and place them to SRAM ways. Wi et al. suggested the region based cache architecture in [16]. They separated a single level of cache into two regions: read and write regions. The read region is prepared for non-write-intensive blocks composed of NVM, while the write region is composed of SRAM for write-intensive blocks. When a block is considered as write-intensive, the block is migrated or placed to the write region. On top of these schemes, combining set schemes were proposed [17, 22, 23]. This insight came from the fact that the write operations among sets are not uniformly distributed. While some sets receive relatively small write requests, other sets are highly utilized. To take advantage of these characteristics, some blocks in the frequently utilized sets are moved to the other sets. To elaborate the prediction algorithm, Quan et al. introduced a prediction table [18] containing the history of the write requests of the LLC. Another prediction table is proposed to store the value of combining addresses of the blocks and program counter of instructions [19]. What distinguishes these works from our scheme is that they have not focused on the CMP environment.

2.1.2 Static approaches

Various methods utilizing the compiler have been proposed. Chen et al. [24] proposed a scheme in which the compiler provides hints to find the write-

intensive block and the hardware is modified to correct the hints. Software dispatch was presented to detect write reuse patterns in [25]. In addition, the migration-intensive blocks are loaded into the SRAM region with the compiler assistance in [26] to mitigate the burden of migration blocks. Moreover, a loop retiming framework was proposed for loops with intensive data array operations to relieve the migration overhead [27]. Another study improves the read performance and energy efficiency guided by the analysis of read bottlenecks [28]. They focused on the recompilation or profiling schemes, while our proposed mechanism modifies the hardware structure and logics.

2.1.3 Hybrid cache architecture for main memory

As the write endurance problem has become important for the main memory, which is based on NVM, many methods have been proposed to prolong its lifetime. They have employed DRAM as a cache for NVM. Qureshi et al. firstly suggested the concept of a small DRAM cache to overcome the latency gap between DRAM and PCM [29]. The mechanism exploits both the short latency of DRAM and the large capacity of PCM by preventing unnecessary access to PCM. They also have shown advanced approaches such as write cancellation and write pausing policies [30] to mitigate the long read access time due to the long write latency. Meanwhile, a scheme proposed in Meza et al. [31] stores the metadata for the last accessed rows into a small buffer to manage the difficulty of fine-granularity DRAM caches. It is found that row buffer misses generate long latencies, and a policy is devised to exploit this observation [32]. They predict the data incurring a row

buffer miss and store it into a DRAM buffer by investigating the row buffer miss counts in PCM. Writeback-aware partitioning offers a new perspective on cache partitioning, taking into account the writeback information [33]. It is innovative in regard to reducing the amount of write access to the PCM main memory by managing the cache partition.

Another approach for the hybrid cache architecture is based on OS support. For PDRAM [34], the researchers introduced a hybrid solution related to software as well as hardware to extend the lifetime of the PCM pages. They modified the OS-level page manager and added a small device to contain the number of write requests for PCM at a page level granularity. Ferreira et al. [35] also inserted a DRAM buffer to decrease the number of read and write requests to PCM via page partitioning. Zhang and Li [36] improved the write endurance and reduced write latency of PCM by exploiting the workload characteristics as an aspect of an OS level paging. New page migration schemes were proposed to track read-bound access NVM pages [37].

All schemes described above are based on the physical features of DRAM or characteristics of OS, thus they are inadequate applied to the SRAM and NVM based LLC, which is the target of the thesis.

2.2 Cache partitioning schemes

To improve the cache efficiency, several methods using stack property have been proposed. The number of cache hit counts of LRU position is monitored to calculate the cache utility of each application or core. Based on the information, the cache is partitioned to minimize the number of total cache misses. Suh et al. [38] dynamically partitioned the LLC and assigned the guided number of cache ways to each application. Even though it successfully raised the cache utility, there was a problem in that the utility information of an application was affected by other applications. To avoid this drawback, Qureshi and Patt [39] introduced a separate utility monitor, which counts the number of hits without interference by other applications. An adaptive placement policy [40] was proposed to load a new block into the local or remote cache for enhancing the efficiency of cache based on stack distance profiling. In addition, compliers used the information to predict the memory behavior of the application [41]. For a real-time system, Liu and Zhang [42] suggested the compilation technique, which improves the worst case data cache performance using the stack distance approach. Most papers on cache partitioning assumed that the LLC consists of SRAM only, hence they do not consider the NVM write counts in their schemes.

Chapter 3

Write avoidance cache coherence protocol

3.1 Limitation of existing cache coherence protocol

We review the legacy cache coherence protocols to get a new insight to reduce the write operations. There are useless write operations in the existing protocol. Generally, memory systems of CMPs are composed of a shared LLC and several private caches which are dedicated to cores [43]. In addition, the cache block is divided into two arrays: tag array and data array. Tag array stores tag bits and cache coherence state, while data array stores block data. When a linefill operation occurs, the requested block data is written to the data array, and the tag bits and cache coherence state are updated to the tag array. Then, the cache block is forwarded and linefilled to the private cache. When a core tries to modify the cache block in the private cache, an invalidation signal is sent to the shared LLC and other private caches to maintain the cache coherence. Thus, the previous write access to the LLC during the linefill operation is considered as the useless write operation, if the cache block in the LLC has been never used until it is invalidated.

Figure 2 illustrates an example of write inefficiency in widely used cache coherence protocols such as MESI or MOESI [44]. In the example, we assume that a core reads and writes a block data of the PC (Private Cache) 1. Table 3 lists the cache states in the figure and their descriptions. When the core tries to read the block data, since the PC1 has no valid block data, the cache controller sends the request for the block data to the LLC.

However, the LLC also has no valid copy; thus, the request is sent to the external sources such as the main memory or other chipsets. When the block data "ABCD" is arrived at the LLC, it is written into the LLC and the state of the LLC is changed to S state, which means the cache block is valid and other private caches may have the same cache block. Then, the block data "ABCD" is forwarded to the PC1.

When the block data is received in the PC1, it is written into the PC1 and the state of the PC1 is changed to E state. After the linefill operation is completed, if the core tries to modify the block data "ABCD" to "EFEF", an invalidation request is sent to the LLC to maintain cache coherence. The purpose of the invalidation request is indicating that the block data of the PC1 is modified and the cache block in the LLC should be invalidated. If the block data "ABCD" in the LLC has not been used until it is invalidated, writing the block data "ABCD" to the LLC during the linefill operation was a useless write operation.

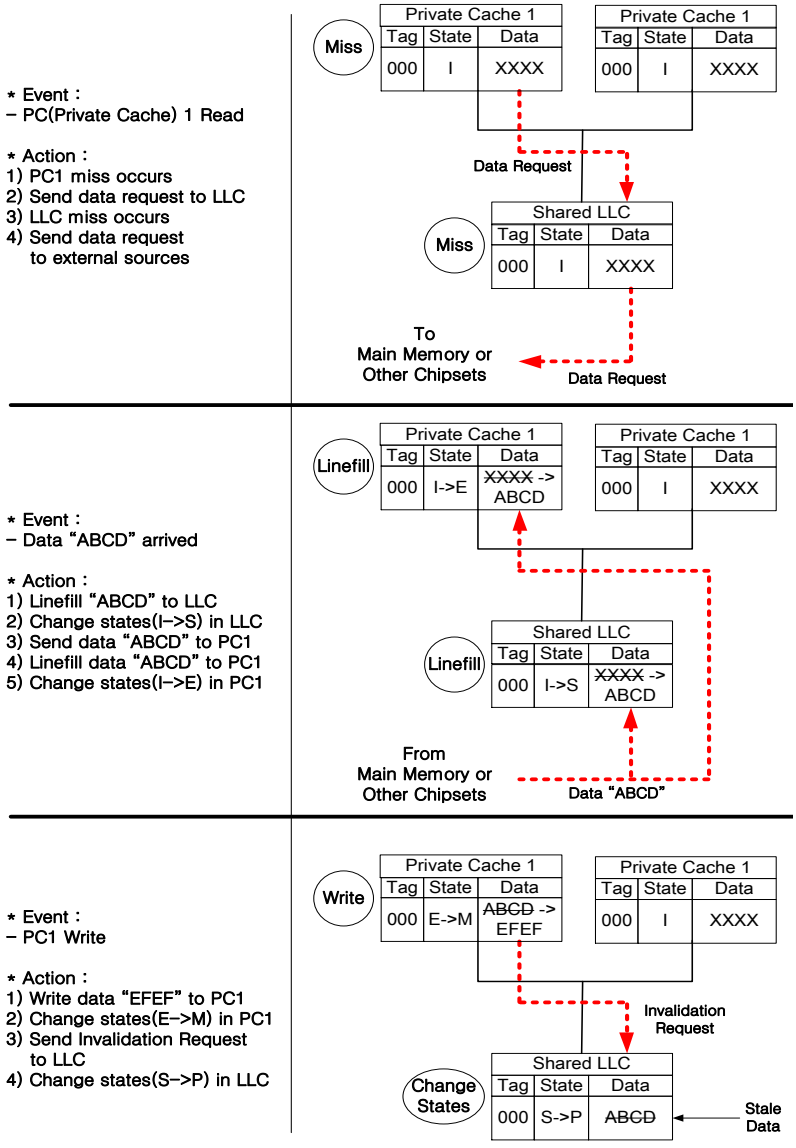


Figure 2: Conventional cache coherence protocol.

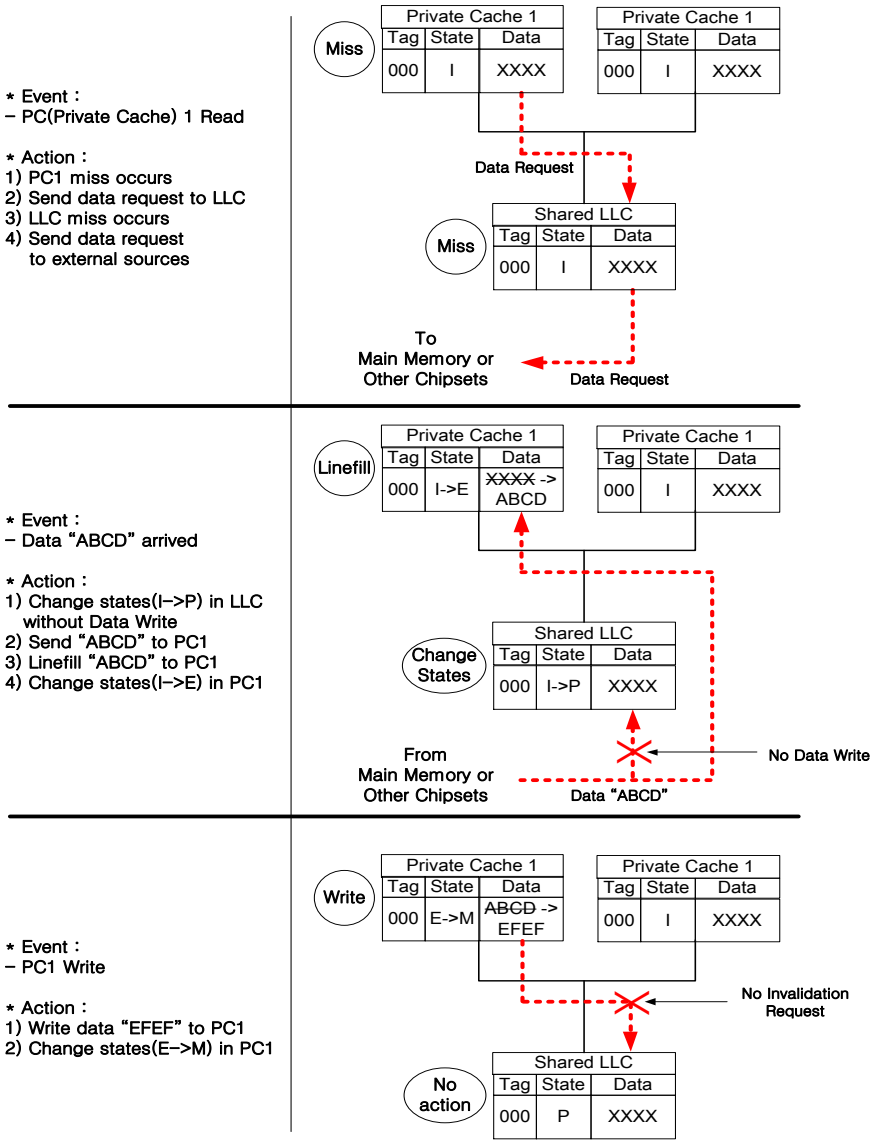


Figure 3: Write avoidance cache coherence protocol (WACC).

Table 3: States and descriptions for write avoidance cache coherence protocol (WACC).

State	Description
I(nvalid)	The cache block is invalid
S(hared)	The cache block has valid block data and other private caches may have valid copy.
E(xclusive)	The cache block has valid block data with exclusive permission and other caches have no valid copy.
M(odified)	The cache block has valid and modified block data. Other caches have no valid copy. This state appears in the private cache only.
P(rivate cache)	The cache block in the LLC has no valid block data, but more than one of the private caches has valid block data. This state appears in the LLC only.

* P state is introduced due to keeping the inclusion property. Modern multiprocessors have employed the inclusive LLC to filter the cache coherence traffic from other chipset or the main memory. Thus, it is needed that a state represents one of the private caches has valid data even the LLC has no valid data.

3.2 Write avoidance cache coherence protocol

To deal with this problem, we suggest a new cache coherence protocol which is called Write avoidance cache coherence (WACC) protocol. In our protocol, the block data of the cache block is not written into the LLC during the linefill operation, while the tag bits and the cache coherence state are updated. Since the block data is not placed in the LLC, one of the private caches has responsibility to provide the valid block data. The block data in the LLC is only updated when it is written-back from the private cache. The writeback is initiated only when no other private cache has the block data in WACC protocol. Therefore, we avoid useless write operation due to modifications of the block data in the private cache.

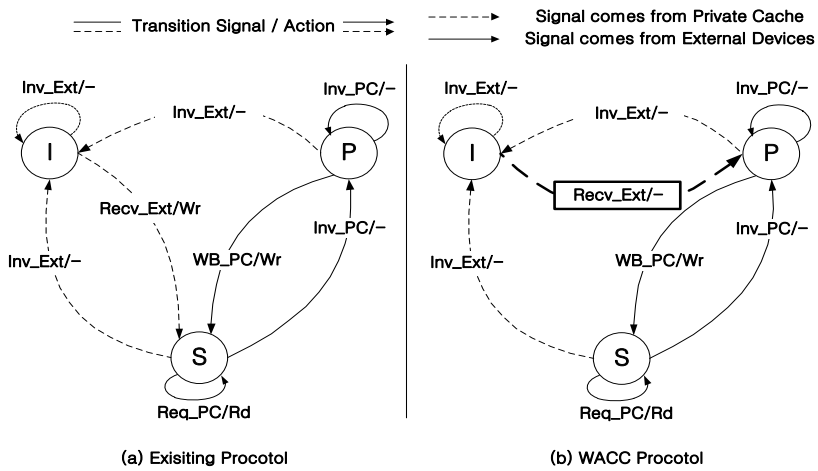


Figure 4: State transition diagrams for WACC.

Figure 3 shows an example of WACC protocol. Unlike the conventional protocols, when the block data ABCD is arrived at the LLC, it is not written to the LLC. Instead, the state is changed to P state and the block data is forwarded to the PC1. When the PC1 is modified to EFEF, there is no need to send an invalidation request to the LLC for the block data ABCD is not written to the LLC. Therefore, one write operation of the LLC and one request for cache coherence is decreased compared to the baseline protocols.

We compare a simple version of the existing MOESI protocol with its modified protocol in Figure 4. Table 4 shows the coherence signals and actions. The transition signal is divided into two parts: {signal}_{source} and the action indicates the operation of the data array. For example, WB_PC/Wr means that if the block is P state and receives the WB signal from a private cache, the block data is written to the data array.

Table 4: Signals/actions and descriptions.

Signal	Description
Inv	Invalidate the cache block if it is valid. This signal is generated when another device tries to modify the block data.
Recv	Provide the block data in the cache block. This signal is generated when a cache hit occurs.
Req	Request the block data for read operation. This signal is generated when a cache miss occurs.
WB	Writeback the block data to the LLC. This signal is generated when a private cache evicts the cache block.
Action	Description
Wr	Write the block data of the received cache block into the data array.
Rd	Read the block data and provide it with the requestor.

As shown in Figure 4(a), when a new cache block is received in the LLC, the state of the cache is transition to S state and the block data is written to the data array in the existing protocol. On the contrary, the state is transition to P state instead of S state in our protocol under the same condition. Furthermore, the write operation is omitted as shown in Figure 4(b). This is because the block data is forwarded without write access to the data array in WACC protocol.

Another point to be considered is that the protocol of the private cache should be changed. The writeback operation is initiated if the cache block in the private cache is modified and evicted in the existing protocols. However, the cache block should be written-back to the LLC in WACC protocol when it is evicted in the private cache regardless of whether the cache block is dirty or not.

Chapter 4

NVM capacity management policy for hybrid cache architecture

4.1 NVM capacity management policy

In this section, we propose two schemes for NVM capacity management policy. First, we introduce a dynamic way adjusting algorithm (DWA) that monitors the optimal number of NVM ways and dynamically adjust the number of active NVM ways [8]. In addition, we also propose a linefill-aware cache partitioning scheme (LCP) to save the dynamic energy consumption by efficiently allocating SRAM ways and NVM ways to cores.

The DWA keeps track of maximum stack distance (MSD), which means the minimum number of ways to maintain the miss rate. If the number of the current active NVM ways is not the optimal value, it is adjusted according to the MSD. In addition, an efficient method to disable NVM ways is required because it is impossible that NVM ways are physically added or removed during execution. Thus, the DWA prevents deactivated NVM ways from victim selection. A newly fetched block is prohibited to be loaded into the disabled NVM ways, which has the effect of virtually deactivating them.

The basic idea of LCP comes from cache partitioning [38, 39, 40], which has been a well-known scheme to improve the performance in CMP systems. The key idea of the cache partitioning is that all cache ways should be efficiently allocated for each application to maximize the hit rate of the LLC. They have contributed the studies of the LLC. However, it is inefficient to apply them directly into HCA because their models assume that all cache ways consist of the same memory type. Even though the cache misses are minimized by the previous cache partitioning schemes, if the linefill operations heavily occur in NVM ways, it fails to reduce the linefill counts of NVM. Therefore, LCP assigns the SRAM ways and the NVM ways to each core based on the change of the NVM linefill counts as well as the NVM write hit counts according to partitioning.

4.1.1 Concept of NVM capacity management policy

This section presents an NVM capacity management policy that resizes the number of NVM ways to fit the demand of applications. This policy comes from the observation that reducing the size of NVM usually decreases the write counts of NVM if the miss rate does not grow. The thesis will propose an analytical model and perform a simulation to verify this observation.

Cache researchers have been investigating the relationship between the size of cache and the miss rate [39]. For many programs, as the cache size grows, the miss rate becomes small. On the contrary, the miss rates of some programs are saturated or remain despite incremental growth of the cache

size. In addition, even the same program always does not require the fixed size of cache. Therefore, the number of requested ways of the cache varies during execution, and the unnecessary ways are disabled without performance degradation.

The number of write accesses to the cache is strongly coupled with the miss rate. Generally, the cache operations are divided into three categories: read hit, write hit, and linefill. Among these operations, write hits and linefill operations compose the write requests. If some read hits are changed to cache misses due to the increasing miss rate, new linefill operations occur as much as the removed read hits. This implies that the total number of write operations are increased. Alternately, if the number of cache misses is not increased, the number of write accesses to the cache remains because the hit counts and miss counts is not changed.

Assume that we minimize the number of NVM ways without generating significant extra cache misses. In that case, the write operations which originally occurred in the deactivated NVM ways are forwarded to SRAM ways or other NVM ways. If a part of write accesses is sent to SRAM ways, the number of write accesses to NVM ways is reduced. Therefore, partial deactivating NVM ways with the stable miss rate highly tends to decrease the write counts of NVM ways.

An illustration is provided in Figure 5 to aid in the understanding of this concept. There are two caches in the example. One of the caches consists of one SRAM and three NVM ways, and another cache is composed of one

SRAM and two NVM ways. The program in our example needs only three ways. For the sake of convenience, suppose that all memory references are write requests.

When the program starts, cache accesses are performed according to the sequence in Figure 5. There is no difference between the two caches in the first three accesses. However, when ”d” miss is encountered, two caches behave differently. While ”d” is placed in the fourth way in cache A, ”a” is replaced with ”d” in cache B. Writing ”d” in the second iteration, SRAM access is made instead of NVM access in cache B. As a result, the number of write to NVM ways is reduced in cache B. The linefill operation of ”d” is forwarded to a SRAM way, and thus one linefill operation and one write hit of NVM ways is reduced.

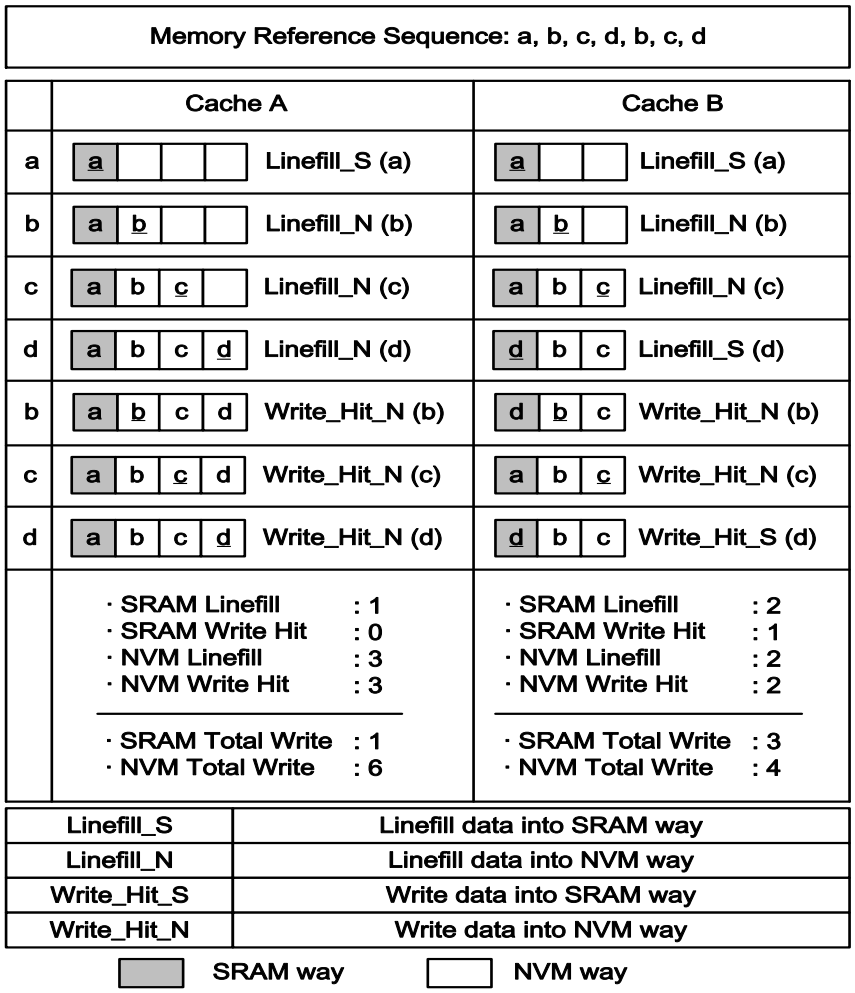


Figure 5: Example for NVM capacity management policy.

4.1.2 Feasibility of NVM capacity management policy

A metric, write intensity of a way (WI), is defined as the portion of write accesses to the way over the write accesses to all ways. It is given by

$$WI_i = \frac{W_i}{W_{total}} \ (1 \leq i \leq T) \tag{4.1}$$

where W_i is the number of write accesses to ith way and W_{total} means the number of total write accesses to the cache, while T is the number of all cache ways. This metric indicates the distribution of write requests among the ways. If all ways have the same write intensity, the write requests are evenly distributed. Unless, write operations occur more frequently in some ways which have higher value than other ways.

Since the total number of write counts is calculated by summation of write counts of each way, it is expressed as

$$W_{total} = \sum_{i=1}^{T} W_i \tag{4.2}$$

The above equation is expressed as form of WI as follows

$$W_{total} = \sum_{i=1}^{T} (WI_i * W_{total})$$

$$= W_{total} * \sum_{i=1}^{T} WI_i \tag{4.3}$$

We rewrite the above equation as form of SRAM ways and NVM ways, and it is given by

$$W_{total} = W_{sram} + W_{nvm}$$

$$= W_{total} * \sum_{i=1}^{S} WI_i + W_{total} * \sum_{i=S+1}^{T} WI_i \tag{4.4}$$

$$W_{sram} = W_{total} * \sum_{i=1}^{S} WI_i \tag{4.5}$$

$$W_{nvm} = W_{total} * \sum_{i=S+1}^{S+N} WI_i = W_{total} * \sum_{i=S+1}^{T} WI_i \tag{4.6}$$

where S is the number of SRAM ways and N is the number of NVM ways, while W_{sram} means the number of write accesses to SRAM ways and W_{nvm} is the number of write accesses to NVM ways. We found that there are three factors that influence the write counts of NVM ways: the number of total counts (W_{total}), the write intensity per way (WI), and the number of NVM ways ($N = T - S$).

So far, the main strategy for reducing the number of write counts of NVM ways has been keeping average WI of NVM ways lower than that of SRAM ways. Throughout previous HCA research, WI is thought as the only important factor among the three factors. It is assumed that N is fixed and W_{total} is not significantly changed. Therefore, they have focused on minimizing WI of NVM ways by detecting write intensive blocks and placing them into SRAM ways. These approaches are successful to reduce write accesses to NVM.

Different from previous approach, we consider N as a variable instead of a constant value. When the number of NVM ways is reduced to N' ($N' < N$), W'_{total}, W'_{sram}, and W'_{nvm} are defined as the number of write accesses to the cache, SRAM ways, and NVM ways:

$$W'_{total} = W'_{sram} + W'_{nvm} \tag{4.7}$$

In addition, we define the altered number of all ways as T' ($T' = S + N' < T$), and Eq. 4.6 is transformed below:

$$W_{nvm} = W_{total} * (\sum_{i=S+1}^{T'} WI_i + \sum_{i=T'+1}^{T} WI_i)$$

$$= \sum_{i=S+1}^{T'} WI_i * W_{total} + \sum_{i=T'+1}^{T} WI_i * W_{total} \tag{4.8}$$

The second term indicates the number of write accesses to the NVM ways that will be removed. If we adjust the number of NVM ways to N', the remaining ways should absorb the write requests of the amount of second term. For simplicity, this term substitute for X and Eq. 4.6 is expressed as follows:

$$X = \sum_{i=T'+1}^{T} WI_i * W_{total} \tag{4.9}$$

$$W_{total} = W_{sram} + (W_{nvm} - X) + X \tag{4.10}$$

Hereby, we introduce a condition that the total write counts are not changed ($W'_{total} = W_{total}$). Under the condition, W'_{total} is given by

$$W'_{total} = W_{sram} + (W_{nvm} - X) + X \tag{4.11}$$

If we divide X into X_{sram} and X_{nvm} that are the write requests of the amount of forwarded to SRAM ways and NVM ways, we obtain

$$W'_{total} = W_{sram} + (W_{nvm} - X) + X_{sram} + X_{nvm}$$

$$= (W_{sram} + X_{sram}) + ((W_{nvm} - X) + X_{nvm}) \tag{4.12}$$

Because W'_{sram} and W'_{nvm} are defined as the number of write accesses to SRAM and NVM in the resized cache, they can be expressed by as following equation:

$$W'_{sram} = W_{sram} + X_{sram} \tag{4.13}$$

$$W'_{nvm} = W_{nvm} - X + X_{nvm} \tag{4.14}$$

Before advancing the discussion, we state that it is assumed that X_{sram} is greater than zero for the simplicity of the model. When the number of ways is changed, the blocks are placed differently than they were. There is a possibility that some write intensive blocks that were originally located in SRAM ways are inserted into NVM ways. In that case, X_{sram} could be zero or minus value. To avoid this problem, we adopt a policy for placing write intensive blocks into SRAM ways as presented [16] to our scheme.

Since X is summation of X_{sram} and X_{nvm}, if X_{sram} is greater than zero, X_{nvm} is given by

$$X_{nvm} < X \tag{4.15}$$

By transforming Eq. 4.14 and substitution W_{nvm} into Eq. 4.15, we obtain

$$W'_{nvm} - W_{nvm} + X < X \tag{4.16}$$

$$W'_{nvm} < W_{nvm} \tag{4.17}$$

Thus, we conclude that fewer NVM ways causes lower write requests to NVM if the miss rate does not grow.

We examined the impact of NVM capacity management on the miss rate, the total write counts, and the write accesses to NVM ways. We assume that the hybrid cache has 4 SRAM ways and 12 NVM ways and that the number of NVM ways varies from 12 to 0. The results are sorted in decreasing order by the number of NVM ways among each application. To improve the readability, we abbreviate SRAM ways to ”S” and NVM ways to ”N”. For example, 4S_2N in the figure means that 4 SRAM ways and 2 NVM ways are used during the simulation.

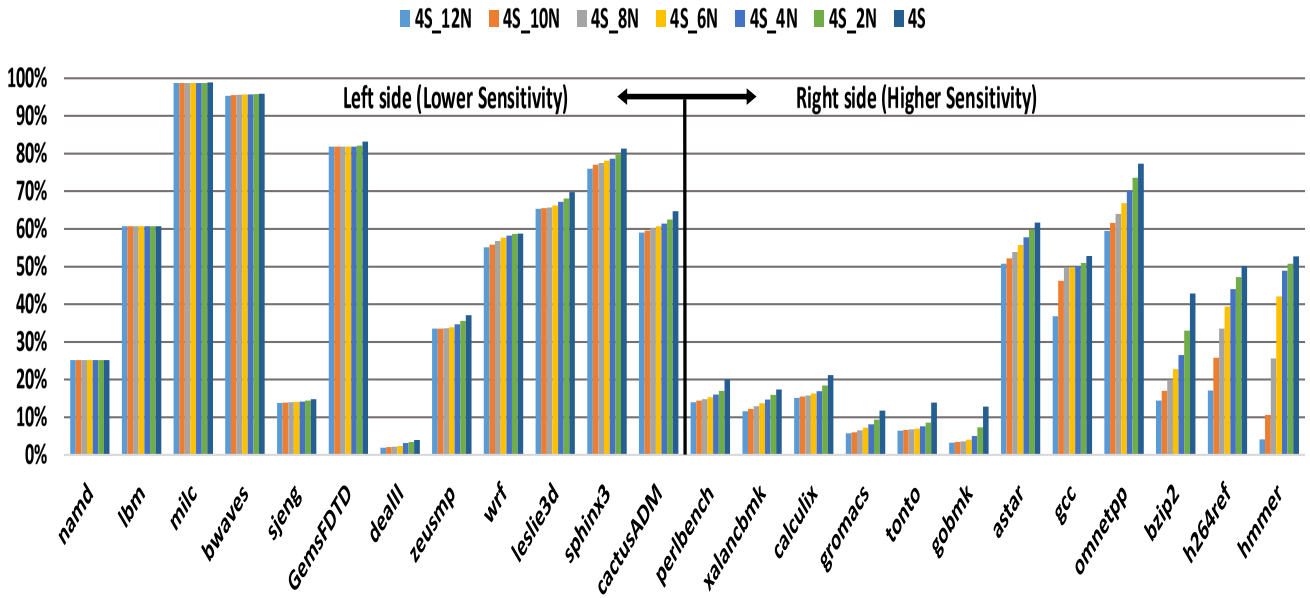


Figure 6: Miss rates with various number of NVM ways.

Figure 6 represents the miss rates with various number of NVM ways to show sensitivity of the miss rate to the size of NVM. We sort all applications by geometric standard deviation (GSD), which represents the amount of dispersion from the geometric mean. In Figure 6, the miss rates of the left applications are not less influenced by the number of NVM ways, while the right side applications are more sensitive to the number of NVM ways. The miss rates of two left most applications such as *namd* and *lbm* remain even when all NVM ways are removed. Part of NVM ways are unnecessary for some left side applications: *milc, bwaves, sjeng, GemsFDTD, dealII,* and *zeusmp*. On the contrary, the growth of the miss rates of the higher sensitive applications is large. Especially, the miss rates of *bzip2* and *h264ref* is multiplied about three times and the miss rate of *hmmer* soars to 12.8 times.

Figure 7 shows normalized write accesses to the HCA with various sizes of NVM. We find that the total write counts of the lower sensitive applications are not greatly increased, while many higher sensitive applications show rapid growth. For the left side applications, only 2.8% of average extra write operations occur. Especially, no change is detected through all sizes of NVM in *namd,lbm,* and *milc*. The number of NVM ways can be decreased to 2 without increasing write counts in *bwaves* and *GemsFDTD*. Other benchmarks such as *sjeng* and *zeusmp* have the same values when NVM ways varies from 12 to 8. On the other hand, the total write counts of the right side applications increase by 29.4% on average.

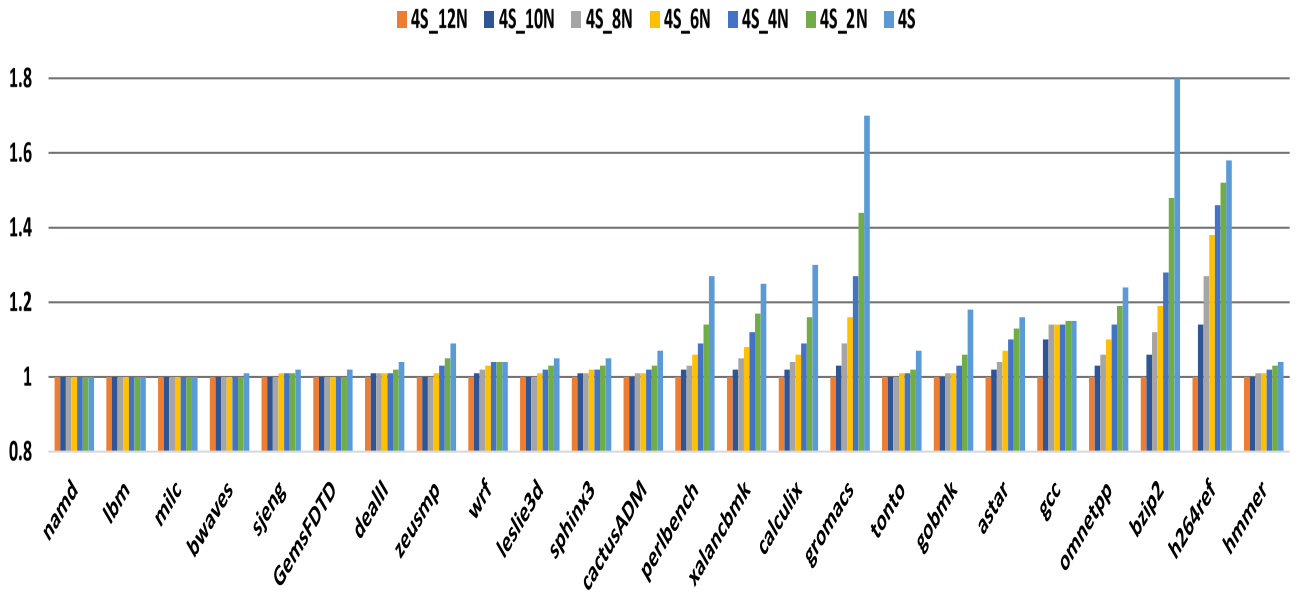


Figure 7: Normalized total write counts of HCA various number of NVM ways. 4S_12N is the standard of normalization.

The normalized write accesses to NVM ways with various number of NVM ways is depicted in Figure 8. As we expected, reducing the number of NVM ways decreases the write accesses to NVM ways in lower sensitive applications. On the other hand, the reduction in the write counts of NVM ways is not guaranteed by resizing the number of active NVM ways in higher sensitive applications. Adjusting NVM ways even results in increasing the write operations of NVM ways in *gobmk, gcc,* and *h264ref*. Some applications such as *gromacs, tonto, bzip2,* and *hmmer* show the similar pattern of the left applications, but their reduction ratios are small.

In summary, we find out that the number of write accesses to NVM ways is usually reduced if resizing the number of active NVM ways does not significantly increase the miss rate by adopting efficient NVM capacity management policy.

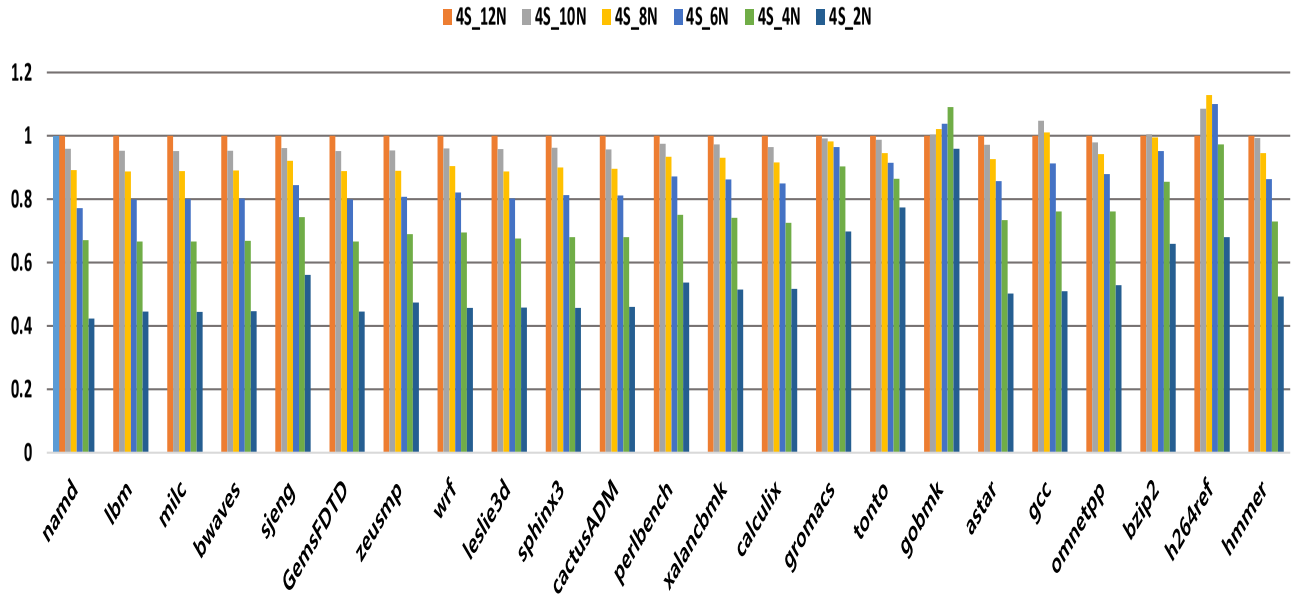


Figure 8: Normalized write counts of NVM with various number of NVM ways. 4S_12N is the standard of normalization.

4.2 Dynamic way adjusting

We propose a dynamic way adjusting algorithm (DWA) to implement NVM capacity management policy. To discover the optimal size of NVM, the maximum stack distance (MSD) is dynamically monitored. Using the MSD, the DWA marks all NVM ways either as ”replaceable way” or ”non-replaceable way” to realize adjusting the number of NVM ways. Replaceable ways are regularly changed to prevent write requests from concentrating on a few NVM ways. This section explains these key ideas and the operations of the DWA.

4.2.1 Maximum stack distance

In order to find the minimum number of ways which sustain the miss rate, we introduce the MSD based on the stack property [39]. It is well known that the LRU replacement policy follows the stack property [45], which means that a cache of a size C always contains all blocks of the cache of size less than C. Assume that the number of sets is a constant value. If a cache block is in an N way cache, it is guaranteed that the block is in the cache, which has more than N ways. A metric related to stack property is the ”stack distance”. When a cache hit regardless of a read hit or a write hit, the stack distance is defined as the LRU order of the hit block. For example, the stack distance of the block at MRU position is one, and that of the LRU position is N in the N way cache. Figure 9 presents the stack distance histogram of a hypothetical application. If the number of the ways is reduced to 3 from 8, the number of

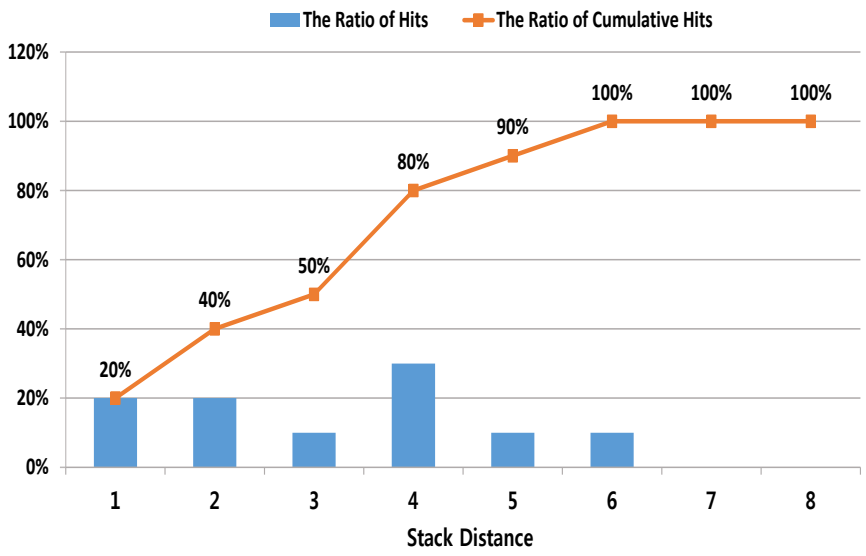


Figure 9: Stack distance histogram.

hits will be halved because the cumulative hits for stack distance 3 is 50%. This means that the miss rate of three-way cache will be increased to 50% in this case. However, if we use 6 ways instead of 8 ways, no additional cache miss occurs. Therefore, the maximum value of the stack distance indicates the minimum number of ways to maintain the hit rate.

We employ an auxiliary tag directory (ATD), a maximum stack distance register (MSDR), and a replaceable way size register (RWSR) to monitor the MSD as shown in Figure 10. The ATD is a separate storage constructed with the same associativity as the main tag array of the cache. It keeps track of the LRU order information and tag bits. When an ATD hit occurs, the MSDR is updated if LRU of the hit block is larger than the current value of the MSDR. The RWSR is updated in two cases. First, if

the MSDR exceeds the RWSR, the RWSR is increased to the MSDR. The condition that the RWSR is smaller than the MSDR means that the current working set needs more cache capacity. Thus additional NVM ways should be replaceable ways by increasing the RWSR. Second, when the value of the RWSR has been larger than that of the MSDR for a while, it is decreased to the value of the MSDR. Keeping the situation in which the RWSR is larger than the MSDR means that unnecessary NVM ways have been used. Therefore, some NVM ways should be deactivated by decreasing the RWSR. To detect this situation and initiate resizing the number of NVM ways, a resizing counter register (RCR) is added. The RCR is increased by 1 when the RWSR is larger than RWSR during the ATD hit operation. Whenever the RWSR is updated to the MSDR, the RCR is reset to 0.

Another consideration in adopting the ATD is the storage overhead. If the ATD has tag information of all sets, the size of the tag array will be doubled. Therefore, to reduce the storage overhead, we use a set sampling policy [46]. The ATD is designed to have only a part of sets which is sampled every 32nd in the proposed algorithm. It is verified that the sampled sets are enough to correctly capture the stack distance value in [46] instead of using all sets.

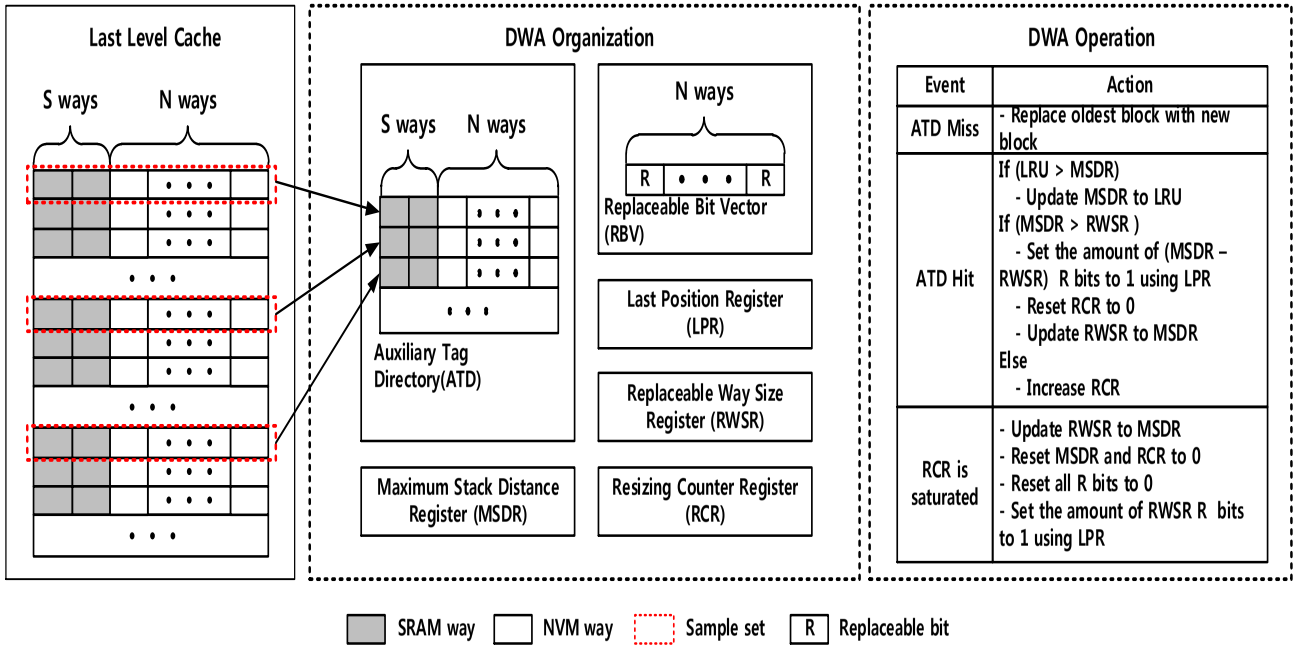


Figure 10: Overall structure of dynamic way adjusting (DWA).

4.2.2 Adjusting the number of NVM ways

Since physical NVM cells are not inserted or deleted according to the change of the MSD, we devise a method to dynamically activate or deactivate NVM ways. To disable unnecessary NVM ways, we introduce the concept of ”replaceable way” and ”non-replaceable way” The replaceable way implies the normal way that participates in all kinds of cache operations, such as read access, write access, and replacement. The non-replaceable way means that it is excluded from block replacement; thus, a new block is not placed into the way. However, when a cache hit occurs, read access and write access are performed, same as the replaceable way. All NVM ways in the DWA are divided into replaceable ways and non-replaceable ways.

The role of the replaceable bit vector (RBV) in Figure 10 is indicating that each way is non-replaceable or not by controlling replaceable (R) bits. Since each R bit is corresponded to each NVM way, the size of R bits is identical to the number of NVM ways. The RBV is altered when the RWSR is changed. If the RWSR is increased, additional R bits are set to 1. Unless, all R bits are updated to rearrange non-replaceable ways.

The cache operation for non-replaceable ways should be different from that for replaceable ways. When a cache hit is occurred to a non-replaceable way, the LRU information is not updated. In the case of a cache miss, the non-replaceable ways are not involved in the victim selection. A detailed description of the management policy is as follows:

1. Cache hit in the replaceable way: If a requested block is in the replaceable ways, the cache operations do not differ from the conventional cache. When a read hit occurs in the replaceable ways, the data is sent to the requestor. In case of a write hit, the data is modified. LRU information is updated in both cases.

2. Cache hit in the non-replaceable way: When the block is in the non-replaceable way, the data is sent to the requestor or the data is written the same as the replaceable way. However, no operation for updating LRU bits occurs because the LRU information of the non-replaceable way is useless in the DWA.

3. Cache Miss: A new block is only placed into the replaceable way. When a cache miss occurs and a requested block arrives, the LRU block in the replaceable ways is selected to load the requested block.

4.2.3 Algorithm of dynamic way adjusting

We rearrange the replaceable ways to avoid lifetime shortening when the replaceable NVM ways are reduced. If some NVM ways are frequently selected as replaceable way during execution, these ways will be worn out earlier than other NVM ways. Thus, we shift the start point of replaceable ways to allow write operations be performed as evenly as possible through the ways. The basic concept is similar to the round robin policy. At the time of selecting the replaceable ways, the NVM way next to the current replaceable ways is chosen for the first replaceable way. The last position register

(LPR) remembers the current last replaceable way to support way shifting. This policy is initiated when RCR is saturated.

Figure 11 shows an example of how this policy works. Assume that the number of the replaceable ways is five and the first three NVM ways are assigned to the replaceable ways. Note that two SRAM ways are always considered the replaceable ways. If the number of the replaceable ways is increased to six, from the fourth NVM way to the sixth NVM way, then the first NVM way is chosen as the replaceable ways.

Figure 12 presents the DWA in detail. When a cache access is confirmed to an ATD hit (line 1), the MSDR is updated if it is not the maximum LRU value (line 2-4). Then, we compare the RWSR with the MSDR to check whether the current size of NVM ways is less than the minimum size of NVM ways (line 5). If the MSDR exceeds the RWSR, some non-replaceable NVM ways are changed to be replaceable from the last NVM way of the current replaceable NVM ways. The amount of activated NVM ways is the difference between the MSDR and the RWSR. The LPR is automatically updated during way adjusting within range from 0 to W_{nvm} (line 6-9). After this adjustment, the RWSR is updated to the MSDR and the RCR is reset to 0 (line 10-11). The replaceable NVM ways are rearranged when the MSDR does not exceed RWSR when RCR is saturated (line 13). If the MSDR is larger than the number of SRAM ways, the RWSR is updated to MSDR (line 14-15). Unless, the RWSR is set to the number of SRAM ways because all SRAM ways are replaceable (line 16-17). As a first step of shifting replaceable ways, all R bits are set to 0 (line 19). Then, from the last

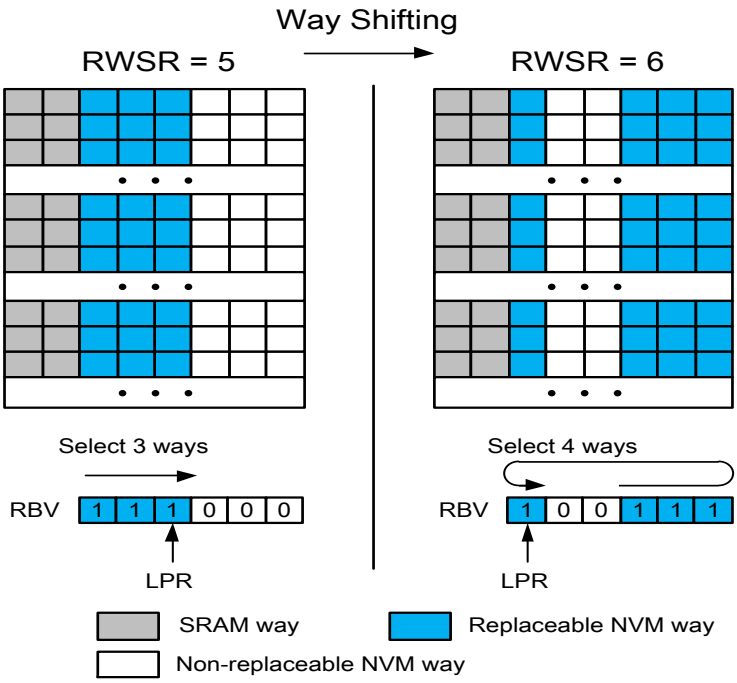


Figure 11: Example of way shifting.

replaceable NVM way, NVM ways of the amount of RWSR are assigned to be replaceable (line 20-23). To keep track of the maximum stack distance again, the MSDR is initialized to 0 and RCR is reset to 0 (line 24-25). If RCR is smaller than the threshold, RCR is increased by 1 (line 27).

Algorithm : Adjust_Replaceable_Ways

Parameters:

$RWSR$: Replaceable way size register
$MSDR$: Maximum stack distance register
LPR: Last position register $(1 \leq LPR \leq W_{nvm})$
RCR: Resizing counter register
$R[x]$: Replaceable bit at xth NVM way

Initial conditions:

$RWSR \leftarrow W_{nvm} + W_{sram}$
$MSDR \leftarrow 1$
$LPR \leftarrow W_{nvm} - 1$
$RCR \leftarrow 0$
All $R[x] \leftarrow 1$

During execution:

```
1 : if ATD hit then
2 :   if hit_block.LRU > MSDR then
3 :     MSDR ← hit_block.LRU
4 :   end if
5 :   if MSDR > RWSR then
6 :     for i ← 1 to (MSDR − RWSR) do
7 :       LPR ← (LPR + 1) % W_nvm
8 :       R[LPR] ← 1
9 :     end for
10:     RWSR ← MSDR
11:     RCR ← 0
12:   else
13:     if RCR is saturated then
14:       if MSDR > W_sram then
15:         RWSR ← MSDR
16:       else
17:         RWSR ← W_sram
18:       end if
19:       All R[x] ← 0
20:       for i ← 1 to (RWSR − W_nvm) do
21:         LPR ← (LPR + 1) % W_nvm
22:         R[LPR] ← 1
23:       end for
24:       MSDR ← 0
25:       RCR ← 0
26:     else
27:       RCR ← RCR + 1
28:     end if
29:   end if
30: end if
```

Figure 12: Algorithm for DWA.

4.3 Cache partitioning for hybrid cache architecture

Modern chip-multiprocessors (CMP) have employed multi-level on-chip caches to address the memory wall problem that is caused by the difference between access latencies of the memory and the processor. Generally, the last-level cache (LLC) occupies the largest area in the cache system and consumes a significant static energy in the CMP. To reduce the area and the leakage power, researchers have considered using non-volatile memory (NVM) [1, 3, 5] as LLC. Unlike the SRAM-based LLC, the NVM-based LLC consumes little leakage power and requires less area with higher density than SRAM. While NVM has these advantages, they also suffer from shortcomings such as longer latency to complete a write operation and higher dynamic energy consumption for a write operation compared to SRAM. Most researchers have focused on minimizing the write counts of NVM because the number of write operations strongly affects the dynamic energy consumption as well as performance.

Hybrid cache architectures (HCA) have been proposed [16, 17, 18, 19, 47] to overcome these limitations of NVM. HCA mainly consists of NVM, but some of them are replaced with SRAM to reduce the number of write requests on NVM. Previous studies concerning HCA have attempted to detect the write-intensive blocks, sets, or ways to allocate these to the SRAM. However, their schemes have not usually focused on reducing the NVM linefill counts, while the portion of NVM linefill operations is larger than

that of NVM write hit operations over the total of write operations to NVM for many applications. In addition, there is no accurate prediction model to estimate the change of the write counts of NVM when the number of SRAM and NVM ways allocated to each core are changed in CMP environments. Since the number of cache ways is closely related to the cache misses, assigning cache ways or releasing cache ways influences the miss rate of the LLC. Even though the write intensity of NVM ways of a core is larger than other cores, providing more SRAM ways with the core does not guarantee reducing the NVM write counts. If a core which hands over SRAM ways to other core generates much more cache misses with the reduced cache capacity, the write counts can be increased due to the extra linefill operations. However, they have not considered this kind of side effects in their schemes.

We propose a novel cache partitioning that is called a linefill-aware cache partitioning scheme (LCP) to reduce the dynamic energy consumption by efficiently allocating SRAM ways and NVM ways to cores. To this end, the thesis presents appropriate metrics and an algorithm for partitioning to realize LCP. We introduce three metrics that represent change of miss counts (ΔM), write counts (ΔW), and NVM write counts ($\Delta NVMW$), respectively. An algorithm for cache partitioning of LCP consists of two steps. First, the number of cache ways for each core is determined in order to reduce the miss counts. Next, the SRAM ways and the NVM ways are allocated to cores to minimize write counts of NVM.

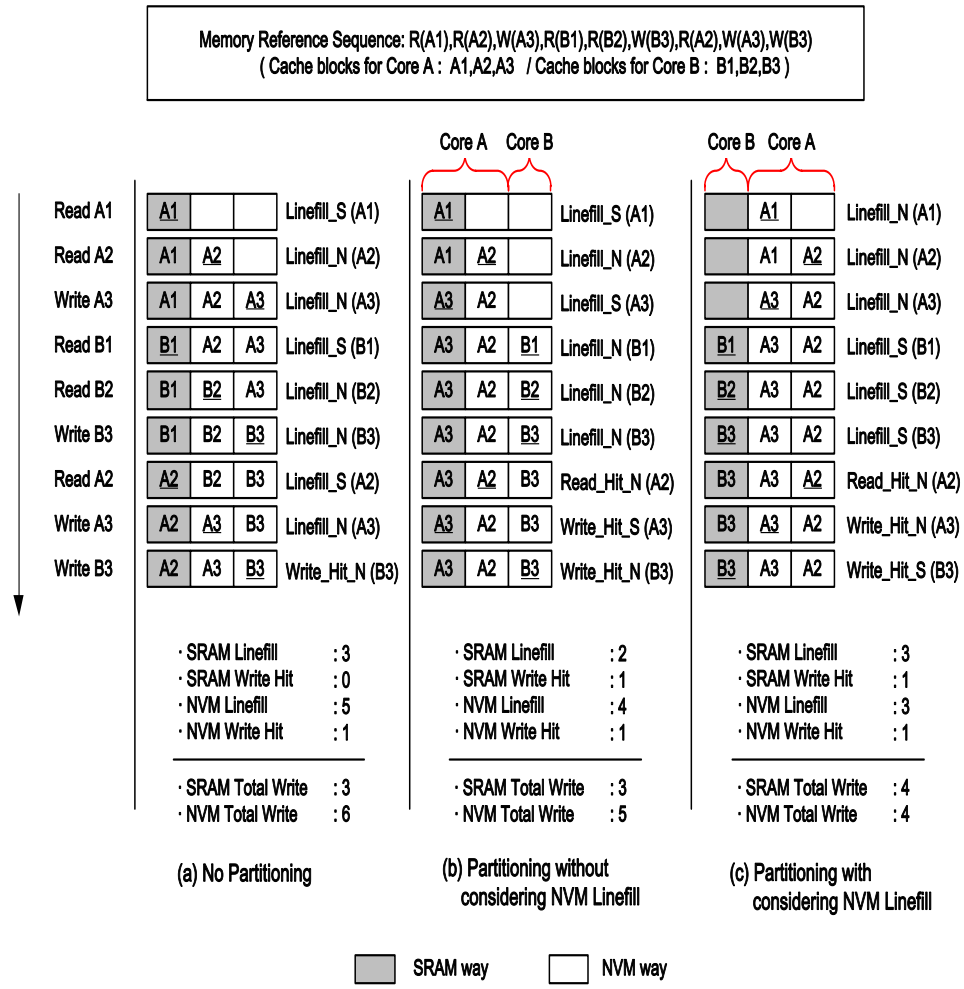


Figure 13: (a) No partitioning is applied. (b) Partitioning without NVM linefill. (c) Partitioning with NVM linefill.

4.3.1 Linefill-aware cache partitioning

To optimize the NVM write counts in HCA, SRAM ways and NVM ways should be efficiently allocated to cores. To help the understanding, we provide an illustration in Figure 13. The cache in this example consists of one SRAM way and two NVM ways. We assumed that there are two cores: core A and core B. A1, A2, and A3 are cache blocks for core A, and B1, B2, and B3 are cache blocks for core B. The cache accesses occur as the memory reference sequence shown in the box of the top in Figure 13.

When there is no special care for the LLC, the total write for the SRAM way is 3 (3 for SRAM linefill) and the NVM total write is 6 (5 for NVM linefill and 1 for NVM write hit), as shown in Figure 13(a). If the cache partitioning only considering the cache misses is applied [39], core A can occupy two cache ways and only one cache way can be assigned to core B (Figure 13(b)). Even though this partitioning decreases two cache misses and one NVM total write, the NVM write counts are not optimized. If a partitioning algorithm can predict the NVM linefill counts as well as the NVM write hit counts for every possible partitioning, the SRAM way should be allocated to core B to minimize the NVM write counts, as shown in Figure 13(c).

Therefore, a new scheme is required to reduce both the NVM write hit counts and the NVM linefill counts, which saves dynamic energy consumption of HCA. This paper devises new metrics to evaluate the effectiveness of cache partitioning schemes and proposes a linefill-aware cache partition-

Table 5: Notation descriptions for metrics.

Notation	Description
$H[i]$	Hit counts of *i*th recency position
$WH[i]$	Write hit counts of *i*th recency position
M_{CONF}	Conflict misses which are the number of cache misses due to partitioning
M_{NON_CONF}	Non-conflict misses which are the number of cache misses regardless of partitioning
$H(N)$	Total cache hit counts when the number of allocated ways is N
$M(N)$	Total cache misses when the number of allocated ways is N
$W(N)$	Total write counts when the number of allocated ways is N
$WH(N)$	Total write hit counts when the number of allocated ways is N
$\Delta M(N,N')$	Miss counts change when the number of allocated ways is changed from N to N'
$\Delta W(N,N')$	Write counts change when the number of allocated ways is changed from N to N'
$\Delta NVMW(N_{SRAM},N'_{SRAM},N_{NVM},N'_{NVM})$	NVM write counts change when the number of allocated SRAM ways is changed from N_{SRAM} to N'_{SRAM} and the number of allocated NVM ways is changed from N_{NVM} to N'_{NVM}

ing scheme (LCP) based on these metrics. Table 5 provides a description of notation we define in this section.

4.3.2 Metrics for cache partitioning

This section describes three metrics for a partitioning decision: Miss counts change (ΔM), write counts change (ΔW), and NVM write counts change ($\Delta NVMW$). We newly devise ΔW and $\Delta NVMW$ and redefine ΔM by revisiting the concept of "the utility" in the previous work [39].

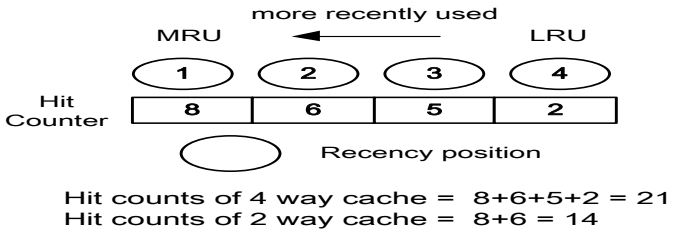


Figure 14: Example of stack property.

- ΔM: the change of the number of cache misses according to adjusting cache capacity for each core. This metric has been usually adopted to decide the cache partition to improve the performance in previous studies.

- ΔW: the change of the write counts for both the SRAM write counts and the NVM write counts according to the change of cache capacity.

- $\Delta NVMW$: the change of the NVM write counts. It is used for the HCA when memory elements are heterogeneous, while two other metrics can be applied in the cache consisting of the same memory element.

The concept of the utility [39] is used to estimate the reduction in the number of cache misses when a core has extra cache ways. The original paper named this concept as ”the utility”, but we redefine it ”ΔM” to clarify its meaning. They noticed that LRU replacement policy followed the stack property [45]. If a cache has the stack property, the cache having more cache ways always contains all blocks of the cache having smaller cache ways when the number of sets remains. Figure 14 presents the hit counts of each

recency position in a 4-way cache. In general, the recency position of the block at MRU position is called position 1, and that of the LRU position is called position 4. In this example, if the number of cache ways is reduced to 2 from 4, we expect that the hit counts of the cache will decrease by one-thirds without performing the experiments for a 2-way cache.

ΔM indicates the change of the miss counts with the change of the number of allocated ways[1]. Let $H[i]$ denote the hit counts of ith recency position of a core and $H(N)$ be the total hit counts when the number of allocated ways is N of the core. A relationship is established between two metrics.

$$H(N) = \sum_{i=1}^{N} H[i] \tag{4.18}$$

Since the increase in the miss counts is the same as the reduction in the hit counts, when the number of allocated ways is changed from N to N' of a core, $\Delta M(N,N')$ is given by

$$\Delta M(N,N') = -(H(N') - H(N)) = \sum_{i=1}^{N} H[i] - \sum_{i=1}^{N'} H[i] \tag{4.19}$$

A new model is built to estimate the change of the number of write operations with the change of the capacity in the cache. Since improving the hit rate is the most important goal in previous studies, ΔM is the only

[1] To clear the meaning of the terminology, the number of cache ways assigned for a core are called "the number of allocated cache ways of the core"

metric for cache partitioning in SRAM-based LLC in CMP environment. However, minimizing the write counts should be considered as well as maximizing the overall hit counts in HCA. Thus, we define a new metric (ΔW) for representing the change of the number of write accesses caused by the change of partitioning.

The change of write counts over the change of the amount of allocated ways is not easily determined, while ΔM is obtained by just accumulating $H[i]$. A cache block of the LLC is updated by two cases. First, when a write hit occurs in the LLC, the corresponding block is overwritten. In addition, if a new block is loaded due to a cache miss, the contents of the block are updated. Therefore, the write counts change (ΔW) is the sum of the write hit counts and the linefill counts.

To find the total write hit counts, we define $WH[i]$ as the write hit counts for ith recency position. The write hit counts $WH(N)$ is expressed in a similar form as the hit counts.

$$WH(N) = \sum_{i=1}^{N} WH[i] \tag{4.20}$$

Calculating the total linefill operations is more complicated than obtaining the total write hit counts because there are two kinds of cache misses to be considered. The first category of the cache miss is called a conflict miss (M_{CONF}), which occurs when a core partially uses the LLC due to cache partitioning. If all cache ways are allocated to the core, the amount of the

conflict miss becomes zero; thus, it varies across resizing the number of allocated ways. On the other hand, there is another kind of cache miss, called a non-conflict miss (M_{NON_CONF}), which occurs regardless of partitioning. In other words, when a core utilizes all cache ways, there is no M_{CONF} in the core, while M_{NON_CONF} can occur. Note that the non-conflict miss is composed of two kinds of misses, usually referred to as capacity and compulsory misses [48]. In our proposal, we use a single term as a non-conflict miss because there is no need to distinguish these misses.

Combining the two cache misses, the miss counts ($M(N)$) can be written as follows:

$$\begin{aligned} M(N) &= M_{CONF} + M_{NON\_CONF} \\ &= H(N_{ALL}) - H(N) + M_{NON\_CONF} \\ &= \sum_{i=1}^{N_{ALL}} H[i] - \sum_{i=1}^{N} H[i] + M_{NON\_CONF} \end{aligned} \tag{4.21}$$

where N_{ALL} is the number of total cache ways in the LLC.

To put it all together, $W(N)$ is expressed as

$$W(N) = WH(N) + M(N) \tag{4.22}$$

Since $\Delta W(N,N')$ means the change of the write counts, we reach the following equation:

$$\Delta W(N,N') = (WH(N') + M(N')) - (WH(N) + M(N)) \tag{4.23}$$

From Eq. 4.20 and Eq. 4.21, we transform Eq. 4.23 into the following:

$$\begin{aligned}\Delta W(N,N') = &(\sum_{i=1}^{N'} WH[i] + \sum_{i=1}^{N_{ALL}} H[i] - \sum_{i=1}^{N'} H[i] + M_{NON\_CONF}) \\ &-(\sum_{i=1}^{N} WH[i] + \sum_{i=1}^{N_{ALL}} H[i] - \sum_{i=1}^{N} H[i] + M_{NON\_CONF})\end{aligned} \tag{4.24}$$

This can be written in this form:

$$\begin{aligned}\Delta W(N,N') = &\sum_{i=1}^{N'} WH[i] - \sum_{i=1}^{N'} H[i] - \sum_{i=1}^{N} WH[i] + \sum_{i=1}^{N} H[i] \\ &+ (\sum_{i=1}^{N_{ALL}} H[i] - \sum_{i=1}^{N_{ALL}} H[i]) + (M_{NON\_CONF} - M_{NON\_CONF})\end{aligned} \tag{4.25}$$

$H(N_{ALL})$ and M_{NON_CONF} in the above equation are removed because they do not change with the number of allocated ways. Therefore, after simplifying Eq. 4.25, this becomes

$$\Delta W(N,N') = \sum_{i=1}^{N'} (WH[i] - H[i]) - \sum_{i=1}^{N} (WH[i] - H[i]) \tag{4.26}$$

To aid the understanding of the equation, we provide illustrations in Figure 15. In this figure, Eq. 4.26 is applied to find the write counts change, while Eq. 4.19 is used to calculate the miss counts change. When the amount of allocated ways is increased to 3 from 2 ($N = 2$ and $N' = 3$), $\Delta M(2,3)$ is -5 and $\Delta W(2,3)$ is -3.

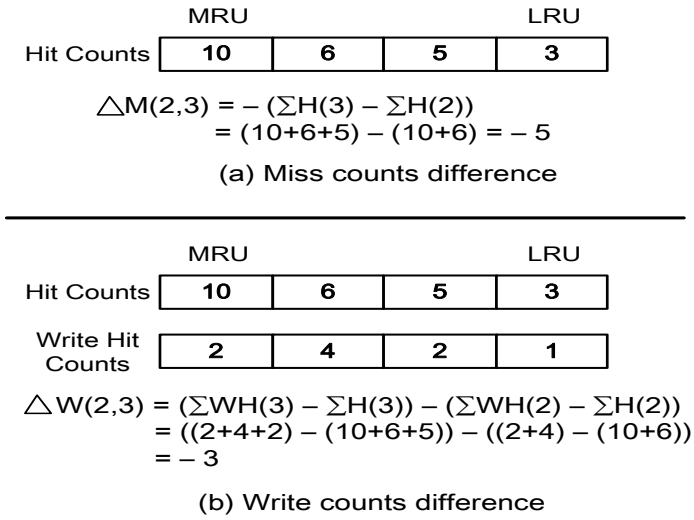


Figure 15: Examples of (a) miss counts change (ΔM) and (b) write counts change (ΔW).

This section describes the NVM write counts change ($\Delta NVMW$) used for calculating the variation of the write accesses to NVM in HCA. In the above section, we showed that the write counts are changed, but it is only applied in the LLC, which has one memory type. Thus, another metric is required to measure the change of NVM write counts. Note that $\Delta NVMW$ has four kinds of parameters because two types of memory elements are considered in this model. N is divided into N_{SRAM} and N_{NVM}, which are the number of allocated SRAM ways and NVM ways before new partitioning is initiated, respectively. Instead of N', N'_{SRAM} and N'_{NVM} are used to indicate how many SRAM ways and NVM ways are allocated to a specific core based on the new partitioning. Therefore, this metric is expressed as $\Delta NVMW(N_{SRAM},N'_{SRAM},N_{NVM},N'_{NVM})$.

We propose a new method to measure the variation of the write counts of NVM because the methods on the stack property cannot calculate the exact change of the write counts of NVM. For example, when a certain NVM way receives five write requests, removing the NVM way does not decrease the write counts of NVM by five. Since the concept of recency position is independent to the order of way, every way can have any recency position and the position usually changes after every cache access. When the number of allocated ways is changed, the blocks are stored into different ways from they were, and the hit counts of each way are not reserved. Therefore, it is impossible to exactly predict the change of the write counts of NVM or SRAM when the number of the allocated cache ways is changed.

Instead, we use a statistical approach to find the NVM write counts. In general, every way has the same probability of receiving write requests, which means write requests are statistically evenly distributed among the ways. Therefore, the portion of the NVM write counts over the all write counts is assumed to be proportional to the ratio of the number of NVM ways over the total number of cache ways.

$$\begin{aligned} &NVMW(N_{SRAM}, N_{NVM}) \approx \\ &W(N_{SRAM} + N_{NVM}) * \frac{N_{NVM}}{N_{SRAM} + N_{NVM}} \end{aligned} \tag{4.27}$$

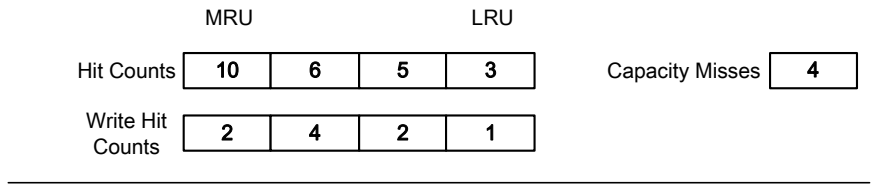
Therefore, $\Delta NVMW$ is calculated as follows:

$$\begin{aligned}
&\Delta NVMW(N_{SRAM}, N'_{SRAM}, N_{NVM}, N'_{NVM}) \\
&= NVMW(N'_{SRAM}, N'_{NVM}) - NVMW(N_{SRAM}, N_{NVM}) \\
&= W(N') * \frac{N'_{NVM}}{N'} - W(N) * \frac{N_{NVM}}{N}
\end{aligned} \tag{4.28}$$

$$\begin{aligned}
&= (WH(N') + M(N') + M_{NON_CONF}) * \frac{N'_{NVM}}{N'} \\
&-(WH(N) + M(N) + M_{NON_CONF}) * \frac{N_{NVM}}{N}
\end{aligned} \tag{4.29}$$

$$\begin{aligned}
&= (\sum_{i=1}^{N'} WH[i] + \sum_{i=N'+1}^{N_{ALL}} H[i] + M_{NON_CONF}) * \frac{N'_{NVM}}{N'} \\
&-(\sum_{i=1}^{N} WH[i] + \sum_{i=N+1}^{N_{ALL}} H[i] + M_{NON_CONF}) * \frac{N_{NVM}}{N}
\end{aligned} \tag{4.30}$$

Figure 16 shows the procedure of calculation of the equation. On top of the write hit counters, a non-conflict miss counter is inserted. A cache in the example is composed of two SRAM ways and two NVM ways. We assume that a core takes one SRAM way and one NVM way at first. If one more way is assigned to the core, there are two options; the core gets either an extra NVM way or SRAM way. For former case, we add an NVM way to the core, $\Delta NVMW$ is increased by 1. On the contrary, the latter case shows that $\Delta NVMW$ becomes -4.



△NVMW(1,1,1,2) = ∑NVMW(1,1) – ∑NVMW(1,2)
= (∑WH(2) + ∑M(2)) * (1 / 2) – (∑WH(3) + ∑M(3)) * (2 / 3)
= ((2+4) + (5+3+4)) * (1 / 2) – ((2+4+2) + 3 + 4) * (2 / 3) = – 1

(a) An NVM way is added (1S1N -> 1S2N)

△NVMW(1,1,1,2) = ∑NVMW(1,1) – ∑NVMW(2,1)
= (∑WH(2) + ∑M(2)) * (1 / 2) – (∑WH(3) + ∑M(3)) * (1 / 3)
= ((2+4) + (10+6+4)) * (1 / 2) – ((2+4+2) + 6 + 4) * (1 / 3) = – 4

(b) An SRAM way is added (1S1N -> 2S1N)

Figure 16: Examples of NVM write counts change ($\Delta NVMW$). Initially, a core owns an SRAM way and an NVM way (1S1N). (a) The core acquires one more NVM way (1S2N). (b) The core acquires one more SRAM way (2S1N).

4.3.3 Algorithm for cache partitioning

The algorithm for LCP consists of two steps to optimize the NVM write counts without increasing cache misses, as shown in Figure 17. The first step is finding the best partitions for optimizing the linefill counts. LCP utilizes ΔM to search for the optimal size of partition in this step. After that, the SRAM partition and NVM partition of each core are determined within its budget determined by the first step, based on ΔW and $\Delta NVMW$. Table 6 lists the description of notation we define in this section.

To make our algorithms more efficient, we employ the concept of the marginal utility approach introduced in UCP [39]. Since prior studies of

Algorithm 1 : Linefill-aware Cache Partitioning

Step 1 : finding the number of allocated cache ways

1 : $U_{ALL} \leftarrow T_{ALL}$ - T_{CORE}
2 : **foreach** *i* ← *all cores* **do**
3 : $A_{ALL}[i] \leftarrow 1$
4 : **end if**
5 : **while** $U_{ALL} > 0$ **do**
6 : min_MU ← ∞
7 : **foreach** *i* ← *all cores* **do**
8 : **for** *w* ← 1 **to** U_{ALL} **do**
9 : MU ← Δ*M* ($A_{ALL}[i]$,$A_{ALL}[i]$+w) / w
10: **if** *MU* < *min_MU* **do**
11: min_MU ← MU
12: $C_{CORE} \leftarrow i$
13: Req ← w
14: **end if**
15: **end for**
16: **end foreach**
17: $A_{ALL}[C_{CORE}] \leftarrow A_{ALL}[C_{CORE}]$ + Req
18: $U_{ALL} \leftarrow U_{ALL}$ - Req
19: **end while**

Step 2 : finding the number of allocated NVM ways

20: $U_{SRAM} \leftarrow T_{SRAM}$
21: **foreach** *i* ← *all cores* **do**
22: $A_{NVM}[i] \leftarrow A_{ALL}[i]$
23: **end foreach**
24: **while** $U_{SRAM} > 0$ **do**
25: **foreach** *i* ← *all cores* **do**
26: min_MU ← ∞
27: **if** $U_{SRAM} > A_{ALL}[i]$ **then**
28: w' ← $A_{ALL}[i]$
29: **else**
30: w' ← U_{SRAM}
31: **end if**
32: **for** *w* ← 1 **to** *w'* **do**
33: **if** $U_{SRAM} == 0$ and $A_{SRAM}[i] == 0$ **do**
34: MU ← Δ*W*($A_{NVM}[i]$,$A_{NVM}[i]$ + w) / w
35: **else**
36: MU ← Δ*NVMW*($A_{SRAM}[i]$,$A_{SRAM}[i]$ + w, $A_{NVM}[i]$,$A_{NVM}[i]$ - w) / w
37: **end if**
38: **if** *MU* < *min_MU* **do**
39: min_MU ← MU
40: $C_{CORE} \leftarrow i$
41: Req ← w
42: **end if**
43: **end for**
44: **end foreach**
45: $A_{SRAM}[C_{CORE}] \leftarrow A_{SRAM}[C_{CORE}]$ + Req
46: $A_{NVM}[C_{CORE}] \leftarrow A_{ALL}[C_{CORE}]$ - $A_{SRAM}[C_{CORE}]$
47: $U_{SRAM} \leftarrow U_{SRAM}$ - Req
48: **end while**

Figure 17: Algorithm of linefill-aware cache partitioning (LCP).

Table 6: Notation descriptions for algorithms.

Notation	Description
T_{ALL}	Number of total cache ways in the LLC
T_{SRAM}	Number of total SRAM ways in the LLC
T_{NVM}	Number of total NVM ways in the LLC
T_{CORE}	Number of total cores
U_{ALL}	Number of unallocated ways
U_{SRAM}	Number of unallocated SRAM ways
U_{NVM}	Number of unallocated NVM ways
$A_{ALL}[i]$	Number of allocated ways per *i*th core
$A_{SRAM}[i]$	Number of allocated SRAM ways for *i*th core
$A_{NVM}[i]$	Number of allocated NVM ways for *i*th core
MU	Marginal utility of metrics
min_MU	Minimum value of marginal utility
Req	Number of requesetd ways to get min_MU
C_{CORE}	A specific core gaining extra cache ways

NVM-based CMP used the greedy algorithm [49, 50], there is a risk of reaching to a suboptimal partitioning, which commonly occurs in greedy algorithms. To avoid this problem, LCP uses the marginal utility. Therefore, our algorithm uses a value which is divided by the number of allocated ways instead of the value directly obtaining from the calculation. For example, if ΔW is -4 and the number of allocated ways is 2, the marginal utility (MU) of ΔW is -2 (= -4 / 2). In addition, the partitioning algorithm is designed to perform the cache repartitioning every 1M cycles because it shows the best efficiency compared with other periods.

Step 1 starts initializing U_{ALL}, which is a key variable of the first loop (line 1). Since each core has at least one way, U_{ALL} has the difference be-

tween the number of total cache ways in the LLC and the number of cores (line 2-4). Step 1 is executed until all ways are assigned to cores (line 5). When each iteration begins, *min_MU* is initialized to infinity; in reality, it has the maximum integer value that a system allows (line 6). For every core, ΔM per way are calculated by varying the number of allocated cache ways (line 7-9). If *MU* is smaller than the currently minimum value of *MU* (line 10), *min_MU* is updated (line 11), and the current core is tentatively indicated as the target core to be allocated more cache ways (line 12). *Req* has the current number of allocated ways (line 13). When the loop ends, the requested ways are allocated to the target core (line 17) and U_{ALL} is updated as well (line 18). Note that this step is performed based on the UCP [39], which is known as one of the best partitioning schemes. Because this step is orthogonal to second step, other partitioning schemes can be used if they provide the better partitioning efficiency.

Step 2 works similar to step 1, but a key variable of the loop becomes U_{SRAM} substituting U_{ALL} and $\Delta NVMW$ and ΔW are used instead of ΔM because SRAM ways are distributed among cores in this step. At first, U_{SRAM} has the number of SRAM cache ways (line 20). The number of the allocated NVM ways for each core is temporarily the number of allocated cache ways, which is determined by the previous step (line 22-24). Another difference from step 1 is that a loop for finding the *min_MU* is iterated when the candidate number of cache ways is from 1 to the maximum value between $A_{ALL}[i]$ and U_{SRAM} (line 27-31). This is because each core cannot have more ways than $A_{ALL}[i]$. $\Delta NVMW$ is basically used to find the value of *MU* (line 36),

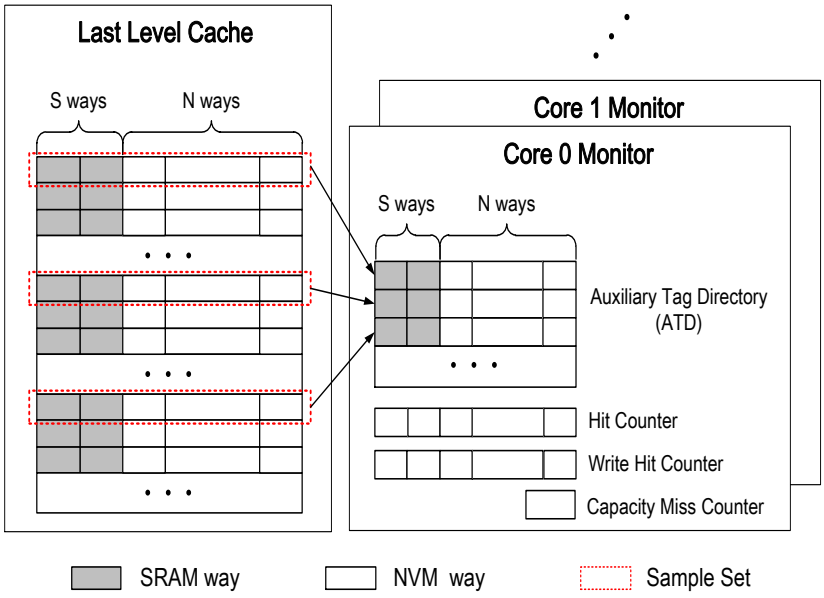


Figure 18: Overall structure of LCP.

however ΔW is applied for simplicity if it is guaranteed that no SRAM way involves calculation (line 34). In this algorithm, the number of NVM ways are simply calculated; we obtain it by subtracting $A_{ALL}[i]$ to $A_{SRAM}[i]$ (line 46).

We extend the conventional utility monitor [39] and utilize a cache partitioning logic of UCP to implement our proposal. Therefore, storage overhead is estimated as less than 1%. The traditional utility monitor contains an auxiliary tag directory (ATD) and hit counters. On top of that, two additional counters are added which are a write hit counter and a non-conflict miss counter, as depicted in Figure 18. As many write hit counters as the number of cache ways are needed, and only a single counter is required for

accumulating the non-conflict misses.

The role of the ATD is keeping track of the recency positions of blocks for each core. Using the ATD, the hit counter indicates the hit counts of each recency position. Similar to the hit counter, the write hit counters store the number of write hit for the corresponding position. The associativity of the hit counter and the write hit counter is the same as the LLC. The non-conflict miss counter is inserted to obtain the total non-conflict miss counts. If a cache miss occurs in the ATD, the non-conflict miss counter is increased by one, while the hit counter is increased when a cache hit occurs in the corresponding recency position.

Assuming that the LLC has 16-way associativity and the size of each counter is 32 bits, the total storage overhead of the LCP is (16 + 1) * 32 bits = 68 bytes. Considering the capacity of the LLC is 2MB in our system, it is obvious that the storage overhead is not significant.

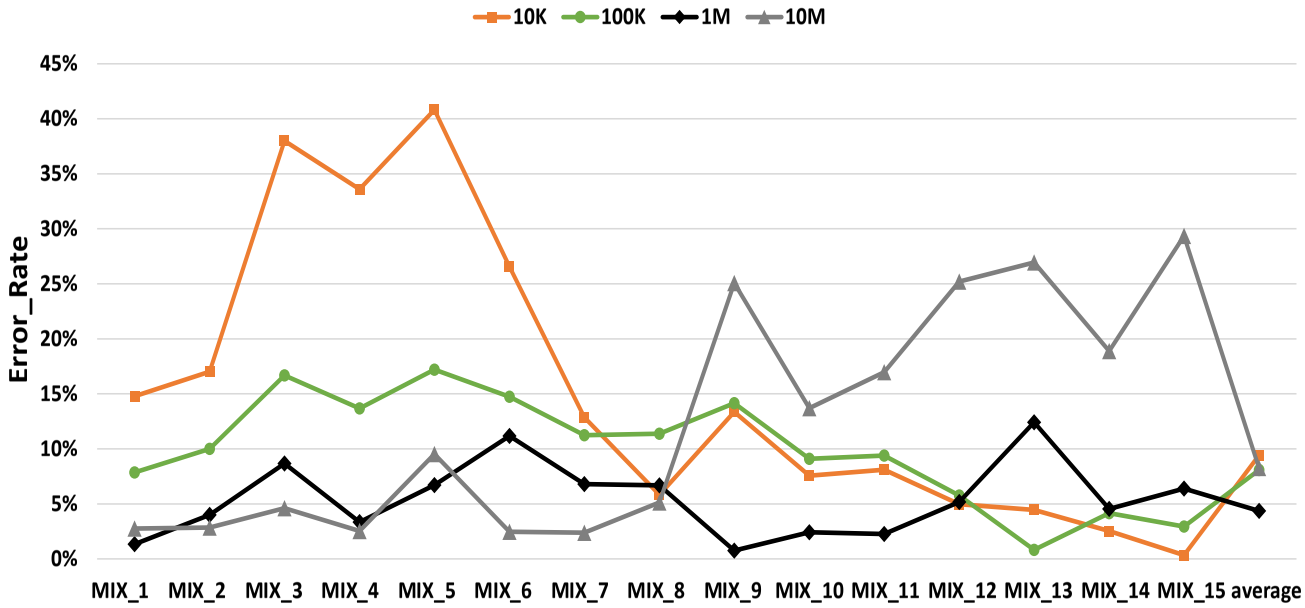


Figure 19: Error rates with various repartitioning period from 10K to 10M.

We start by analyzing how accurate the proposed algorithm predicts the NVM write counts. Whenever the cache partitioning is done, the expected NVM write counts during the execution period is accumulated. At the end of the program execution, the difference between the predicted value and the measured value is used to calculate the error value of the algorithm. In this way, we estimate the error rate of our algorithms as follows:

$$ErrorRate = \frac{|Predicted\ NVM\ Writes - Measured\ NVM\ Writes|}{Predicted\ NVM\ Writes} * 100 \tag{4.31}$$

Figure 19 summarizes error rates of our algorithm with various sizes of repartitioning periods from 10K to 10M. LCP utilizes the statistics of each period to predict the behavior of the next. If a previous period has a similar access pattern of the following period, this approach will be effective. Unfortunately, if partitioning occurs in the middle of transition of working sets in the program, the information gathered by the ATD during the current period does not represent the next period. In this case, the accuracy of hit counts, write hit counts, and cache misses will decrease. Thus, we have experimented with several repartitioning periods and the consequential change of the accuracy. The proposed LCP with the 1M period cycle shows that the error rate is 4.3%, which is meaningfully lower than the error rate of other period sizes. Therefore, we choose 1M as the repartitioning period for our proposal.

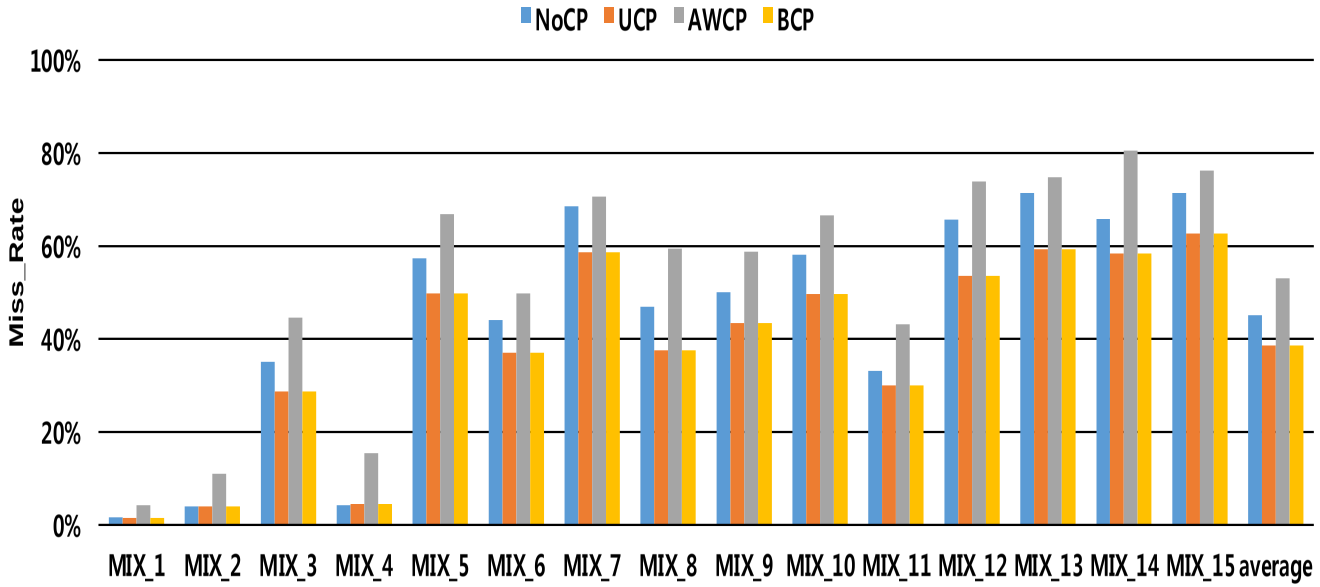


Figure 20: Miss rates with four schemes.

The miss rates for all workloads are given in Figure 20 for NoCP, BSABM, AWCP, and LCP. AWCP shows the worst miss rate for all benchmark programs because the number of cache ways for each core is adjusted according to its NVM write intensity. Even though this approach is beneficial to reducing the number of write counts, it is not helpful to improve the total hit counts. The miss rate of BSABM is the nearly same as NoCP because they use a similar replacement policy. The miss rate of LCP is decreased by 4.3% over NoCP, and the difference between average miss rate of AWCP and LCP is 13.7%. While the efficiency of LCP varies significantly depending on characteristics of workload, the miss rates of all applications are decreased. For MIX_4, the miss rate of LCP is lower than that of AWCP by 21.9%.

4.4 Overhead of NVM capacity management policy

Table 7 shows the storage overhead of the DWA. We assume that the system uses a 40-bit physical address space. To keep track of the MSD, an entry of the ATD has a separate tag and LRU bits. The each ATD has 64 entries and 256 entries because the number of sample sets is 64 and 256 respectively. The size of R bits is 12 as the number of NVM ways is 12. The DWA also needs three kinds of 4-bit registers and a 2-bit resizing counter register. Both HCAs have about less than 1% extra area. With a low hardware overhead, our proposal achieved the dynamic energy saving and write endurance en-

Table 7: Storage overhead.

Component	HCA with STT-RAM	HCA with PCM
ATD entry	LRU + Tag + Valid = 4 + 22 + 1 = 27 bits 27 bits * 16 way = 54 bytes	LRU + Tag + Valid = 4 + 20 + 1 = 25 bits 25 bits * 16 way = 50 bytes
ATD	54 bytes * 64 sets = 3.8KB	50 bytes * 256 sets = 12.5KB
R bits	12 bits	12 bits
LPR	4 bits	4 bits
MSDR	4 bits	4 bits
RWSR	4 bits	4 bits
RCR	2 bits	2 bits
Overhead for LCP	(16 + 1) * 32 bits = 68 bytes	(16 + 1) * 32 bits = 68 bytes
Total	about 4KB (0.1%)	about 13KB (0.31%)

hancement. For the LCP, as we discussed earlier, the total storage overhead of the LCP is (16 + 1) * 32 bits = 68 bytes on top of the extra storage of the DWA. Therefore, the storage overhead of both schemes is not significant.

Another consideration for cache partitioning is the timing overhead of obtaining the optimal value. To investigate the timing overhead, we calculated the latencies of the algorithm in detail as shown in Table 8. According to Eq. 4.19 one iteration of the main loop of step 1 requires one addition, one subtraction, one division, one comparison, and one assignment. The latencies of an adder and a comparator are one cycle and the latency of a divider is thirteen cycles in modern processors [43], thus one iteration takes 17 cycles (we assume that each register captures the value in a cycle). Ac-

Table 8: Timing overhead.

Component	Cycles
Step1 Initialization (line 1-4)	2 cycles
Step1 Main loop (line 6-16)	17 cycles
Step1 Result assigning (line 17-18)	2 cycles
Step2 Initialization (line 20-23)	3 cycles
Step2 Main loop preparation (line 24-31)	2 cycles
Step2 Main loop (line 32-44)	36 cycles
Step2 Result assigning (line 45-47)	3 cycles
Total	851 cycles (0.9%)

cording to Eq. 4.30, one iteration of the main loop of the step 2 requires three additions, one multiplication, two divisions, one comparison, and one assignment. The latency of a multiplier is five cycles in modern processors [43], thus one iteration takes 36 cycles.

The initialization steps are executed once for every partitioning. The main loop in step one of LCP is iterated 24.95 times and the main loop in step two is iterated 10.21 times. The other parts of the algorithm are executed 4.57 times and 2.31 times for each step respectively. Therefore, the algorithm takes 851 cycles to identify the average of the partitioning (2+17*25+2*5+3+2*3+36*11+3*3 = 851). Considering that the period of partitioning is 1M, the latency of the algorithm does not have an influence on the overall performance.

Chapter 5

Experimental results

5.1 Experimental environment

We simulated our approach with PARSEC benchmark suite [11] for evaluating WACC. The gem5 simulator is used to evaluate the normalized energy and normalized lifetime of our protocol [9]. The overall simulation parameters are shown in Table 9. We assume that the cache coherence protocol is a MOESI protocol. In addition, LLC is composed of STT-RAM because STT-RAM is considered as the right alternative among several types of NVM [51]. The power value of STT-RAM is derived from the previous work [52].

For DWA, a simulation was performed using Macsim [10] which is a trace-driven and cycle level simulator. It is designed to thoroughly model the detailed microarchitectural behavior, including pipeline stages and memory systems. Our baseline system has a three level cache hierarchy. The L1 and L2 caches are composed of the SRAM memory. Table 9 shows our baseline processor configurations in detail. Since STT-RAM and PCM are widely studied among several kinds of NVM, the LLC has two hybrid cache configurations: STT-RAM with SRAM, PCM with SRAM. We examined our proposal on multi core configuration which has 4 cores as well. We used

SPEC CINT2006 and SPEC CFP2006 of the SPEC CPU2006 benchmark suite [12]. Because the benchmark programs with the reference input set take a very long time to run, we simulated 500M instructions of the region selected by Pinpoints [53, 54] which is a well-known tool to find the representative regions. To compare our proposal with previous studies, we also conducted the experiments with prediction table based cache line replacement and management policy (PTHCM) [18]. For multi core system simulation, we generated ten workloads by mixing six applications as listed in Table 12.

In addition, the standard of normalization in our results is the baseline hybrid cache, which is operated as a conventional cache except that it consists of both SRAM and STT-RAM cells. Thus, the baseline hybrid cache has no special policy such as the DWA or the PTHCM. For DWA, note that write intensity block migration policy is always applied. Finally, we assume that cache hierarchy maintains inclusion property in our proposal as like many modern processors such as the Intel i7 processor [43] or ARM CORTEX-A57 processor [55].

We have performed experiments to evaluate the proposed cache partitioning scheme with Macsim [10] for LCP. Table 9 presents the system parameters used for the simulation. It has four cores and a two-level cache hierarchy. The capacity of the L1 instruction and data caches are 32KB, and they are 4-way associative caches. The LLC (L2) cache is a 2MB 16-way cache, which is composed of 4-way SRAM and 12-way NVM. The line size of all caches is 64B.

Table 9: Processor configurations.

WACC	
Cores	4
L1 Inst / Data Cache	64KB, 2-way, 64B line
L2 Unified Cache	2MB, 16-way, 64B line
Memory	64bit bus width , 4 read/write ports
Function Units	6 IALU, 2 IMULT, 4 FPALU, 2 FPMULT

DWA	
Core Type	x86, out-of-order, 2GHz
Core Count	1 / 4
INT / MEM / FP	4 / 4 / 4
Branch Predictor	gshare predictor, 16 history length
ROB Size	256
I/D Cache	16KB, 4-way, 64B blocks, 1-cycle latency
L2 Cache	512KB, 8-way, 64B blocks, 5-cycle latency
Hybrid LLC with STT-RAM	4MB(4-way SRAM and 12-way STT-RAM), 64B blocks SRAM: 10-cycle latency STT-RAM: 10-cycle (read) and 45-cycle (write) latency
Hybrid LLC with PCM	16MB(4-way SRAM and 12-way PCM), 64B blocks SRAM: 10-cycle latency PCM: 19-cycle (read) and 93-cycle (write) latency
Memory Latency	200 cycles

LCP	
Core Type	x86, out-of-order, 2GHz
Core Count	4
INT / MEM / FP	4 / 4 / 4
Branch Predictor	gshare predictor, 16 history length
ROB Size	256
I/D Cache	32KB, 4-way, 64B blocks, 2-cycle latency
Hybrid LLC	2MB(4-way SRAM and 12-way STT-RAM), 64B blocks
Memory Latency	200 cycles

We used SPEC CINT2006 and SPEC CFP2006 of the SPEC CPU2006 benchmark suite for the simulation [12] for LCP. To evaluate the efficiency of our proposal across write intensive and non-write intensive applications, workloads are created based on write counts per kilo-instructions (WBKI). At first, we sorted the applications by increasing the order based on WBKI as shown in Table 10 and divided them into three categories: such as low, mid, and high. Mixing four benchmarks from the three categories, we generated 15 workloads as listed in Table 11 (The number of combination of selecting 4 applications from 3 categories with repetitions is 15 and applications in each category are randomly selected.) Each trace is collected by Pinpoints [53], which is widely used to extract the representative regions.

There are four schemes tested in our simulation: the baseline which uses no partitioning scheme (NoCP), block swapping and active block migration (BSABM) [49], access-aware cache partitioning policy (AWCP) [50], and LCP proposed in the thesis. NoCP has no partitioning scheme and follows the LRU replacement. To compare the previous studies with our proposal, BSABM and AWCP, which are available for the HCA-based LLC in CMP, are included for the experiment.

To fairly compare the results of our proposal and previous studies, we used the same parameters of STT-RAM that were used in the previous study [50]; the dynamic energy consumption of cache operation for an SRAM cache bank 0.609nJ, while the read energy for an STT-RAM cache bank is 0.598nJ and the write energy is 4.375nJ.

Table 10: Write counts per kilo-instructions for LCP.

Type	Benchmark	WPKI	Type	Benchmark	WPKI
Low	dealII	0.90	Mid	zeusmp	30.92
	gamess	1.04		cactusADM	41.78
	gromacs	1.79		gcc	51.96
	povray	2.31		omnetpp	65.46
	perlbench	2.38	High	milc	75.94
	h264ref	4.13		wrf	92.29
	calculix	7.56		libquantum	114.29
	xalancbmk	8.10		GemsFDTD	133.44
Mid	gobmk	11.20		leslie3d	138.10
	hmmer	12.99		soplex	145.47
	tonto	13.53		lbm	221.45
	bzip2	15.75		mcf	228.77

Table 11: Multi-core workloads for LCP.

Workload	Benchmarks
MIX_1	dealII(L), gamess(L), calculix(L), xalancbmk(L)
MIX_2	gamess(L), gromacs(L), h264ref(L), cactusADM(M)
MIX_3	dealII(L), povray(L), xalancbmk(L), lbm(H)
MIX_4	gromacs(L), povray(L), gcc(M), omnetpp(M)
MIX_5	povray(L), perlbench(L), cactusADM(M), libquantum(H)
MIX_6	dealII(L), gamess(L), soplex(H), lbm(H)
MIX_7	xalancbmk(L), gobmk(M), cactusADM(M), omnetpp(M)
MIX_8	dealII(L), gcc(M), omnetpp(M), mcf(H)
MIX_9	povray(L), zeusmp(M), wrf(H), lbm(H)
MIX_10	povray(L), libquantum(H), lbm(H), mcf(H)
MIX_11	gobmk(M), hmmer(M), gcc(M), omnetpp(M)
MIX_12	gobmk(M), tonto(M), omnetpp(M), lbm(H)
MIX_13	hmmer(M), bzip2(M), leslie3d(H), lbm(H)
MIX_14	hmmer(M), GemsFDTD(H), leslie3d(H), mcf(H)
MIX_15	milc(H), wrf(H), lbm(H), mcf(H)

Table 12: Multi-core workloads for DWA.

Workload	Benchmarks
MIX_1	bwaves, calculix, wrf, gromacs
MIX_2	bwaves, calculix, wrf, hmmer
MIX_3	bwaves, calculix, wrf, h264ref
MIX_4	bwaves, calculix, gromacs, hmmer
MIX_5	bwaves, calculix, gromacs, h264ref
MIX_6	bwaves, calculix, hmmer, h264ref
MIX_7	bwaves, wrf, gromacs, hmmer
MIX_8	bwaves, wrf, gromacs, h264ref
MIX_9	bwaves, wrf, hmmer, h264ref
MIX_10	bwaves, gromacs, hmmer, h264ref
MIX_11	calculix, wrf, gromacs, hmmer
MIX_12	calculix, wrf, gromacs, h264ref
MIX_13	calculix, wrf, hmmer, h264ref
MIX_14	calculix, gromacs, hmmer, h264ref
MIX_15	wrf, gromacs, hmmer, h264ref

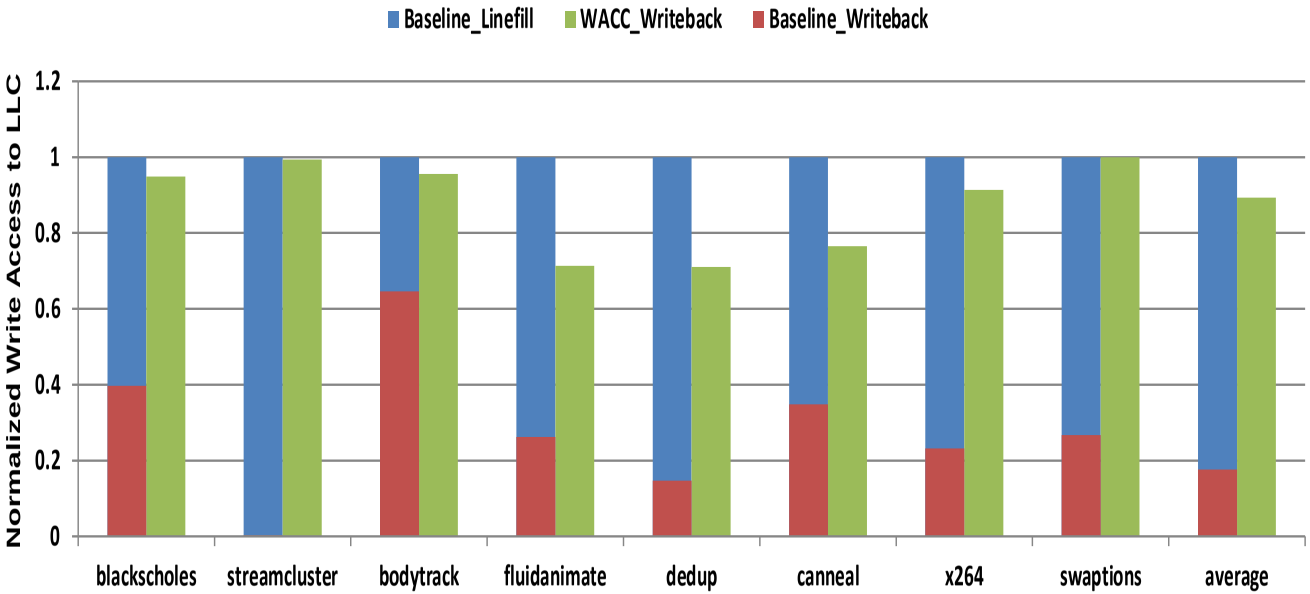


Figure 21: Normalized number of the access to LLC of WACC protocol compared to the MOESI protocol.

5.2 Write access to NVM

Figure 21 presents the normalized number of the read and write access to LLC in our protocol compared to the baseline MOESI protocol. Note that write access is divided into writeback access and linefill access. As a result, 13.2% of the write operations were decreased on average. The noticeable result is that the number of the writeback access was increased, while there were no linefill operation. When a cache block is evicted in a private cache, the writeback operation is not required in the existing protocols if the cache block is not modified. This is because the LLC already has the valid block data if the cache block is clean. On the contrary, the writeback operation should be initiated if no other private cache has the valid copy during cache replacement in WACC protocol. This difference generates the extra write-back operations. However, the total number of the write access in WACC protocol is smaller than that of other protocols because the reduction in the linefill operation is much larger than the increment in the writeback operation.

We first examined the write counts of NVM ways as depicted in Figure 22 and Figure 23. About 75.4% reduction and 77.2% reduction in the number of write accesses is achieved on average in the DWA for HCAs with STT-RAM and PCM, respectively, while the decrement on the number of write accesses to NVM ways of PTHCM are about 5.7% and 11.0%.

From the two figures, we discover that the write access reduction ratio of the DWA follows the sensitivity of the miss rate to the number of NVM

ways. First, low sensitive applications require a small number of NVM ways; therefore, the number of write accesses to NVM is largely reduced. On the contrary, highly sensitive applications show only a little change of write access because they have very little room for the DWA. To show this trend clearly, we calculate the reduction ratio of each category. For the left side applications, 92.2% reduction and 88.3% reduction in the write counts of STT-RAM and PCM ways is achieved on average, while 22.6% reduction and 55.6% reduction in the number of write accesses is achieved on average for the right side applications.

Furthermore, we combined the PTHCM with the DWA to check that it is orthogonally effective with other HCA algorithms. Since our proposal does not affect the fundamentals of operation of other HCA algorithms, the DWA can create a synergy effect. The results show that the PTHCM with the DWA (PTHCM_DWA) achieved the best results among four HCA algorithms as it showed 77.6% reduction and 80.0% reduction in write counts of NVM ways. Combining PTHCM with DWA reduces the write access to NVM more 8.9% when only DWA is applied for STT-RAM. In addition, PTHCM_DWA shows the lower NVM write counts by 11.0%. Therefore, we conclude that merging two algorithms takes advantage of both algorithms successfully.

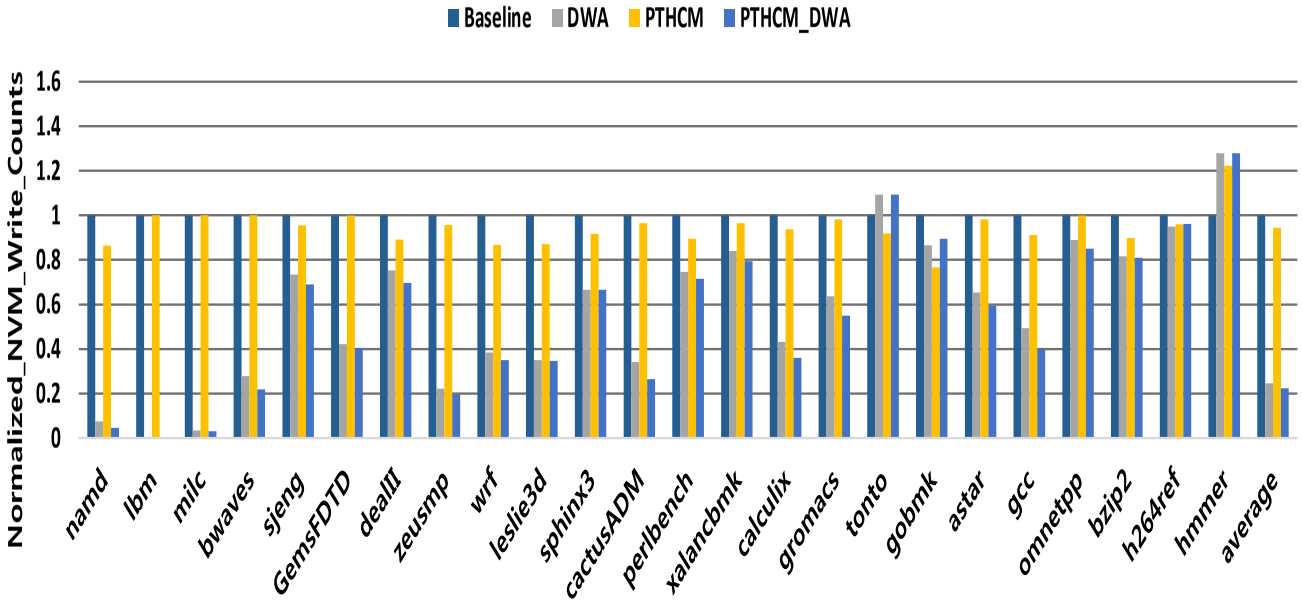


Figure 22: Normalized NVM write counts of DWA with STT-RAM.

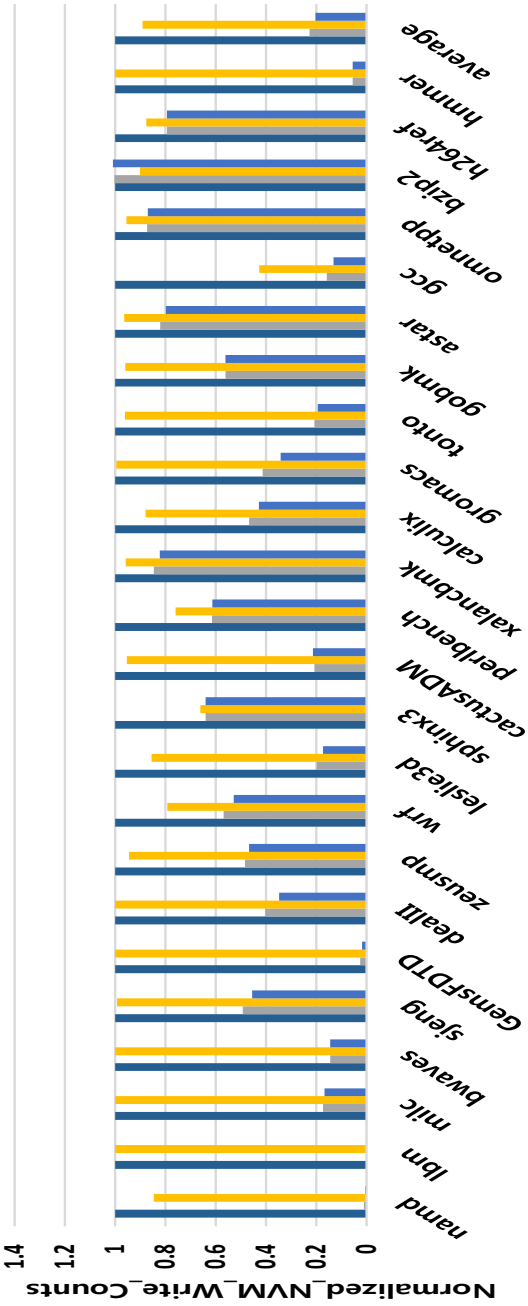


Figure 23: Normalized NVM write counts of DWA with PCM.

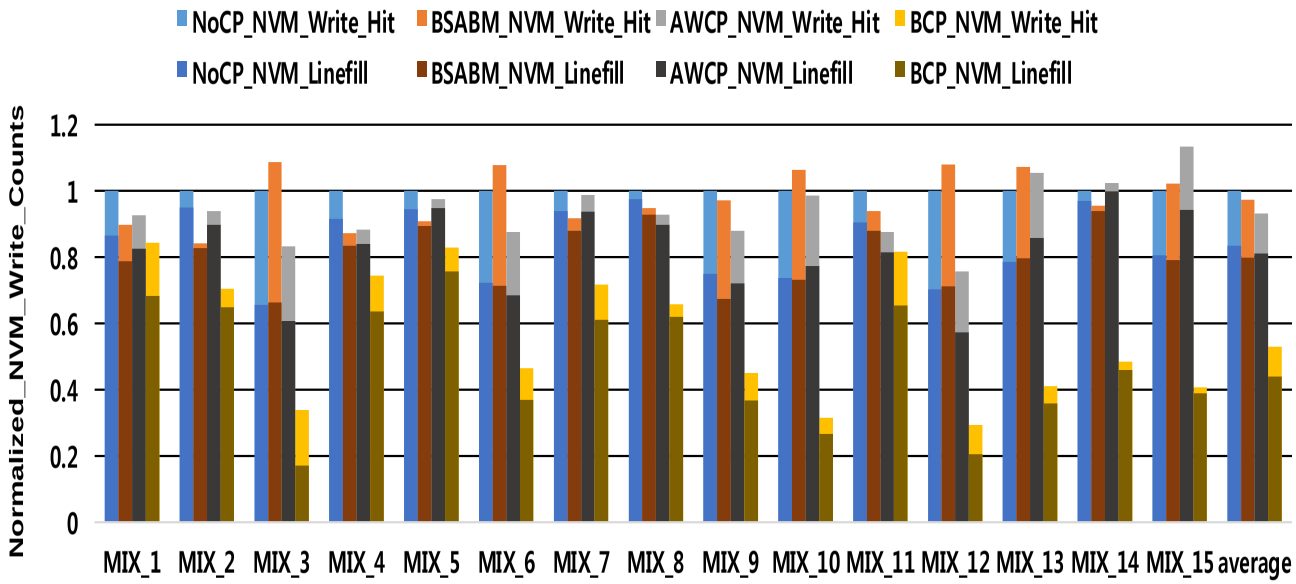


Figure 24: Normalized NVM write counts with four schemes.

Next, we analyze the NVM write counts of BSABM, AWCP, and LCP normalized to NoCP as depicted in Figure 24. The average value in the figure indicates the geometric mean of all workloads. BSABM and AWCP decreased the NVM write counts by 2.6% and 6.7%, respectively. LCP achieved a 46.9% reduction in the NVM write counts, which is the much better than previous studies. To investigate these results further, we divide the total NVM write counts into the NVM write hit counts and the NVM write linefill counts. At first, we found that the linefill operation occupies a significant portion of the NVM write counts. While the portion of the write hit counts is 16.5% on average, the portion of the NVM linefill counts is 83.5%. BSABM, AWCP, and LCP reduced the NVM write hit counts by 21.7%, 26.4%, and 39.2%, respectively. LCP shows the best results, and the previous schemes for HCA also achieved the meaningful reduction in the NVM write hit counts. On the contrary, the reduction ratio of the NVM linefill counts of BSABM and AWCP are only 4.3% and 2.8%, while LCP reduced the NVM linefill counts by 47.4%. These results confirm that LCP accomplishes the reduction in the NVM write counts by reducing the NVM linefill counts significantly as we intended.

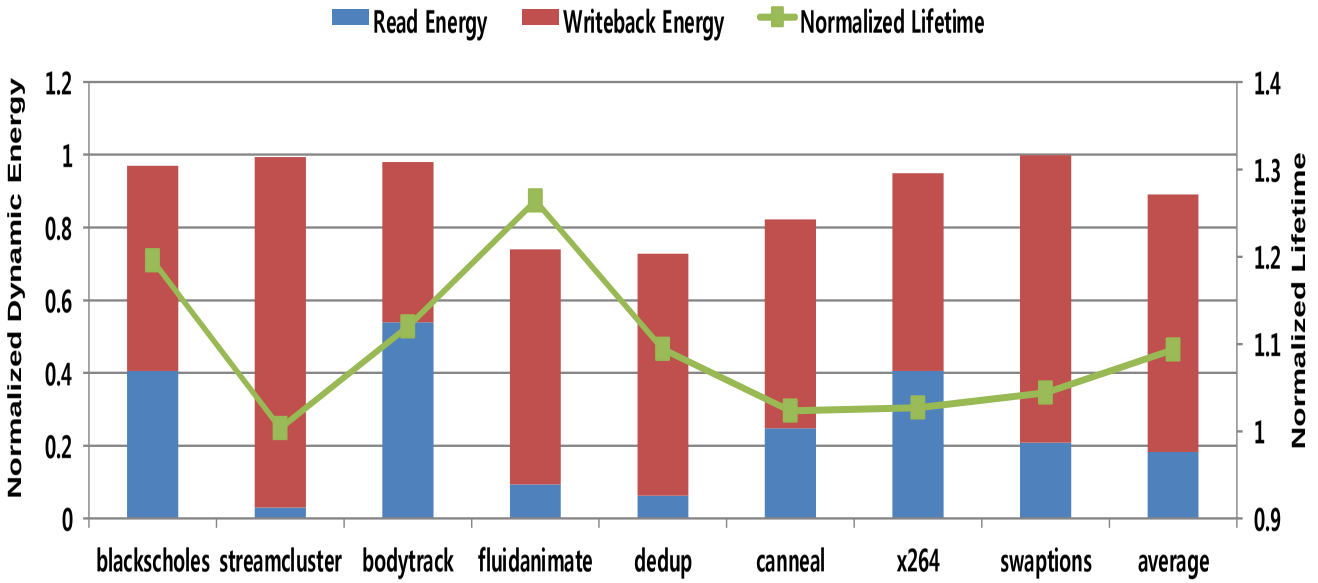


Figure 25: Normalized dynamic energy consumption and lifetime of WACC compared to the baseline MOESI protocol.

5.3 Dynamic energy consumption

We show the normalized dynamic energy consumption and lifetime in Figure 25. Since the dynamic energy in write operation dominates the dynamic energy consumption in read operation, the reduction of the write operations leads to reducing the total dynamic energy consumption. Our protocol achieves 27.1% energy savings at maximum and 10.8% energy savings on average. In addition, WACC protocol also extends the lifetime of the LLC because the lifetime of STT-RAM is inversely proportional to the number of write access to the LLC. The improvement of average write endurance in WACC protocol is 26.3% at maximum and 9.3% on average.

We investigated the normalized dynamic energy consumption compared to the baseline hybrid cache as shown in Figure 26 and Figure 27, which also present the portion of the write energy consumption of NVM over the total dynamic energy consumption. The results of HCA with STT-RAM show that the DWA achieved 26.4% reduction in the total dynamic energy consumption. The dynamic energy consumption of the PTHCM and the PTHCM_DWA was saved 2.3% and 28.4% over the baseline hybrid cache, respectively. For HCA with PCM, the DWA saved 27.4% of dynamic energy consumption, while the PTHCM and the PTHCM_DWA reduced the dynamic energy consumption by 2.7% and 30.0%. The trend of reduction is similar to that of reduction in the write accesses. This is because the dynamic energy consumption is mainly affected by the write accesses to STT-RAM.

Based on the observation of these figures, the write energy consumption of NVM occupies a significant portion of the total dynamic energy consumption. In the baseline hybrid cache, 78.6% and 56.0% of the dynamic energy was consumed due to the write accesses to STT-RAM and PCM ways. Therefore, we conclude that the number of write accesses to NVM ways is the most important factor for dynamic energy consumption. The results show that the portion of write dynamic energy of NVM ways was reduced to 32.8% and 14.7% in the DWA. The dynamic energy consumption of NVM write operations of the PTHCM occupies 74.3% and 48.8% of the total dynamic energy consumption. For the PTHCM_DWA, the portion is reduced to 30.0% and 14.1%. The reduction trend is also similar to that of the write access reduction. Therefore, the reduction in the dynamic energy consumption mainly comes from the reduction of the write energy consumption of NVM.

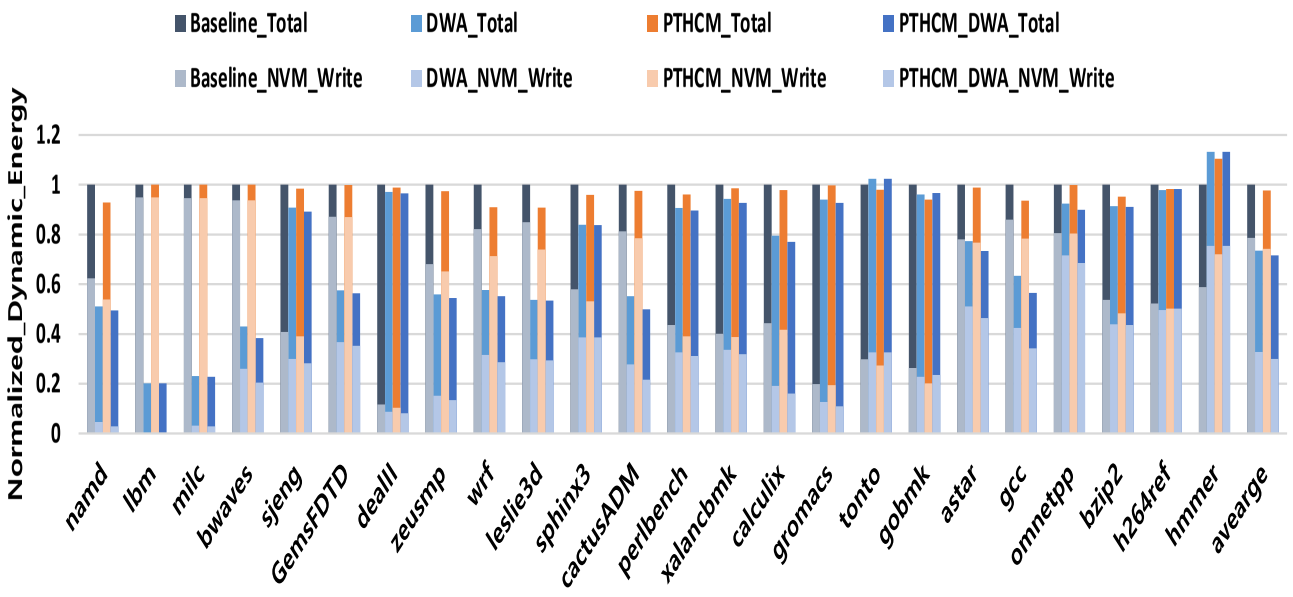


Figure 26: Normalized dynamic energy consumption of DWA with STT-RAM.

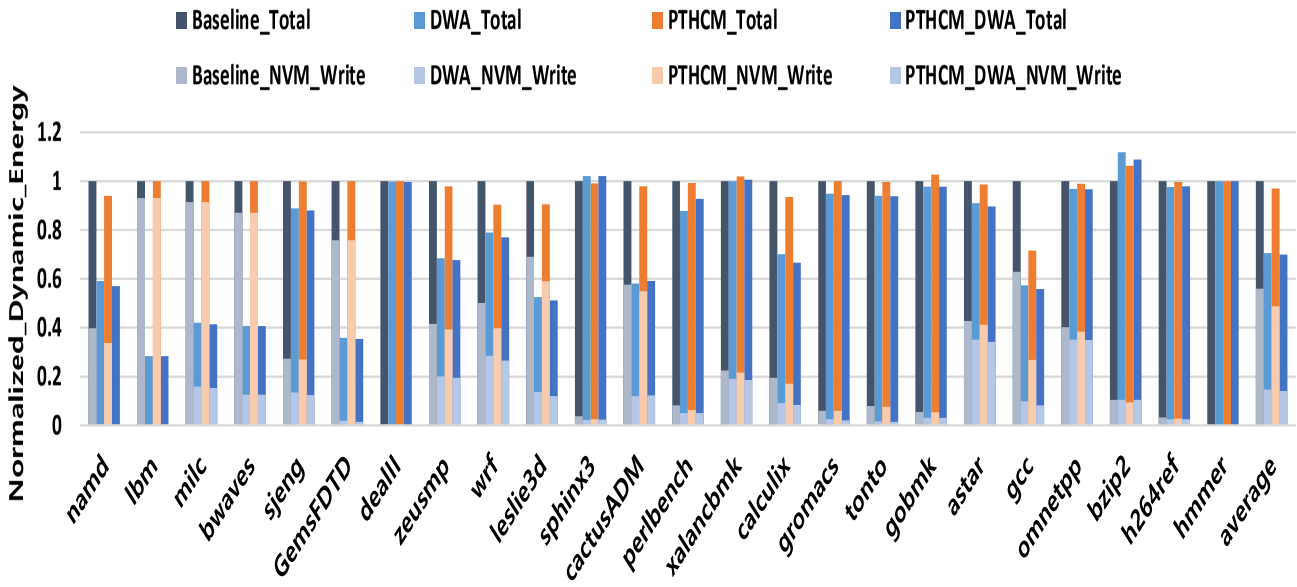


Figure 27: Normalized dynamic energy consumption of DWA with PCM.

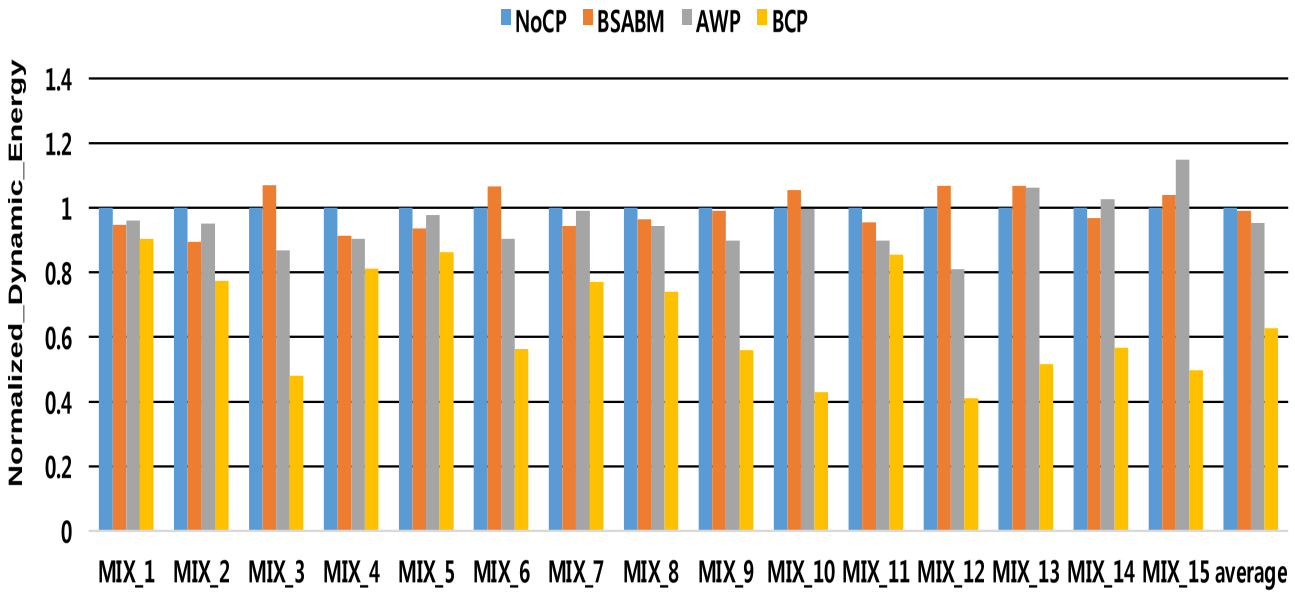


Figure 28: Normalized dynamic energy consumption compared to NoCP.

The normalized dynamic energy consumption of four schemes are presented in Figure 28. LCP saved 37.2%, 36.6%, and 34.1% of dynamic energy consumption over NoCP, BSABM, and AWCP, respectively. The trends of the dynamic energy reduction are similar to those of the normalized NVM write counts, while the variation is small. For MIX_12, the dynamic energy consumption is reduced by nearly 60% compared to AWCP at maximum, while the difference between AWCP and LCP is less than 1% for MIX_1. The reason for this similarity is that the NVM write counts is a main contributor to the total energy consumption; thus, reducing the number of NVM write accesses to the LLC highly influenced the total dynamic energy consumption.

5.4 Lifetime

We estimated the normalized lifetime as shown in Figure 29 and Figure 30. There is a general consensus among researchers that PCM has a limited lifetime. However, opinions are different about the write endurance of STT-RAM. Many studies assume that its write endurance is high enough, and thus they set aside the lifetime problem. On the other hand, another group argues that the assumption is unrealistic [19, 56]. Since determining the correctness of their claims is not the focus in the thesis, the results of both types of NVM are presented.

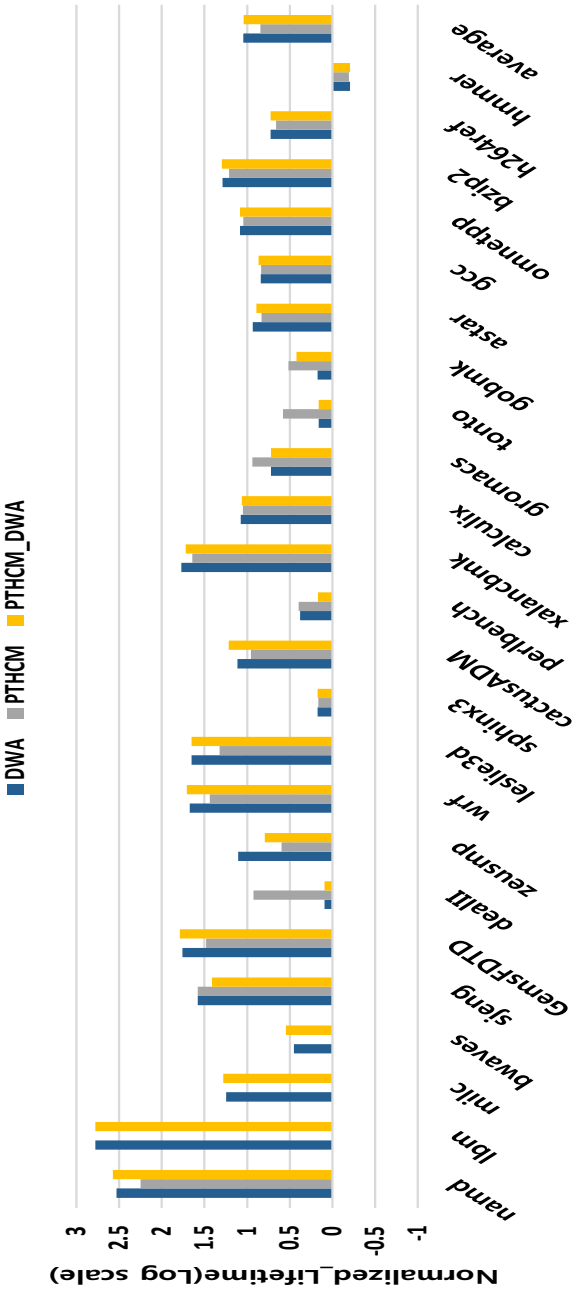


Figure 29: Normalized lifetime of DWA with STT-RAM.

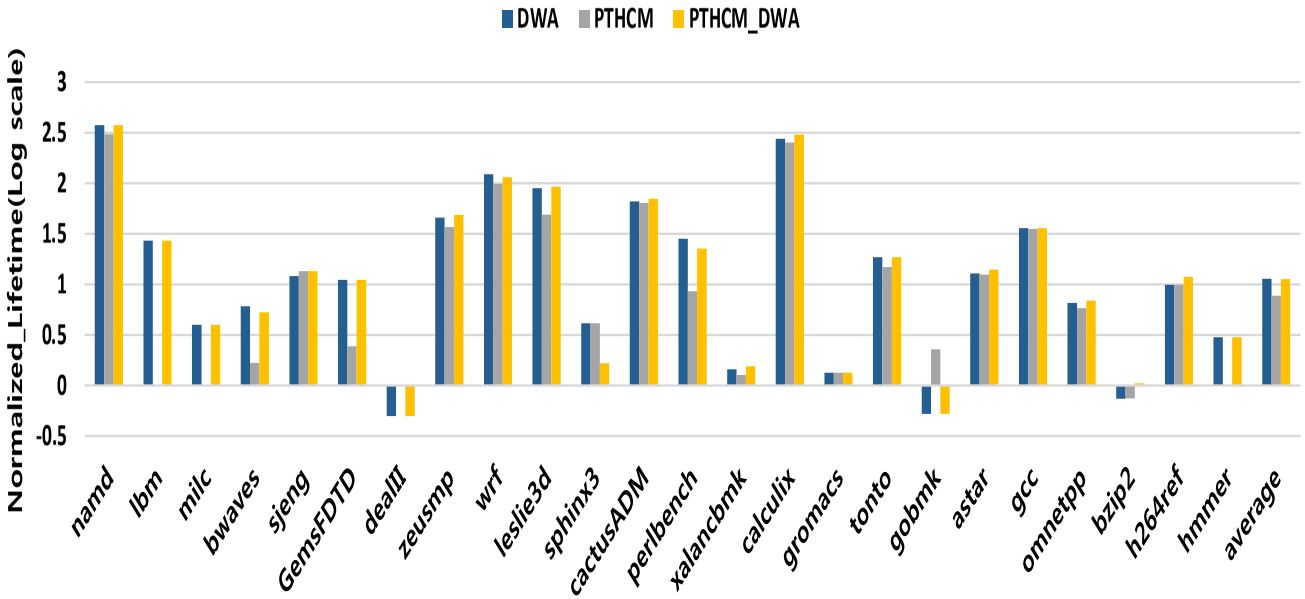


Figure 30: Normalized lifetime of DWA with PCM.

Notice that the results of two figures are presented in log scale because the lifetime of some applications were extended significantly. Especially, the write endurance of *namd* and *lbm* was increased by more than 300 times. For these applications, the number of replaceable ways was almost always less than the number of SRAM ways. Since NVM ways were rarely used in the DWA, the lifetime soared up. The PTHCM_DWA extended the lifetime by 10.9 times and 11.3 times for HCAs with STT-RAM and PCM, respectively.

To confirm that our proposal does not increase the miss rate significantly, we present the miss rates of each HCA configuration compared to the baseline hybrid cache in Figure 31 and Figure 32. The miss rate of the DWA was increased only by 1.8% and 1.9% for HCAs with STT-RAM and PCM, respectively, while the PTHCM decreased the miss rate by 1%. Since the PTHCM did not improve the miss rate meaningfully, the miss rate of the PTHCM_DWA followed the miss rate of the DWA. Therefore, the miss rates of the DWA and the PTHCM_DWA are very similar and the PTHCM_DWA increased the miss rate by 1.9% and 1.9% on average which are the same values of the DWA. As expected, this result confirms that our proposed algorithm does not significantly increase the miss rate.

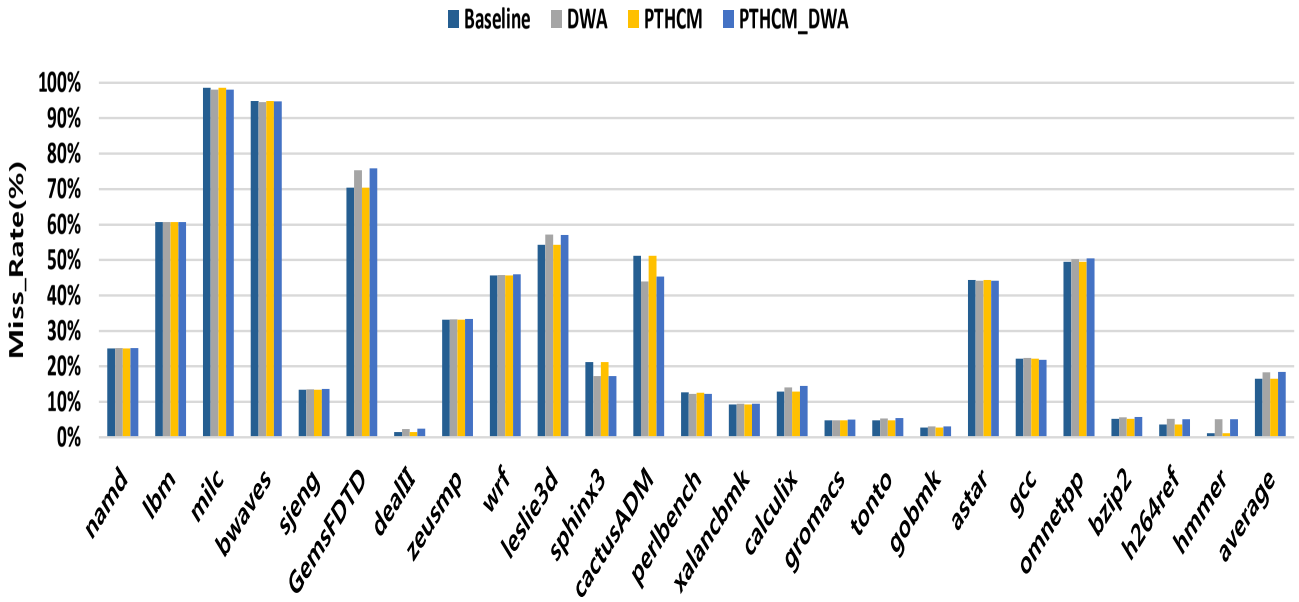


Figure 31: Miss rates with various HCA confiruations with STT-RAM.

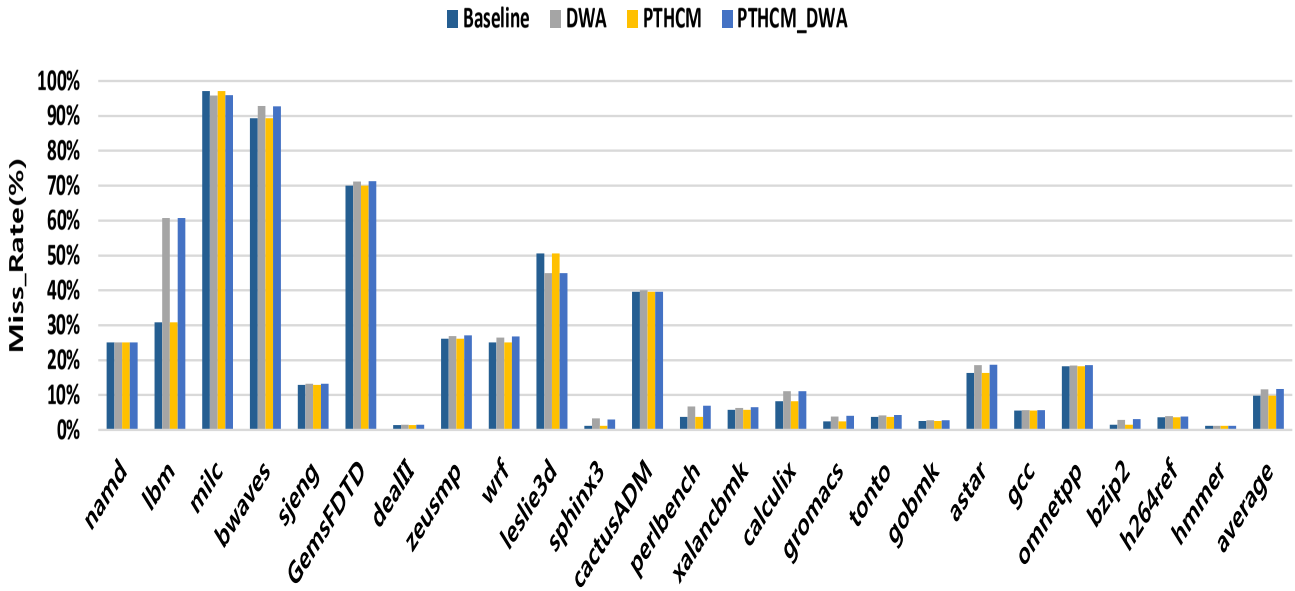


Figure 32: Miss rates with various DWA configurations with PCM.

5.5 Multi-core environment

We investigated several metrics for multi-core environments as shown in Figure 33 and Figure 34. For multi core system simulation, we generated ten workloads by mixing six applications as listed in Table 12. The two benchmarks for low sensitivity are *bwaves* and *calculix*, while *hmmer* and *h264ref* represent high sensitivity. Other two benchmarks such as *wrf* and *gromacs* are selected as the middle range of sensitive programs.

First of all, a significant reduction in the write accesses was achieved in both HCA configurations. The DWA removed 80.7% of write accesses on average, while the average write reduction ratio of six benchmarks is 61.3% for HCA with STT-RAM in single-core environments. This result means that our proposal has the extendibility for the multi-core system. In case of HCA with PCM, the average reduction ratio of multi-core results is 59.4%, while each application removed 76.3% of write accesses on average. Even though the results of HCA with PCM are less impressive compared to HCA with STT-RAM, our proposal still removed a great deal of unnecessary NVM write operations. The results of dynamic energy consumption are consistent with the trend of the write accesses to NVM. For HCAs with STT-RAM and PCM, 55.5% and 33.7% of dynamic energy consumption were saved, respectively. The lifetime was prolonged by 1.76 times and 1.35 times on average.

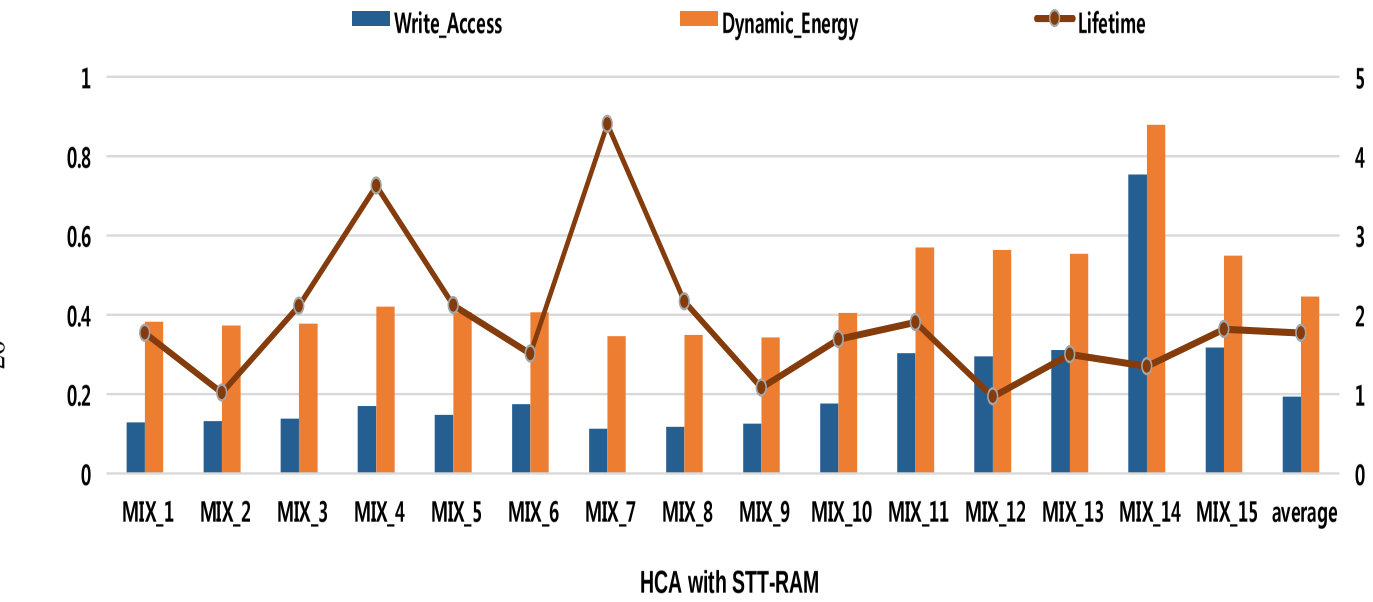


Figure 33: Normalized write access, dynamic energy consumption, and lifetime of HCA with STT-RAM with the multi-core workloads.

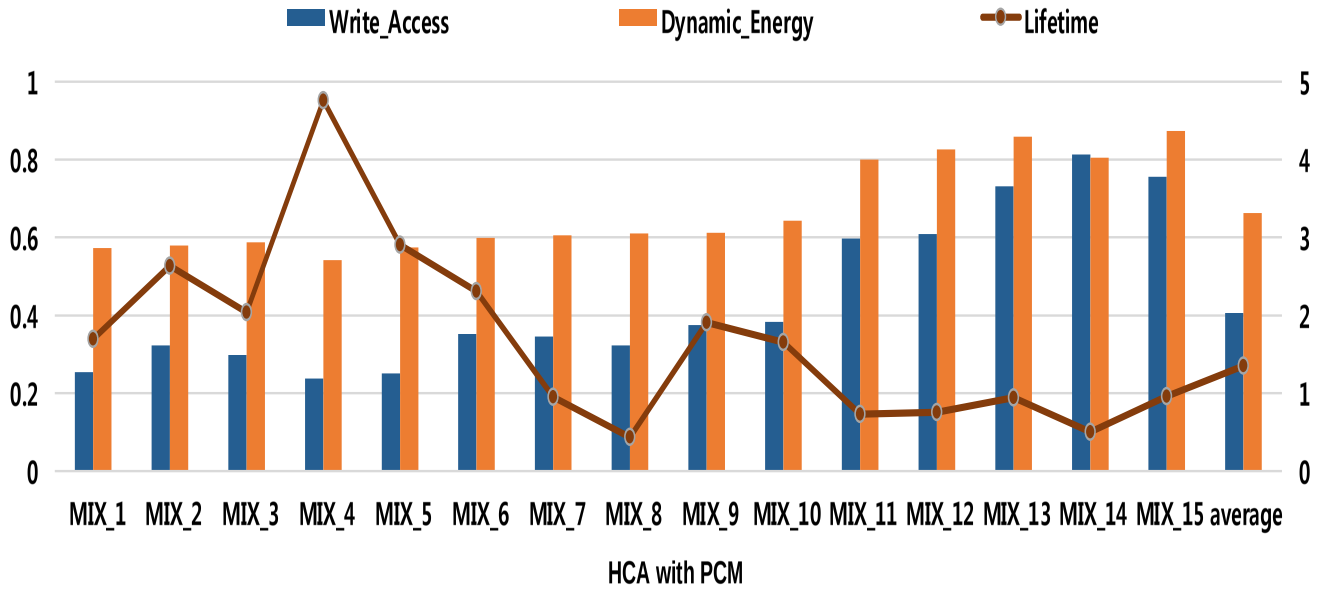


Figure 34: Normalized write access, dynamic energy consumption, and lifetime of HCA with PCM with the multi-core workloads.

To represent the performance improvement in a multi-core environment, three metrics usually are presented – nstruction per cycle (IPC) throughput, weighted speedup, and fairness – which have their own purposes [57]. They usually are defined as follows:

$$IPC\ throughput = \sum_{i=1}^{n} IPC_i \tag{5.1}$$

$$Weighted\ Speedup = \sum_{i=1}^{n} \frac{IPC_i^{MP}}{IPC_i^{SP}} \tag{5.2}$$

$$Fairness = \frac{n}{\sum_{i=1}^{n} \frac{IPC_i^{SP}}{IPC_i^{MP}}} \tag{5.3}$$

where IPC_i^{SP} is the IPC of ith program under single program mode (SP) and IPC_i^{MP} is the IPC under multi-program mode (MP). IPC throughput is simply and intuitively defined as the sum of the IPCs of the all applications. The weighted speedup is proposed to equalize the contribution of programs using normalized IPCs [58]. Luo et al. argued that harmonic mean is more suitable to represent the fairness than weighted speedup [59].

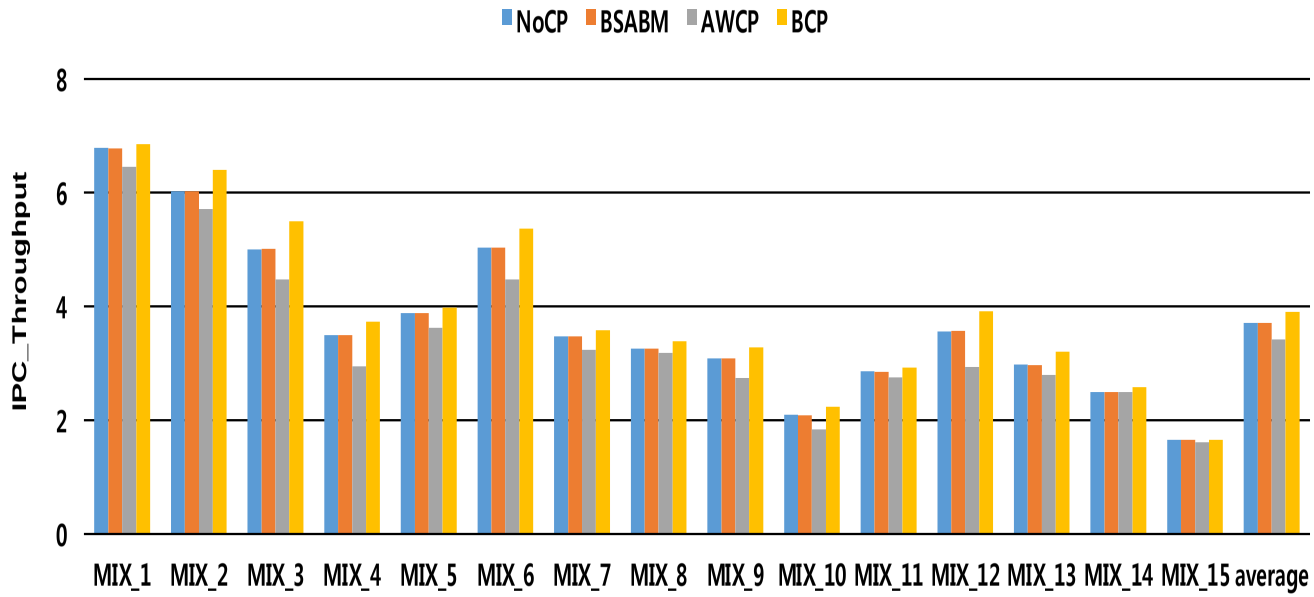


Figure 35: IPC throughput with four schemes.

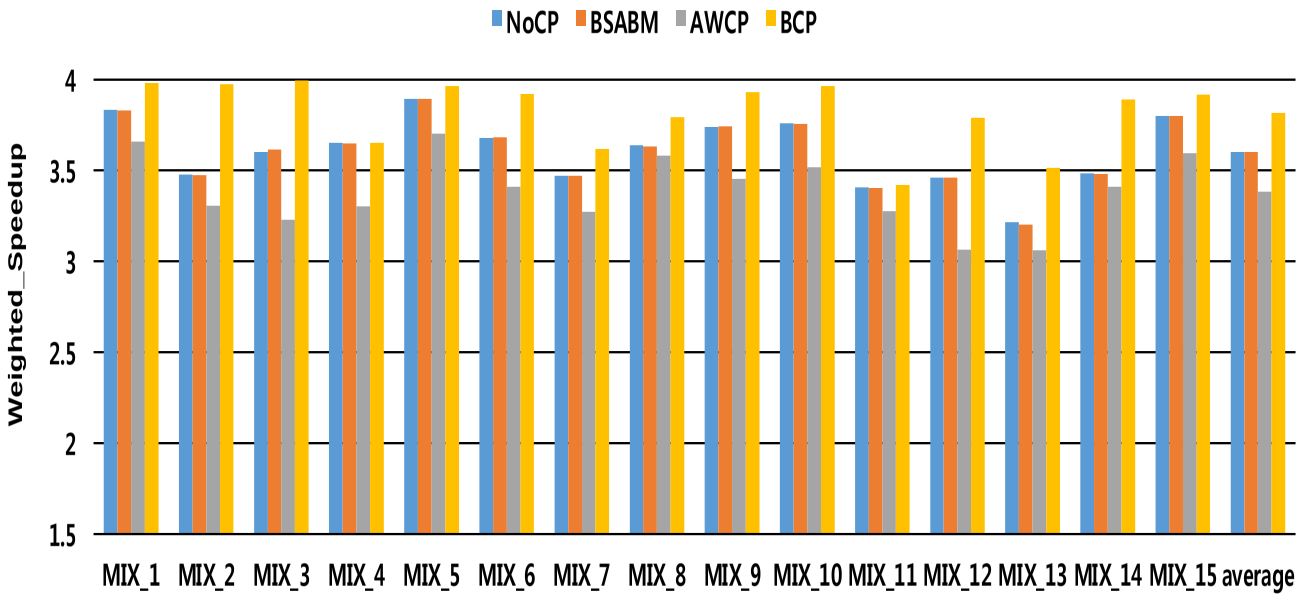


Figure 36: Weighted speedup with four schemes.

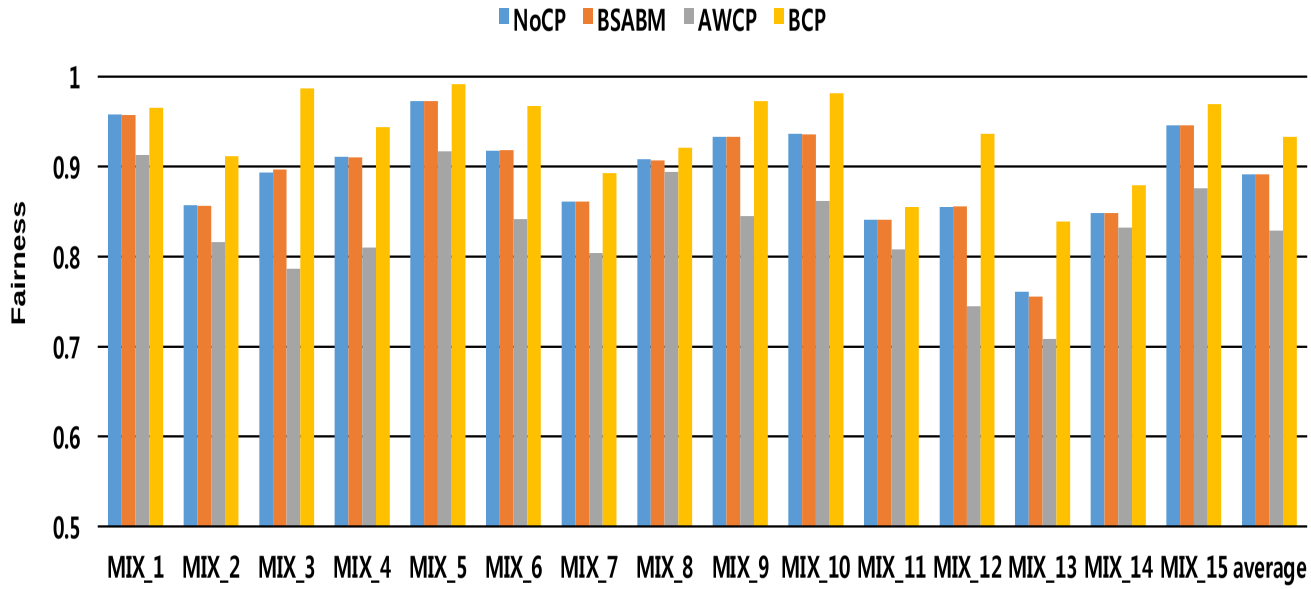


Figure 37: Fairness with four schemes.

Therefore, we plot three metrics in Figure 35, Figure 36, and Figure 37 for different schemes. LCP outperforms NoCP and AWCP by 5.0% and 14.3% in terms of IPC throughput as depicted in Figure 35. In addition, our scheme improved the weighted speedup by 5.6% and 11.4% for NoCP and AWCP as shown in Figure 36. Finally, Figure 37 compares the fairness improvement for four schemes; the fairness of LCP is improved to 0.93, while NoCP and AWCP have 0.89 and 0.83, respectively. The IPC throughput improvement is maximized for MIX_3, whereas MIX_2 shows the best weighted speedup improvement compared to AWCP. The fairness of the applications of MIX_12 is most increased.

Chapter 6

Conclusion

6.1 Conclusion

In the thesis, three proposals have been provided to compensate for identified weaknesses of NVM: write avoidance cache coherence protocol (WACC), dynamic way adjusting scheme (DWA), and linefill-aware cache partitioning (LCP).

We proposed a novel cache coherence protocol to eliminate useless write operations of LLC for a multi-core system. Based on the analysis of the existing protocols, it was found that they generated useless write accesses to the LLC during the linefill operation. Thus, our protocol,which is called WACC, modifies the cache states without storing the block data during linefill. This write policy reduced the number of write access attempts to the LLC, which led to improvements in the energy consumption and lifetime. The simulation result showed that the reduction of maximum energy consumption in WACC protocol is 27.1% and the lifetime extension is 26.3% at maximum in STT-RAM based LLC.

The thesis introduced the concept of an NVM capacity management policy for reducing the number of write accesses to NVM. This policy is implemented by two methods called dynamic way adjusting scheme (DWA) and linefill-aware cache partitioning (LCP). DWA dynamically resized the number of active NVM ways to improve the dynamic energy consumption and the lifetime of the components. To adjust the number of NVM ways, the maximum stack distance is dynamically monitored and rearranging of the replaceable NVM ways is regularly performed. The proposed policy reduced the number of write accesses to STT-RAM by about 77.6% and PCM by 79.6%. The results also showed that HCAs with STT-RAM and PCM achieves 30.0% reduction and 28.4% in dynamic energy consumption. The lifetime of the two HCAs was prolonged by 10.9 times and 11.3 times over a conventional hybrid cache system. Both HCAs can achieve these improvements without any meaningful miss rate increment. While the portion of the NVM linefill operations, over the write counts, is about 83.5% in our experimental results, previous studies have not considered the linefill operations to NVM in CMP environments during partitioning.

We also proposed LCP, to minimize the NVM write counts, in consideration of the NVM linefill counts, as well as the NVM write hit counts. In the thesis, three kinds of metrics were introduced to analyze the efficiency of adjusting the cache partitioning; if a core gets or loses ways, how many the miss counts, write counts, and NVM write counts are changed. A cache partitioning algorithm for LCP is proposed to provide the best partitioning through a two-step approach based on these metrics. We have shown that

the proposed LCP predicts the NVM write counts with less than a 5% error rate and reduces the dynamic energy consumption by 34.1% on average with improved performance.

6.2 Future work

We will extend the findings of thesis in two ways. First, we plan to combine our proposal with schemes for non-uniformity of write operations among sets which are inspired that the write varies across different cache sets. They separated the physical mapping and logical mapping of cache sets and stored data between sets. The key idea is decent, but there is a pitfall to simply merge LCP with the inter-set variation wear leveling scheme (ISWLs). Since the data is possible to be placed in a different set, they violate the stack property which our scheme is based on. Keeping track of all recency position of remapped blocks would not be a feasible method because it needs a significant area overhead and consumes a lot of dynamic energy. Hence, we are developing a new method to efficiently bond LCP and ISWLs.

In addition, we will consider combining data bypassing techniques to the proposed scheme. Even though cache bypassing techniques are apparently promising schemes for NVM, they cannot be directly applied to our mechanism because the inclusion property is not maintained in most of their schemes. We will investigate a new scheme that both keeps inclusion property and utilizes the bypass schemes.

Bibliography

[1] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, *et al.*, "A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram," in *Proceedings of IEEE International Electron Devices Meeting*, pp. 459–462, IEEE, 2005.

[2] H. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[3] N. Yamada, E. Ohno, K. Nishiuchi, N. Akahira, and M. Takao, "Rapid-phase transitions of gete-sb2te3 pseudobinary amorphous thin films for an optical disk memory," *Journal of Applied Physics*, vol. 69, no. 5, pp. 2849–2856, 1991.

[4] A. Driskill-Smith, S. Watts, D. Apalkov, D. Druist, X. Tang, Z. Diao, X. Luo, A. Ong, V. Nikitin, and E. Chen, "Non-volatile spin-transfer torque ram (stt-ram): An analysis of chip data, thermal stability and scalability," in *Proceedings of IEEE International Memory Workshop*, pp. 1–3, IEEE, 2010.

[5] T. Sumi, Y. Judai, K. Hirano, T. Ito, T. Mikawa, M. Takeo, M. Azuma, S.-i. Hayashi, Y. Uemoto, K. Arita, *et al.*, "Ferroelectric nonvolatile memory technology and its applications," *Japanese Journal of Applied Physics*, vol. 35, no. 2S, p. 1516, 1996.

[6] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[7] J. H. Choi, J. W. Kwak, and C. S. Jhon, "Write avoidance cache coherence protocol for non-volatile memory as last-level cache in

chip-multiprocessor," *IEICE Transactions on Information and Systems*, vol. 97, no. 8, pp. 2166–2169, 2014.

[8] J. H. Choi and G. H. Park, "Demand-aware nvm capacity management policy for hybrid cache architecture," *Computer Journal*, advance online publication, 2015, doi:10.1093/comjnl/bxv103.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[10] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pp. 72–81, ACM, 2008.

[12] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[13] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[14] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[15] S. Lee, K. Kang, and C.-M. Kyung, "Runtime thermal management for 3-d chip-multiprocessors with hybrid sram/mram l2 cache," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 23, no. 3, pp. 520–533, 2014.

[16] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie, "Power and performance of read-write aware hybrid caches with non-volatile memories," in *Proceedings of International Conference on Design, Automation and Test in Europe*, pp. 737–742, IEEE, 2009.

[17] J. Li, L. Shi, C. J. Xue, C. Yang, and Y. Xu, "Exploiting set-level write non-uniformity for energy-efficient nvm-based hybrid cache," in *Proceedings of International Symposium on Embedded Systems for Real-Time Multimedia*, pp. 19–28, IEEE, 2011.

[18] B. Quan, T. Zhang, T. Chen, and J. Wu, "Prediction table based management policy for stt-ram and sram hybrid cache," in *Proceedings of International Conference on Computing and Convergence Technology*, pp. 1092–1097, IEEE, 2012.

[19] J. Ahn, S. Yoo, and K. Choi, "Write intensity prediction for energy-efficient non-volatile caches," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 223–228, IEEE, 2013.

[20] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 34–45, ACM, 2009.

[21] J. H. Choi, J. W. Kwak, S. T. Jhang, and C. S. Jhon, "Adaptive cache compression for non-volatile memories in embedded system," in *Proceedings of International Conference on Research in Adaptive and Convergent Systems*, pp. 52–57, ACM, 2014.

[22] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad, "High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 79–84, IEEE, 2011.

[23] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi, "i 2 wap: Improving non-volatile cache lifetime by reducing inter-and intra-set write vari-

ations," in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 234–245, IEEE, 2013.

[24] Y.-T. Chen, J. Cong, H. Huang, C. Liu, R. Prabhakar, and G. Reinman, "Static and dynamic co-optimizations for blocks mapping in hybrid caches," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 237–242, ACM, 2012.

[25] Y. Li, Y. Chen, and A. K. Jones, "A software approach for combating asymmetries of non-volatile memories," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 191–196, ACM, 2012.

[26] Q. Li, M. Zhao, C. J. Xue, and Y. He, "Compiler-assisted preferred caching for embedded systems with stt-ram based hybrid cache," *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 109–118, 2012.

[27] K. Qiu, M. Zhao, C. Fu, L. Shi, and C. J. Xue, "Migration-aware loop retiming for stt-ram based hybrid cache for embedded systems," in *Proceedings of International Conference on Application-Specific Systems, Architectures and Processors*, pp. 83–86, IEEE, 2013.

[28] Y. Li, Y. Zhang, H. Li, Y. Chen, and A. K. Jones, "C1c: A configurable, compiler-guided stt-ram l1 cache," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 4, p. 52, 2013.

[29] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 24–33, 2009.

[30] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montaño, "Improving read performance of phase change memories via write cancellation and write pausing," in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 1–11, IEEE, 2010.

[31] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, 2012.

[32] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *Proceedings of International Conference on Computer Design*, pp. 337–344, IEEE, 2012.

[33] M. Zhou, Y. Du, B. Childers, R. Melhem, and D. Mossé, "Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, p. 53, 2012.

[34] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: a hybrid pram and dram main memory system," in *Proceedings of Internaional Conference on Design Automation Conference*, pp. 664–669, IEEE, 2009.

[35] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing pcm main memory lifetime," in *Proceedings of Internaional Conference on Design, Automation and Test in Europe*, pp. 914–919, IEEE, 2010.

[36] W. Zhang and T. Li, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pp. 101–112, IEEE, 2009.

[37] H. Seok, Y. Park, and K. H. Park, "Migration based page caching algorithm for a hybrid main memory of dram and pram," in *Applied Computing, International Symposium on*, pp. 595–599, ACM, 2011.

[38] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[39] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Microarchitecture, IEEE/ACM International Symposium on*, pp. 423–432, IEEE Computer Society, 2006.

[40] A. Samih, Y. Solihin, and A. Krishna, "Evaluating placement policies for managing capacity sharing in cmp architectures with private caches," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 3, p. 15, 2011.

[41] C. CaBcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of International Conference on Supercomputing*, pp. 150–159, ACM, 2003.

[42] Y. Liu and W. Zhang, "Exploiting stack distance to estimate worst-case data cache performance," in *Proceedings of International Symposium on Applied Computing*, pp. 1979–1983, ACM, 2009.

[43] "The intel 64 and ia-32 architectures software developer's manual." http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf. accessed 3-Mar-2014.

[44] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[45] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.

[46] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 167–178, 2006.

[47] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie, "Adaptive placement and migration policy for an stt-ram-based hybrid cache,"

[48] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2011.

[49] J. Li, C. J. Xue, and Y. Xu, "Stt-ram based energy-efficiency hybrid cache for cmps," in *Proceedings of International Conference on VLSI and System-on-Chip*, pp. 31–36, IEEE, 2011.

[50] S.-M. Syu, Y.-H. Shao, and I.-C. Lin, "High-endurance hybrid cache design in cmp architecture with cache partitioning and access-aware policy," in *Proceedings of International Conference on Great Lakes Symposium on VLSI*, pp. 19–24, ACM, 2013.

[51] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for stt-ram using early write termination," in *Proceedings of International Conference on Computer-Aided Design-Digest of Technical Papers*, pp. 264–268, IEEE, 2009.

[52] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3d stacked mram l2 cache for cmps," in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 239–249, IEEE, 2009.

[53] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation," in *Proceedings of International Symposium on Microarchitecture*, pp. 81–92, IEEE Computer Society, 2004.

[54] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, pp. 190–200, ACM, 2005.

[55] "Arm cortex-a57 processor." "`http://www.arm.com/products/processors/cortex-a/cortex-a57-processor.php`" (accessed 1-Sep-2015).

[56] J. Wang, Y. Tim, W.-F. Wong, Z.-L. Ong, Z. Sun, and H. H. Li, "A coherent hybrid sram and stt-ram l1 cache architecture for shared memory multicores.," in *Proceeding of Asia and South Pacific Design Automation Conference*, pp. 610–615, IEEE, 2014.

[57] L. Eeckhout, "Computer architecture performance evaluation methods," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–145, 2010.

[58] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous mutlithreading processor," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 234–244, 2000.

[59] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors.," in *Performance Analysis of Systems and Software, International Symposium on*, pp. 164–171, IEEE, 2001.

초록

비휘발성 메모리 기반의 최종 레벨 캐시를 위한 쓰기 회피 기법

비휘발성 메모리는 높은 집적성과 낮은 정적 전력 소모량이라는 특성으로 인해 최종 레벨 캐시로 사용되기에 유력한 기술로 떠오르고 있다. 그러나 비휘발성 메모리는 쓰기 작업을 위해 많은 전력과 시간을 소모하고, 제한된 수명을 가진다는 단점이 있기 때문에 이를 보완하기 위한 방법이 없다면 최종 레벨 캐시로 사용되기 어렵다. 본 논문에서 비휘발성 메모리의 단점을 보완하기 위해 쓰기 회피 기법들을 제시하였다. 먼저, 멀티 코어 환경에서 쓰기 횟수를 줄이기 위한 캐시 일관성 정책(Write avoidance cache coherence protocol)을 제시하였고, 이종 캐시 구조(Hybrid cache architecture) 에서 쓰기 회수를 최소화하기 위한 2가지 기법을 제안하였다. 첫번째 기법은 NVM way을 동적으로 조정하는 방식이며(Dynamic way adjusting), 다른 기법은 linefill을 고려한 캐시 분할 기법(Linefill-aware cache partitioning)이다.

우선 본 논문에서는 쓰기 횟수를 줄이기 위한 새로운 캐시 일관성 정책을 제안한다. 새로운 정책을 사용하는 시스템에서는 상위 레벨 캐시에 동일한 데이터가 있는 경우, 최종 레벨 캐시에서는 태그 정보만 저장하고 데이터 정보는 기록하지 않는다. 따라서 상위 레벨 캐시에서 쓰기 수정이 일어났을 때, 불필요한 쓰기를 줄일 수 있게 된다.

다음으로 이종 캐시 구조 환경하에서 비휘발성 메모리의 크기를 제한하여 쓰기 횟수를 줄이는 기법을 제안한다. 이종 캐시 구조는 비휘발성 메모리의 일부를 휘발성 메모리인 SRAM로 교체하여 두 가지 종류의 메모리가 하나의 캐시에 존재하는 구조이다. 통계적으로 비휘발성 메모리의 way의 비율이 많아질수록 전체 쓰기 작업에서 비휘발성 메모리의 쓰기 작업의 비율 또한 커지게 된다. 그런데 모든 프로그램이 항상 전체 메모리를 요구하는 것은 아니다. 프로그램에 따라서 또는 실행 시간에 따라서 메모리의 일부만을 요구할 때도 있다. 그러한 경우에는 필요한 만큼만 비휘발성 메모리를 사용하도록 메모리의 크기를 제한한다면 성능의 저하 없이 비휘발성 메모리의 쓰기 횟수를 줄일 수 있다.

또한, 본 논문에서는 이종 캐시 구조를 사용하는 멀티 코어 시스템에서 비휘발성 메모리의 쓰기 회수를 최소화하는 캐시 분할(Cache partitioning)을 제안한다. 기존의 캐시 분할 방식들은 휘발성 메모리를 사용한 동종 캐시 구조를 사용하기 때문에, 각 코어에 할당할 way의 수만 계산하였다. 그러나 이종 캐시 구조에서는 각 코어가 사용할 전체 way의 수뿐만 아니라 비휘발성 메모리 way의 수와 휘발성 메모리 way의 수를 따로 구해야 한다. 그렇지 않으면 휘발성 메모리 way가 비효율적으로 코어에 분배되어, 전체적인 비휘발성 메모리의 쓰기 회수가 최적화되지 않는다. 따라서, 본 논문에서는 일정한 주기마다 캐시 분할 방식을 바꾸어 가면서 비휘발성 메모리의 쓰기 회수를 최소화하는 캐시 분할 구성을 찾아낸다.

실험을 수행한 결과, Write avoidance cache coherence protocol을 적용하게 되면 전력 소모량은 13.2%가 감소하며, Dynamic way adjusting와 Linefill-aware cache partitioning을 적용하는 경우 각각 전력 소모량이 26.9%와 37.2% 감소하였다.