UNIVERSITY OF CALIFORNIA
SANTA CRUZ

TOWARDS SUSTAINABLE NON-VOLATILE MEMORY: MACHINE LEARNING AND MEMORY-AWARE DATA STRUCTURES FOR ENERGY EFFICIENCY AND LONGEVITY

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE AND ENGINEERING

by

Saeed Kargar

September 2024

The Dissertation of Saeed Kargar
is approved:

Table of Contents

List of Figures

List of Tables

Abstract

Towards Sustainable Non-Volatile Memory: Machine Learning and Memory-Aware Data Structures for Energy Efficiency and Longevity

by

Saeed Kargar

Non-volatile memory (NVM) technology has revolutionized memory systems with its non-volatility and near-zero standby power consumption, making it a promising alternative to traditional DRAMs. However, NVMs also face significant challenges, particularly limited write endurance and high energy consumption, which impede their widespread adoption. This thesis contributes to the integration of NVM technologies into the memory hierarchy, addressing these challenges with software-level solutions, including AI-based techniques and data structure-based methods. These approaches enhance energy efficiency and write endurance, improving the practicality and longevity of NVM devices. This thesis is organized into four main parts:

In the first part, we conduct a comprehensive evaluation of real-world NVM devices, such as Optane memory, to explore the effects of memory awareness on performance, energy consumption, and lifetime. Our experiments reveal that memory-aware strategies significantly increase device lifetime, decrease power consumption, and improve system latency. This chapter underscores the importance of integrating recent advances from the NVM storage community into existing and future data management systems.

In the second part, we propose Predict and Write (PNW), a K/V-store that uses a clustering-based machine learning approach to extend the lifetime of NVMs. PNW reduces the number of bit flips for PUT/UPDATE operations by determining the optimal memory location for updated values. By leveraging the indirection level of K/V-stores, PNW organizes NVM addresses in a dynamic address pool clustered by data value similarity. Our results demonstrate that PNW can reduce total bit flips by up to 85% and cache lines by 56% compared to the state of the art.

In the third part, we introduce E2-NVM, a software-level memory-aware storage layer designed to improve the Energy efficiency and write Endurance (E2) of NVMs. E2-NVM employs a Variational Autoencoder (VAE) based design to judiciously direct write operations to memory segments that minimize bit flips. This solution, which can be combined with existing indexing and hardware-based methods, demonstrates a reduction in energy consumption by up to 56% in real-world evaluations on an Optane memory device.

The final part of this thesis presents a software-level data storage layer solution that uses a indexing data structure to improve the energy consumption and write endurance of NVMs. In this work, we presented the case for memory-awareness and showed that by judiciously selecting memory locations for new writes and updates we can reduce bit flipping and consequently improve the energy efficiency and write endurance of NVM devices. We took this concept and built Hamming Tree, with which existing data stores can be augmented, to make them memory-aware. Hamming Tree tackles the challenges associated with mapping free memory locations based on the hamming distance of their content. Our evaluations on an Intel Optane memory device show that Hamming Tree can achieve up to a 67.8% improvement in energy efficiency.

Overall, the methods proposed in this thesis are software-level solutions aimed at improving the performance of NVMs. PNW is the simplest, suitable for systems with fixed memory segment sizes and basic hardware resources, though advanced components can enhance its performance. E2-NVM is more advanced, handling large, variable-sized memory segments and requiring GPUs for maximum efficiency. Hamming Tree, implemented as an indexing data structure, is fast but sensitive to the number of indexed items. This thesis presents a comprehensive framework for enhancing NVM efficiency and longevity, addressing critical challenges like energy consumption and write endurance.

Dedicated

to

my father, whose wisdom, unwavering support, and boundless belief in me have always guided my path;

my mother, for her endless love and her heartfelt pride in my progress;

my sister, for standing by my side and helping me through tough times; and

my brother, for sharing in the joy of my successes.

Acknowledgments

I extend my sincere gratitude to my advisor, Professor Faisal Nawab, for his invaluable support and mentorship. I am also deeply thankful to Professor Chen Qian for graciously agreeing to serve as my co-advisor, and to Professor Katia Obraczka for her invaluable contributions as a committee member.

Chapter 1

Understanding Non-Volatile Memories: Challenges and Opportunities for Enhanced Energy Efficiency and Endurance

1.1 Introduction

Nowadays, we are witnessing the emergence of new non-volatile memory (NVM) technologies that are remarkably changing the landscape of memory systems. They are persistent, have high density, byte addressable, and require near-zero standby power [62], which makes them of great interest in the database and storage systems communities. Furthermore, NVMs provide up to 10x higher bandwidth and 100x lower access latency compared to SSDs [23, 48]. However, because they also present some limitations, such

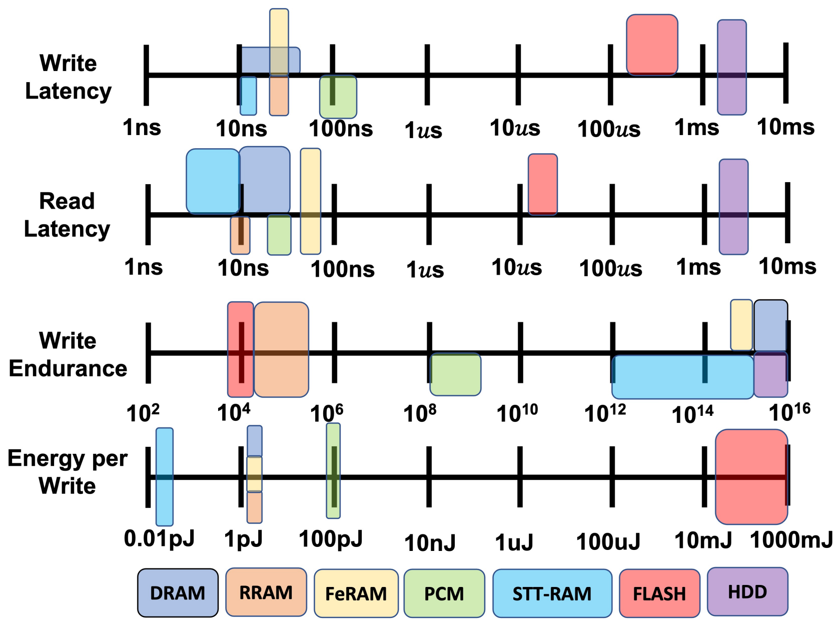


Figure 1.1: Comparison of device properties of memory and storage technologies [82]

as limited write endurance, which is significantly lower than DRAM write endurance, and high write energy consumption (Figure 1.1), adopting the current systems to use NVM while maximizing their potential is proving to be challenging. Additionally, unlike flash storage, cells are written individually in many NVM technologies such as Phase Change Memory (PCM). This means that some cells may receive a much higher number of writes than others during a given period, and as a result wear out sooner. So, any system design needs to take these limitations into consideration before deciding to utilize NVMs.

The scientific community has conducted an extensive amount of research on integrating these new technologies in current systems. Furthermore, these emerging technologies are carving out their own place in the memory hierarchy. As persistent memories are usually larger than DRAM in capacity, researchers in the database community usually take advantage of its persistence to boost performance of the system by making the in-memory database persistent, which results in capacity expansion and recovery cost reduction [37,79,101,152,153]. However, when designing a database management system, it is critical to fully consider the characteristics of NVMs, including their limitations, to take advantage of their hardware potentials.

In this chapter, we survey recent work in both storage and database communities, where a substantial amount of work has been done in the adoption of NVMs. In particular, we focus on the common limitations that are shared among most NVMs—i.e. low write endurance, high energy consumption, and intrinsic asymmetric properties of reads and writes. Table 1.1 compares the main characteristics of these memory technologies. This chapter provides information on how to integrate recent advances from the NVM storage community into existing and future data management systems.

1.2 Background

We now introduce some concepts which will be useful throughout this thesis. We refer the reader to previous works for a detailed background on NVM architecture [17, 112, 151].

1.2.1 NVM Properties

Non-Volatile Memory (NVM) has the potential of transforming the memory architecture in data management systems due to their characteristics such as persistency,

	Storage Class Memory device property							
	Cost/bit	Reliability	Flexibility	Write Endurance	Write Energy	Scalability	Write latency	Density
PCM	✓	✓	✓	✗	✗	✓	✗	✓
FeRAM	✗	✗	✓	✓	✓	✗	✓	✗
STT-RAM	✗	✗	✗	✓	✗	✗	✓	✗
ReRAM	✓	✗	✓	✗	✗	✓	✗	✓
SRAM	✗	✗	✗	✓	✗	✗	✓	✗
NAND Flash	✓	✗	✓	✗	✗	✗	✓	✓
NOR Flash	✗	✓	✓	✗	✗	✗	✗	✗
3D XPoint	✓	✓	✓	✗	✗	✓	✗	✓

Table 1.1: Device properties of Emerging Nonvolatile Memories.

high density, byte addressability, and requiring near-zero standby power [62]. However, they also have their own drawbacks that may vary from technology to technology: some have problems with write endurance, some have higher write latency than the read latency, some have low density and do not scale well, and so on. So, researchers have been trying to overcome these limitations to efficiently utilize them.

As far as write endurance is concerned, most NVM technologies, such as PCM, ReRAM, 3D XPoint and Flash memory, have lower write endurance (usually between 10^5 to 10^9) compared to the traditional memory technologies, such as DRAM, which have virtually unlimited lifetime (10^{16}). Even in some NVM technologies such as STT-RAM with higher lifetime (at least 10^{12}), certain usage conditions, such as frequent accesses with a high applied bias voltage or a long pulse, may aggravate their lifetime

Contribution	Selected Studies
Lifetime Improvement	[82], [2], [119], [52], [59], [1], [129], [130], [40], [133], [155] [68], [120], [5], [149], [137], [16], [166], [81], [83], [69] [100], [19], [163], [22], [126], [3], [61]
Performance	[82], [40], [133], [155], [100], [83], [60], [32], [105], [37] [98], [79], [18], [121], [99], [101], [12] [49], [65], [147], [25], [161], [27], [150], [160], [88], [97] [45], [33], [36], [141], [92], [96], [50], [134], [13]
Energy Efficiency	[82], [119], [129], [16], [166], [81], [83], [68], [69], [60] [18], [128], [138], [139], [77], [28], [95], [102] [70], [162], [140], [136], [90], [135], [63]

Table 1.2: Selected studies categorized by their contribution.

and reliability [78, 127]. Furthermore, in many use cases, some of NVM technologies, such as PCM, STT-RAM and ReRAM, are used in the lower-level cache hierarchies instead of memory level, which expose them to aggressively high number of writes. So, in these cases, write endurance can prevent the adoption of NVM in lower-level caches.

Improving the lifetime of NVMs is only one aspect researchers must address. NVM write operations demand significantly more current and power compared to DRAMs and their own read operations. For instance, flipping an individual bit in PCM requires around 50 pJ/b, compared to writing a DRAM page, which needs only 1 pJ/b [16]. Table 1.2 highlights recent studies aimed at enhancing the lifetime, performance, or energy efficiency of various NVM technologies. Researchers address these issues through different approaches, which can be categorized into three main strategies: (1) designing new data structures and database management systems tailored to NVM limitations, (2) redesigning existing data structures to be more NVM-friendly, and (3) creating

SCM Type	Selected Studies
PCM	[82], [68], [120], [5], [137], [16], [166], [81], [83], [163], [61] [60], [32], [79], [99], [160], [139], [102], [135], [63]
ReRAM	[2], [119], [52], [59], [155], [68], [138], [19], [70], [136], [90]
STT-RAM	[2], [59], [1], [129], [130], [22], [126], [3] [18], [121], [134], [128], [95], [162]
NAND Flash	[40], [133], [100], [105], [12], [49], [25] [27], [88], [45], [96], [13]
3D XPoint	[81], [83], [69], [37], [79], [99], [101], [65], [147], [161] [27], [150], [97], [33], [36], [141], [92], [50], [28]

Table 1.3: Selected studies categorized by their SCM type.

specialized hardware-level or software-level solutions to mitigate NVM drawbacks and limitations.

1.2.2 NVM Technologies

Despite having many benefits over DRAMs, NVM technologies have some limitations in common, such as high energy consumption, low write endurance and asymmetric read/write costs (Table 1.4), which needs to be considered before using them. So, researchers come up with different solutions to deal with these limitations, from redesigning the conventional data structures to proposing hardware-level methods, to deploy these new technologies in their systems. Among all the challenges that NVMs face, in this thesis, we focus on (1) low endurance, high energy consumption, and asymmetric read/write related problems and (2) how researchers in different communities, from databases to storage systems to embedded systems and distributed systems, overcome

Table 1.4: Comparison of memory technologies [77, 144]

Category	Read Latency	Write Latency	Write Endurance
HDD	5ms	5ms	$\geq 10^{15}$
DRAM	$50 \sim 60ns$	$50 \sim 60ns$	$\geq 10^{16}$
PCM	$50 \sim 70ns$	$120 \sim 150ns$	$10^8 \sim 10^9$
ReRAM	$10ns$	$50ns$	10^{11}
SLC Flash	$25\mu s$	$500\mu s$	$10^4 \sim 10^5$
STT-RAM	$10 \sim 35ns$	$50ns$	$\geq 10^{15}$

these limitations. Table 1.3 classifies the research studies from Table 1.2 based on their memory technologies. The NVM technologies that we cover are:

Spin-Torque Transfer RAM (STT-RAM), which is a variation of MRAM, switches the memory states using spin-transfer torque. Using spin-polarized current for setting bits makes the cell structure simple and small. The most noticeable advantage of this memory is to have a high efficiency and write endurance even compared to DRAM. These characteristics make it as one of the top alternatives to the current technologies such as DRAM and NOR Flash. Despite having the mentioned advantages, this technology comes at a price: having low density which makes it hard to scale [122].

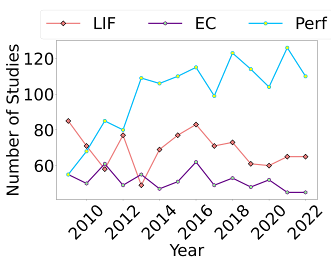
Resistive RAM(ReRAM) is one of the most promising NVM technologies that can take the place of DRAM. It's simple structure, easy fabrication, high scalability, and compatibility with the existing CMOS technology make this technology a good candidate to be used in many applications in various fields from Neuromorphic Computing to logic applications. In ReRAM, by applying current to a cell, the state of the cell can be switched. Despite all its advantages, it has some drawbacks, such as having low write endurance and inconsistent switching mechanism, which makes researchers look for methods to improve its performance and mitigate its disadvantages to use it in the existing systems [157].

Phase-Change Random Access Memory (PCM), which is arguably the most mature of the NVM technologies, consists of phase-change materials that switches between two different phases with distinct properties: an amorphous phase, with high electrical resistivity, and a crystalline phase, with low electrical resistivity. So, in PCMs, there are two main operations: SET operation and RESET operation. These operations are controlled by electrical current as follows: while in the RESET operation High-power are used to place the memory cell into the high-resistance RESET state, for the SET operation, moderate power but longer duration pulses are used to return the cell to the low-resistance SET state. Although PCM scales well and has write endurance comparable to that of NAND Flash (10^8–10^9), which makes it a viable alternate for future high-speed storage devices. However, its asymmetric read/write costs in terms of energy consumption and latency, and lower write endurance compared to DRAM (10^{16}), limit the PCM adoption in system architecture.

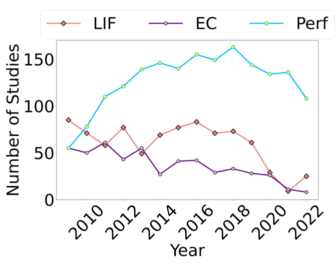
3D XPoint is a relatively recent NVM technology that is developed by the Intel and Micron jointly since 2015. This technology is usually considered as a type of PCM although Intel has never disclosed its technical details [157]. This technology presents many advantages, such as having a high density and low access latency, which makes it one of the best candidates to replace DRAM. However, this technology poses some challenges, which need to be considered. First, its write energy consumption is relatively high. Second, although its write latency is relatively low, it is still orders of magnitude high compared to DRAM. Furthermore, its memory cell write endurance (10^8–10^9) is orders of magnitude lower than that for DRAM although it is claimed that it is enough to survive continuous operation as the main memory for a few years [4].

Flash memory is an electronic NVM storage medium that can be electrically erased and reprogrammed. It is probably the most wide-spread NVM technology since it has the performance better than traditional storage. There are two types of nonvolatile semiconductor flash memory: NOR and NAND. While NOR Flash has low latency and can be programmed at byte granularity, which is suitable for IoT system devices like GPS and e-readers that do not require as much memory, NAND Flash is more like HDD with page-based programming granularity, which is suitable for sequential data such as video, audio, and so on [7]. Despite all the mentioned advantages, this technology shares some disadvantages that most NVM technologies have in common: having limited write endurance, low density, energy constraints, asymmetric read/write latency and persistence.

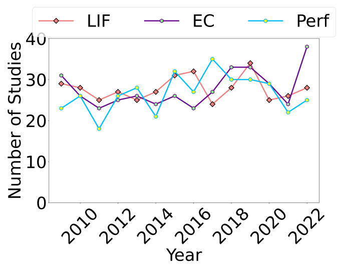
FeRAM, which has similar structure as DRAM bit cell except, achieve non-volatility, it utilizes a ferroelectric layer instead of a dielectric layer. This technology has been gaining popularity, especially in industrial IoT applications and autonomous vehicles field, for its unique characteristics such as having very high life endurance, low write time speed, and low write energy consumption [4]. However, like any other technology in NVM category, it faces some limitations that hinders this technology from its widespread use, such as much lower storage densities than Flash, storage capacity



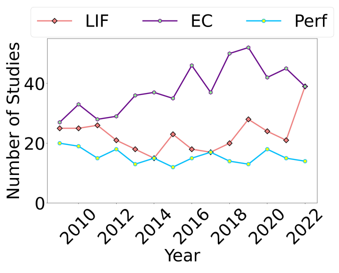
(a) PCM



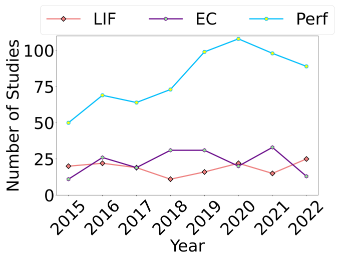
(b) NAND Flash



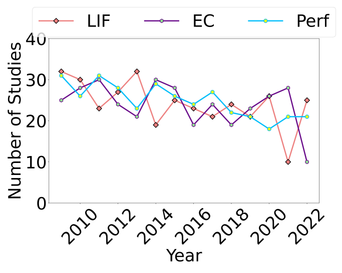
(c) ReRAM



(d) STT-RAM



(e) 3D XPoint



(f) Others

Figure 1.2: The number of studies that are done to improve the lifetime (LIF), energy consumption (EC), and performance (Perf) of different NVM technologies.

limitations and higher cost.

Figure 1.2 illustrates the number of studies that are carried out in the three main categories, i.e., improving lifetime (LIF), performance (Perf), and energy consumption (EC), on different types of NVM technologies from 2009. These results can reveal how much each of the mentioned fields could draw researchers' attention in different NVM technologies over time.

1.2.3 NVM challenges

1.2.3.1 Write endurance

Most NVM technologies, such as PCM, ReRAM, 3D XPoint and Flash memory, have low write endurance (the number of writes that can be applied to a segment of storage media before it becomes unreliable.) The write endurance in these technologies is on the order of 10^5–10^9 writes, which is significantly lower than DRAM write endurance (in the order of 10^{16} write) [82,112]. Even in some NVM technologies such as STT-RAM, which has an endurance of at least 10^{12} writes, certain usage conditions, such as frequent accesses with a high applied bias voltage or a long pulse, may aggravate their lifetime and reliability [78,127]. Furthermore, in many use cases, some of these technologies, such as PCM, STT-RAM and ReRAM, are used in the lower-level cache hierarchies instead of memory level, which expose them to aggressively high number of writes. So, in these cases, write endurance can prevent the adoption of NVM in lower-level caches.

Table 1.2 illustrates some of the selected studies that are proposed recently in an attempt to solve the limited write endurance of NVMs. Although most of the solutions that are presented to improve the lifetime of NVM, such as reducing write amplification, local write optimization and memory awareness, can be applied to various NVM technologies with write endurance problem, in this thesis, we focus on Phase Change Memory (PCM) and how its lifetime can be improved through bit flip reduction.

As mentioned earlier, Phase-Change Memory (PCM) is currently the most mature and widely used NVM technology under research. It is employed in various memory hierarchies, from L2 cache to storage medium, highlighting its potential to replace conventional memory technologies. In Section 2.3, we examine the impact of reducing bit flips on PCM performance, energy consumption, and lifespan. Our experiments demonstrate that minimizing the number of written cells not only extends the device's average lifespan but also significantly reduces power consumption and improves system latency. Additionally, this dissertation aims to bridge the gap between the NVM storage and data management communities by integrating recent advances from the former into existing and future data management systems.

1.2.3.2 Energy Consumption

Bit flip reduction represents the design principle of minimizing how many bits are flipped from 0 to 1 or 1 to 0 when a write is applied to a memory segment. In a PCM device, as the number of bit flips decreases, the write endurance improves. Reducing the number of written cells also means that the energy consumption of the system drops significantly. For instance, based on an experiment conducted by [138], when the number of different bits in the write data and the overwritten content varies from 0% to 100%, the energy consumption can vary from nearly 0 to over 10000 pJ. So, targeting bit flip reduction is a worthy investment in the NVM context, and there are a large body of research, such as [16, 68, 69, 81, 83, 138, 139], targeting bit flip reduction to extend the NVM lifetime and decrease their energy consumption.

To see how this difference affects the system's energy consumption, we have conducted an experiment on a real Optane memory device (Section 2.3). For the

	Content
Old item	0xABCDA23B BCDABCDA BCDAB01A ... 1CDA23BA CCDAAB00 ACCDAB01
New item	0xABCDABC1 BCDABCDA BCDAB01A ... 1CDA23BA CCDAAB00 ABCDABC8

Figure 1.3: An example of replacing a memory content with a similar content used in PNW [81]

experiment, we have used the Persistent Memory Development Kit (PMDK) [1], formerly known as NVML. As we will see in the results, reducing the number of bit flips can have positive effects on the energy consumption of the NVM device. This experiment alongside the ones that we have done to analyze the impacts of bit flip reduction on latency and lifetime of PCMs show that reducing the number of bit flips in the NVM devices, such as PCM, whose controllers are optimized by only flipping bits when the old value of a cell differs from the value being written to it can lead to improvement in lifetime, latency, and energy consumption of the device.

It is worth noting that although we tested our results primarily on Optane, a type of PCM, our designs are applicable to other PCM-based technologies, such as Phase-Change Random Access Memory (PRAM) and Resistive RAM (RRAM), which can also benefit from bit flip reduction. For instance, our deep learning-based method, E2-NVM (Section 4), focuses on improving system energy consumption. This method is particularly attractive for applications using low-power PCM devices, such as those relying on energy-harvesting systems or batteries, including the Internet of Things (IoT) and mobile devices, where power conservation is crucial [15, 21, 113].

[1]Persistent Memory Development Kit https://pmem.io/pmdk/.

1.2.3.3 Latency

Not only does reducing the number of bit flips save the energy consumption of the system, but also it can improve the latency of write operations. Figure 2.2 (the top one) shows that write latency also improves when bit flips are reduced. The main reason behind this is when the content of the old data and the data that is being written is similar, the total number of writes can also be reduced, which results in improving write latency. This process can reduce the number of writes in two ways: (1) the first way is by writing new items in-place to replace a similar old value in terms of hamming distance. This leads to decreasing NVM word writes (i.e., the number of modified words in a cache line.) (2) In the second way, writing similar contents decreases the number of NVM line writes, respectively cache lines needed to be written per item. For example, suppose that the page size in a system is 4KB. In this scenario, if the items are similar to each other in terms of the hamming distance, fewer number of cache lines are needed to fulfill the request (suppose each part in Figure 1.3 is a cache line).

Chapter 2

Prior work

There have been plenty of methods proposed for improving NVM write endurance and energy consumption from hardware-focused methods to software-focused ones. According to their general strategies, these methods can be categorized into three main groups:

1) the storage community developed write optimization techniques that are mostly based on a *Read-Before-Write* (RBW) pattern [154]. In RBW, a write operation w to a memory location x is always preceded with a read of x. The value to be written by w is compared with the old content of x, and only the bits that are different are written. This reduces the number of flipped bits, which increases write endurance [154]. Other approaches built on RBW to increase write endurance by masking or changing the value to be written if it leads to reducing bit flips [6,31,42,73,118]. Flip-N-Write (FNW), for example, checks whether flipping the bits of the write operation would lead to decreasing the number of bit flips [31].

2) the data management community tackled the problem of write endurance

by minimize write operations via techniques such as caching [10, 29, 117] and delayed merging [79,99], or by designing specialized data structures that require fewer writes [166]. Therefore, data structures that are designed to be deployed on NVM should be designed in a way to exploit the advantages and avoid the disadvantages of the technologies. For example, data structures for disks are block-oriented and work the best for sequential access. However, those designed for flash reduce write amplification, which is the main concern in flash technologies [16]. As we will explain later, techniques that focus on reducing write amplification can result in increasing write endurance although this is not always the case [16].

3) The proposed methods in the third group also increase the write endurance of PCMs through a new approach called *memory-awareness*. The main idea behind this approach is to take advantage of having knowledge of the memory content in advance because read operations are less expensive than write operations in PCMs. Prior methods pick the memory location for a write operation arbitrarily (new data items select an arbitrary location in memory, and updates to data items overwrite the previously-chosen location.) The methods in this group judiciously pick a memory location that is similar to the value to be written. When the new value and the value to be overwritten are similar, this reduces the number of bit flips required, as similar values result in fewer changes. Numerous methods employing this approach as a foundation will be explored in Section 2.4.

In the following, we present the main concerns, challenges, and limitations of state-of-the-art methods that have utilized NVMs in their designs by dividing them into three main groups based on the trends and solutions they propose to solve the problem of write endurance and energy consumption. We also identify the short- and long-term research opportunities in this space.

2.1 Wear leveling

Wear leveling has been studied extensively for Flash-based storage devices [14, 54, 89, 94, 103]. In these methods, the wear-leveling algorithms usually keep track of the storage blocks and remap those blocks that are written heavily in a given time quanta to the lowest wear-out blocks. In storage class memory, wear leveling has almost the same objective, which is extending the lifetime of NVM devices by distributing the writes evenly across the memory blocks of NVM so that no *hot* area reaches its maximum lifespan by extremely high concentration of write operations [26, 31, 42, 68, 124, 138, 158]. These methods usually are implemented in the memory controller level to protect NVMs. Wear-leveling is usually transparent for upper-level applications and they can simply access to the same content using the same logical address (they are unaware of the physical address where the data are stored.) According to the mapping strategies, existing wear-leveling schemes can be divided into two main groups:

(1) table-based wear-leveling (TBWL) techniques that store the logical addresses (LA), their corresponding physical addresses (PA), and the frequency of accesses. In this way, the storage table of the wear-leveling can be eliminated. When the number of writes to a specific physical address (PA) goes beyond a threshold, its content is swapped by

the wear-leveling method [68, 164].

For example, In [123], the authors propose Fine-Grain Wear Leveling (FGWL), which shifts cache lines within a page to achieve uniform wear out of all lines in the page. Also, when a PCM page is read, it is realigned. The pages are written from the Write Queue to the PCM in a line-shifted format. For a system with 4 lines per page, for instance, the rotate amount is between 0 and 3 lines. The rotate value of 0 means the page is stored in a traditional manner. If it is 1, then the Line 0 of the address is being shifted one line and stored in Line 1 of the physical PCM page, line 1 of the address in stored in line 2, line 2 in 3, and finally, line 3 of address space is stored in Line 0. When a PCM page is read, it is realigned. The pages are written from the Write Queue to the PCM in a line-shifted format.

Self-adaptive wear-leveling (SAWL) algorithm [68] is another wear-leveling scheme that dynamically adjusts the wear-leveling granularity to accommodate more useful addresses in the cache, thus improving cache hit rate. This method distributes the writes across the cells of entire memory, thus achieving suitable tradeoff between the lifetime and cache hit rate.

(2) Algebraic-based wear-leveling methods [124,132,143] are another group that try to distribute the incoming writes to avoid concentrating writes on specific physical locations and creating the hot areas. To this end, they usually replace the address-mapping table in the table-based wear-leveling algorithms with hardware structure, which are more space efficient [124]. In this way, they can increase the lifetime of NVMs orders of magnitude compared to the based line where there is no wear-leveling.

Start-Gap [124] is one of the first methods that proposed a wear-leveling method based on the algebraic mapping between the logical and physical addresses. In this technique, there are two registers (Start and Gap) that do the wear-leveling. When a new write comes to the memory, Start-Gap moves one line from its location to a neighboring location. While the Gap register keeps track of the number of lines moved, Start register counts the number of times that all the available lines have moved. Finally, the mapping between logical and physical address is done by a simple arithmetic operation of Gap and Start registers, which eliminates the need for storing the address-mapping table in the memory.

In [156], the authors propose a hardware-based wear-leveling scheme named Security Refresh, which performs dynamic randomization for placing PCM data. In this method, an embedded controller inside each PCM is responsible for preventing adversaries from tampering the bus interface or aggregating meaningful information via side channels [156]. They also applied designed some attacks to analyze the wear-out distribution using Security Refresh.

Although wear-leveling strategies have been successful in preventing creation of hot locations and extending the lifetime of NVMs, the controller cannot guarantee that some cells will not wear out much faster than the average. The reason is that distributing writes evenly across the memory space does not necessarily mean that the individual cells within the words also be flipped/written evenly. That is why, in some extreme cases, even with the protection of state-of-the-art wear-leveling schemes, wear-out attacks such as Remapping Timing Attack [66] and Row Buffer Hit [110] can wear out NVM as fast as 137 seconds [110]. Therefore, hardware techniques such as FNW [31], CDE [42], FPC [59], and Flip-Mirror-Rotate [118], which will be explained in the next section, have been proposed to focus on reducing the number of bit flips within a given word instead of just distributing writes uniformly across the device.

2.2 Reducing write amplification

Many data storage and indexing solutions target the reduction of write amplification to optimize the utilization of I/O bandwidth. This is done via various techniques, including delaying the consolidation of writes [79,99], caching [10,29,117], and others [166]. With the introduction of NVM to the memory hierarchy, it turns out that reducing write amplification can have the positive side-effect of increasing NVM write endurance since less data is written. However, this is not an easy task to do due to the fact that all the existing data structures and database systems have been designed for DRAMs and HDDs, where the challenges of the lifespan of memory segments and the energy consumption of writes are not as significant in DRAM/HDD as they are in NVM. However, as discussed before, when it comes to NVMs, write operation needs to be performed wisely. So, the proposed methods in this group reduce the write amplification in an attempt to decrease the average number of updated cells and as a result increases the lifetime of NVMs.

To achieve this, many methods re-design existing data structures and database systems to mitigate the write amplification issue caused by them instead of designing and building new ones from scratch. The reason behind this is that existing data structures and database systems have undergone decades of research that makes them extremely efficient and makes building alternatives from scratch an arduous task.

Log-Structured Merge-tree (LSM-tree) is one of those data structures that has been widely adopted for use in the storage layer of modern NoSQL systems, and as a result, has attracted a large body of research, from both the database community and the storage systems community, that try to improve various aspects of LSM-trees by using NVMs [37, 79, 105]. NoveLSM [79] is one of these methods. This method is a persistent LSM-based key-value storage system designed to take advantage of having a non-volatile memory in its design. To tackle NVM's limited write endurance, NoveLSM comes up with a new design, where only the parts of the key/value store that do not need to be changed frequently, such as immutable memtables, are handled by NVM. On the other hand, other parts, such as mutable memtables, which need constant updates and data movements, are placed on DRAM, which do not have any restrictions on write operation.

WiscKey [105] is another work, which proposes a persistent LSM-tree-based key-value store, which has been derived from the popular LSM-tree implementation, LevelDB. Although, like the other methods in this category, WiscKey focuses on decreasing write amplification, it achieve this through a different and simple way, which is separating keys from values. This method observes that since the indexing is done by keys, and not values, they do not need to be bundled together when they are stored in the LSM-tree. So, in this method, only keys are kept sorted in the LSM-tree, while values are stored separately in a log. Through this insight, they have reduced write amplification by avoiding the unnecessary movement of values while sorting. Although this technique is originally proposed for SSDs, it can be generalized to storage class memories, which suffer from the same limitation.

Another data structure that has been redesigned to utilize NVMs is B+-Trees, which is used widely in K/V data stores [29, 65]. Fingerprinting Persistent Tree (FPTree) [117] is a hybrid SCM-DRAM persistent and concurrent B+-Tree that is designed specifically for NVMs. This method aims to decrease write amplification on NVMs. To do so, in this method, leaf nodes are persisted in SCM while inner nodes are placed in DRAM and rebuilt upon recovery.

LB+-Tree [101] is another method that changes the structure of the conventional data structures to take advantage of NVMs. In this work, to improve the insertion performance of LB+-Tree, three techniques are proposed: (1) entry moving, which creates empty slots in the first line of a leaf node to reduce the number of NVM line writes, (2) logless node split, which targets the logging overhead, and (3) distributed headers. LB+-Tree improves the performance of the insertion operation compared to the other methods.

DPTree (Differential Persistent Tree) [165] batches multiple writes in DRAM persistently and later merge them into a PM component to decrease the persistence overhead, which is imposed to the system because of the persist primitive that the existing PM index structures use to guarantee consistency in case of failure.

In [30], the authors target the tail latency of tree-based index structures, which the result of the internal structural refinement operations (SROs) and the inter-thread interferences. To reach to this aim, uTree introduces a shadow linked-list layer to the leaf nodes of a B+-tree to minimize the SRO overhead. This method has succeeded in improving throughput and latency.

Hash-based indexing structures have also been good candidates to utilize NVMs due to their nature of typically causing high write amplification and that they are vastly used in various applications and systems [69,108,152,166]. A lot of effort has been made to improve hash-based indexing structures for byte-addressable persistent memory, and almost all of them focus on decreasing the write amplification to reach their goal. Path hashing [166] is an example of these hash-based indexes, which is designed specifically for NVMs. The basic idea of path hashing is to leverage a position sharing method to resolve the hash-collision problem, which usually results in a high number of extra writes or write amplifications.

Dash [104] proposes a persistent hashing scheme that aims to solve the performance (scalability) and functionality challenges that persistent hash tables face by reducing both unnecessary reads and writes. To achieve this, Dash takes advantage of some previous techniques, such as fingerprinting, optimistic locking, and combine them with some novel methods such as bucket load balancing technique. They have tested their method on real Optane DCPMM and the results show that Dash can improve throughput significantly. It is worth noting that the methods that the methods that we mentioned in this section are merely some examples of the whole group of methods 1.2 that utilize NVMs in their work in different ways.

Although the methods in this category tackle the problem of energy consumption and write endurance by minimizing write amplification, they conflate the problem of energy efficiency and write endurance with the problem of write amplification. While in many cases a technique that leads to reducing write amplification has the side-effect of increasing energy efficiency and write endurance, this is not always the case as shown by prior work and our own evaluations [16, 82].

All these methods offer different advantages and disadvantages. These methods invite exploring how they can be integrated with existing data management systems to enable them to improve the lifetime of NVMs. Some of these methods are independent from the application (and often implemented as a hardware method) which means that augmenting them within existing data management systems is a straight-forward task. Other approaches—especially ones based on masking—require domain knowledge on the application using them. There is an opportunity for data management researchers to find ways to adapt these methods to work with existing data management systems. This would entail learning the write patterns of data management systems and translating this knowledge into appropriate masking techniques that are based on the methods presented above.

2.3 Bit Flip Reduction

Although reducing write amplification is a promising way to extend the PCMs' lifespan, it does not necessarily lead to the best opportunities to reduce bit flipping and

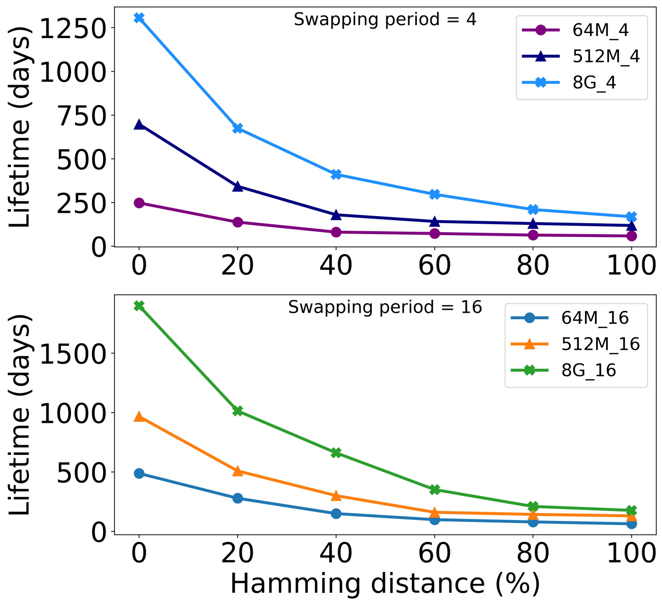


Figure 2.1: The impact of capacity and swapping period on PCM's lifetime when the percentage of hamming distance between the write data and the overwritten content changes.

increasing write endurance [16,81]. This is because—unlike flash—PCM cells are written individually, which means that the number of flipped bits is more important to optimize than the number of written words [16]. Therefore, focusing on reducing bit flips is a viable solution that can both save energy and extend the life of PCMs.

To see how this difference affects NVM's performance in terms of lifetime, energy consumption, and latency, we have conducted some experiments on a PCM device and a real Optane memory. Figure 2.1 shows the lifetime of a PCM device is affected by the number of bit flips that occur during write operations. As we can see in this figure, when the average percentage of the hamming distance between the old value of the cell and the value that is going to be written increases, the lifetime of the device decreases.

This figure also shows swapping period plays an important role in the lifetime of the device. Based on the results, for a certain hamming distance, when the swapping period is high, the device lasts longer because having lower swapping period means more write amplification, which increases the overall flipping bits. However, setting lower swapping period increases the risk of certain cells being worn out sooner than others, especially when there is a malicious attack. Figure 2.1 also shows that when the capacity of the device is low, e.g., in mobile and embedded systems, memory cells can wear out faster.

To see how bit flip reduction affects the latency and energy consumption of a NVM device, we have conducted a simple experiment on a real Optane memory device (Section 2.3) using the Persistent Memory Development Kit (PMDK) [1], formerly known as NVML. In this test, first, we allocate a contiguous region of N Optane blocks of 256B. During each "round" of the experiment, we first initialize all the blocks with random data, and then update the blocks with new data with content that is x% different than the data that is already in the block (hamming distance). We use PMDK's transactions to persist writes. We measure the latency and energy consumption of the socket for each round. Figure 2.2 shows that by overwriting similar content, which needs less bit flipping, we can save a huge amount of energy and improve latency.

Generally, the existing bit flipping reduction methods can be divided into two main categories: RBW-based techniques and Memory-awareness.

[1]Persistent Memory Development Kit https://pmem.io/pmdk/.

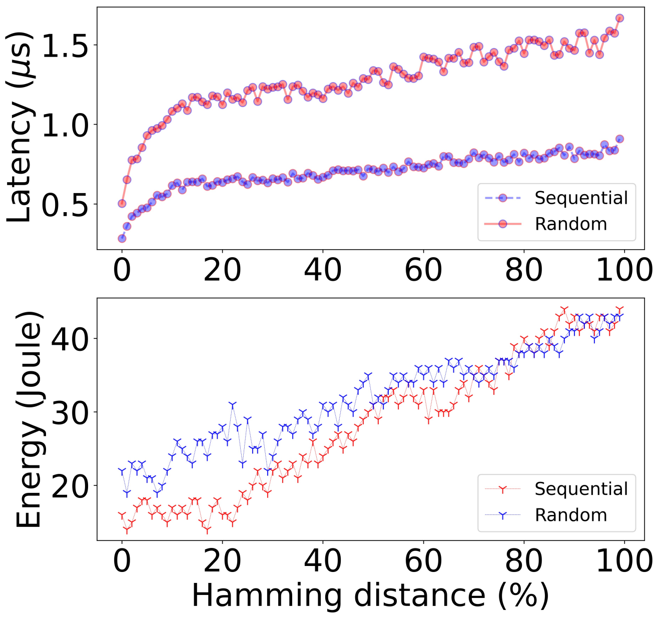


Figure 2.2: The latency and memory energy consumption on a real Intel Optane memory device for read and write operations with different percentages of hamming distance.

2.3.1 RBW-based techniques

One of the most important characteristics of Read Before Write (RBW) technique and its variants is their simplicity and efficiency in dealing with the lifetime issue of NVM. So, there has been a large body of research that uses various types of this method in their works. In this category, there are various techniques, such as caching [10, 29, 117] and the Read-Before-Write (RBW) technique [154], to decrease the number of bit flips.

RBW is one of the most popular techniques, which has been widely utilized by various approaches [6, 31, 42, 73, 118], to reduce the number of bit flips is the Read-Before-Write (RBW) technique [154], in which the content of an old memory block is read before it is overwritten with the new data (Figure 2.3). This technique replaces

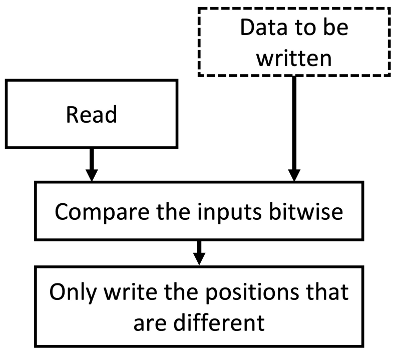

Data to be written
Read
Compare the inputs bitwise
Only write the positions that are different

Figure 2.3: Read Before Write (RBW) technique.

each PCM write operation with a more efficient read-modify-write operation. Reading before writing allows comparing the bits of the old and new data, updating only the bits that differ.

Flip-n-Write (FNW) [31] is one of the most popular methods and became the building block of many other techniques in this area. This method compares the current content of the memory location (the old data) with the content to-be-written (the new data). This enables FNW to decide whether to write the new data in its original format or to flip it before writing it if that leads to reducing the number of bit flips. (A flag is used so that future operations know whether to flip the content before reading.) This method guarantees that the number of bit flips in PCM is always less than half the total number of written bits (excluding the flag bit).

DCW [42] finds common patterns and then compresses data to reduce the number of bit flips in PCM. Like Flip-N-Write, DCW replaces a write operation with a read-modify-write process. It starts comparing the new data and the old data from the first bit to the last one. The most significant difference between DCW and Frip-N-Write is that in DCW, the maximum number of bit flips is still N (the word width).

Captopril [73] is another recent proposal for reducing bit flips in PCMs. This method masks some “hot locations”, where bits are flipped more, to reduce the number of bit flips. In this method, the authors compare every write with 4 predefined sequences of bits to decide which bits need to be flipped and which ones need to be written in their original form. This method suffers from relatively high overhead. More importantly, it is rigid and would only work on predefined applications.

Flip-Mirror-Rotate [118] is another method that is built upon Flip-N-Write [31] and FPC [6] to reduce the number of flipped bits. Like Captopril, this method uses only predefined patterns to mask some bits, which means it would only work on predefined applications.

MinShift [107] proposes reduces the total number of update bits to SCMs. The main idea of this method is that if the hamming distance falls between two specific bounds, the new data is rotated to change the hamming distance. Although this method is simple, it suffers from high overhead.

In [46], the authors use a combination of MinShift and Flip-N-Write to decrease the number of written bits. They compute the minimum amount of some possible states to choose a pattern to encode the data. This method has advantages and disadvantages of both methods.

2.4 Memory-awareness

As discussed earlier, the writing operations in PCM takes much more energy than reading operation. To save energy, the PCM controller can avoid writing to a cell whose content is the desired value. It means that the lifetime of the cell and the consumed energy in the write operation depends on the number of bits that are actually being flipped by the write rather than the number of words or bits that are written.

Although focusing on bit flipping reduction technique seems a reasonable choice, the methods in this category fail to achieve its full potential because the existing methods miss a crucial opportunity. Prior methods pick the memory location for a write operation arbitrarily (new data items select an arbitrary location in memory, and updates to data items overwrite the previously-chosen location.) This misses the opportunity to judiciously pick a memory location that is similar to the value to be written. When the new value and the value to be overwritten are similar, this means that the number of bit flips is going to be lower. Reducing the number of bit flips increases write endurance and reduces power consumption. We call this approach "memory awareness."

2.4.1 Content-aware methods

The methods in this group, by being aware of the memory content, try to redirect write requests to overwrite specific memory locations based on their content-aware replacement policies to decrease the energy consumption of the system and extend the lifetime of the device. For example, the authors in [158] proposes an encoded content-aware cache replacement policy to reduce the total switched bits in spin-torque magnetic random-access memory (STT-MRAM) caches. To do this, instead of replacing the LRU block, the victim block is chosen among the blocks whose contents are most similar to the missed one. To avoid the comparison of the entire 512 bits of the blocks, each block is encoded using 8 bits, which incur low space overhead. The reason behind this encoding is that when the contents of two blocks are dominated by certain bit value, there is a good chance that the content similarity of the two blocks is high, hence may lower the switch bits when one double word replaces the other [158].

Data Content-aware (DATACON) [138] is a recent mechanism that reduces the latency and energy of PCM writes by redirecting the write requests to a new physical address within memory to overwrite memory locations containing all-zeros or all-ones depending on the content of the incoming writes. DATACON is implemented inside the memory controller. To keep track of the all-zeros and all-ones memory locations and their address translations, the memory controller needs to maintain a table. Although this method takes advantage of being content aware, it needs to be implemented inside the existing memory controller, which is non-trivial task. Moreover, the average number of bit flips it can save is highly dependent on the workload since it uses two fixed patterns to save bit flips.

In [16], the authors modifies the common data structures based on the idea of pointer distance to minimize the number of bit flips on PCMs. In this method, instead of building a doubly-linked list, for instance, XOR linked lists are used, which allows each node to store only the XOR between the previous and next node instead of storing the previous and next nodes. The results show that storing the XOR of two pointers, which are likely to contain similar higher-order bits, reduces the number of bit flips, which can lead to reducing power consumption.

2.4.2 AI-based methods

Machine learning and deep learning are revolutionizing various fields, from power systems and transport to storage and database systems [9, 51, 55, 91]. While the use of machine learning in the area of database management and storage systems is not entirely new, leveraging it to extend the lifetime, energy efficiency and latency of NVMs has not been explored until our introduction of this approach in “Predict and Write.” In our work, Predict and Write, we introduced the concept of “AI-based memory-aware” methods for the first time in this domain. The core idea of AI-based methods is to utilize machine learning techniques to learn the existing write patterns on Non-Volatile Memory (NVM) devices. This enabled us to direct incoming write operations to the optimal memory locations, minimizing the number of bit flips. By doing so, we can significantly enhance the energy efficiency and longevity of NVM devices.

As discussed earlier in Section 2.3, our extensive tests demonstrate a positive correlation between reducing bit flips and improving the lifetime, energy efficiency, and latency of NVM devices. To achieve this reduction in bit flips, we leveraged AI-based models that analyze the data distribution of existing data to identify the optimal free memory locations for incoming writes. This approach minimizes bit flips and enhances the overall performance and longevity of NVM devices.

Reducing bit flips using software-level solutions addresses two major challenges faced by hardware solutions. First, hardware algorithms need to be small and simplistic to fit within the memory controller, which limits their complexity and effectiveness. Second, developing hardware-based methods is often inaccessible to researchers, as most storage solutions for wear leveling and bit flip reduction are proprietary and require new hardware manufacturing.

To achieve this, we designed various software-level memory-aware solutions that harness the power of machine learning and deep learning models to enhance the adaptability of NVM devices in existing database and storage systems. Our software-level implementations map free memory locations based on their hamming distance, avoiding the computational and space constraints of memory controller-based solutions. Incoming write operations are intercepted and directed to free memory locations with similar hamming distances. This mapping is facilitated by training machine learning and deep learning models on the free memory locations available on the device. Implementing these learning models in software, rather than in the memory controller, allows for more efficient and sophisticated solutions. In this thesis, we present and discuss the challenges encountered in applying learning models to this problem, including the use of efficient models that combine clustering, dynamic address pools, and indexing data structures.

Chapter 3

Predict and Write: Using K-Means Clustering to Extend the Lifetime of NVM Storage

In this chapter, we introduce our first work "Predict and Write" (PNW) [81], an NVM-based K/V store that uses a dynamic approach to minimize bit flips adapting to new applications and dynamic workload changes. PNW decreases both the number of NVM line writes as well as the number of NVM word writes (see section 3.3.1).

We leverage machine learning (ML) to continuously learn a model that reflects the existing write patterns of a given workload. The model learns to cluster memory locations in NVM enabling the placement of future writes to locations that minimize the amount of bit flips. Furthermore, by periodically retraining the ML model, it adapts dynamically to a changing workload without the need for user intervention. It is worth noting that unlike the previous methods, which are based on the RBW technique, PNW does not depend on NVM hardware modifications. This is because we do not require using hardware-based read-modify-write operations before write operations as we can avoid writing similar data at a larger granularity (*e.g.* a cache line). However, future work on combining PNW with custom hardware support could further reduce the number bit flips at the bit or byte level.

Our design consists of a ML model, a hash index, a table for storing metadata named the *dynamic address pool* and a data zone to store the actual data or K/V pairs (see section 3.2.1). We also show in the evaluation section that the performance benefits obtained from the ML technique significantly outweigh its overhead in terms of space cost and time. This is the case even when the ML models are running on CPUs without using specialized hardware. Future extensions of our proposal to use methods that process the ML model on specialized hardware such as accelerators and TPUs would further improve the efficiency of our approach [91].

3.1 Predict and Write

Whenever there is a need for updating memory in-place, the number of bit flips depends on the hamming distance between the old data—currently in the memory location—and the new data, which is going to overwrite the memory location. PNW reduces bit flips by avoiding in-place updates and, instead, finding a new memory location for each write that would minimize the hamming distance. By placing the write

Table 3.1: An example of a PCM with 6 elements

Cluster	Index	Content
1	0	’0’, ’0’, ’0’, ’0’, ’0’, ’1’, ’1’, ’1’
	1	’0’, ’0’, ’0’, ’0’, ’1’, ’0’, ’1’, ’1’
2	2	’0’, ’0’, ’1’, ’0’, ’1’, ’1’, ’0’, ’0’
	3	’0’, ’0’, ’1’, ’1’, ’1’, ’1’, ’0’, ’0’
3	4	’1’, ’1’, ’0’, ’1’, ’0’, ’0’, ’0’, ’0’
	5	’0’, ’1’, ’1’, ’1’, ’0’, ’0’, ’0’, ’0’

operation in the right memory location that minimizes the hamming distance between the old and the new data, the number of bit flips can be significantly reduced. While promising for reducing bit flips, this technique introduces several challenges. First, it requires an indirection layer to map a logical value to its current physical location. As the write unit size of NVMs is a byte, storing these mappings on the byte level introduces a significant overhead. Second, the technique requires computing the hamming distance between the new (to-be-written) data and all the available physical data locations. Computing the similarity between all locations is prohibitive.

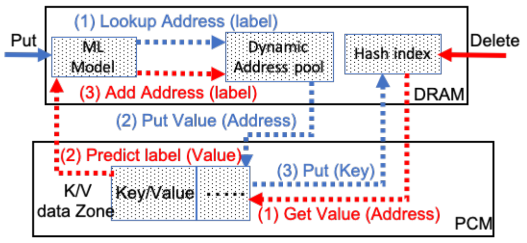
The first challenge is addressed by leveraging a K/V store that already implements an indirection layer to map keys to values. To address the second challenge (finding the right memory location for a write operation to minimize the hamming distance), we introduce a machine learning approach based on k-means clustering.

The intuition behind our clustering approach is that we cluster similar memory locations in terms of the bit patterns of their contents. Using this clustering, we can quickly retrieve a new memory location for a PUT operation such that the hamming distance between the new to-be-written data and the old memory location where it will be written is minimized. We do not need to perform k-means clustering for each PUT/DELETE operation; instead, it is sufficient to perform clustering periodically. We evaluate the training frequency and its effect on reducing bit flips in Section 3.3.6.

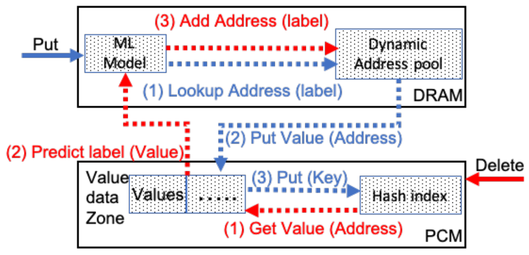
To illustrate our approach, consider a storage system that is using a PCM as its persistent memory with a capacity of six equal sized (8 words) entries, managed by a free-list which we refer to as the *dynamic-address-pool* (Table 3.1). Now, suppose that we have two PUT operations that write the following new data items, d1: ['0', '0', '0', '0', '1', '1', '1', '1'] and d2: ['1', '1', '1', '1', '0', '0', '0', '0'].

In a regular system, where updates are applied in place, there exists only one option to write the data and hence the reduction of bit flips with techniques such as FNW is limited. PNW, on the other hand, determines the best memory location to write the new data by computing the minimum hamming distance between the new data and existing free memory locations maintained in a table called "dynamic address pool". Computing all hamming distances grows in complexity with the number of entries in the dynamic address pool and hence becomes intractable. To overcome this problem, PNW groups the entries in the dynamic address pool into clusters according to their hamming distance.

For instance, we can group the elements from the example in Table 3.1 into three clusters where indexes 0 and 1 form cluster 1, indexes 2 and 3 form cluster 2, and indexes 4 and 5 form cluster 3. Now, if we receive the same new items d1 and d2, we direct them to clusters that are closest to them, which are clusters 1 and 3, respectively.



(a) Proposed architecture for small keys



(b) Proposed architecture for large keys

Figure 3.1: An example of procedures which serve K/V PUT and DELETE operations for a) small and b) large keys.

These items are grouped together because the K-means model assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. In this example, the centroids for the first, second, and third clusters would be [0. 0. 0. 0. 0.5 0.5 1. 1.], [0. 0. 1. 0.5 1. 1. 0. 0.], and [0.5 1. 0.5 1. 0. 0. 0. 0.], respectively. Because the variations within clusters are minimal, the data points are homogeneous (similar) within the same cluster. In this scenario, wherever we decide to write the items within their corresponding clusters, we will end up writing only 1 bit for each item, without any extra flag bits. This is a simple example of how PNW works.

It is worth noting that PNW reduces the number of writes in two ways: (1) the first way is by writing new items in-place to replace a similar old value in terms of hamming distance. This leads to PNW decreasing NVM word writes (*i.e.*, the number of modified words in a cache line.) (2) In the second way, PNW decreases the number of NVM line writes, respectively cache lines needed to be written per item. For example, suppose that the page size in our system is 4KB as shown in Figure 1.3. In this scenario, if the items are similar to each other in terms of the hamming distance, fewer number of cache lines are needed to fulfill the request (suppose each part in Figure 1.3 is a cache line). This enables PNW to decrease NVM word writes in addition to NVM line writes.

3.2 Key-Value Store Design

In this section, we present the design of our K/V store utilizing the Predict-and-Write technique. We first describe the ML model and then discuss the capabilities supported by our proposed K/V store.

3.2.1 Overview and system model

Our design consists of a ML model, a hash index, a table for storing available (free) NVM addresses *dynamic address pool*, and the *K/V data zone* to store the K-V pairs. In Figure 3.1, we show a K/V store on a DRAM-NVM hybrid memory layout using our PNW method. Our data store implementation supports K/V operations including GET, PUT, and DELETE.

3.2.1.1 Machine Learning Model

Our proposed machine learning method learns the existing data distribution among real-world workloads to decrease the bit flips in write operations. We utilize an unsupervised ML model that is able to cluster data elements into a number of clusters based on their similarity. In particular, we leverage K-means clustering to cluster the

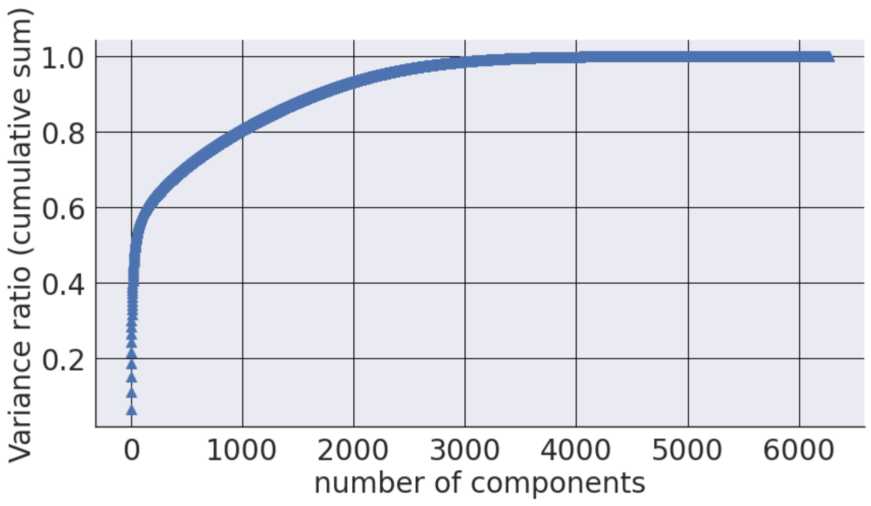


Figure 3.2: PCA variance ratio according to the number of principal components.

available data on PCM. The size of the buckets (the unit of the value size) can vary ranging from a word size to the size of a page or even the size of a document depending on the system.

In our system, each memory location is encoded as a vector of bits, each of which is used as a feature/dimension. The entire data zone can be encoded as a 2D tensor (that is, an array of vectors) of shape (n, m), where the first axis (n) represents the samples (old data) and the second axis (m) represents the features. Because the size of the buckets can be very large (thousands of bits), it can lead to a problem referred to as the "curse of dimensionality", which increases the training time and space complexity of the model significantly.

Addressing the Curse of Dimensionality To tackle the curse of dimensionality problem, we use Principal Component Analysis (PCA) on the data sets used in PNW reducing the number of dimensions before training the model. Although PCA is applicable to all data sets, it is especially useful for the ones with a very large number of features. Projecting data to a lower dimensional subspace is very common in different areas such as meteorology, image processing, and genomics analysis, especially before K-means clustering is applied [44,75,114,159]. The main basis of PCA-based dimension reduction is to keep only the principle components (features) which explain the most variance in the original data [20]. Figure 3.2 shows the PCA variance ratio according to the number of principal components for MNIST, which is one of the data sets we use in our tests. In this example, we only keep the first 1000 principal components (features) because they are enough to represent more than 80% of the variance in the data.

Determining the Number of Clusters Another important decision that needs to be made before training the model is to determine the number of clusters (K). There are a number of ways to determine the optimal value for K [24]. In this work, we use one of the most common techniques called the "elbow method" [76,109,142]. The elbow method is expressed as the following Sum of Squared Error (SSE) [142]:

$$SSE(X, \Pi) = \sum_{i=1}^{K} \sum_{x_j \in C_i} \|x_j - m_i\|_2^2 \tag{3.1}$$

where $\|.\|_2$ denotes the Euclidean (L2) norm, $m_i = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j$ is the centroid of cluster C_i where the cardinality is $|C_i|$, $\Pi = \{C_1, C_2, ..., C_K\}$, and $X = \{x_1, ..., x_i, ..., x_N\}$ (N is the feature vector).

In this method, the value for SSE is calculated as we increase the number of clusters. To determine the optimal number of clusters, we identify a sharp decrease

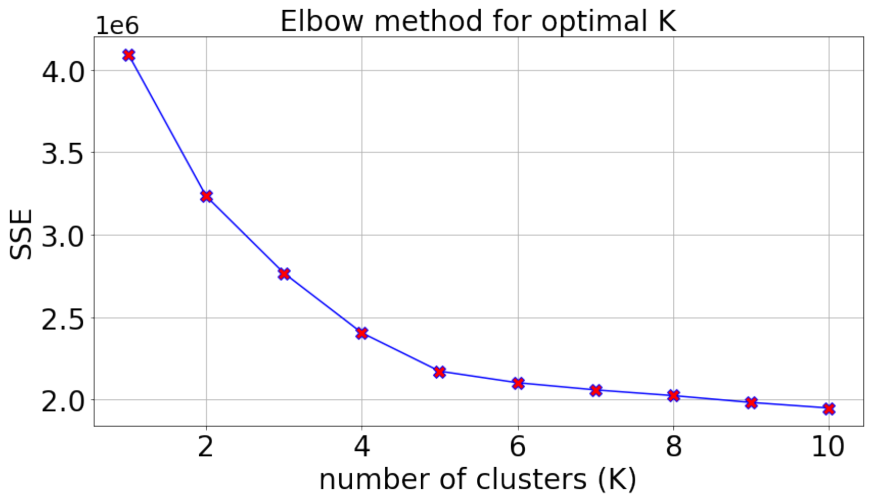


Figure 3.3: Sum of Square Error graph to find the optimal K.

known as the "elbow" or "knee", which suggests the optimal value for K [109, 142, 145]. Figure 3.3 shows an example of choosing the optimal K by seeing the significant decrease in the SSE graph, which is in K = 5 (the data set is MNIST).

The ML model is constructed on DRAM as it does not need to be persistent and can be reconstructed after a crash. By constructing the model on DRAM, we take advantage of both DRAM's high write endurance and DRAM's high speed. Another advantage of our proposed method is that this model can be replaced by any customized learning model.

3.2.1.2 Dynamic address pool

The *dynamic address pool* is a table that contains a number of entries, equal to the number of clusters in the ML model (Figure 3.4). Each entry in the dynamic address pool contains a free-list of the available memory locations that belong to the same cluster, as it is learned by the ML model.

Initialization. The first step of initialization is creating a K-means clustering model based on the number of clusters we want to have, and then training the model on all the available data in the NVM storage called the data zone (Algorithm 1). The next step is to label data items in each memory location (line 3). Finally, we add the available addresses on the data zone to their corresponding entry in the dynamic address pool (lines 4 and 5). Now, when a PUT request is received by the system, the ML model finds its label, and based on that label, the dynamic address pool returns one address from corresponding cluster. We maintain a flag for each address in the dynamic address pool to indicate whether it is available. We also remove memory addresses out of the dynamic address pool when they are allocated to a K/V pair and reinsert them afterwards to ameliorate the cost of keeping a flag per address in terms of lookup time.

It is worth noting that the storage overhead of the dynamic address pool is proportional to the number of pointers that are stored per value. As a result, for large values, the size of the table does not grow significantly. For small values, however, the number of addresses that needs to be stored per value can grow substantially. To limit the table size, we set a fixed number of entries in the table, so the size of the table cannot not grow to more than a specific maximum threshold. In this way, the table is used by adding addresses in and removing them from the table until the number of available addresses goes under a minimum threshold, called the load factor, which is described in details in section. 3.2.3.

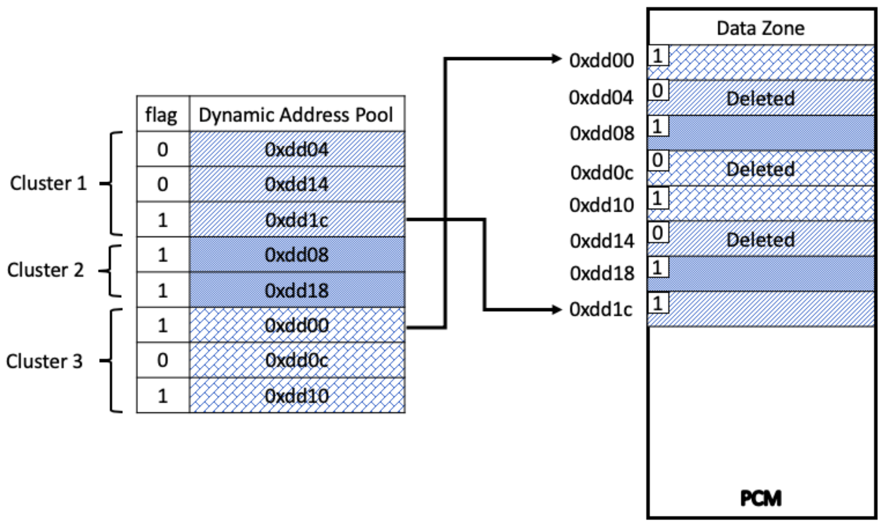


Figure 3.4: Dynamic Address Pool.

Algorithm 1: Initialization

```
// n_clusters:  number of clusters
// D′ and A: content and addresses of the data zone
// DAP: Dynamic Address Pool
// N: len(D′)
1: model = KMeans(n_clusters)
2: model.train(D′)
3: labels = model.labels_
4: for (i:=0, i<N, i++)
5:    DAP[labels[i]].append(A(i))
```

3.2.1.3 Hash index

Indexing is critical in designing K/V stores. Our hash index component maps each key to the memory location that contains its value in the NVM data zone. To build indexes that support K/V operations, there exists a variety of choices ranging from B+-Tree to LSM trees to hashmaps. The operational efficiency of each indexing structure varies from one implementation to another and hence the optimal implementation is application specific. For the existing implementation, we choose hash indexing, however, it can be replaced with any other indexing data structure. The only requirement of the indexing structure is that it can map logical keys to arbitrary physical memory addresses.

We have two choices to store the indexing structure:

- If we place the indexing structure into PCM (Figure 3.1(b)), there is no need to rebuild it during the recovery from a crash. However, it also introduces extra writes to the NVM because of the write amplification problem induced by indexing data structures such as B+Trees and hash indexing. It is a good design choice when the size of the keys are large because in that case the wear-out cost of the hash index is negligible. However, for small keys, it represents a problem which we mitigate by leveraging data structures such as NVM-friendly hashing indexes [166].
- Another design choice is to build the indexing structure on DRAM (Figure 3.1(a)). This architecture is particularly beneficial when the size of the keys are small. In this case, we do not pay any cost for the extra bit flipping that is caused by the write amplification of the indexing structures. Nonetheless, we need to build the

whole data structure from scratch during recovery after a crash.

In the evaluation, we build and persist a write-friendly hash index in PCM as introduced in [166]. We perform the tests based on this design to explore the worst case scenario of putting the hash index on PCM in terms of extra bit flips introduced by write amplification. Also, for every entry in the hash index, there is a flag bit that shows whether the corresponding key is available or not. In particular, whenever we receive a delete request, we can reset its corresponding bit in the hash index to reflect that the corresponding index does not exist anymore instead of deleting it. We can do the same procedure for deleting a K/V pair from the data zone.

3.2.2 Supported K/V Operations

3.2.2.1 PUT Operation

PUT and UPDATE operations are executed as follows. As shown in Figure 3.1, when our system receives a write request such as PUT, the model is queried to determine the cluster that is closest to the value-to-be-written in terms of their hamming distance. Then, a memory address is returned from that cluster by using the *dynamic address pool*. Then, the K/V pair is written into the returned address, which is in the K/V zone on NVM. Finally, the newly-added index entry is added to the hash index (step 3).

Algorithm 2 illustrates the pseudo-code of the write operation under the PNW scheme (Figure 3.1). The first step of PNW is to find its label, which is equal to its corresponding entry in dynamic address pool, using the ML model (line 1). Next, we

Algorithm 2: Write operation

```
// D′ and D: old and new (key,value)
// DAP: Dynamic Address Pool
Write (D: (key,value)){
 1: E = model.predict(D); //predict the entry
 2: A = DAP.get(E);//get the address
 3: D′= Read(A); //old (key,value)
 4: DAP.remove(A) //remove the address from DAP
 5: for each bit in {D} and {D′}
 6:   if they differ, update memory bit
 7: HI.put(D, A) //update the hash index}
```

Algorithm 3: DELETE operation

```
// D′: old key
// DAP: Dynamic Address Pool
// HI: Hash Index
Delete (D′: key){
 1: A = HI.get(D′); // get the address
 2: Reset-Flag-Bit(A); // delete
 3: E = model.predict(Read(A)); // predict the entry
 4: DAP.update(A:address, E:entry); // add the address
    back to DAP
}
```

select one of the available addresses from the corresponding entry in the dynamic address pool, and write the data to the address (lines 2 and 3). Next, we need to remove the selected address (A) from the cluster's free-list in the dynamic address pool (line 4). Finally, only the bits (in the buffer D) that are different than the data in PCM (D') are actually updated (lines 5 and 6). We also need to update the hash index at the end to enable finding the value for future lookups (line 7).

3.2.2.2 DELETE operation

Algorithm 3 illustrates PNW's delete operation (also see Figure 3.1). The delete procedure is accomplished by the following steps. In step 1, to find the item in the K/V data zone, the delete request is directed to the hash index, and then the associated entry is deleted from the K/V data zone by resetting the associated flag bit (lines 1 and 2). In this step, the delete operation is completed; however, to make the system more efficient, we recycle the recently freed address back to the dynamic address pool by finding the label of the deleted data (line 3), and then adding the freed address to the corresponding entry in the dynamic address pool (line 4). In this way, the address can be used again in the future, and the model is re-trained less frequently.

3.2.2.3 UPDATE Operations

An update operation can be implemented in two different ways:

- If we care about the write endurance more than latency, the update operation consists of the delete operation plus the PUT operation in order to prevent bit flipping as much as possible. It means that the item that has to be updated is first deleted from NVM (delete operation), and then its new place is found by in a dynamic address pool (PUT operation) using the model. It is worth noting that we can do the DELETE-PUT process asynchronously to mitigate the latency problem. In other words, the system can retain synchronous updates to K/V items and the hash index in NVM, and for the dynamic address pool in DRAM, it can be asynchronously updated through the model in the background to hide the extra latency.

- On the other hand, if the application cares about latency more than the other factors, especially wear-leveling, the request just needs to go through the *hash index* to find its place in the K/V data zone and then update the item in place without any further changes since it does not affect the dynamic address pool. In this way, we sacrifice wear-leveling to achieve lower latency.

In our system and evaluations, we follow the first approach as our main goal is to increase write endurance. However, it turns out—as we present in experimental evaluations—that minimizing bit flips is also good for performance alleviating the trade-off between write endurance and latency.

3.2.2.4 GET Operation

Read operations in our system are straight-forward as they do not lead to changing any data structures. Specifically, a get request goes through the hash index to find its corresponding value from the K/V data zone, and then the read value is returned.

3.2.3 Additional design considerations

It is possible that all the available addresses of a cluster (called cluster C) are utilized. In this case, if the model sends a request that requires a new address from cluster C, the dynamic address pool will not be able to serve this request because there are no more addresses available in that cluster. To avoid this problem, we define a load factor for the K/V data zone on the NVM. Setting the load factor to x percent, means that when x percent of the available addresses in the K/V data zone are used, the K/V data zone needs to be extended. To add new memory addresses to the data zone, we need to train a new model. It is worth noting that, unlike traditional methods, we do not need to move or change anything in the hash table on NVM because they still have valid information. The only things that need to be changed are the model and dynamic address pool, which are both located on DRAM. So, our method to expand the size of a cluster does not impose any extra writes to the NVM.

The main reason behind defining the load factor is to prevent latency spikes or stalls in the system. The load factor is similar in principle to the load factors that are used in hashing schemes as a way to monitor the space utilization of the system to prevent hash collisions. In other words, the load factor is going to warn us that the system will need to be retrained in the near future. So, before this happens, we can re-train a new model, by adding some new memory locations to the K/V data zone, in the background while the system is running. Then, we can switch to the new model and table before the previous model gets stuck. In this case, we can hide the re-training latency and the system works without disruptions due to retraining. We have done some tests in the next section to figure out the best time to start training a new model before the old one is full to keep the system working smoothly. PNW supports any size of key/values from 32-bit word size to the page sizes of 4KB to the size of a document. Thereby, the way in which data elements are provided to the models depends on the K/V pair size. For instance, small (e.g. 64 bit) data elements can be directly passed to the model, while for large data element (e.g. 4KB) we first apply dimensionality reduction using PCA before passing the data to the model.

3.3 Evaluation

3.3.1 Methodology

In this section, we evaluate our proposed method using different metrics focusing on the reduction in writes and bit flips. We leverage a collection of real and synthetic data sets. Since only insert and delete requests cause mutating the state of the NVM, we insert n items into the K/V store followed by deleting 0.5n items (except for section 3.3.6). Also, we do not make any assumption about the access pattern within or across clusters. So, we simply apply the K-means clustering (from the scikit-learn library) based on the available memory locations on PCM. We compare PNW with both RBW solutions and K/V stores. For the former, we compare with the writes on the storage component of

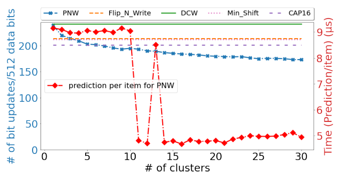
PNW, which is the data zone.

We compare our results against other methods described in Section 2, such as FNW [31], DCW [154], Captopril [73] and MinShift [107]. For synthetic data sets, our sample K/V store system has at least 10M buckets. When there are 10M buckets, for instance, we first warm-up K/V stores with 10M key/values. This means that we store some items as “old data” before starting our tests. The data type and distributes of these items differ depending on the test. “old data” is used for the initial training of the ML model.

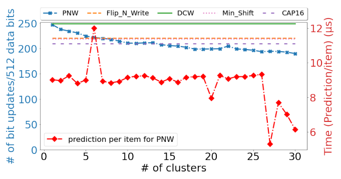
To compare PNW’s results with other methods, we tune their parameters in such a way that they achieve their best performance. For example, we allow MinShift to shift n times, where n is the size of the item instead of the size of the word, which means it always results in its best performance in terms of the number of bit flips [107]. With respect to Captopril, we also considered its best case, which happens when the blocks are partitioned into n = 16 segments [73].

Unless we mention otherwise, we execute the K/V operations with randomly selected key/values from the same generator. As real NVM DIMMs are not available for us yet, we emulate NVM using DRAM similar to prior works [67, 116, 146, 152]. We assume an access latency of the latest 3D-XPoint of 600ns [72, 115].

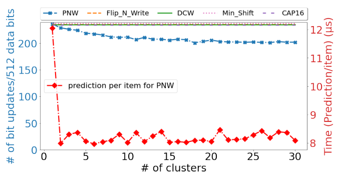
The experiments are executed on an Intel Core i7 processor running at 2.2 GHz with 2 cores (4 logical cores), each of which has 256KB L2 Cache and 4MB L3 Cache using 8 GB of RAM, running macOS Catalina (version 10.15.4). The reason that we run the tests on a local computer without any GPU support is to get a sense of how



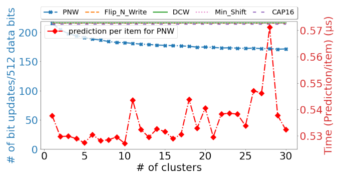
(a) Amazon samples



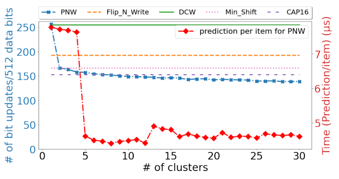
(b) 3D road network



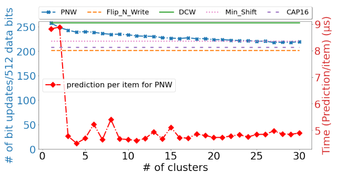
(c) Sherbrooke



(d) traffic surveillance



(e) normal distribution



(f) uniform distribution

Figure 3.5: The average number of actual bit updates per writing 512 bits as well as the latency of prediction per item in PNW for the real-world textual and numerical data sets (a-b), multimedia data sets (c-d), and synthetic data sets (e-f).

our methods work on an ordinary system without any unique capabilities. We test our proposed method using various data sets, which can be categorized as real-world textual and numerical data, real-world multimedia data including image and video data sets, and finally, hard-to-cluster synthetic data sets. In the following subsections, we show the results of the tests on these data sets and analyze them.

3.3.2 Real-world textual and numerical data sets

The first data set is called Amazon Access Samples Data Set [1], containing 30K log entries. Although this data set has 20K attributes, in this test, only less than 10% of them are used for each sample. For this test, we first have set aside 5K buckets as the

[1]https://archive.ics.uci.edu/ml/datasets/Amazon Access Samples

"old data" on the NVM memory and then warmed up the system by writing 5K items from the data set into our buckets. Then, we replaced this "old data" with new incoming data from the same data set (the remaining 25K items). Figure 3.5(a) illustrates that when there are one or two clusters, the number of written bits in our method is more than FNW. Nevertheless, when the number of clusters is more than 2, we start to get better results until we reach between 15%(compared to CAP16) to 70%(compared to the conventional method) improvements compared to the other methods when the number of clusters is 30.

The next real-world data set, i.e., 3D Road Network Data Set [57,86], contains information of road networks in North Jutland, Denmark (covering a region of 185 x 135 km^2). We used the same setup as above for this data set containing 434874 entries. In this test, we chose 100K buckets as "old memory" and warmed up the system by 100K entries from the 3D Road Network Data Set. The results are shown in Figure 3.5(b). When the number of clusters is big enough (here k = 14), PNW starts to outperform all the other methods in terms of the number of bit flips until it gets its highest performance when k=30 (between 10% to 63% improvements compared to the other methods).

Finally, the last real-world data set is one of the collections of a database called DocWord, which consists of five text collections in the form of "bags-of-words". This collection, which is called PubMed abstracts [47], consists of 730 million words in total. For doing the tests, we first created 100M buckets as the "old data" storing data from the PubMed data set. Then, we wrote the new incoming data items from the same data set on the previous data items stored on the buckets and kept track of the number of the updated bits per 512 bits.

3.3.3 Real-world multimedia data sets

In the first set of tests, we have used some video data sets to see what happens if a system, for instance, a CCTV recorder, uses an NVM media as its persistence memory. We have used two video data sets: 1) The Sherbrooke video data set [74], representing a two-minute-long video (with resolution 800x600). 2) A Traffic Surveillance video [11], collected from seven intersections in the Danish cities of Aalborg and Viborg, containing 21 five-minute sequences of two cameras including RGB and thermal data. The resolution of both cameras is 640x480 pixels, and the frame rate is fixed at 20 fps. In this test, we just used one sequence of RGB camera called "day sequence 2".

For the first data set (Sherbrooke), we stored the first 30 seconds of this video as the old data, and for the second one (Seq2), we stored the first one minute of the video as the old data and used the remaining of the video as the new data. The results are shown in Figure 3.5(c) and 3.5(d), respectively. These figures show that our method outperforms the other ones in both data sets. For the first data set (Sherbrooke), PNW improves the other methods between 14% to 60% and for the second one (Seq2), we outperforms the other ones between 21% to 67%.

The next data set is one of the most widely used data sets for machine learning research, and especially for computer vision algorithms, i.e., CIFAR-10 data set [93]. This data set is a subset of the 80 million tiny images data set and consists of 60,000 32x32 color images, grouped into ten different classes. Similar to the previous experiments, we first set aside 10K of these images as the old data to fill out the 10K buckets we created as our NVM system. Then, the new incoming data items are written in place of the old ones one by one.

3.3.4 Hard-to-cluster synthetic data sets

In this section, we are going to observe the behavior of PNW on some synthetic data sets that do not follow any specific data distribution. The reason of doing these tests is to discover the limitations of our ML-based method and analyze them to give the readers a clearer view of the possible applications of PNW. To perform these tests, we start with a synthetic data set that shows a clear pattern and then test two more data distributions that are completely different in terms of their data pattern.

For the synthetic data sets, we used 32-bit keys and values. We also generated two types of integer data (normal and uniformly distributed), ranging from 0 to 2^{32}. For random integers, we generated them via a pseudo-random number generator. For the normal data set, we generated a synthetic data set of 100M unique values sampled from a normal distribution with $\mu = 2^{31}$ and $\sigma = 2^{28}$ to test our method. In all synthetic data set tests, the confidence interval was less than 10^3 for 95% confidence level.

First, we show the results of the first synthetic data set, following a regular pattern. Figure 3.5(e) shows the results for different number of clusters ranging from k=1 to k=30 for normal distribution. We have compared the performance of PNW to the other ones in terms of the number of bits updated/written per 512 bits. In this figure, we observe that when we pick k=1, the result for PNW is not different from

DCW since both do the same thing if there is no clustering. Our approach enhances the results of DCW and FNW more than 40% and 25%, respectively, when the number of clusters is more than 10. It also outperforms MinShift and Captopril more than 15% and 10%, respectively. Also, the delay is almost 5μs to 6μs most of the time.

In the second experiment, we did the same, but for a different data distribution, i.e. uniform random distribution, to learn more about the behavior of our method. Data sets like this one are highly random, and as a result, difficult to learn using an ML model. The results are depicted in Figure 3.5(f) showing that although our method has succeeded in improving the results for DCW, MinShift, and the conventional method by almost 15%, 5%, and 60%, respectively, it lags behind FNW and CAP16 for this data set as expected for the random data set.

In some of the previous results, there are anomalies where the number of bit flips suddenly jumps while increasing the number of clusters. Such anomalies are due to the unpredictability of ML-based methods. However, we expect that such anomalies would be normalized during extended operation.

3.3.5 End-to-end write latency

In the following, we are going to measure the write latency, which includes the time spent on 1) predicting a cluster number, 2) finding an empty bucket within the dynamic address pool, and 3) writing the key/value on NVM. We do this test to measure the overhead of our method.

In Figure 3.6, we show the write latency comparisons for various data sets. In

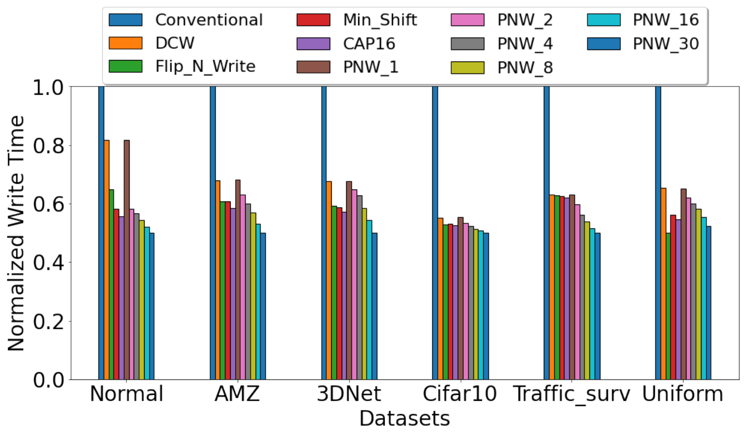


Figure 3.6: End-to-end write latency comparison for various data sets.

this test, we use the normal and uniform data distributions, Amazon Access Samples, 3D Road Network, CIFAR, and the *day sequence 2* traffic surveillance video. For our method, we had to train the model based on the old data, filling out the dynamic address pool, and then writing the new data. The write latency is calculated based on the number of cache lines that are written per item. In this test, we observe that each method that updates fewer bits has a higher chance of having a lower write latency because it has to update fewer cache lines than the others.

Figure 3.6 shows the normalized time of the write operation required by different methods. As illustrated in this figure, our proposed method, when the number of clusters is enough, can outperform the others even though it has to perform two additional steps. The reason is that our method performs fewer write operations than the other ones, and it makes up the time it spends on the extra steps. However, for the uniform data distribution, we could not do the same since PNW is not able to find a clear pattern

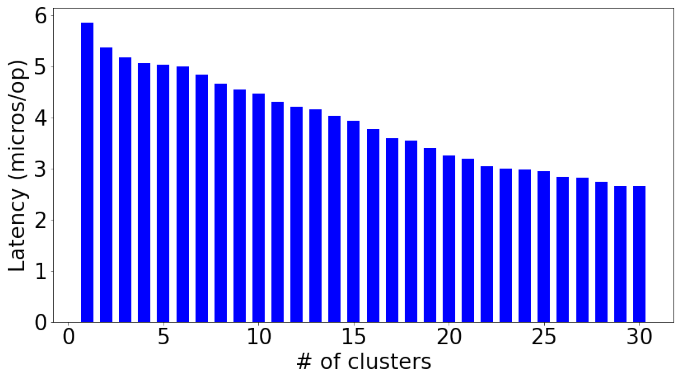


Figure 3.7: The impact of choosing the number of clusters (K) on the average write latency for the PubMed abstracts data set.

among the data items to make up the extra steps.

Figure 3.7 compares the average write latency for different number of clusters (K) on the PubMed abstract data set. In this test, to see the impact of K on latency, we invoke insert and delete operations on the system in a 1:1 ratio. Note that the value of K does not affect the lookup request latency because in the lookup, the request does not go through the model or the dynamic address pool. This test shows that by increasing K, latency decreases because all the items within a cluster become more similar (in terms of hamming distance). So, the new items can be written by replacing old ones with a fewer number of cache line writes, which leads to decreasing latency.

In the next test, we compare PNW with recent K/V stores to see its performance in terms of the number of written cache lines. Like the previous test, since only insert and delete requests cause writes to NVMs, we first insert n items into the system and then delete 0.5n items. FP-Tree [117] is a hybrid SCM-DRAM persistent B+-Tree method

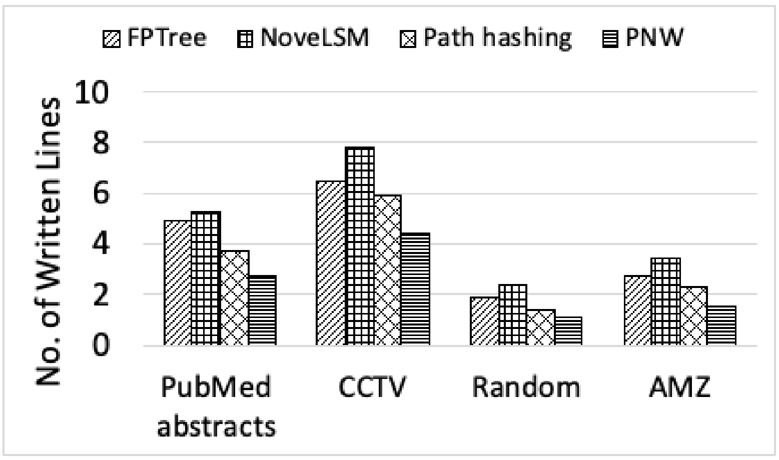


Figure 3.8: The average number of written cache lines for each request.

that we implement and compared PNW with. The second persistent K/V store that we compare PNW with is NoveLSM [79], which is a persistent LSM-based K/V storage system. It is designed to exploit non-volatile memories in an attempt to provide low latency and high throughput to applications. We also implement a hashing scheme that is designed for NVMs called Path hashing [166]. It is worth noting that for this test, we implement PNW as shown in Figure 3.1(a).

Figure 3.8 shows the average number of written cache lines for each request. The number of written cache lines per request in FPTree and NoveLSM is higher than others because they modify more items to process a request. Although the number of written lines in path hashing is fewer than the others, its written lines are higher than PNW because: 1) It incurs more writes when re-hashing to handle conflicts, and 2) like other methods, it is not “memory-aware”. PNW has the fewest written cache lines mostly because it can save some cache lines per request because of replacing the old

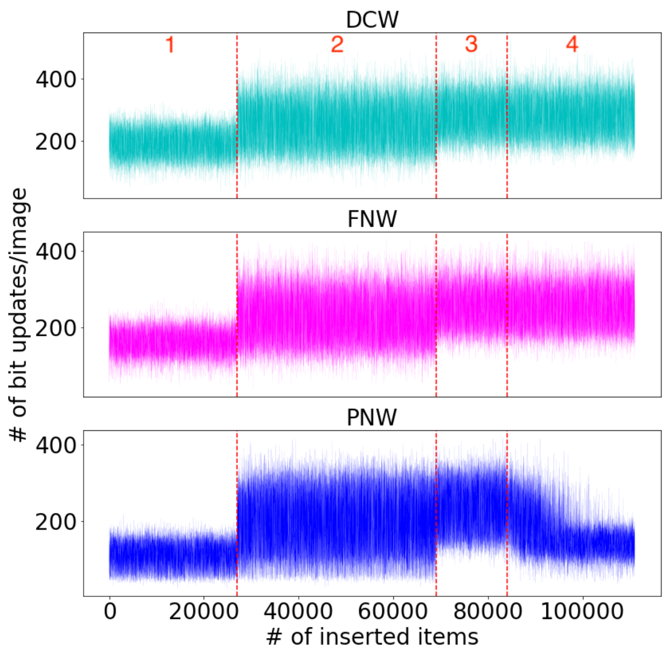
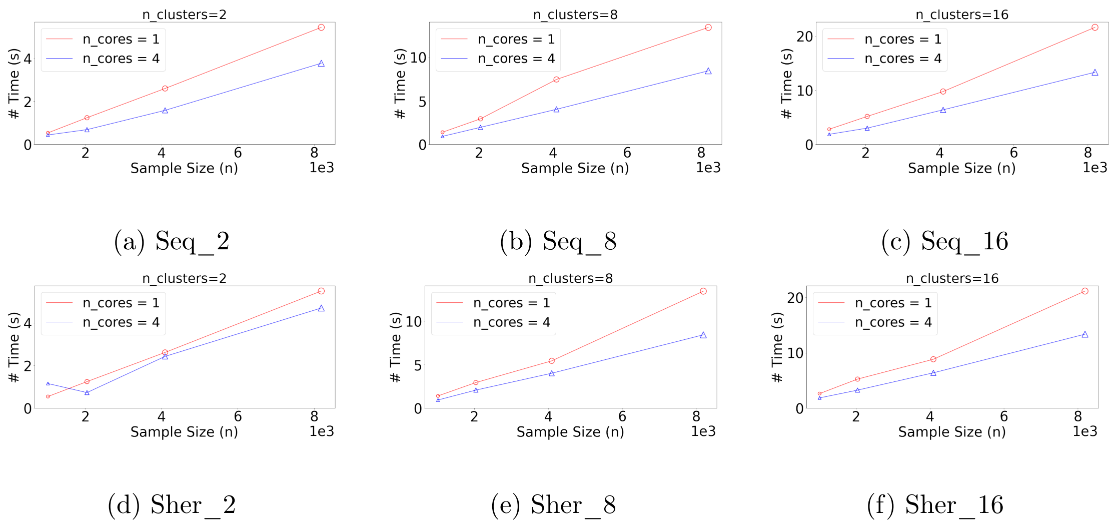


Figure 3.9: The performance change by converting the workload from MNIST into Fashion-MNIST over time.



(a) Seq_2

(b) Seq_8

(c) Seq_16

(d) Sher_2

(e) Sher_8

(f) Sher_16

Figure 3.10: PNW's average model training time for different data sets using single core versus multi-core processing.

items by similar new items. We also observe that for some data sets the average number of written cache lines is higher for all methods because of the larger item size.

3.3.6 Training overhead

To see how rapidly can our method adapt to changing workloads, we conduct the last experiment to track the behavior of our method while changing the workload. You can see the results in Figure 3.9. In this test, we use two data sets from the Keras library, i.e., MNIST database of handwritten digits and Fashion-MNIST database of fashion articles, each of which contains 60,000 28x28 gray scale images, along with a test set of 10,000 images. For this test, we did the follows steps:

- Phase 1: we stored 28K images from the MNIST data set as the old data. After training the model and creating the dynamic address pool, we started streaming 27K images from the same data set (MNIST) as the new data into the system to overwrite the old data. As we can see in Figure 3.9, there is no noticeable change in the performance of the system in the first 27K frames. Even at the end of this stage, where the old data is almost completely replaced with the new one, we still do not see any substantial change in the performance.
- Phase 2: we send a mixture of items from two different data sets, i.e., Fashion-MNIST and MNIST, at the ratio of 2 to 1. We shuffled 15K of MNIST images with 30K of Fashion-MNIST and then sent them to the system as the new incoming data. As it is obvious in the figure, the performance is affected immediately (the

number of updated bits increases) since two-third of the incoming data are entirely different from the previous ones and as a result have a larger hamming distance.

- Phase 3: In this phase, we sent 12K images only from the second data set, i.e., Fashion-MNIST. The number of updated bits fluctuated less since the old data contains the items mostly from Fashion-MNIST, and the incoming data is also from the same one too.
- Phase 4: In this phase, we continued sending 28K images from the second data set (Fashion-MNIST) with one difference: we re-trained our model on the old data, which contains the images from the Fashion-MNIST data set now. As you can see in the figure, the results got better and fluctuated less.

As a result, we have seen that, depending on the application and the workload, we do not always have to re-train the model rapidly, and we can use the same model for a certain amount of time before it needs to be re-trained. This allows us to do the retraining in the background lazily and update the model periodically.

PNW is designed to enable re-training in the background while the current model is serving requests. However, to set the load factor to its correct value, PNW needs to know when to start re-training the model before the old one becomes inefficient, i.e. the system's performance decreases in terms of the number of bit flips. This is of great importance because we might not want to give all the available resources to the model since the system needs to serve the requests without any problem while the new model is being re-trained. We performed additional experiments to evaluate the costs for re-training a new model using different number of the available cores (Figure. 3.10). These experiments are performed on the traffic surveillance [11] and the Sherbrooke video data sets [74].

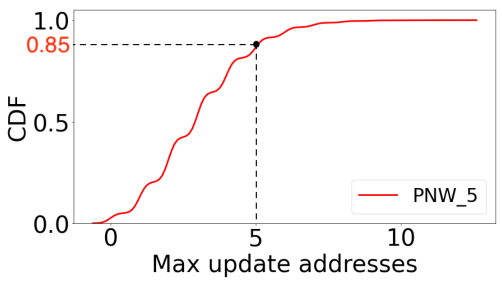
In this test (Figure. 3.10), we calculate the time needed for re-training the model for 2, 4, 8, and 16 clusters. In each case, we did the test on two different modes: 1) running the model on a single core; and 2) running the model on all 4 cores. As we can see from the results, as the value of k and the sample size increases, the model needs more time to be re-trained. For instance, for training a model with k=16 clusters on more than 8000 samples/frames (Figure. 3.10(a)), we need almost 20 and 13 seconds if we use one and 4 cores, respectively. This can give us an idea of setting the load factor in a way that we have enough time to finish re-training the new model before the old model becomes inefficient. So, if we have more than one core available for us in the system to train the model, multi-core processing is worth it when the sample size is big enough.

3.3.7 Wear-leveling

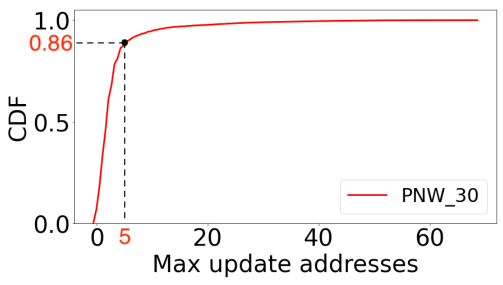
Aside from decreasing the number of writes, wear-leveling is equally important to extend the lifetime of PCM. The reason is that some blocks of PCM may receive a much higher number of writes than the other blocks, and as a result, wear out sooner [112, 151]. Therefore, to observe the performance of PNW in terms of the distribution of the maximum number of bit flips and the wear-leveling of PCM, we conduct two more tests. In these tests, we run PNW in two different modes, i.e. for k =5 and k = 30 clusters, on the combination of MNIST and Fashion-MNIST data sets. Like the previous test, we first warm up the data zone with 28K items from the combination of both data sets. Then, we stream 112K writes from the same data sets to the system. During the test, we also perform delete actions to make space for incoming writes. In other words, each word in the data zone is updated 4 times on average.

Figure. 3.11 shows the maximum number of times the addresses in the data zone are written as a cumulative distribution function (CDF). In other words, this figure illustrates the estimation of the likelihood to observe an address in the data zone of PCM that is written less than or equal to a specific number of times. For example, as we can see in Figure. 3.11, the estimated likelihood to observe an address in the PCM data zone to be written less than or equal to 5 ($P\ (X \leq 5)$) is 85% and 86% when we have $k = 5$ and $k = 30$ clusters, respectively. We also observe that more than 99% of the addresses in the data zone experience no more than 10 writes for $k = 5$ and 15 writes for $k = 30$. This results show that, regardless of the number of clusters, PNW distributes write activities across the whole PCM chip.

Finally, we analyze the wear-leveling of memory bits as CDFs. Figure. 3.12 illustrates the estimation of the likelihood to observe a memory bit in the data zone of PCM that is written less than or equal to a specific number of times. For instance, we observe that while the estimated likelihood of a memory bit being written less than or equal to 4 times is 74% for $k=5$ clusters (Figure. 3.12(a)), this likelihood rises to 98% when $k=30$ (Figure. 3.12(b)). This important observation shows an interesting fact about PNW: By increasing the number of clusters, bit flips are distributed more evenly

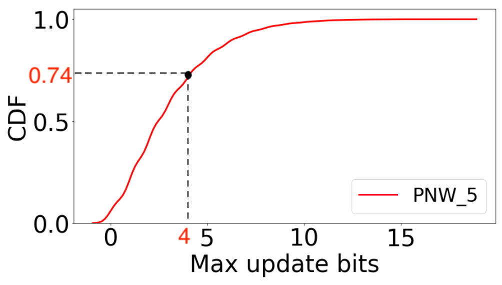


(a) k = 5

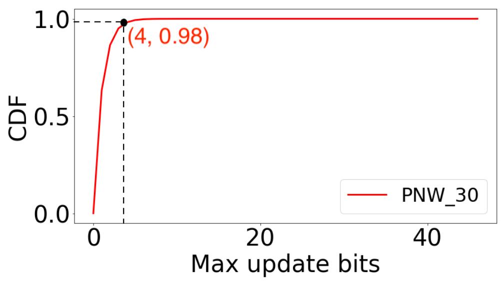


(b) k = 30

Figure 3.11: The maximum update addresses as CDFs by applying PNW with a) k=5 and b) k=30 clusters.



(a) k = 5



(b) k = 30

Figure 3.12: Wear-leveling as CDFs by applying PNW with a) k=5 and b) k=30 clusters.

across the whole data zone of the PCM chip, and as a result, the lifetime of PCM is extended more. The reason behind this is that when the number of clusters increases, the items within the clusters become more similar to each other. Therefore, regardless of the number of clusters, PNW evenly distributes writes not only in the address level but also in the bit level.

3.4 Conclusion

Building storage systems, such as K/V stores, on hybrid memory, creates unprecedented opportunities to utilize fast memory access to achieve enhanced performance compared to those on traditional hard disks or flash-based solid-state drives (SSDs). In this work, we improve the write bandwidth, write energy, write latency, and write endurance of NVMs through *Predict and Write* (PNW), a K/V store that uses a clustering-based machine learning approach to extend the lifetime of NVMs using machine learning.

To this end, we bring a well-known unsupervised machine learning algorithm, called K-means clustering, into NVMs territory. In our proposed method, we have built the model on DRAM based on the existing old data on PCM. In this way, we try to prevent data items from being moved on PCM by making the model change, which is on DRAM. In other words, by bringing all the extra writes from NVM to DRAM, PNW does not impose any extra writes to NVM. We examined the performance of our proposed approach with others in terms of different factors such as the number of writes and the latency for various workloads, on both synthetic and real-world data, with different distributions of data. The results show that our method outperform existing solutions and that the benefit of using a ML model outweigh its overhead. Based on the results, by choosing the right target memory location for a given PUT/UPDATE operation, PNW has succeeded in reducing the number of total bit flips and cache lines over the state of the art.

Chapter 4

E2-NVM: A Memory-Aware Write Scheme to Improve Energy Efficiency and Write Endurance of NVMs using Variational Autoencoders

4.1 Introduction

Although our initial work, PNW [81], paved the way for utilizing machine learning to extend the lifetime of NVMs, in E2-NVM, we made significant advancements to enhance its speed and adaptability. As you will see later in this chapter, these improvements are evident in both performance enhancements and new design considerations. Performance-wise, incorporating Variational Autoencoders (VAE) has improved accuracy and reduced latency, addressing fundamental overheads in our previous machine learning-based design. In terms of new designs and considerations, we have (1) introduced padding strategies to manage data with variable sizes, (2) addressed the swapping nature of underlying memory controllers, and (3) last but not least, thoroughly studied and evaluated the implications of our work on energy efficiency.

In this chapter, we present E2-NVM [80], a software-level memory-aware solution, which is implemented in software and does not suffer from the compute and space constraints of solutions implemented in the memory controller. E2-NVM is implemented as a storage layer that maps free memory locations according to their hamming distance. Incoming write operations are then intercepted, and placed on a free memory location that is similar in terms of their hamming distance. To perform this mapping, E2-NVM trains a deep learning model using the free memory locations. The use of deep learning is possible because E2-NVM is implemented in software rather than in the memory controller. In this section, we present and discuss the challenges we faced in applying VAE to this problem. This includes using an efficient model, which is a combination of a VAE and a clustering model, to overcome the limitations of traditional clustering methods when they have to deal with high dimensional data. We also tackle the problem of supporting memory segments of variable sizes. We propose a data padding strategy that allows using the same VAE model for memory segments with different sizes. What distinguishes E2-NVM from content-aware methods is that it works in the software-level instead of the memory controller. This means that it has access to more compute and state to perform more complex mapping operations. In particular, using deep learning requires more compute (to train and process the model) and state space (to store the mappings and model).

4.1.1 System Model

The system model consists of hardware and software components. E2-NVM does not require any special hardware. We consider a hybrid DRAM-NVM architecture, where both devices are placed on the memory bus. The NVM device, in addition to the memory segments, contains a *memory controller* that intercepts all operations to NVM. The memory controller may utilize a wear leveling solution that swaps memory segments periodically. The details of wear leveling methods are typically proprietary. However, prior work has indicated that wear leveling approaches perform a memory segment swap every ψ write operations. Typically, the value of ψ is in the order of 10s of writes [68]. E2-NVM is a storage layer that sits between software applications (such as data stores) and the hardware components.

4.1.2 Motivation: Software-Level Bit Flipping Reduction

To see how bit flip reduction affects the latency and energy consumption of a NVM device, we have conducted an experiment on a real Optane memory device, which is one type of PCM, using the Persistent Memory Development Kit (PMDK) [1], formerly known as NVML. In this test, first, we allocate a contiguous region of N Optane blocks of 256B. During each "round" of the experiment, we first initialize all the blocks with random data, and then update the blocks with new data with content that is x%

[1] Persistent Memory Development Kit https://pmem.io/pmdk/.

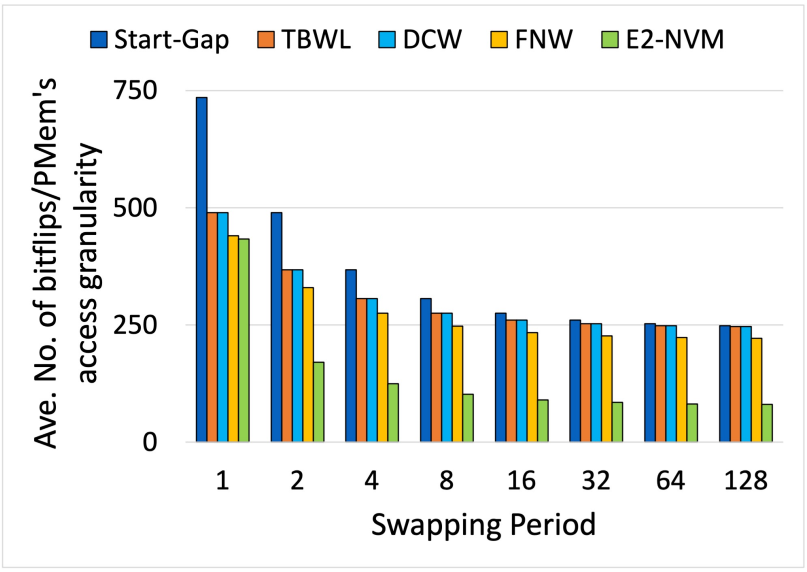


Figure 4.1: The average number of bit updates for different wear-leveling techniques when swapping period changes.

different than the data that is already in the block (hamming distance). We use PMDK's transactions to persist writes. We measure the latency and energy consumption for each round. Figure 2.2 shows that by overwriting similar content, which needs less bit flipping, we can achieve an average energy savings of up to 56%. The experiments also show the potential of improving write latency which is important as it can offset some of the overhead that is incurred by software-level solutions that aim to reduce bit flips. This improvement in latency is due to the ability to write fewer cache lines when the cache line to be written is identical to the one in the memory segment. In this case, the memory controller avoids writing them, which reduces the average latency [81].

This potential of reducing bit flips using software-level solutions overcomes two challenges that faced hardware solutions: The first is that to be deployed on hardware, algorithms need to be small and simplistic—in terms of computation power and memory space—to fit in the memory controller. The second is that developing hardware-based methods is not accessible to researchers. This is evident by how most storage solutions for wear leveling and bit flip reduction are proprietary and requires manufacturing new hardware to implement a new solution. It is worth noting that although we provided our results on Optane, which is one type of PCM, E2-NVM is applicable to other phase change material-based technologies, such as phase-change random access memory (PRAM) and Resistive RAM (RRAM), which can benefit from bit flip reduction. Since E2-NVM's main focus is to improve energy consumption of the system, our proposed method can be especially attractive to the applications that use low power PCM devices due to relying on energy-harvesting systems or batteries [21], such as the Internet of Things (IoT) and mobile devices.

Figure 4.1 shows how E2-NVM can achieve its goals despite the interference and segment swapping from the underlying memory controller. We used Amazon Access Samples Data Set [2], which is described in the evaluation section. We also show how E2-NVM compare with prior hardware-based bit flip-reduction techniques that we describe in more detail in the evaluation section [31, 73, 81, 107, 154]. The figure shows the performance of E2-NVM while varying the frequency, ψ, of the underlying wear-leveling swapping of memory segments (this experiment utilizes an emulation of the memory controller as such parameters cannot be manipulated on typical real memory controllers.) When the frequency ψ is 1, then the swap is performed for every write operation, which

[2]Amazon Access Samples. UCI Machine Learning Repository, 2011

means that E2-NVM judicious memory segment choice is swapped. This leads to not observing the benefits of the software-level approach. (A low ψ value is also not good for hardware-based methods because it means that more bit flips are incurred due to frequent swapping.) As we increase ψ to normal levels, E2-NVM shows that software-level approaches are capable of significant improvement.

4.2 E2-NVM Design

4.2.1 Variational Autoencoder (VAE)

The representation ability of dimensionality reduction techniques like PCA is limited at scale. Many applications from power systems to health care to storage systems and database systems use deep learning as a feasible alternative that can provide low dimensional learned features with lower preprocessing and training delay while preserving intrinsic local structure in data [9, 55, 58, 148]. In this work, we choose Variational Autoencoders as our model of choice for several reasons: high representation ability, fast training, and the ability to jointly perform clustering and model training [58]. VAE can be considered as a generative variant of Auto Encoder (AE), as it enforces the latent code of AE to follow a predefined distribution [111].

Our VAE consists of an encoder, a decoder, and a loss function. The encoder part is a deep neural network with weights and biases θ, which takes a memory segment x as input and encodes it into a latent (hidden) representation space z, which has much less features/dimensions than x. The main responsibility of the encoder is to learn an efficient compression of the data x into this lower-dimensional space z. The decoder part is another deep neural network with weights and biases ϕ, which takes the latent representation z produced by the encoder and outputs the parameters to the probability distribution of the data. Finally, since the loss function of the VAE is the negative log-likelihood with a regulizer, the total loss is $\sum_{i=1}^{N} l_i$ for the total data points, where the loss function l_i for a single data point x_i is calculated as bellow:

$$l_i(\theta, \phi) = -\mathbb{E}_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + \mathbb{KL}(q_\theta(z|x_i)||p(z))$$

where $q_\theta(z|x_i)$ and $p_\phi(x_i|z)$ denote the encoder's and decoder's distributions, respectively. The first term is the reconstruction loss (expected negative log-likelihood of the ith data point). The expectation is taken with respect to the encoder's distribution over the representations. This is the reason that makes the decoder learn to reconstruct the data. The second term is the Kullback-Leibler divergence between the encoder's distribution $q_\theta(z \mid x)q$ and $p(z)$, which measures how p is close to q. It is worth noting that, in the VAEs, p is specified as a standard normal distribution with mean 0 and variance 1 [8].

4.2.2 E2-NVM Design

At the core of E2-NVM is a VAE, an unsupervised ML model, which in combination with K-means clustering maps memory locations into clusters based on the similarity of their content. The VAE-based clustering model is trained/re-trained based on the bit-wise contents of the available memory locations/segments on PCM, and learns the existing data distribution in memory. E2-NVM integrates the VAE's reconstruction

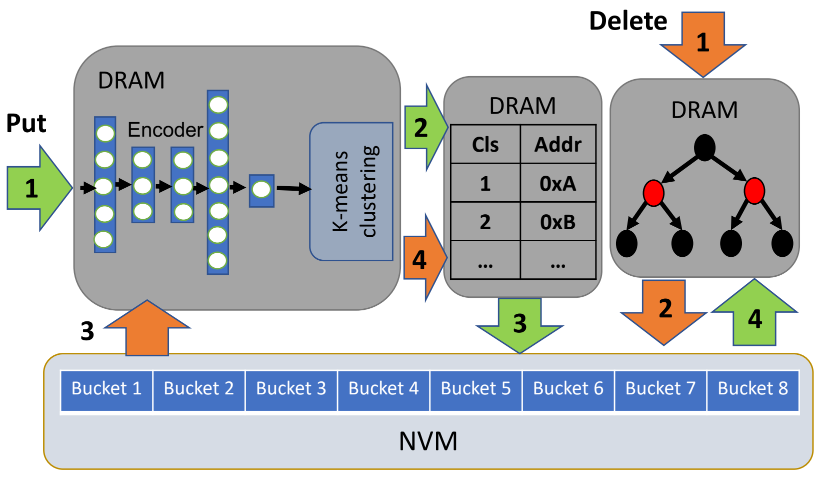


Figure 4.2: Memory layout of a E2-NVM-based key-value store.

loss and the K-means clustering loss to jointly train cluster label assignment and learning of suitable features for clustering. In other words, E2-NVM jointly optimizes (a) reduction of high-dimensional input to low-dimensional latent space representation and (b) clustering in the low-dimensional latent space. E2-NVM can scale to much larger input sizes compared to prior work since clustering is performed on the latent space. Moreover, latent space generated by the jointly optimized VAE-based clustering model has better representation ability compared to traditional techniques such as PCA.

Each memory location is encoded as a vector of bits, each of which is used as a feature/dimension. The entire data zone/memory pool can be encoded as a 2D tensor (that is, an array of vectors) of shape (n, m), where the first axis (n) represents the samples (old data) and the second axis (m) represents the features. The model takes the input of size m and downsizes it to a low-dimensional latent space (e.g., size 10) and feeds it to the K-means clustering model. E2-NVM also employs a padding strategy (§4.3) to accommodate data items of arbitrary size.

4.2.3 E2-NVM Integration and Operations

We present a persistent key/value store that is built on a hybrid DRAM-NVM memory using E2-NVM (Figure 4.2.)

4.2.3.1 System Model

The components of the system are:

E2-NVM. This component includes the VAE and k-means clustering models. This component is trained using the data on NVM to enable creating k clusters based on hamming distance similarity. After training, only the encoder part of the VAE and the K-means clustering models are needed. In operation, this component is used to predict the cluster of a given data object.

Cluster-to-memory dynamic address pool. This component is responsible for tracking the free memory segments that belong to each cluster. It is implemented as a simple mapping data structure where the key is the cluster id and the value is a list of all the free memory addresses that belong to the cluster. This map is initially populated in the initialization phase when E2-NVM is trained. The population is performed by running the prediction algorithm using E2-NVM on the free memory addresses. During normal operation, the mapping structure is mutated in response to operations. A PUT operation would result in removing the chosen address from the pool and the DELETE operation results in adding (recycling) the deleted memory address back to the pool. It is worth noting that, when a write request comes to the system and its corresponding cluster is found, E2-NVM returns the first available address in the cluster. Although searching within clusters can result in finding the perfect matches, we do not do that since all the similar addresses in terms of their hamming distance are grouped into one cluster. So, we just take the first available address in the cluster knowing that it will have a very similar content to the request. Our evaluation shows that this design decision allows good bit flip reduction even without having to search for the ideal address within a cluster.

The storage overhead of the dynamic address pool is proportional to the number of memory segments (or buckets) in the memory pool. Given a specific memory pool size, for large memory segments, the size of the table does not grow significantly. For small values, however, the number of addresses that needs to be stored per memory segment can grow substantially. To limit the table size, we have two options: (1) Setting a fixed number of entries in the table, so the size of the table cannot not grow to more than a specific maximum threshold. (2) Depending on the size of the memory pool, choosing the size of the memory segments in a way that while limiting the size of the table within a specific threshold, we achieve the expected energy performance. To find the most efficient segment size for a specific size memory pool, we have conducted some tests, which are discussed in section 4.3.1.4.

Data index. This is the component that corresponds to the application. In this case, it is a key-value store. This indexing component maps each key to the memory location that contains its value in NVM. Although we use red-black tree in our example, depending on the application, it can be replaced by any indexing data structure such as skip lists, hash indexing, LSM trees, etc. Our tree indexing component maps each key to the memory location that contains its value in NVM.

NVM storage. This component represents the NVM persistent storage used to store data for the key-value store. The storage is divided into fixed-sized memory segments. It contains free memory addresses that can be used for future write requests and is mapped by the dynamic address pool. Also, it contains allocated memory segments that are mapped by the data index structure. In our abstraction we divide NVM storage into fixed sized buckets. An implementation may utilize different instances of this architecture with buckets of different sizes. This allows utilizing memory with different bucket sizes, similar to how memory allocation techniques may maintain different allocations with multiple fixed sizes.

4.2.3.2 Data operations

Key-value operations can be divided into operations that write data (PUT and UPDATE), delete data (DELETE), and read data (GET and SCAN). Each one of these three types is implemented via an algorithm that would potentially mutate the state of the four components in Figure 4.2. Performing these operations needs to be performed carefully to ensure that the state of each component is consistent—e.g., after deleting a data object, the memory segment that corresponds to it must be recycled and added back to the dynamic address pool so it can be used by future write operations.

Write operation algorithm. Algorithm 4 illustrates the pseudo-code of the write operation under the E2-NVM scheme (shown by the green arrows in Figure 4.2). To locate the appropriate memory location for an incoming key/value write, i.e., new

Algorithm 4: Write operation

```
// D′ and D: old and new (key,value)
// DAP: Cluster to Memory Dynamic Address Pool
Write (D: (key,value)){
 1: E = E2-NVM-model.predict(D); //predict the entry
 2: A = DAP.get(E);//get the address
 3: D′= Read(A); //old (key,value)
 4: DAP.remove(A) //remove the address from DAP
 5: for each bit in {D} and {D′}
 6:   if they differ, update memory bit
 7: RB-Tree.put(D, A) //update the index}
```

Algorithm 5: DELETE operation

```
// D′: old key
// DAP: Dynamic Address Pool
// RB-Tree:  Red-Black Tree
Delete (D′: key){
 1: A = RB-Tree.get(D′); //get the address
 2: Reset-Flag-Bit(A);//delete
 3: E = E2-NVM-model.predict(Read(A)); //predict the
    entry
 4: DAP.update(A:address, E:entry);//add the address
    back to DAP}
```

PUT or UPDATE operations, the input data is first processed by E2-NVM's encoder (step 1). In E2-NVM, the input data is transformed to the latent space representation using the VAE encoder. Then, using K-means clustering, the low-dimensional representation is mapped to the cluster that is closest to the value-to-be-written in terms of hamming distance (line 1 of the algorithm). Then, the chosen cluster is passed to the dynamic address pool (step 2 in the figure and line 2 in the algorithm). The dynamic address pool chooses a memory segment from the cluster and assigns it to the write operation. After removing the address from DAP (line 4), the write operation is applied to the chosen address in NVM (step 3 and lines 5 and 6). Finally, an index entry is added to the application's data index (step 4 and line 7).

It is worth noting that for serving an incoming write request, E2-NVM chooses the first available memory segment within the cluster. Although searching within clusters can result in finding the perfect matches, we do not do that since all the similar addresses in terms of their hamming distance are grouped into one cluster. So, we just take the first available address in the cluster knowing that it will have a very similar content to the request. Our evaluation shows that this design decision allows good bit flip reduction even without having to search for the ideal address within a cluster.

DELETE operation algorithm. Algorithm 5 illustrates the delete operation (shown by the red arrows in Figure 4.2). To perform a DELETE operation, the key is first sent to the tree to find the item's location in the key-value store in NVM (step 1, line1). The associated entry is then deleted from the key-value store by resetting the associated flag bit (step 2, line 2). We recycle the recently freed address back to the free list; the address (and its content) is sent through the encoder and then K-means clustering to find a suitable cluster (step 3, line 3). Then, the memory address is added to the corresponding cluster in the dynamic address pool so it can be used for future operations (step 4, line4).

Read operation algorithm. The GET operation is sent through the data indexing tree to find its location in NVM storage.

SCAN operation algorithm. Similar to GET operations, a SCAN operation is directed to the indexing data structure to find the range of key-value pairs to be read and returned to the user.

Resilience and concurrency considerations. For ease of exposition, the algorithms presented above did not factor in complications due to resilience and concurrency concerns. Specifically, a machine crash in the middle of a write or delete algorithm might lead to an inconsistent state. For example, if a crash occurs after step 2 in the write algorithm (a memory address is chosen and deleted from the dynamic address pool) but before the rest of the steps (so that the data index was not updated to include the newly-written data object) would create an inconsistency. This can be problematic because the chosen memory segment is now neither in the dynamic address pool nor the data index, rendering it wasted. To overcome this type of anomalies, standard recovery and logging methods—such as redo and undo logging—that ensure the atomicity of the operations are applied (i.e., an operation would be either performed to completion or not at all.) Similarly, performing operations concurrently might lead to unexpected anomalies. To overcome these anomalies, standard concurrency control mechanisms—

such as locking—can be applied. Both resilience/recovery and concurrency control are parts of many key-value stores, which enables the integration with E2-NVM to utilize these methods instead of building another layer of recovery/concurrency control.

4.2.4 E2-NVM benefits

VAE-based clustering employed by E2-NVM offers several benefits over prior memory-aware techniques: lower training time, higher accuracy, and better scalability. To demonstrate the benefits of E2-NVM, in Figure 4.3, we compare E2-NVM with two methods: K-means clustering and K-means clustering in combination with PCA as employed by a recent memory-aware clustering method PNW [81]. In the comparison, we use the MNIST dataset. We measure two key metrics of performance, latency and number of bit flips. A system with lower latency (preprocessing and training) and lower number of bit flips has better performance.

We train the clustering models on NVIDIA Tesla K80 GPU. We group the incoming data of different sizes into 20 clusters. The size of the input data is $70,000$ and the number of features or the latent space representation is varied from 32 to 16384. Figure 4.3 shows that when the number of features—here the number of bits—increases beyond a couple of thousands, pre-processing latency of K-means clustering is extremely high. The results show that using K-means clustering alone (without PCA) is not a feasible choice for item sizes of kilobytes or more since the pre-processing time goes up exponentially with the increase in the number of features (the number of bits). For large data sizes, the second mode (PCA + K-means) is the only viable choice under

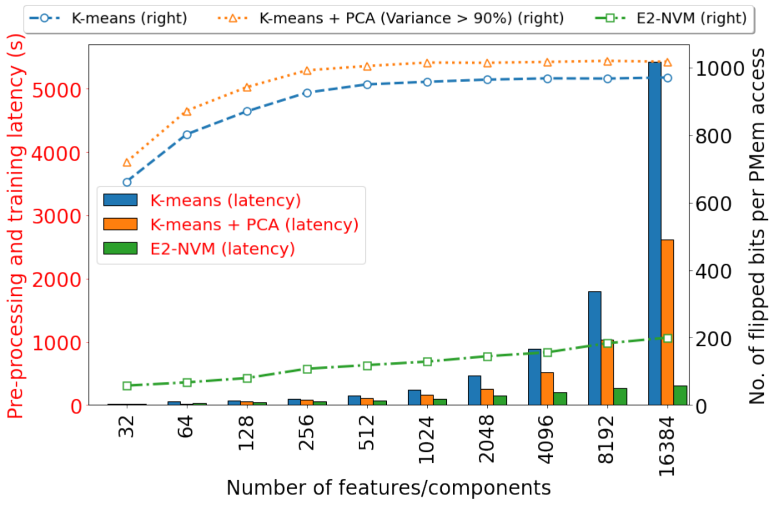


Figure 4.3: Comparison of E2-NVM with PNW (K-means alone and K-means+PCA) in terms of the number of bit flips and latency.

PNW due to latency constraints. However, due to loss of information arising from dimensionality reduction with PCA, clustering efficiency is affected. Hence, the number of bit flips increases when moving from K-means to K-means+PCA. For example, in Figure 4.3, the number of principal components (features) for each size of the data is enough to represent more than 90% of the variance in the data. However, this increases the number of bit flips.

Figure 4.3 also presents the results for the VAE-based clustering model of E2-NVM. This model needs significantly less time than PNW for training, which includes both VAE and K-means clustering training. This is because the VAE can decrease the dimensionality from tens of thousands to hundreds very fast while also minimizing data loss. E2-NVM minimizes both latency and the number of bit flips significantly. This enables E2-NVM to support memory-awareness for data with larger sizes in the order of kilo to megabytes.

4.3 The padding strategy

4.3.1 Padding Strategies

Overview. In machine learning methods, the input size needs to be defined when the model is created. To support inputs of various sizes, we propose a padding strategy. In this strategy, first, our deep learning model is trained on fixed-sized features, for instance w, like the other deep learning models. When the model is ready, it can serve input data with size w. But, for input data with a different size, p, which is smaller than w, we need to transform it into an item with size w. To do that, we use padding, which is adding q=w-p bits to the item to fit the input layer of our DL model. The ultimate goal is to pad the inputs of smaller sizes in such a way that it ends up in the cluster with the most similar items, which means minimizing the number of bit flips. It is worth noting that the padded part with size q is not a part of the main data and is added to the data just for clustering purposes. Only the actual data of size p is written to the target memory and the remaining q bits are ignored (not written to storage). Figure 4.4 shows an example of E2-NVM's different padding strategies on an input data d1:[0,0,0,1].

Challenges. The padding strategy decides (a) where the padded bits should be placed relative to the original input data, and (b) what bits to use in padding. The padding strategy influences the accuracy of detection and consequently finding a suitable

Table 4.1: An example of a PCM with 12 memory segments.

Cluster	Index	Content
0	0	[0, 0, 1, 1, 1, 1, 0, 1]
	1	[0, 0, 1, 0, 1, 1, 0, 0]
	2	[0, 0, 1, 1, 1, 1, 0, 0]
	3	[0, 0, 1, 1, 1, 0, 0, 0]
1	4	[1, 0, 0, 0, 1, 0, 1, 1]
	5	[0, 0, 0, 0, 1, 0, 1, 1]
	6	[0, 0, 0, 0, 1, 1, 1, 1]
	7	[0, 0, 0, 0, 1, 0, 1, 0]
2	8	[1, 0, 1, 1, 0, 0, 0, 0]
	9	[0, 1, 1, 1, 0, 0, 1, 0]
	10	[1, 1, 1, 1, 0, 0, 0, 0]
	11	[1, 1, 0, 1, 0, 0, 0, 0]

memory segment. However, there is a trade-off between the complexity of the padding strategy and the effect on the accuracy of the deep learning model. In the remainder of this section, we discuss a number of strategies (see Figure 4.4) in the spectrum of this trade-off.

Padding location. The location of the padded bits can be one of the followings: (i) padding bits before the input data (beginning-padding), (ii) padding bits after the input data (end-padding), or (iii) padding bits are split, with one half before the input data and the other half after the input data (middle-padding).

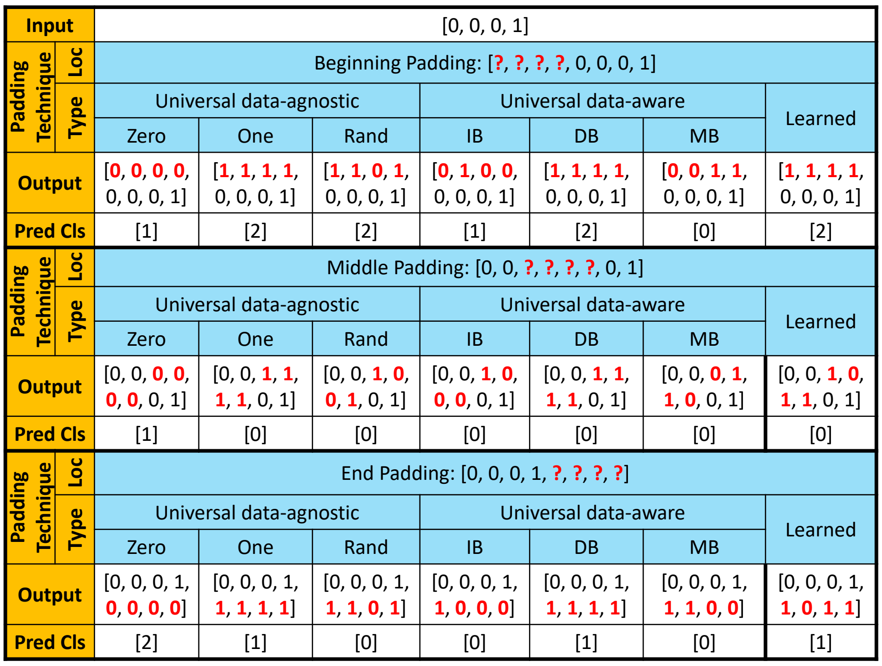
4.3.1.1 Padding Type: Universal data-agnostic padding

The simplest padding strategy—in terms of complexity and overhead—is universal data-agnostic padding, where the padding bits are generated given a simple fixed rule independent of data. Specifically, there are three types that we experiment with: (i) zero padding where all the padded bits are 0 bits, (ii) one padding where all the padded bits are 1 bits, and (iii) random padding where the padded bits are chosen randomly.

To illustrate how our padding strategy works, consider a storage system that is using a PCM as its persistent memory with a capacity of 12 equal sized memory segments, managed by a free-list, which we refer to as the dynamic-address-pool (Table 4.1). In this example E2-NVM groups these memory locations into 3 different clusters. Now, suppose that we have a new data object d1: [0,0,0,1] that we want to insert into the PCM using universal data-agnostic with beginning-padding location ([?,?,?,?,0,0,0,1]). Using one padding, 8-4=4 bits are added before the input data (in the question marks) to get [$\underline{1}$,$\underline{1}$,$\underline{1}$,$\underline{1}$,0,0,0,1]. Then, when it is sent to the model, cluster 2 is predicted to be the best cluster and one memory location is selected within this cluster (Figure 4.4). Note that the leftmost 4 bits are not written to the memory since they were added to the data to be able to utilize the deep learning model. Although this padding type is very simple, our DL model might not be utilized to its full potential since the padded bits added to the input data might not reflect the existing distribution in the memory content.

4.3.1.2 Padding Type: Universal data-aware padding

In this scheme, we aim to find a padding strategy that chooses the padding bits based on the content of the data objects in NVM or the content of the input items. The intuition behind data-aware padding strategies is that if the padding pattern matches



Input	[0, 0, 0, 1]						
Padding Technique – Loc	Beginning Padding: [**?**, **?**, **?**, **?**, 0, 0, 0, 1]						
Padding Technique – Type	Universal data-agnostic			Universal data-aware			Learned
	Zero	One	Rand	IB	DB	MB	
Output	[**0**, **0**, **0**, **0**, 0, 0, 0, 1]	[**1**, **1**, **1**, **1**, 0, 0, 0, 1]	[**1**, **1**, **0**, **1**, 0, 0, 0, 1]	[**0**, **1**, **0**, **0**, 0, 0, 0, 1]	[**1**, **1**, **1**, **1**, 0, 0, 0, 1]	[**0**, **0**, **1**, **1**, 0, 0, 0, 1]	[**1**, **1**, **1**, **1**, 0, 0, 0, 1]
Pred Cls	[1]	[2]	[2]	[1]	[2]	[0]	[2]
Padding Technique – Loc	Middle Padding: [0, 0, **?**, **?**, **?**, **?**, 0, 1]						
Padding Technique – Type	Universal data-agnostic			Universal data-aware			Learned
	Zero	One	Rand	IB	DB	MB	
Output	[0, 0, **0**, **0**, **0**, **0**, 0, 1]	[0, 0, **1**, **1**, **1**, **1**, 0, 1]	[0, 0, **1**, **0**, **0**, **1**, 0, 1]	[0, 0, **1**, **0**, **0**, **0**, 0, 1]	[0, 0, **1**, **1**, **1**, **1**, 0, 1]	[0, 0, **0**, **1**, **1**, **0**, 0, 1]	[0, 0, **1**, **0**, **1**, **1**, 0, 1]
Pred Cls	[1]	[0]	[0]	[0]	[0]	[0]	[0]
Padding Technique – Loc	End Padding: [0, 0, 0, 1, **?**, **?**, **?**, **?**]						
Padding Technique – Type	Universal data-agnostic			Universal data-aware			Learned
	Zero	One	Rand	IB	DB	MB	
Output	[0, 0, 0, 1, **0**, **0**, **0**, **0**]	[0, 0, 0, 1, **1**, **1**, **1**, **1**]	[0, 0, 0, 1, **1**, **1**, **0**, **1**]	[0, 0, 0, 1, **1**, **0**, **0**, **0**]	[0, 0, 0, 1, **1**, **1**, **1**, **1**]	[0, 0, 0, 1, **1**, **1**, **0**, **0**]	[0, 0, 0, 1, **1**, **0**, **1**, **1**]
Pred Cls	[2]	[1]	[0]	[0]	[1]	[0]	[1]

Figure 4.4: An example of applying E2-NVM's different padding strategies on an input data d1:[0,0,0,1] based on the memory pool defined in Table 4.1.

patterns in data, the VAE can perform more efficient clustering. In other word, we want to choose the content of padded parts in a way that maximizes the probability of mapping input items to right clusters and minimizes the number of bit flips. For the universal data-aware padding strategy, we propose three different padding schemes: (i) input-based padding (IB), (ii) dataset-based padding (DB), and (iii) memory-based padding (MB).

In input-based padding, the content of the padded part for an incoming data item is determined based on the distribution of ones and zeros in the input item. For instance, for the system we described in this section, if it receives d1:[0,0,0,1], the padded part will contain 1s and 0s with probability of 0.25 and 0.75, respectively, which is the same as the probability of 1's and 0's in d1. So, for the middle padding, the output format would be [0,0,$\underline{1}$,$\underline{0}$,$\underline{0}$,$\underline{0}$,0,1], which results in cluster dataset-based padding uses the distribution of 1's and 0's in all the items it has received so far. For memory-based padding, we choose the probabilities based on the content of the existing memory locations on NVM that are going to be replaced by the new incoming items.

Although universal data-aware padding strategies generally result in better clustering decisions compared to data-agnostic padding, there are still some items that are directed to the wrong clusters. This is because the distribution of 0's and 1's might not reflect the best padding to be performed on the input item. The next strategy aims to overcome this challenge. In the last padding strategy, we have designed a padding strategy to not only generate the padded content based on the incoming new data items but also be aware of the content of the whole old memory, which is going to be written by the incoming items.

4.3.1.3 Padding Type: Learned padding

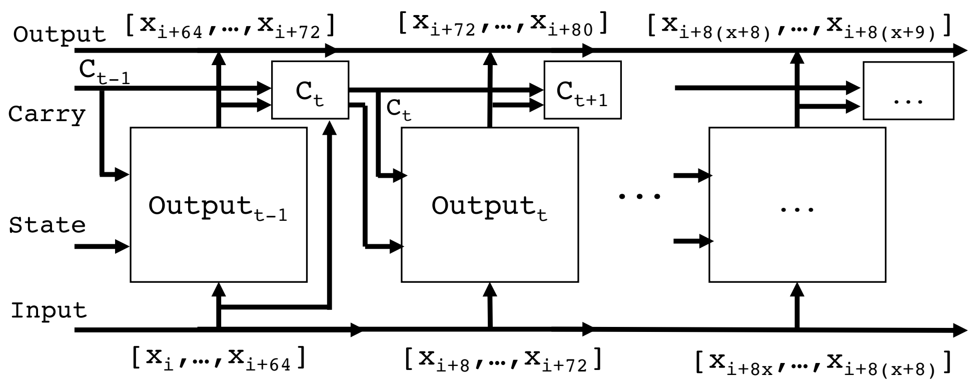
The main shortcoming of both data-aware and data-agnostic padding techniques is that they do not always generate a padding that is tailored specifically to both the input item as well as prior data, some of which that are used in training the DL model. We overcome this by designing a learning-based strategy, where we train a model that enables us to predict the best padding strategy for a given input item which takes into account all the existing data. The learned padding model takes an incoming data item of arbitrary size as input and generates the padding bits as output. The intuition behind this padding strategy is that the padding model is trying to predict the best padding strategy that will place the incoming item in the right cluster.

To this aim, we utilize an Long Short-Term Memory (LSTM) model to generate more meaningful padded data, so E2-NVM can predict similar memory locations with higher accuracy, which in turn improves the energy consumption of the system. Figure 4.5 shows the architecture of our proposed LSTM whose underlying algorithm was developed by Hochreiter and Schmidhuber in 1997 [64]. This figure shows that the model has a hidden state where it represents the state of the current timestamp, which is known as short term memory. In addition to that LSTM also has a cell state represented by C_{t-1} and C_t for previous and current timestamp respectively (known as long term memory).

As it is shown in Figure 4.5, E2-NVM utilizes an LSTM with a sliding window strategy which takes as input 64 bits and predicts 8 bits in a single step. The window is slid by 8 bits after each prediction to generate the required number of padded bits. To illustrate the goal of our learned padding approach, consider the same storage system in Table 4.1. For the sake of simplicity, suppose that, in this example, our LSTM model takes input size of 7 bits and predicts 1 bit at a time. Now, let's further assume that our system receives [1,0,1,1,0,0,0],[0,1,1,1,0,0,1], [1,1,1,1,0,0,0], [1,0,0,0,1,0,1], [0,0,0,0,1,0,1], and [0,0,0,0,1,1,1]. Since the memory contents are 8 bits and E2-NVM works on input sizes of 8 bits, we feed them to our following simple LSTM model to make them 8 bits:

```
# define model
model = Sequential()
model.add(LSTM(10, input_shape=(1,7)))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mse', optimizer='adam')
model.fit(X, y, epochs=20, shuffle=False, verbose=0)
# make predictions
yhat = model.predict(X, verbose=0)
```

As it is clear in Table 4.1, the best case scenario happens when their eighths bits are predicted as 0, 0, 0, 1, 1, and 1, which is congruent with the results of the LSTM model: [0.006], [0.024], [-0.027], [1.056], [0.869], and [1.038]. As a result, all the mentioned items are assigned to their correct clusters, which in turn improves system's energy efficiency.



Output
Carry
State
Input
C_{t-1}
C_t
C_{t+1}
Output_{t-1}
Output_t
...
$[x_{i+64},\dots,x_{i+72}]$
$[x_{i+72},\dots,x_{i+80}]$
$[x_{i+8(x+8)},\dots,x_{i+8(x+9)}]$
$[x_i,\dots,x_{i+64}]$
$[x_{i+8},\dots,x_{i+72}]$
$[x_{i+8x},\dots,x_{i+8(x+8)}]$

Figure 4.5: The anatomy of the LSTM model used in E2-NVM.

4.3.1.4 Additional design considerations

As we discussed before, E2-NVM is built on DRAM, and might need to index a large portion of NVM, which usually comes at much bigger sizes than DRAM. This means that if the memory segments that we use in the system are small (for example 1KB) and we want to index a NVM of size 1 TB, we will need to have E2-NVM index around 1 billion memory segments, which takes a lot of space in DRAM. To solve this problem, in E2-NVM, instead of indexing the whole NVM device at the beginning, a dynamic incremental approach can be adopted, which starts by indexing a portion of the memory, and as time progresses, more addresses that were not initially mapped can be added incrementally to DAP. Also, free memory locations are dynamically added back (recycled) to the free lists of DAP after DELETE operations.

Figure 4.6 illustrates the amount of memory that E2-NVM uses for indexing different number of memory segments for the PubMed data set [47]. The results show that although indexing a smaller number of memory segments takes less space, it also

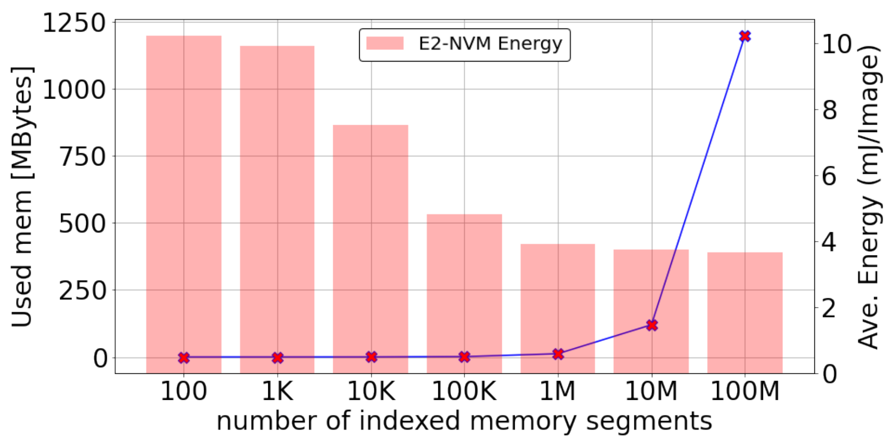


Figure 4.6: E2-NVM's memory and energy consumption for indexing different numbers of memory segments.

means that not only do we need to retrain the DL model more frequently, but we will also have fewer choices to find the most similar location for incoming writes, which results in increasing the number of bit flips and energy consumption. From the results presented in Figure 4.6, we observe that by having 100K to 1M memory segments, we can have the best of both worlds. While we do not see any tangible performance degradation in terms the energy consumption, indexing this number of memory segments will consume a couple of MBs in memory. Furthermore, by indexing more than 1M memory segments, we do not see any significant improvements in energy consumption.

To overcome the overhead incurred due to small key-value pairs, batching can be applied so that small writes are grouped together to form larger writes to memory segments. This way, E2-NVM needs to map the free memory locations based on the batch size rather than the key-value pair size, which leads to reducing E2-NVM footprint. It is worth noting that we do not make assumptions about word/byte-alignment. However, we want to note that padding bytes are not stored. So, it does not impact the storage of the data. Rather, padding is only performed for the prediction part of E2-NVM. The reason for this is that we want to allow data with variable size to be applied to the model.

As a result, selecting very small memory segments (less than 1KB) when the memory pool size is very big (64 GB or more) is not a good design decision because although the NVM's write energy consumption is very low, the system suffers from scalability issues (dynamic address pool takes a big space in DRAM). Likewise, although selecting big memory segments takes up small space in DRAM, it is not energy efficient because we will have fewer choices to find the most similar locations for the incoming writes, which results in increasing the number of bit flips and energy consumption. Therefore, this test can help us find the most efficient memory segment size. This depends on (1) the size of the area in NVM that we want to index (memory pool size), (2) the size of DRAM that we use, and (3) the amount of energy consumption that the system can tolerate.

Another important design consideration is to guarantee that the dynamic address pool will never run out of free memory addresses and E2-NVM will always be able to serve the incoming requests. To this end, we set a minimum threshold to number of addresses in each cluster and will trigger the re-training process in the background when one of the clusters reaches to the threshold. After the new model is ready, we switch to the new model. It is worth noting that we do not need to train the model in E2-NVM as long as the performance is not affected substantially (please see Section 4.4.3 for more

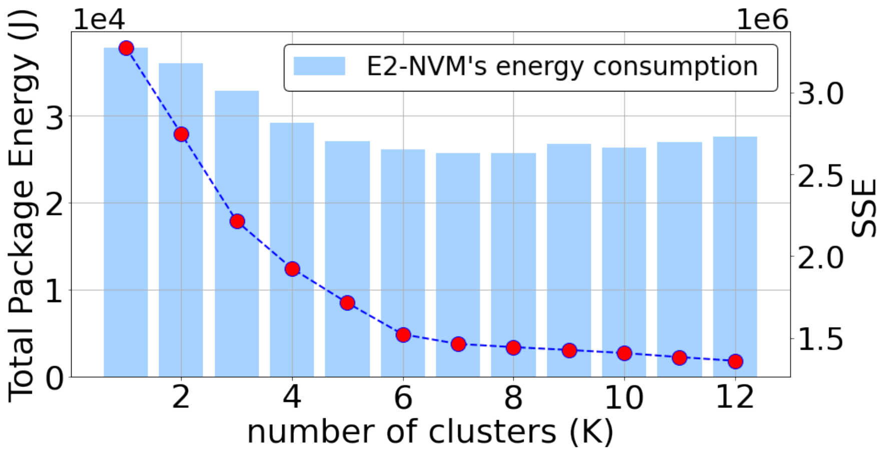
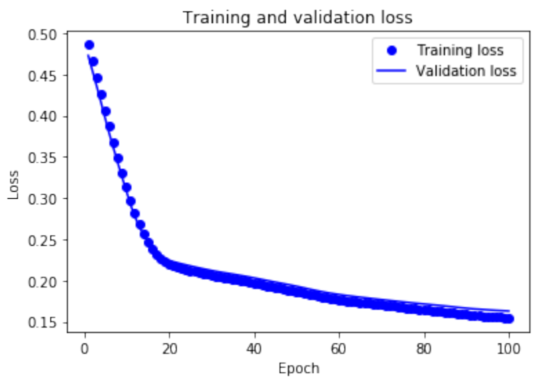


Figure 4.7: Sum of Square Error graph versus E2-NVM's energy consumption to find the optimal K.

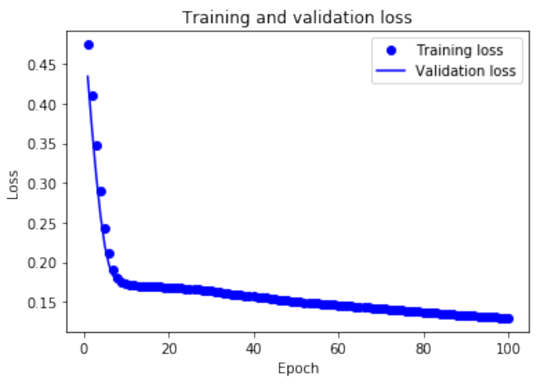
details).

Determining the Number of Clusters. A decision that needs to be made before E2-NVM starts training its DL model is to determine the number of clusters (K). There are a number of factors that need to be taken into consideration: (1) As the number of clusters increases, the memory contents inside clusters become more similar to each other, and it saves more bit flips, which means consuming less energy (see Section 4.4). (2) Although having more clusters saves more energy when writing on NVM, it also means that E2-NVM needs more time to finish training its DL model, which means increasing latency and energy consumption of the system.

Figure 4.7 shows the results of the test that we conducted to determine the optimal value for K. For energy-consumption, our experiments show a"valley" trend (the blue bar graph in the figure): the energy-consumption is relatively high at both the lowest and highest Ks, and is relatively low at intermediate Ks. This is because, in low Ks, the



(a) AMZ



(b) CCTV

Figure 4.8: The training and validation loss for feature extraction during training for different datasets.

number of clusters is too small and the contents inside each cluster are not very similar to each other, which leads to high energy consumption in NVM. Conversely, increasing the number of clusters beyond a certain level increases the total energy consumption while providing only a limited return on bit flip reduction, also leading to energy inefficiency in DRAM and CPU. This “energy-valley” trend indicates that choosing the best value for K in an energy-aware fashion requires a good trade-off between the amount of energy that the system requires for writing on NVM and the energy that E2-NVM needs to train its DL model.

In this work, we use the “elbow method” [81, 109], which is one of the most common techniques that is used to find the optimal value for K in K-means clustering.

The elbow method is expressed as the following Sum of Squared Error (SSE) [142]:

$$SSE(X, \Pi) = \sum_{i=1}^{K} \sum_{x_j \in C_i} \|x_j - m_i\|_2^2 \tag{4.1}$$

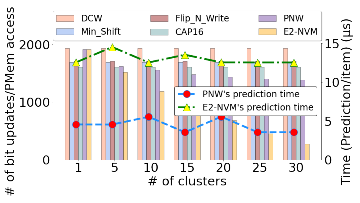
where $\|.\|_2$ denotes the Euclidean (L2) norm, $m_i = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j$ is the centroid of cluster C_i where the cardinality is $|C_i|$, $\Pi = \{C_1, C_2, ..., C_K\}$, and $X = \{x_1, ..., x_i, ..., x_N\}$ (N is the feature vector).

To determine the optimal number of clusters, we identify a sharp decrease known as the "elbow" or "knee", which suggests the optimal value for K [81, 109, 142]. Figure. 4.7 shows an example of choosing the optimal K by seeing the significant decrease in the SSE graph, which is in K = 6 (the data set is CIFAR-10). As we can see in this figure, this value is also a good estimation for the energy consumption of E2-NVM for different values of K.

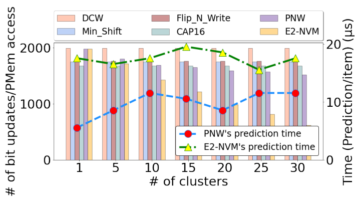
4.4 Experiments

4.4.1 Methodology

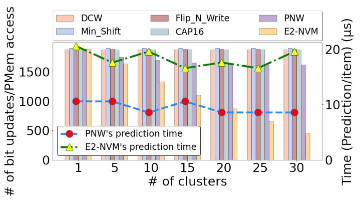
The experiments are executed on (1) an Intel Core i7 processor running at 4.7 GHz with 4 cores, each of which has 1MB L2 Cache and 12MB L3 Cache using 32GB of Intel® Optane™ Memory Series 3D Xpoint™ and 16 GB of DRAM, and (2) an Intel Xeon® @2.6GHz with 16 cores, an Nvidia Tesla K20m GPU with 5GB memory using 32 GB DDR4 main memory and a 256 GB SSD hard drive. The machine has 32GB DDR3 main memory, 128GB of Intel® Optane™ Persistent Memory 200 Series (PMEM



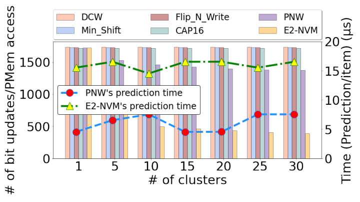
(a) Amazon Samples



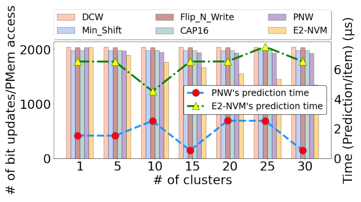
(b) 3D Road Net



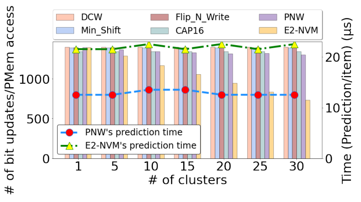
(c) Sherbrooke



(d) Traffic Surveillance



(e) CIFAR-10



(f) PubMed

Figure 4.9: The average number of actual bit updates per PMem's cache line access granularity as well as the latency of prediction per item in E2-NVM for the real-world textual and multimedia datasets.

Module), and a 256 GB SSD hard drive. We use the latter machine to get the results in Figures 4.6, 4.10, 4.16 and 4.18. Although the amount of energy might be different for different setups, both machines showed similar behavior in terms of the relationship between the number of flipped bits and energy consumption. We utilize thread-safe methods in E2-NVM. This is the case for the data structures that we utilize to maintain address pools and mapping (contention in other parts such as the VAE would not lead to concurrency anomalies since operations on them are read-only).

There are two methods to measure energy consumption: (1) Power Monitors, which use hardware tools to measure the actual power of the device. Despite being very precise, they are extremely difficult to set up. (2) Energy Profilers, which are vastly used by researchers, do not require any special hardware, or power sensors, and estimate the power cost of different hardware using estimation models [3]. In this thesis, we use an energy profiler named Perf, which is a performance analysis tool and a part of the Intel's RAPL interface [56, 87]. we measured the energy and power consumption of the memory (both DRAM and PMEM), while running our tests using the perf [39] tool:

```
$ perf stat -a -r 5 -e power/energy-cores/, \
  power/energy-ram/, power/energy-gpu/, \
  power/energy-pkg/, power/energy-psys/ ./test
```

This tool provides the collection of energy measurements from various components of a computer system such as: cores, Intel's GPUs, package (all the core and un-core components), DRAM, total power consumption of a node, and so on. Also, the sampling

[3]http://luiscruz.github.io/2021/07/20/measuring-energy.html

rate in our tests is 1000 samples per second.

4.4.2 Overview and setup

In this section, we evaluate our proposed method in terms of bit flips, energy efficiency, and performance. We perform experiments on a real Intel Optane memory device to measure energy efficiency and performance overhead. Also, we perform experiments with emulated Optane memory to measure bit flip reduction (which cannot be measured using the real device.)

We compare E2-NVM with two main groups of solutions: 1) persistent K/V stores that use specialized data structures to deal with the limitations of NVMs [79, 81, 106, 117, 166]. These methods generally focus on reducing write amplification. 2) hardware-based bit flip optimization methods that use the RBW technique to alleviate the limitations of NVMs [31,73,107,154]. Unlike the previous category, this group focuses directly on decreasing the number of bit flips.

4.4.2.1 Workloads

We have used various types of real-world and synthetic workloads in our evaluations.

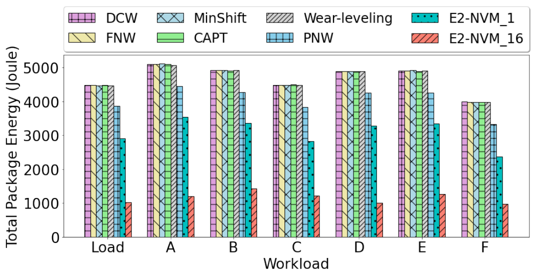
Synthetic workloads. In the first synthetic workload, we run the YCSB benchmark [34] to evaluate E2-NVM. We load a 10-GB data set into the database as the "old data" in the load phase. Then, we run the workloads one by one, and compare the results. The six core workloads that we used in our tests have different read-write ratios and access patterns: Workload-A has 50% reads and 50% updates, Workload-B has

95% reads and 5% updates, and Workload-C has 100% reads; the keys are chosen from a Zipfian distribution, and the updates operate on already-existing keys. Workload-D involves 95% reads and 5% inserting new keys (temporally weighted distribution). Workload-E involves 95% range queries and 5% inserting new keys (Zipfian distribution), while Workload-F has 50% read-modify-writes and 50% reads.

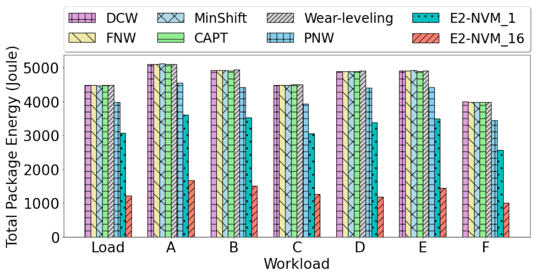
Real-world workloads (Numerical). We use Amazon Access Samples [47] that contain 30K access log entries. We also use the 3D Road Network Data Set [57, 86] that contains 434874 entries of road networks information of North Jutland, Denmark. Finally, we use the collections of the DocWord database named "PubMed", which consists of 730 million entries [47].

Real-world workloads (Images). We use two of the most widely used datasets for machine learning and computer vision research, MNIST and CIFAR-10 datasets [93]. The former is a dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images. The latter dataset is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images, which are grouped into 10 different classes.

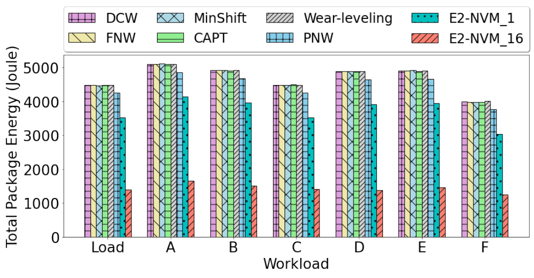
Real-world workloads (Videos). In the last set of tests, we use two video datasets: 1) The Sherbrooke video dataset [74], which is more than two-minute-long video (with resolution 800x600), which was filmed at the Sherbrooke/Amherst intersection in Montreal by a camera located a couple of meters above the ground, and 2) Traffic Surveillance video [11], which is collected from seven intersections in the Danish cities of Aalborg and Viborg. In this test, we use two sequences of RGB cameras called CCTV1



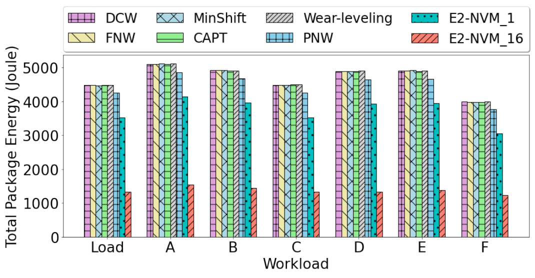
1KB



2KB



4KB



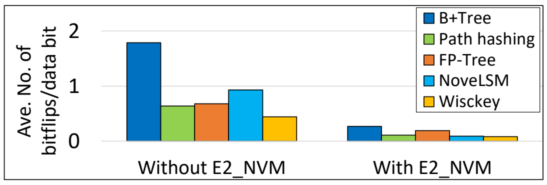
16KB

Figure 4.10: The average amount of energy consumed per PMem's cache line access granularity when memory segment size changes for YCSB workloads.

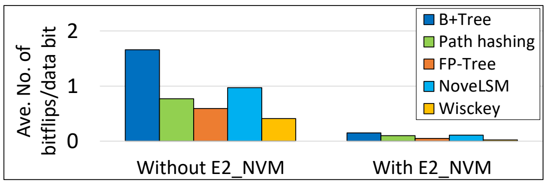
and CCTV2. For the first dataset, we stored the first 30 seconds of this video as the old data and then we replaced it with the rest of the video as the new data. We did the same with the second datasets with one difference and that is storing first one minute of the video as the old data and using the rest of the video as the new data.

4.4.3 Evaluation Results

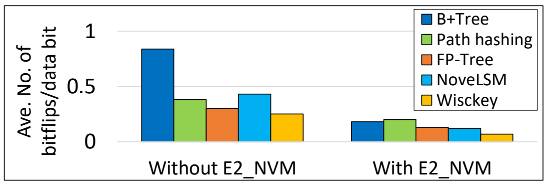
Deep learning model characteristics. In the first set of experiments, Figure 4.8 shows the E2-NVM's learning curves, which is a metric to show how well the DL model is "generalizing" the learned patterns. We have evaluated our DL model



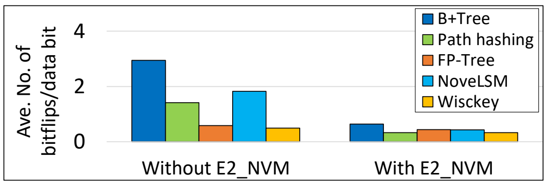
(a)Amazon Samples



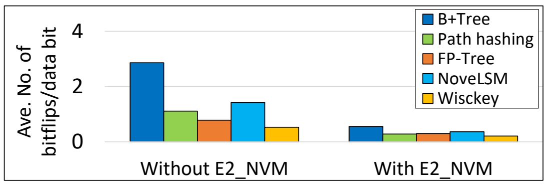
(b)3D Road Network



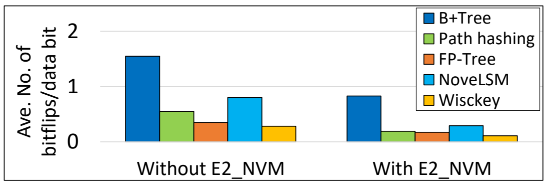
(c)Sherbrooke



(d)Log-normal distribution



(e)Zipf data distribution



PubMed

Figure 4.11: The impact of augmenting E2-NVM to data stores in terms of the average number of bit updates per writing 1 data bit.

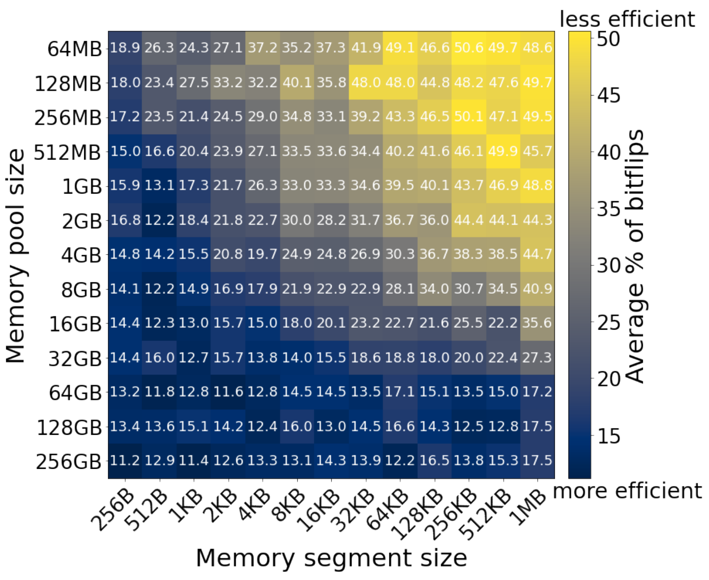


Figure 4.12: E2-NVM's performance in terms of the total memory energy for different memory segment and memory pool sizes.

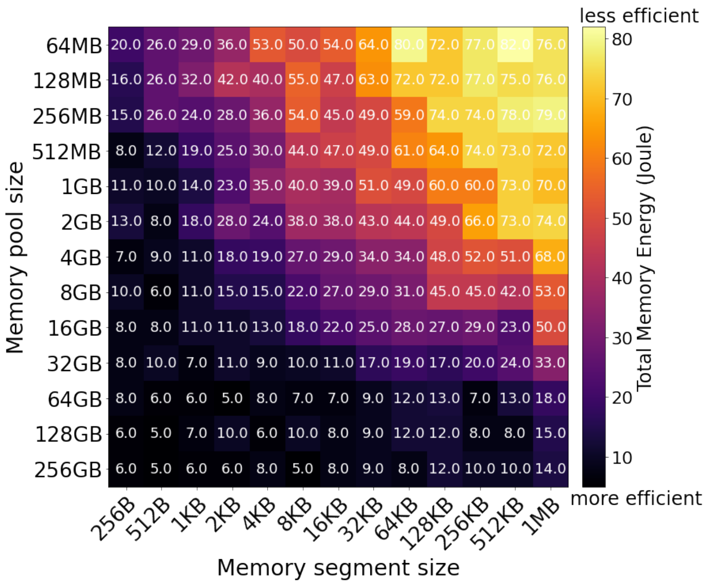


Figure 4.13: E2-NVM's performance in terms of the average updated bits ratio for different memory segment and memory pool sizes.

on the training dataset and on a hold-out validation dataset, which is completely isolated from the training dataset. In this figure, we have used the loss metric whereby smaller values indicate better learning and a value of 0 indicates that the training dataset was learned perfectly, and no mistakes were made. As we can see from the results, our deep learning model converges very quickly, which shows the ability of E2-NVM to learn and generalize the existing patterns in memory segments.

Comparison with RBW and memory-aware methods. Figure 4.9 shows the results for comparing E2-NVM with the following RBW methods: DCW [154], MinShift [107], FNW (Flip 'n Write) [31], and Captopril [73]. We also compare with a clustering-based memory-aware solution, PNW [81]. We vary the number of clusters, k, ranging from k=1 to k=30 for different datasets (note that k=1 is a baseline that aims to show the performance without getting the benefits of the clustering-based methods. Also, the only methods that are impacted by increasing the number of clusters are the clustering-based methods, PNW and E2-NVM.). We have compared the performance of E2-NVM to others in terms of the number of bits updated/written per PMem access. In this figure, when we pick k=1, the result for E2-NVM, PNW, and DCW are the same since they all behave similarly with no clustering. Figure 4.9 shows that our method enhances the previous ML-based method (PNW [81]) by up to 3.2x and the baseline bit flip optimized methods by up to 4.23x.

We have also calculated the model prediction latency for both PNW and E2-NVM. In our method, the delay is higher compared to PNW because E2-NVM performs two predictions, one by the DL model and the other by the clustering ML model.

This highlights a performance-accuracy trade-off (where performance is the prediction overhead and accuracy is the accuracy of finding a similar memory segment) where E2-NVM favors accuracy but introduces more overhead compared to PNW.

In the next experiment, we calculate the average amount of energy that is consumed per PMem's cache line access granularity when the memory segment size changes for YCSB workloads on a real Optane memory device. Figure 4.10 shows that by choosing smaller segment size, our method can save more energy since the model can predict memory segments with higher accuracy, which leads to minimizing the number of bit flips. This figure also shows another factor that affects the energy consumption of our method—the number of clusters. When there are more clusters, the similarity among the items within a cluster increases, which leads to fewer number of bit flips and less energy consumption.

Augmenting E2-NVM to existing NVM data structures. In the next experiment, we tested the performance of the implemented methods, such as B+-Tree [29], Wisckey [105], Path Hashing [166], FP-Tree [117], and NoveLSM [79]), in terms of the number of updated bits in two different ways: before plugging to E2-NVM, and after plugging to E2-NVM. The results are shown in Figure 4.11. When not plugged to E2-NVM, B+-Tree has the worst performance because, in a regular B+-Tree [29], the items in leaf nodes need to be sorted, which increases the number of movements and bit flips. After we tested the performance of the methods in terms of bit flips, we plugged them to E2-NVM and repeated the same tests. After plugging each method to E2-NVM, their performance improves by up to 91% by preventing a lot of unnecessary bits from being

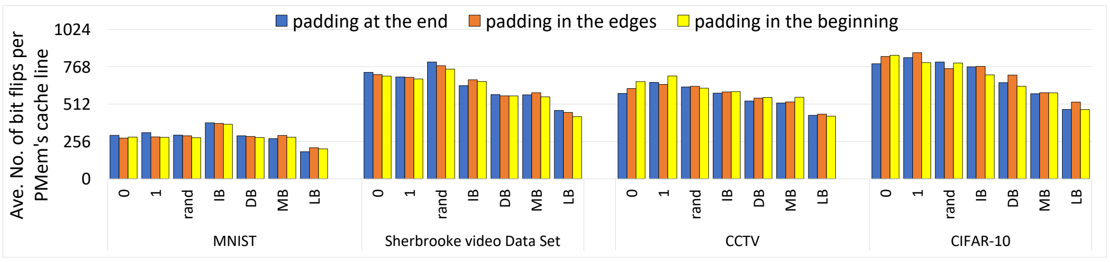


Figure 4.14: The average number of bit flips per word after applying different padding strategies.

flipped. These tests validate the ability to plug existing data indexing structures to E2-NVM to reduce bit flipping.

The effect of memory pool size and memory segment size on the total memory energy. Figures 4.12 and 4.13 illustrate the overall performance of E2-NVM in terms of energy efficiency and updated bits for different memory segment and pool size choices for the mixture of all the real workloads in this work. For the overall energy efficiency shown in Figure 4.13, we observe that the E2-NVM's power consumption increases as the ratio of the memory segment size to the memory pool size increases, which also aligns with results of the average bit flips ratio shown in Figure 4.12. Based on this observation, we conclude that the smaller we choose the size of the memory segments compared to the size of the data zone (pool size), the more tree nodes are available for E2-NVM, which also results in fewer bit flips and less energy consumption.

Padding evaluations. In multimedia applications, such as CCTV and video storage, the size of data objects is typically fixed depending on the chosen resolution.

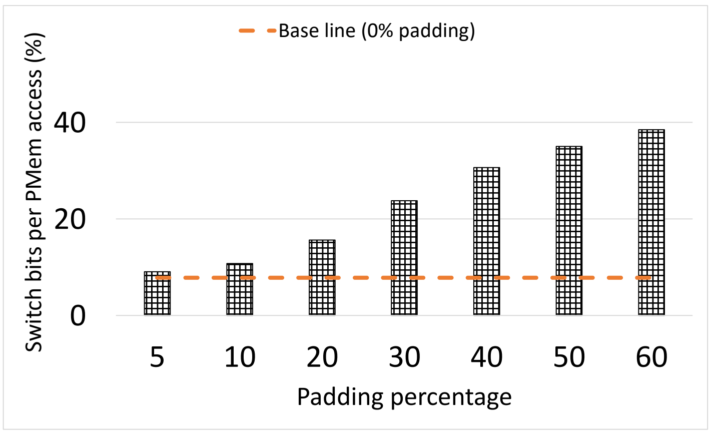


Figure 4.15: The number of bit flips when different percentages of the video frame size is padded by the learned padding scheme for the CCTV dataset.

This enables us to use E2-NVM directly as we can fix the input size of the model to match the data object size. However, to generalize our solution to handle data items of arbitrary sizes using the same model, we introduced padding methods. Figure 4.14 shows the performance of our proposed padding strategies. In this experiment, there are three padding positions. In the first one, the original data is at the beginning and the padding part is concatenated to the rightmost side of the data (padding at the end). In the second position, the data stays in the middle of the frame and padded bits are split into equal parts and appended to both sides of the data (padding in the edges). Finally, in the third position, the data is placed at the rightmost side of the frame and the padding part is attached to the left side of the data (padding in the beginning).

For each test, first, the DL model is trained on the training dataset (80% of the complete dataset). To generate the test set, we crop one-third of the data items in the positions that we mentioned and then pad it. So, for each dataset, we test 7 padding strategies and across 3 different padding positions.

In Figure 4.14, we make several observations about padding strategies, i.e., input-based (IB), dataset-based (DB), memory-based (MB), learned-based(LB), zero (0), and one (1) paddings. First, data-aware padding schemes outperforms the data-agnostic schemes in terms of reducing the number of bit flips. Second, learned padding scheme has the highest performance in terms of the average number of bit flips. Third, padding in the edges has higher variance compared to padding at the beginning or end. To summarize, the performance of padding improves with increasing complexity of models, from data-agnostic schemes to data-aware schemes and the learned padding.

The accuracy of mapping a data item to the most appropriate cluster drops when a large fraction of bits are padded bits. In Figure 4.15, we evaluate the impact of padding by measuring the number of bit flips under different percentages of data frames are being padded. In this experiment, we first trained the DL model on the CCTV's training dataset. Then, we cut off different percentages of the frames of the testing dataset and feed them to E2-NVM. Since the size of the input is smaller than what the model has been trained on, E2-NVM uses the learned padding strategy to fit the data items into the original frame sizes. In other words, in this experiment, we generate the missing parts of the frames in the testing dataset, so we can compare it with the baseline where the frames of the testing dataset were intact (the 0% padding line in the figure).

Figure 4.15 shows that when there is no padding (0% padding), the number of bit flips is minimum, the best performance. The reason is that when the input data frames are the same size as the training data frames, E2-NVM can find the best cluster for the incoming data items and minimize the number of bit flips. For padded data, we measure the number of bit flips per word based on the written bits only (padded bits are not written to NVM.) When the percentage of padding increases, it becomes more difficult for the system to predict the best padded part and find the most similar clusters. However, when the percentage of padding is low (10%), there is minimal loss in performance. This motivates combining the padding strategy with batching—as described in the padding section—to reduce the percentage of padding.

Memory overhead analysis of E2-NVM in terms of energy consumption. To see how training and re-training of our deep learning model affect the energy consumption of the system, we conducted an experiment to track the behavior of our method as time goes by. In this test, we first created a transactional object store with a total pool size of 8GB and element/segment size of 64KB in the Intel Optane DC PMM (in the App Direct Mode) and seeded it with data items from ImageNet [41], which contains over 14 million labeled images. Also, we re-sized the images to fit the size of the elements (64KB) in the pool. Figure 4.16 shows the result in terms of the amount of total package energy that is consumed after a specific time. In this test, we did the following steps, which are marked on Figure 4.16 from 1 to 4: (1) We trained our DL model on the memory contents of the pool before we start accepting the write requests (stage 1 in the figure). (2) After our DL model is trained, we started writing new data items on the existing memory contents of the pool from the ImageNet data set. In this step, we overwrote the pool with the items from the same data set (combination of both existing and new images) five times. (3) In step 3, we retrained the model to reflect the changes in the data distribution. It is worth noting that, for the sake of this experiment, we stopped writing on NVM until the retraining process finishes although, in E2-NVM, the writing process does not have to be stopped because the retraining is done in the background lazily. (4) In the last step, we resumed writing new data on NVM. We overwrote the pool with ImageNet data for four more times. As we saw this test, the total energy that E2-NVM saves up by writing similar contents will make up its energy overhead caused by training, retraining and prediction.

Another important observation that we can make from this test is that we can make a precise estimation of the cost of the re-training process in terms of energy consumption and latency by looking at the the cost of the training process of the same model at the initialization phase (stage 1 in Figure 4.16). The reason behind is that the training/re-training cost depends on the size of the data set (memory pool), dimensionality (memory segment), the complexity of the deep learning model that we use, and the hardware that we run our model on, which all are the same for both training and re-training processes.

Adaptability of E2-NVM to the dynamic changes. To analyze the behavior of E2-NVM in different scenarios and its adaptability to the system's changes, we conduct the last experiment in a dynamic environment when the content of the memory and the incoming workload changes over the course of time. You can see the results in Figure 4.17. In this test, we use three image data sets from Tensorflow, i.e., MNIST, Fashion-MNIST, and CIFAR-10. For this test, we did the following scenarios:

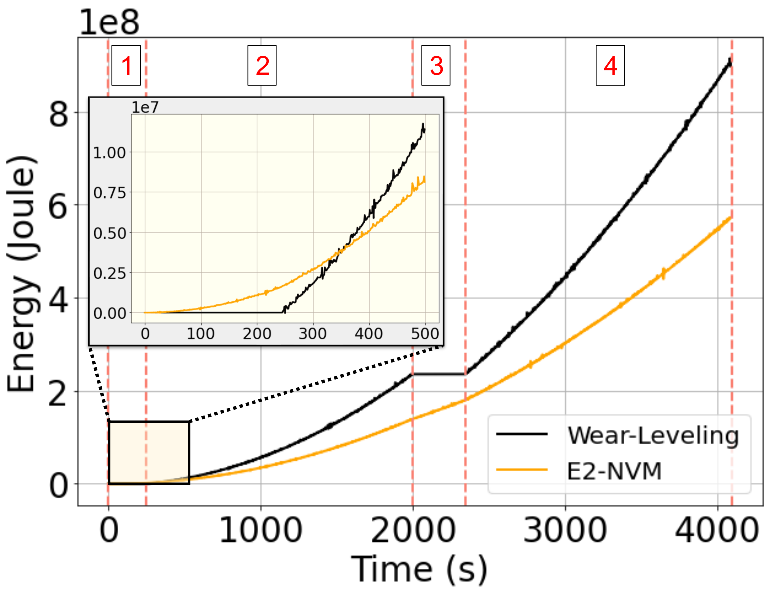


Figure 4.16: Tracking the package energy sampled every 1ms for E2-NVM when it goes through periodic training, re-training and writing phases compared to the wear-leveling technique over time.

Scenario 1: we seeded the data zone in NVM with a completely random content and then trained the E2-NVM model on this content. After training the model and creating the cluster-to-memory dynamic address pool, we started streaming 54K images (Figure 4.17 part I) from MNIST as the new data into the system to overwrite the old data followed by deleting half of the items to make the system dynamic. As we see from the results (Fig 4.17 part I-a), because the content that the model is trained on and the incoming writes are different, the number of bit flips fluctuates a lot, but as time goes by, fluctuations become narrower toward the bottom (Figure 4.17 part I-b) due to the fact that E2-NVM updates the cluster contents by recycling the deleted items and bringing them back to the cycle.

Scenario 2: we trained the model one more time with the current content and updated the dynamic address pool. Then, we continued streaming 27K images from the same data set (MNIST) as the new data into the system to overwrite the old data. Figure 4.17 shows that fluctuations and the average number of bit flips decrease. Even at the end of this stage, where the old data is almost completely replaced with the new one, we still do not see significant changes in performance (Fig 4.17 part II).

Scenario 3: Starting from this point, we send a mixture of 27K items from two different data sets, i.e., Fashion-MNIST and MNIST, at the ratio of 1 to 2. Figure 4.17 part III shows that the performance is affected immediately (the number of updated bits increases) since two-third of the incoming data are entirely from the content that the model has never seen before and results in a larger hamming distance.

Scenario 4 (part IV in the figure): we sent 30K images from the third data set, i.e., CIFAR-10. The number of updated bits fluctuated more since (1) the old data contains the items from completely different data sets and (2) the model has never seen (been trained) the incoming data.

Scenario 5: In this phase, we continued sending 28K images from CIFAR-10 with one difference: we re-trained our model on the existing content. Figure 4.17 part V-c shows that the results improve very fast since the data set and the content that the model is trained on are from the same type. As a result, we have seen that, depending on the application and the workload, we do not always have to re-train the model rapidly, and we can use the same model for a certain amount of time before it needs to be re-trained. This allows us to do the re-training in the background lazily and update the model periodically while the current model is serving the requests. To this aim, E2-NVM needs to know when to start re-training the model before the old one becomes inefficient, i.e. the system's performance decreases in terms of energy consumption. This is of great importance because we might not want to give all the available resources to the model since the system needs to serve the requests without any problem while the new model is being re-trained. We performed additional tests to evaluate the costs for re-training a new model (Figure 4.18). This experiment is performed on ImageNet [41].

In Figure 4.18, we measured the re-training time of the model per epoch for a different number of memory segments. We run these tests on the second hardware setup. Based on the results, the model needs more time and energy to be re-trained as the number of memory segments increases. This gives us an idea of setting the load factor

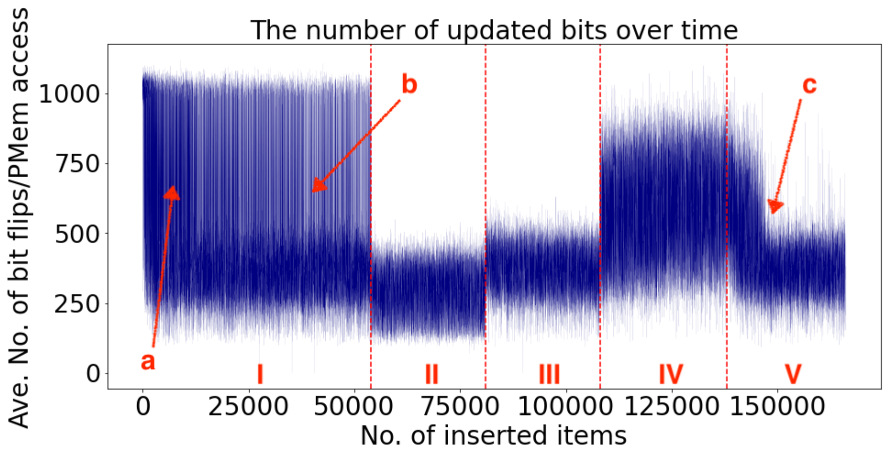


Figure 4.17: Tracking the performance of E2-NVM by changing the memory content and incoming writes over the course of time.

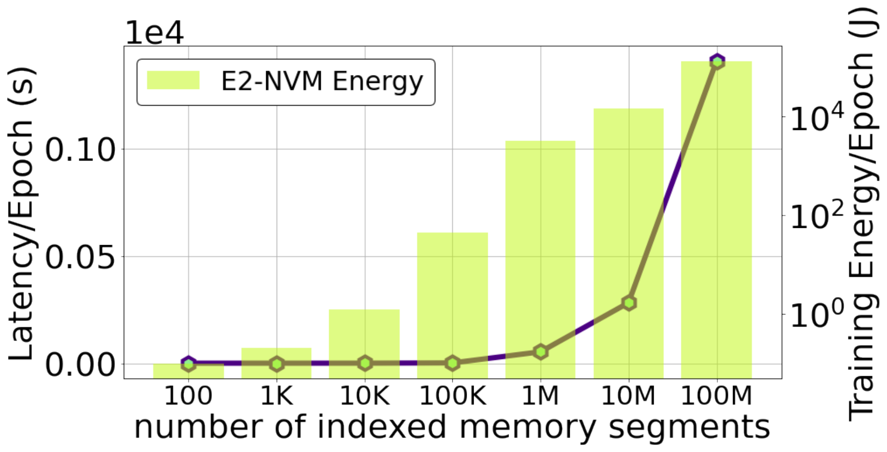


Figure 4.18: E2-NVM's training costs in terms of latency and energy consumption per epoch for indexing different number of memory segments.

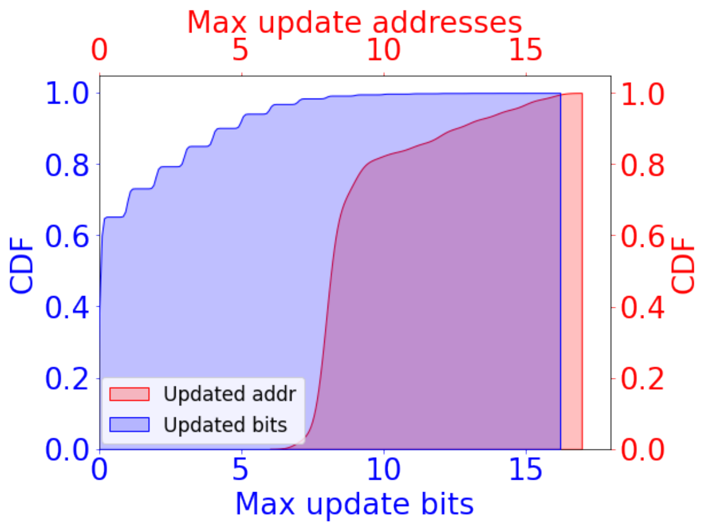


Figure 4.19: The maximum update addresses and wear-leveling as CDFs by applying E2-NVM with k=30 clusters.

so that we have enough time to finish re-training the new model before the old model becomes inefficient. Training the model sooner than this threshold would not result in a noticeable performance improvement since the pattern of the bits in the content is almost the same as the training time. Also, by waiting too long before re-training the model, the system misses the opportunity to improve the performance considerably.

Although decreasing the number of writes is important, wear-leveling is equally important to extend the lifetime of PCM. The reason is that some blocks of the memory device may receive a much higher number of writes than the other blocks, and as a result, wear out sooner [81]. Therefore, to observe the performance of E2-NVM in terms of the distribution of the maximum number of bit flips and the wear-leveling of PCM, we conduct two more tests. In these tests, we run E2-NVM when k = 30 clusters, on the mixture of MNIST and Fashion-MNIST data sets. For this test, we first warm up the data zone with 28K items from the combination of both data sets. Then, we stream 112K writes from the same data sets to the system. During the test, we also perform delete actions to make space for incoming writes (each word in the data zone is updated 4 times on average).

Figure 4.19 shows the maximum number of times the addresses in the data zone are written and the wear-leveling of memory bits as a cumulative distribution function (CDF). This figure illustrates two results: (1) the estimation of the likelihood to observe an address in the data zone of PCM that is written less than or equal to a specific number of times, and (2) the estimation of the likelihood to observe a memory bit in the data zone of PCM that is written less than or equal to a specific number of times. For example, as we can see in Figure 4.19, the estimated likelihood to observe an address in the PCM data zone to be written less than or equal to 10 (P $(X \leq 10)$) is 81% (red color). Similarly, we observe that the estimated likelihood of a memory bit being written less than or equal to 5 times is 85%. This likelihood rises to 98% when a memory bit being written equals to 7 times (blue color). This important observation shows that (1) E2-NVM distributes write activities across the whole PCM chip, and (2) E2-NVM distributes bit flips evenly across the whole data zone of the PCM chip, and as a result, the lifetime of PCM is extended more.

4.5 Conclusion

In E2-NVM, we have explored the use of software-level approaches to improve energy efficiency and write endurance of NVMs. Specifically, a deep learning model is used to map memory locations based on the hamming distance of their content. This mapping is used when new writes arrive to assign them to memory location with similar content. This reduces the number of bit flips, which leads to better write endurance and energy efficiency.

Chapter 5

Hamming Tree: The Case for Energy-Aware Indexing for NVMs

As stated earlier, the existing methods, in which writes are generally updated in place, miss a crucial opportunity to increase energy efficiency and write endurance significantly. This opportunity is to be *memory-aware.* Picking the memory location for a write operation arbitrarily (new data items select an arbitrary location in memory, and updates to data items overwrite the previously-chosen location.) misses the opportunity to judiciously pick a memory location that is similar to the value to be written (in terms of their hamming distance.) When the new value and the value to be overwritten are similar, this means that the number of bit flips is going to be lower. Reducing the number of bit flips increases write endurance and reduces power consumption [68,81–84,138,158].

To enable memory-awareness, the supported data structure needs to perform out-of-place updates. This may introduce some performance overhead when updating data. However, the advantage of enabling memory-awareness outweighs the overhead caused by making the data structure perform out-of-place updates. Later in this chapter, we will show this experimentally with evaluations with a high percentage of updates in the workload.

In this chapter, we introduce a novel approach distinct from the methodologies discussed earlier in this thesis. This new work involves an indexing-based data structure, implemented at the software level in the data storage layer, called *Hamming Tree* [83, 85]. Hamming Tree is designed for NVM-based data management systems to increase energy efficiency and write endurance by enabling existing indexing structures to select a memory location for their writes that would minimize bit flips. To this end, Hamming Tree maps all available (free) memory locations according to their hamming distance. When a write w is invoked, Hamming Tree is traversed to find a free memory location with content similar to the value of w. Hamming Tree's design innovation is based on employing a recursive method of comparing the density of 0/1 bits for each free memory segment.

In our evaluation, we augment Hamming Tree with four existing indexing structures: B+-tree, LSM-based persistent K/V store called NoveLSM [79], a cache optimized NVM index called FP-Tree [117], and a write-friendly hashing scheme [166]. We performed real evaluations on an Optane memory device that show that Hamming Tree can reduce energy consumption by up to 67.8%.

5.1 System Model

The system model consists of hardware and software components. We assume the use of existing hardware components and do not require any special hardware. In hardware, we consider a hybrid DRAM-NVM architecture, where both devices are placed on the memory bus. The NVM device, in addition to the memory segments, contains a *memory controller* that intercepts all operations to NVM. The memory controller may utilize a wear leveling solution that swaps memory segments periodically. The details of wear leveling methods are typically proprietary. However, prior work has indicated that wear leveling approaches perform a memory segment swap every k write operations (we provide some details of these approaches in the related work section.) Typically, the value of k is in the order of 10s of writes [68]. Some memory controllers also adopt bit flipping reduction technologies such as ones based on RBW [31, 42, 59]. We implement our Hamming Tree solution in the software-level. Hamming Tree is a storage layer that sits between software applications (such as data stores) and the hardware components. Therefore, Hamming Tree can be thought of as a library that can be used by existing data storage systems.

5.1.1 Software-Level Bit Flip Reduction

To see how bit flip reduction affects the system's energy consumption and performance, we have conducted a simple experiment on a real Optane memory device. We used the Persistent Memory Development Kit (PMDK) [1], formerly known as NVML.

[1]Persistent Memory Development Kit https://pmem.io/pmdk/.

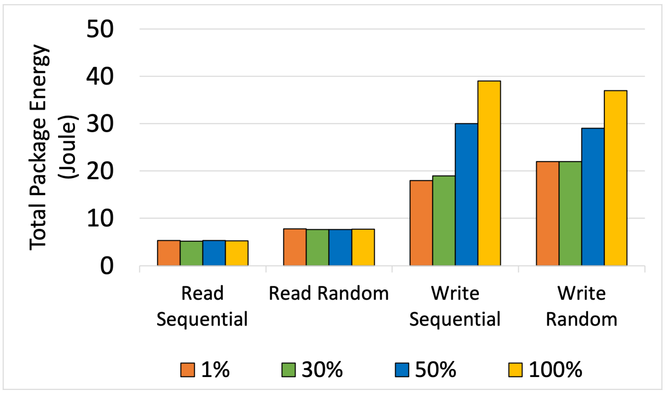


Figure 5.1: Total memory energy consumption on a real Intel Optane memory device for read and write operations with different percentages of hamming distance.

In this test, first, we allocate a contiguous region of N Optane blocks of 256B. During each "round" of the experiment, we first initialize all the blocks with random data, and then update the blocks with new data with content that is x% different than the data that is already in the block (hamming distance). We use PMDK's transactions to persist writes. We measure the energy consumption of the socket for each round. Fig. 5.1 shows that overwriting similar content, which needs less bit flipping, consumes less energy. This shows that reducing bit flips has the potential of better energy efficiency. Furthermore, it shows that this can be achieved by solutions in the software-level, despite the interference of the memory controller and other software/hardware components (we discuss this further in the rest of the section.) Fig. 5.2 shows that write latency also improves when bit flips are reduced. This can offset some of the overhead introduced by software-level methods to pick memory segments that would reduce bit flips, such as Hamming Tree.

One potential problem that might arise when using a software-level method to

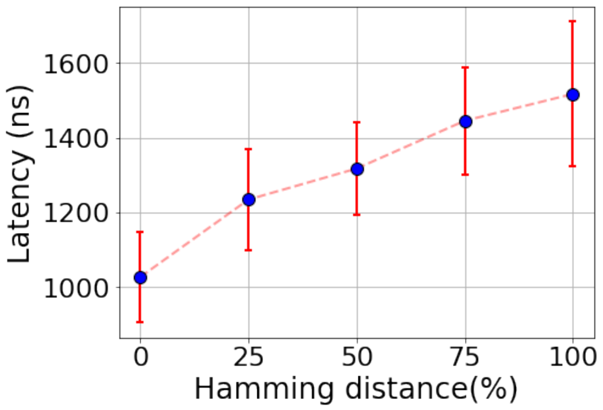


Figure 5.2: The write latency in a real Intel Optane memory device for different percentages of hamming distance.

control where a new write is applied in the NVM device is that the memory controller's wear leveling method might interfere with the process. Specifically, the wear leveling algorithm might swap the destination memory segment before the write operation is applied to it. However, swapping in wear leveling methods—as we describe in more details later—is only applied once every ψ writes, where ψ is typically in the order of tens of writes [68, 80]. Therefore, swapping only affects the memory location choice of Hamming Tree once every ψ writes. We show in our evaluations that Hamming Tree achieves significant improvement over other methods even with a small number ψ.

The potential of reducing bit flips using software-level solutions overcomes two challenges that faced hardware-based solutions: The first is that to be deployed on hardware, algorithms need to be small and simplistic—in terms of computation power and memory—to fit in the memory controller. The second limit is that developing hardware-based methods is not accessible to researchers and practitioners. This is

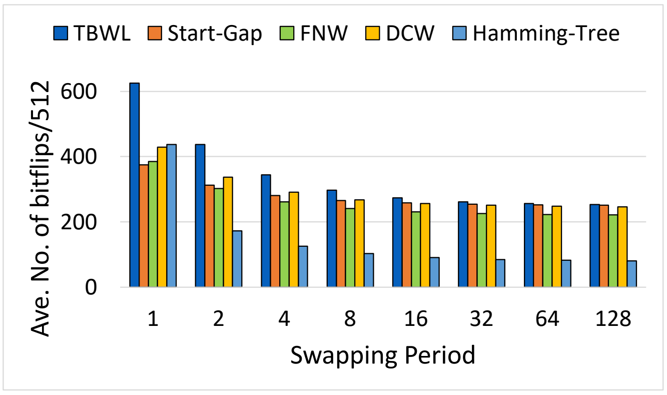


Figure 5.3: The average number of bit updates for different wear-leveling techniques when the swapping period changes. (TBWL: Table-based Wear-Leveling [164], Start-Gap [124], FNW: Flip-n-Write [31], DCW: Data Comparison Write [42])

evident by how most storage solutions for wear leveling and bit flip reduction are proprietary and require manufacturing new hardware to implement a new solution.

Fig. 5.3 shows our results comparing software-level Hamming Tree with hardware-level wear-leveling and bit flip reduction techniques (We list the names and references of the compared functions in the figure's caption and discuss some of them in more detail in the related work section.). In this test, we used Amazon Access Samples Data Set [47], which is described in the evaluation section. The figure shows the performance of Hamming Tree while varying the frequency, ψ, of the underlying wear-leveling swapping of memory segments (this experiment utilizes an emulation of the memory controller as such parameters cannot be manipulated on real memory controllers.) When the frequency ψ is 1, then the swap is performed for every write operation, which means that Hamming Tree's judicious memory segment choice is swapped. This leads to not observing the benefits of the software-level approach. (A low ψ value is also not good

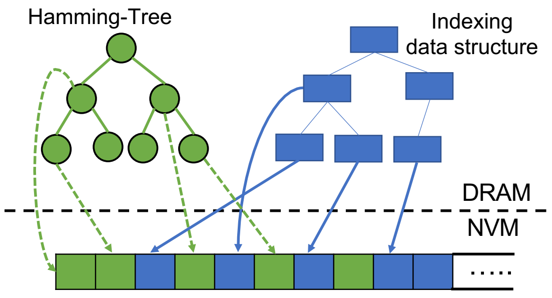


Figure 5.4: The storage and memory layout of Hamming Tree.

for hardware-based methods because it means that more bit flips are incurred due to frequent swapping.) However, as we increase ψ to normal levels, Hamming Tree shows that software-level approaches are capable of significant improvement beyond what is achievable by hardware-level methods.

It is worth noting that although we provided our results on Optane, which is one type of PCM, Hamming Tree is applicable to other phase change material-based technologies, such as phase-change random access memory (PRAM) and Resistive RAM (RRAM), which can benefit from bit flip reduction. Since Hamming Tree's main focus is to improve the energy consumption of the system, our proposed method can be especially attractive to the applications that use low-power PCM devices due to relying on energy-harvesting systems or batteries [21], such as the Internet of Things (IoT) and mobile devices, in which conserving power is one of the main concerns [15, 113].

5.2 Hamming Tree Design

In this section, we present the design of Hamming Tree. Our objective is to select memory locations for the incoming writes that minimize the hamming distance, and hence, the number of bit flips. Hamming Tree is a data structure that achieves this objective through organizing free memory locations based on their hamming distance. In a regular system, where updates are applied in place, there exists only one option to write the data and hence the reduction of bit flips with techniques such as FNW [31] is limited. Hamming Tree, on the other hand, determines the best existing free memory location in terms of hamming distance to minimize the number of bit flips. Hamming Tree can be built on any tree-based data structure, such as B-Tree or RB-Tree (in this implementation, we use B-Tree). Then, Hamming Tree can be used as a storage layer for data indexes in applications or data stores. The data indexing structure handles the regular—application-level—indexing of keys and values, and Hamming Tree handles the storage-level mapping of free memory locations for future writes and updates. There are no restrictions on which indexing structures can be used as long as they support key-value operations. Therefore, we can augment a wide-range of indexing structures such as ones based on B-Trees, LSM, hash tables, and others with Hamming Tree. We show how Hamming Tree can be augmented with various solutions throughout the design and evaluation sections.

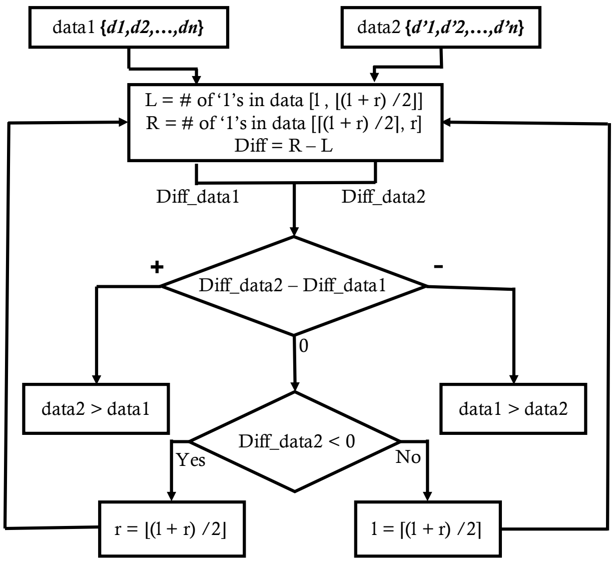


Figure 5.5: The comparison method in Hamming Tree.

5.2.1 Overview and System Model

Hamming Tree is a DRAM-NVM based data structure that can be added to existing data indexing technologies—whether they are designed for NVM or not—to improve their performance in terms of NVM write endurance and energy consumption. For this work, we assume a hybrid memory architecture. In this architecture, both DRAM and NVM are on the same main memory level, which means that managing them can be done under a single physical address space [43]. Fig. 5.4 shows an example of the storage and memory layout of Hamming Tree. The indexing data structure (here shown as a tree data structure) is indexing memory locations that are used (the blue memory locations contain the values of allocated data objects). Hamming Tree indexes free memory locations according to their contents' hamming distance (the green memory locations are free to be used by future writes).

Hamming Tree Mapping Intuition. Whenever there is a need for updating memory in-place, the number of bit flips depends on the hamming distance between the old data—currently in the memory location— and the new data, which is going to overwrite the memory location. Like the other memory-aware techniques, such as PNW [81], Hamming Tree reduces bit flips by avoiding in-place updates and, instead, finding a new memory location for each write that would minimize the hamming distance. By placing the write operation in the right memory location that minimizes the hamming distance between the old and the new data, the number of bit flips can be significantly reduced. Hamming Tree uses an underlying indexing structure such as B-Tree or RB-Tree, to map available (free) memory locations of NVM. The only difference between Hamming Tree and other tree-based indexing data structures is the way it compares the items and orders them; Hamming Tree orders free memory locations according to their hamming distance. For example, in a regular B-Tree, number 1 is ordered before number 8 because 1 is less than 8. However, in Hamming Tree the two numbers are compared and ordered based on the density of 0 and 1 (0/1) bits. Specifically, the intuition behind Hamming Tree is to map memory locations that have the same density of 0/1 bits in different segments together. For instance, the memory locations with a high density of 1's in the left-most segments are considered *smaller* than others; memory locations with a high density of 1's in the center-most segments are considered in the middle; and memory locations with a high density of 1's in the right-most segments are considered *larger* than others. The spectrum between these extremes is mapped according to the density of 0/1 bits in smaller segments of each memory location. With this mapping, Hamming Tree enables a new write to find a memory location that matches its density of 0/1 bits, which means that the selected memory location's bit-wise content is similar to the new write content. This leads to reducing bit flips as the new write is applied to a memory location with similar content.

Hamming-Distance-Based Comparison Function. Fig. 5.5 shows the flowchart of the Hamming Tree comparison method. The comparison of two items d1 and d2 starts by measuring the density of 1's to 0's between the left half and the right half of the data items. A "Diff" function returns the difference of the number of 1 bits in the right segment to the number of 1 bits in the left segment (a positive Diff value represents that the number of 1's in the right segment is higher than the left segment, and vice versa). Diff is applied to both d1 and d2 and they are compared. As shown in Fig. 5.5, the values of "l" and "r" are pointers to the beginning and the end of the higher density part of the data items being compared, and they change as the "Diff" function is called in the next steps. If the size of the memory segment is n bits, the initial values of l and r would be 0 and n-1, respectively, and they change in the next calls depending on the density of 1's in the data item. If Diff(d1) is smaller than Diff(d2), then d2 is considered greater than d1 (this reflects that a higher density of 1's in the right segment translates to being *bigger*). Similarly, if Diff(d1) is greater than Diff(d2), then d1 is considered bigger than d2. If Diff(d1) and Diff(d2) are equivalent, then we recursively measure the density difference in the half with more 1's. This continues until we find a segment of an item that has a higher density compared to the corresponding segment of the other item, and then they are ordered accordingly.

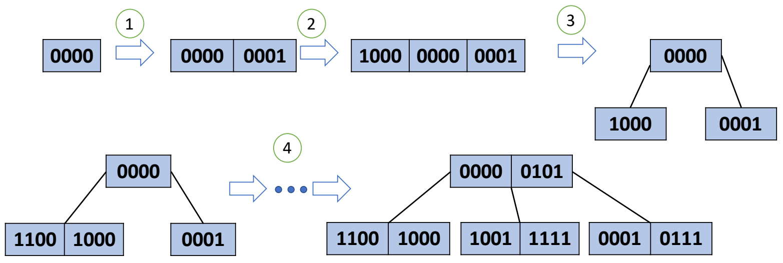


Figure 5.6: An example of how Hamming Tree is formed.

Hamming Tree Evolution. Fig. 5.6 illustrates an example of Hamming Tree which is built on a B-Tree of order m=3. Initially, the tree has item '0000_b' (represented in bits). Then, '0001_b' (with Diff = R-L = 1) is added to the right of '0000_b' with Diff = 0. This is because Diff(0001_b)=1 is larger than Diff(0000_b)=0 (see the flowchart in Fig. 5.5). Then. when the next item '1000_b' is added to Hamming Tree, it is placed on the leftmost position because its Diff is -1, which is less than others (step 2). In this step, all the three items are inserted at the root node because this is a B-Tree of order m=3. Let us now insert item '1100_b'. Since root node is full, in step 3, it will first split into a root and two child nodes, then item '1100_b' will be inserted into the appropriate child node. Based on the flowchart in Fig. 5.5, the reason that item '1100_b' is directed to the left child is that Diff(1100_b)=-2 is less than Diff(0000_b)=0. Likewise, in the left child, because Diff(1100_b)=-2 is less than Diff(1000_b)=-1, it is inserted to the left of item 1000_b. The rest of the items are added one by one in the same way (step 4).

Hamming Tree Density Comparison Arithmetic. In the following, we first define various operations on density magnitude representations (i.e., $<_c$, $>_c$, $=_c$,

$\geq_c$, $\leq_c$)[2], and then show that the density comparison in Hamming Tree, which is based on the density magnitude representations, is totally ordered. We prove total order by showing that the comparison function ensures reflexivity, antisymmetry, transitivity, and trichotomy [38, 131].

Assume two memory segments P1 and P2—b bits long. Our definitions below are based on comparing the density difference (dd) of 1's between the right and left halves of the sub-segment q—$b/2^i$ bits long—of the memory segment P1 in the i^{th} recursive call, $dd(P1, q, i)$, and the density difference of P2 in the i^{th} recursive call, $dd(P2, q, i)$, as binary numbers. Notice that we keep calling dd recursively from $i = 0$ to $i = log_2(b)$ until, in the $i = k^{th}$ call ($0 \leq k < log_2(b)$), $dd(P1, q, k)! = dd(P2, q, k)$.

Definition 1: $P1 \geq_c P2$ if and only if $dd(P1, q, i) \geq dd(P2, q, i)$ for the first i where for all smaller i's, they are equal. For example, consider that $P1 = 0011$ and $P2 = 1100$ (b=4); here, since for $k = 0$, $q = b/2^i = 4/1 = 4$, $dd(P1, 4, 0) = 2 - 0 = 2$, and $dd(P2, 4, 0) = 0 - 2 = -2$, $P1 \geq_c P2$.

Definition 2: $P1 =_c P2$ if and only if $dd(P1, q, i) = dd(P2, q, i)$ for all i in range $[0, log2(b))$. For instance, consider that P1=1011 and $P2 = 0111$; since for $i = 0$, $dd(P1, 4, 0) = 2-1 = 1$ equals to $dd(P2, 4, 0) = 2-1 = 1$, we call the function for $i = 1$, which yields in $dd(P1, 2, 1) = 1 - 1 = 0$ and $dd(P2, 4, 1) = 1 - 1 = 0$. Since, there does not exist any $i = k$ such that $dd(P1, q, i)! = dd(P2, q, i)$, $P1 =_c P2$.

Now Let P be a set of memory segments, and let $\sim$ be a comparison relation

[2]To avoid confusion, we use the subscript c with the operations (for example, $\geq_c$) to denote operations for comparison arithmetic to distinguish them from regular arithmetic.

$\geq_c$, $\leq_c$, or $=_c$ on P.

Lemma 1: (Reflexivity) For all x $\in$ P x$\sim$x.

Proof: Calling $dd(x, q, i)$ on both sides will result in $dd(x, q, i) = dd(x, q, i)$ for all i in range $[0, log_2(b))$, which, based on Definition 2, means $x =_c x$.

Lemma 2: (Antisymmetric) For all $x, y \in P$, if $x \sim y$ and $y \sim x$, then $x =_c y$.

Proof: First consider $x \leq_c y$, then, based on Definition 1, for $i = k1$, $dd(x, q, i) \leq dd(y, q, i)$, which means that for all i in $0 \leq i \leq k1 < log_2(b)$, $dd(x, q, i) = dd(y, q, i)$. Now consider $y \leq_c x$. Then, there exist $k2$ such that $dd(y, q, i) < dd(x, q, i)$, which means that for all i in $0 \leq i \leq k2 < log2(b)$, $dd(y, q, i) = dd(x, q, i)$. Therefore, there does not exist any $i = k$ such that $dd(x, q, i)! = dd(y, q, i)$, which means $x =_c y$.

Lemma 3: (transitivity) For all $x, y, z \in P$, if $x \sim y$ and $y \sim z$, then $x \sim z$.

Proof: Suppose that $x \leq_c y$, then, based on Definition 1, for $k1$, $dd(x, q, k1) < dd(y, q, k1)$, which means that for all i in $0 \leq i \leq k1 < log_2(b)$, $dd(x, q, i) = dd(y, q, i)$. Now suppose that $y \leq_c z$. Then, there exists $k2$ such that $dd(x, q, k2) \leq dd(y, q, k2) < dd(z, q, k2)$, which means that for all i in $0 \leq i \leq k2 < log_2(b)$, $dd(x, q, i) \leq dd(y, q, i) = dd(z, q, i)$. Therefore, there exists $0 \leq k3 \leq k2$ such that $dd(x, q, k3) < dd(z, q, k3)$, which means $x \leq_c z (x \sim z)$.

(trichotomy) For all $a, b \in P$, $a <_c b$, $a =_c b$, or $a >_c b$.

Proof: Based on our assumption, for any arbitrary memory segment $a \in P$ with b bits long, calling density difference returns a number $n \in \mathbb{Z}$, which shows its 1's density difference across sub-segment q in the i^{th} call. Comparing any two numbers $n1, n2 \in \mathbb{Z}$ results in $n1 < n2$, $n1 = n2$, or $n1 > n2$, which, based on the definitions

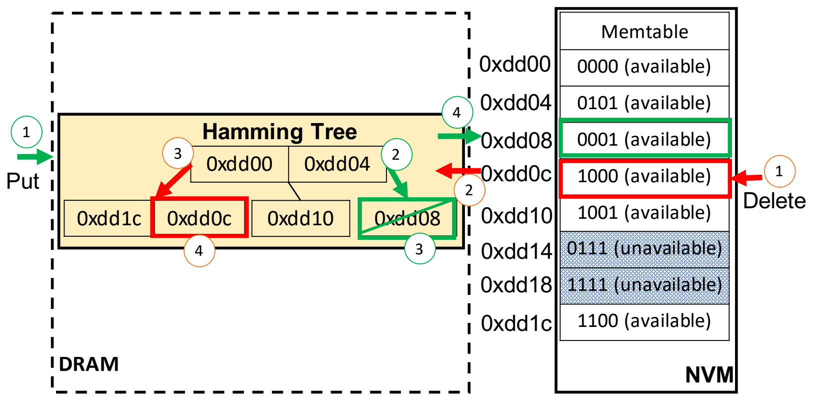


Figure 5.7: An example of procedures which serve key-value PUT and DELETE operations in Hamming Tree.

above, follows that $a <_c b$, $a =_c b$, or $a >_c b$ after a maximum $log_2(b)$ calls.

The first three lemmas prove that density comparison, which is based on density magnitude representations, in a Hamming Tree is partially ordered on the comparison relations, such as $\geq_c$ and $\leq_c$, and addition of the trichotomy proves that density comparison in a $HammingTree$ is totally ordered [38, 131].

Hamming Tree Mapping Structure. Fig. 5.7 provides an example of Hamming Tree's mapping structure where Hamming Tree is in DRAM and maps the available (free) memory locations in the NVM data zone, in which the actual data or K/V pairs are stored. Hamming Tree does not need to be persisted in NVM because it can be reconstructed during recovery. When a DELETE, PUT, or UPDATE operation is applied, Hamming Tree is traversed and updated accordingly to perform the operations and find the best free memory location. (We show more details about how operations are performed later in Section 5.2.2.)

Augmentation Benefits. One of the main advantages of Hamming Tree is that it can be added as a storage layer to existing key-value store. This is important because it reduces the barrier of adopting technologies that improve energy efficiency and write endurance. This is not the case for many other technologies that are tied to a specific indexing design or require special hardware. Additionally, it enables the use of existing non-NVM data structures—that are optimized and have undergone extensive study and research—to work on NVM without having to be redesigned to mitigate the energy and write endurance challenges.

Diversity of available memory segments. In Hamming Tree, when a write request comes to the system, it needs to return an available memory address with most similar bit pattern to the incoming write even if Hamming Tree does not find an exact or similar match to the incoming requests. So, in our method, when it reaches to a leaf node and no match is found, one of the last visited memory locations in the search path can be selected as the target. By going closer to the leaf nodes, the contents become more similar to the new data. It is worth noting that when initializing Hamming Tree, we do not write the area of NVM that Hamming Tree is going to cover. We use the existing content to build the Hamming Tree to avoid performing extra writes/bit flips. If the distribution of the writes does not change significantly, we expect to find more similar bit densities as time progresses.

5.2.2 Hamming Tree Operations

We now show how DELETE, PUT, and UPDATE operations on the indexing structure are handled by Hamming Tree.

DELETE Operation A DELETE operation on the indexing structure means that a data object is removed from the data store. This leads to freeing the memory location that was associated with that data object. The consequence of freeing a memory location is that it needs to be added to Hamming Tree so that it is available to be used for future data objects. Fig. 5.7 shows an example of how a DELETE request, which is shown in red-colored steps, leads to adding a new entry in Hamming Tree. The first step is to mark the memory address 0xdd0c as available. This step is important for recovering Hamming Tree since it is maintained in DRAM. The second step is to issue an insert request to Hamming Tree to add the address 0xdd0c. Finally, the third step is to add the address to Hamming Tree. This step involves traversing Hamming Tree and potentially changing its structure according to the degree and balance of the tree, similar to what is presented in Fig. 5.6.

PUT Operation. A PUT operations on the indexing structure means that a data object is added to the data store. This leads to allocating a free memory location. This free memory location would be selected by traversing Hamming Tree. After selecting the memory location for the new data object, Hamming Tree needs to reflect this selection by removing the address from its structure. The green-colored steps in Fig. 5.7 shows an example of how a PUT request leads to removing an entry from

Hamming Tree. The first step is that a PUT request is intercepted by Hamming Tree. Then, Hamming Tree is traversed according to the content of the PUT operation to find a free memory location with a similar content (in terms of hamming distance).

Assume that the PUT operation is attempting to write with content 1110_b. The traversal starts at the root of Hamming Tree, and 1110_b is compared with the content of nodes in the root (0000_b from 0xdd00 and 0101_b from 0xdd04). 1110_b is smaller than both (according to Hamming Tree compare function), so we traverse through the left-most pointer. Since this is a leaf node, we find the entry that has the content that is most similar to the content of the PUT operation (1110_b) in terms of hamming distance. There are two entries (0xdd1c with content 1100_b and 0xdd0c with content 1000_b). The entry 0xdd1c which has content 1100_b is closer—in terms of hamming distance—to 1110_b. This entry is picked and removed from Hamming Tree (step 3 in Fig. 5.7). Then, the address is returned to the indexing structure so that the write can be applied to 0xdd1c and the address is marked unavailable.

UPDATE Operation. An UPDATE operation is treated as a sequence of a DELETE operation followed by a PUT operation. This would enable finding the best location for the updated data item to minimize bit flips—instead of updating in-place that would be faster but would potentially lead to more bit flips. The delete-insert process can be done concurrently to reduce the latency overhead (*i.e.*, the indexing structure changes the data object pointer to the new memory location and write the update to it immediately while Hamming Tree recycles the old memory location in the background).

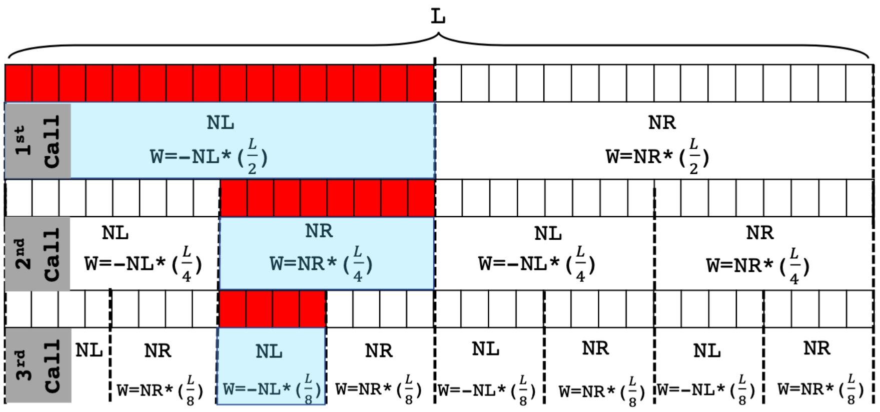
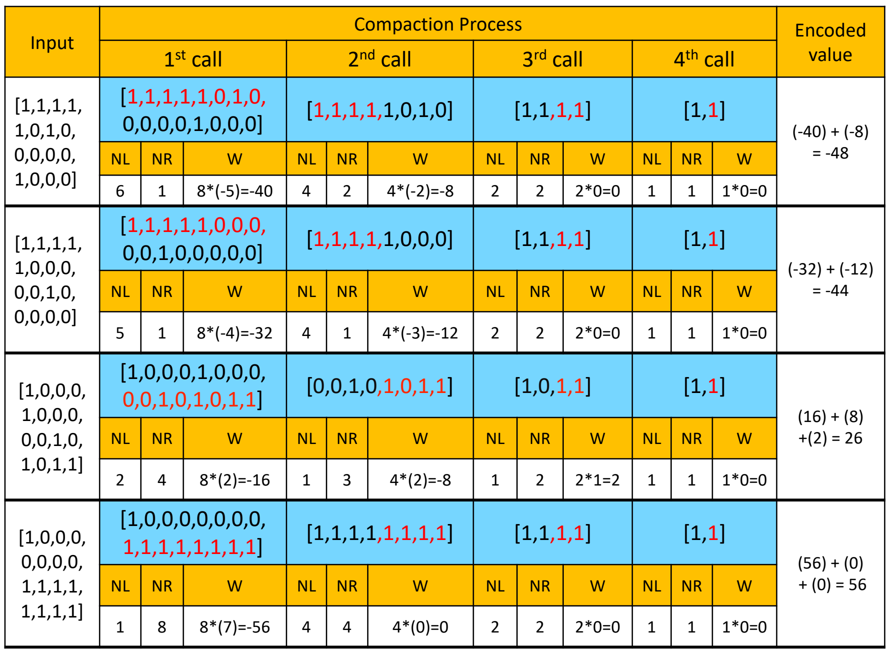
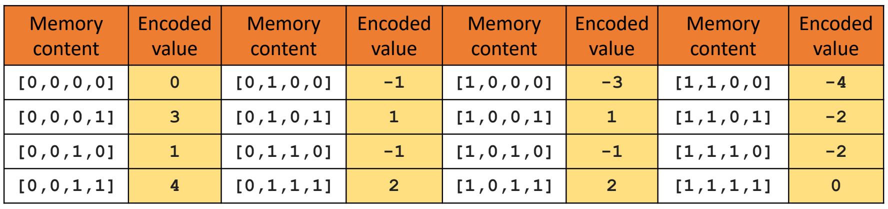


Figure 5.8: Hamming Tree's compaction strategy with three calls (the highlighted areas indicate the half with higher density of 1's).



Input	Compaction Process												Encoded value
	1st call			2nd call			3rd call			4th call			
[1,1,1,1, 1,0,1,0, 0,0,0,0, 1,0,0,0]	[1,1,1,1,1,0,1,0, 0,0,0,0,1,0,0,0]			[1,1,1,1,1,0,1,0]			[1,1,1,1]			[1,1]			(-40) + (-8) = -48
	NL	NR	W	NL	NR	W	NL	NR	W	NL	NR	W	
	6	1	8*(-5)=-40	4	2	4*(-2)=-8	2	2	2*0=0	1	1	1*0=0	
[1,1,1,1, 1,0,0,0, 0,0,1,0, 0,0,0,0]	[1,1,1,1,1,0,0,0, 0,0,1,0,0,0,0,0]			[1,1,1,1,1,0,0,0]			[1,1,1,1]			[1,1]			(-32) + (-12) = -44
	NL	NR	W	NL	NR	W	NL	NR	W	NL	NR	W	
	5	1	8*(-4)=-32	4	1	4*(-3)=-12	2	2	2*0=0	1	1	1*0=0	
[1,0,0,0, 1,0,0,0, 0,0,1,0, 1,0,1,1]	[1,0,0,0,1,0,0,0, 0,0,1,0,1,0,1,1]			[0,0,1,0,1,0,1,1]			[1,0,1,1]			[1,1]			(16) + (8) +(2) = 26
	NL	NR	W	NL	NR	W	NL	NR	W	NL	NR	W	
	2	4	8*(2)=-16	1	3	4*(2)=-8	1	2	2*1=2	1	1	1*0=0	
[1,0,0,0, 0,0,0,0, 1,1,1,1, 1,1,1,1]	[1,0,0,0,0,0,0,0, 1,1,1,1,1,1,1,1]			[1,1,1,1,1,1,1,1]			[1,1,1,1]			[1,1]			(56) + (0) + (0) = 56
	NL	NR	W	NL	NR	W	NL	NR	W	NL	NR	W	
	1	8	8*(7)=-56	4	4	4*(0)=0	2	2	2*0=0	1	1	1*0=0	

Figure 5.9: examples of Hamming Tree's compaction strategy.



Memory content	Encoded value	Memory content	Encoded value	Memory content	Encoded value	Memory content	Encoded value
[0,0,0,0]	0	[0,1,0,0]	-1	[1,0,0,0]	-3	[1,1,0,0]	-4
[0,0,0,1]	3	[0,1,0,1]	1	[1,0,0,1]	1	[1,1,0,1]	-2
[0,0,1,0]	1	[0,1,1,0]	-1	[1,0,1,0]	-1	[1,1,1,0]	-2
[0,0,1,1]	4	[0,1,1,1]	2	[1,0,1,1]	2	[1,1,1,1]	0

Figure 5.10: The compacted values for all the 4-bit inputs.

5.2.3 Compact Content Representation

5.2.3.1 Overview and Motivation

In Hamming Tree, when write requests come to the system, we need to traverse the Hamming Tree to find the target memory, so every step in traversing Hamming Tree leads to reading from NVM. This can incur significant overhead when the size of values is big, which is the case in multimedia applications that we are interested in. In this section, we present an extension of Hamming Tree to overcome this I/O overhead. This method is a compact numerical representation of content that we can augment in Hamming Tree to enable us to perform the traversal and comparison operations without the need of the original content from NVM. However, this is challenging because the compact representation must allow us to perform the compare operation which is based on the density of 0/1 bits. Therefore, we cannot use regular compression techniques as we cannot perform our special compare operation on the compressed content. (Note that there exist compression techniques that allow arithmetic operations, however, they cannot be used in our case because our compare function is based on the special property of the density of 0/1 bits.) We overcome this challenge by designing a specialized compaction strategy that would compact content and allow using the compare operation that is based on 0/1 bit density on the compacted form. We present our proposal next.

5.2.3.2 Compaction Strategy

In this section, we introduce our compaction method that encodes the content of available memory entries in a compact numerical representation, so they can be stored with their corresponding addresses in Hamming Tree instead of the values themselves. Fig. 5.8 illustrates our proposed compaction strategy. As shown in this figure, we build on the methodology used in the comparison function of Hamming Tree to recursively calculate a *density magnitude* for segments of a memory location. In every step, the density of each half of the memory location content is compared, and the difference in the density of 1's is added to the numerical representation (therefore, a positive value indicates more 1 bits in the right half, and vice versa). Then, this is repeated by calling the compaction function but only with the half with the most 1's (the highlighted parts in Fig. 5.8). This continues until we reach a segment of size 1 bit. At each recursion, we assign a lower weight to the added score, which enables the representation to put more priority on the density difference in larger segments (L/2 in the first call, L/4 in the second call and so on). Eventually, the sign and magnitude of the number represents, roughly, the density distribution across segments of the memory location's content. The formal compaction formula is described by the following recursive function:

$$T(A, L, H, n) = W(A, L, H) \times \tfrac{n}{2} + \begin{cases} \text{T(A, L, } \lfloor \frac{L+H}{2} \rfloor \text{, } \frac{n}{2}) & \text{if W<0} \\ \text{T(A, } \lceil \frac{L+H}{2} \rceil \text{, H, } \frac{n}{2}) & \text{if W}\geq 0 \end{cases} \tag{5.1}$$

Where A is the input in bits, n is the size of the input A, L is the start and H is the end index positions, respectively. Also, the termination point for this recursive equation is L≥H when T becomes 0. The value for W is calculated as follows:

$$W(A, L, H) = \begin{cases} N_R & \text{if } N_R \text{ - } N_L > 0 \\ -N_L & \text{if } N_R \text{ - } N_L < 0 \\ 0 & N_R \text{ - } N_L = 0 \end{cases} \tag{5.2}$$

where N_L and N_R are the number of 1's in the left and right sides of A[L,H], respectively.

Fig. 5.9 shows some examples of how our proposed compaction strategy works. As it is shown in this table, the input size is 16 bits, which means that we perform $log_2 16{=}4$ recursive calls. In each call, we calculate W based on the difference of the number of 1's in the right (NR) and in the left (NL) halves of the input. To make the compaction strategy clearer, consider this example of calculating the numerical density representation for A = '1,1,1,1,1,0,1,0,0,0,0,0,1,0,0,0_b', which is the first example in Fig. 5.9. Based on Equation. 5.2, W(A, 0, 16) = N_R - N_L = 1 - 6 = -5 < 0, so, in the first call, Equation. 5.1 is calculated as follows: 1) T(A, 0, 15) = W(A, 0, 15) × (16/2) + T(A, 0, $\lfloor \frac{0+15}{2} \rfloor$) = (-5)×8 + T(A, 0, 7). In the second call, W(A, 0, 7) = N_R - N_L = 2 - 4 = -2 < 0, so based on the equation, T(A, 0, 7) = W(A, 0, 7) × (8/2) + T(A, 0, $\lfloor \frac{0+7}{2} \rfloor$) = (-2)×4 + T(A, 0, 3). In the next call, T(A, 0, 3)= W(A, 0, 3) × (4/2) + T(A, $\lceil \frac{0+3}{2} \rceil$, 3) = 0 × 2 + T(A, 2, 3). And to get the final value, T(A, 2, 3) = W(A, 2, 3) × (2/2) + T(A, $\lceil \frac{2+3}{2} \rceil$, 3) = 0 × 1 + 0 = 0. So, the final encoded value for our input would be T(A, 0, 16) = $(-40) + (-8) = -48$. Fig. 5.10 shows the encoded values for all the combinations of 4-bit inputs. It is worth noting that, in the compaction strategy, having the same numbers for two different patterns, such as [1,1,0,1] and [1,1,1,0]—which are both encoded to -2—does not lead to a problematic situation since they will be treated as having the same key in the index which is handled by indexing data structures. Also, in terms of finding similar memory segments, our goal is to encode densities, and having the same number is an indication in typical cases that the densities are similar. Therefore, picking either one when a request is received with a similar density number is typically sufficient.

5.2.3.3 Space Complexity analysis of our proposed encoding scheme

In this section, we discuss the space complexity of content compaction in Hamming Tree. Specifically, we now derive the size of the compacted numerical representation given the size of the original content. Let's suppose that we want to encode an item A with the size of n bits. Based on Equation. 5.1, T is called on A recursively until the final value is calculated. Here, for clarity of exposition, we include the size of the input in the equations only, and not the start/end indexes and the item A itself. For example, instead of T(A, L, H, n), we just use T(n). First, the encoding function is called for the first time on the whole item (size n).

$$T(n) = W \times \frac{n}{2} + T(\frac{n}{2}) \tag{5.3}$$

Now, based on Equation. 5.2, because the maximum value of W for an item of size n is $\frac{n}{2}$, Equation. 5.3 can be re-written as follows:

$$T(n) \leq \frac{n}{2} \times \frac{n}{2} + T(\frac{n}{2}) = \frac{n^2}{4} + T(\frac{n}{2}) \tag{5.4}$$

Then, by recursively substituting the value of function T we get the expansion and series:

$$T(n) \leq \frac{n^2}{4} + \frac{n^2}{16} + \frac{n^2}{64} + ... + T(1) = \sum_{i=1}^{\log_2 n} \frac{n^2}{4^i} \tag{5.5}$$

We derive—via a well-known series formulation—the following:

$$T(n) \leq \frac{n^2 - 1}{3} \tag{5.6}$$

This means that compacting an original value with size n would result in a compacted numerical value that is less than $\frac{n^2-1}{3}$, which is in the order of $\mathcal{O}(n^2)$. Such a value can be stored using only $\mathcal{O}(\log n)$ bits. For example, for encoding values of sizes 1KB, 1MB, and 1GB, the size of their corresponding encoded values would be 13, 23, and 33 bits, respectively. Therefore, we can eliminate the need to read from NVM while traversing Hamming Tree by including compact representations of the contents of free memory locations. This compact representation would only add negligible size relative to the original size of the content.

5.2.3.4 Total order

In Section 5.2.1 (under "Hamming Tree Density Comparison Arithmetic"), we showed that the original compare function is totally ordered. Here, we discuss the total order guarantee of the compacted variant that we presented above. In the compacted

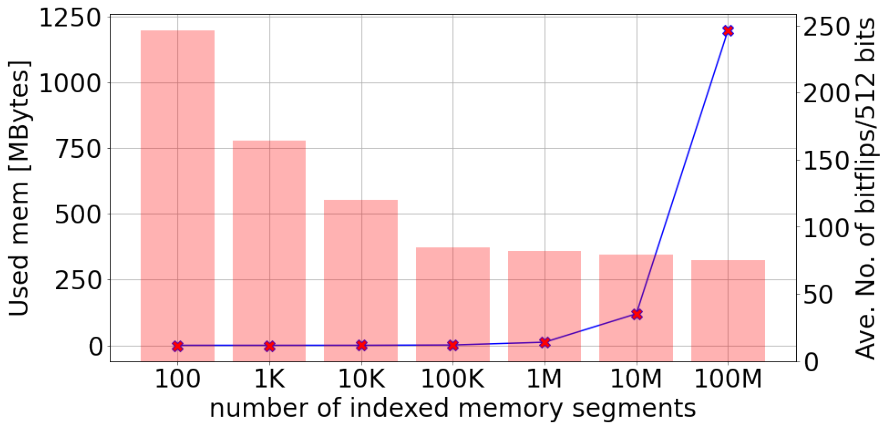


Figure 5.11: Hamming Tree's memory consumption (line graph) and performance (bar graph) for indexing different numbers of memory segments.

strategy, each memory segment is represented as a number that is calculated using the function T presented in Equation 5.1. This means that a set of memory segments is totally-ordered based on the number calculated by the function T of the memory segments in the set.

5.2.4 Indexing granularity and Batching

Hamming Tree organizes the available memory addresses on NVM based on their content. Hamming Tree can be built based on either the content of the keys, values, or both depending on their size. For instance, when the size of the keys are much smaller than the values, it is more reasonable to build the Hamming Tree based on either values or both keys and values. The reason behind this is that when the size of values are much bigger than the size of the keys, then the number of bit flips caused by the values is going to dominate the total number of bit flips caused by writing the key/value pairs. On the other hand, if the size of the keys is not negligible compared to the size of the values, building the Hamming Tree is better with both keys and values.

As we discussed before, Hamming Tree is built on DRAM, and might need to index a large portion of NVM, which usually comes at much bigger sizes than DRAM. This means that if the memory segments that we use in the system are small (for example 1KB) and we want to index a NVM of size 1 TB, we will need to have Hamming Tree index around 1 billion memory segments, which takes a lot of space in DRAM. To solve this problem, Hamming Tree indexes just a portion of the whole available memory that it needs to index and it brings more free memory locations as needed and add them to its free memory locations.

Fig. 5.11 illustrates the amount of memory that Hamming Tree uses for indexing different numbers of memory segments for PubMed data set [47]. The results show that by indexing less number of memory segments, Hamming Tree will take less space in the DRAM. However, it also means that we will have fewer choices to find the most similar location for the incoming writes, which results in increasing the number of bit flips (Fig. 5.11.) However, based on the results, by having 100K to 1M memory segments, we can have the best of both worlds. While we do not see any tangible performance degradation, indexing this number of memory segments will just take a couple of MBs in the memory.

In addition to compact representation that we discussed earlier, to overcome the overhead incurred due to small key-value pairs, batching can be applied so that small writes are grouped together to form larger writes to memory segments. This way, Hamming Tree needs to map the free memory locations based on the batch size rather than the key-value pair size, which leads to reducing Hamming Tree footprint.

5.2.5 Case Study: Augmenting with LSM Tree

We present an example of a Log-Structured Merge (LSM)-Tree-based key-value store that is inspired from LevelDB [53] augmented with Hamming Tree.

5.2.5.1 Background on LSM-Trees

A LSM-Tree consists of n levels, where typically the first level is in-memory and the rest of the levels are in flash/disk. Each level has a threshold on the number of pages that it can contain. Once the threshold is exceeded, some or all of the pages are pushed and merged with the next level. An insert or update operation is buffered in an in-memory data structure called a memtable in the first level of the LSM Tree. Once the memtable is full, it becomes an (immutable) memtable. Eventually, when the number of memtables exceeds the threshold of the first LSM level, these memtables are merged with the pages in the next level (called SSTables). This continues for the following levels.

In addition to the LSM-Tree, the data store needs to log every insert or update to a persistent recovery log before inserting them to the in-memory memtable to avoid data loss in case of a power failure or system crash. However, this is not the case if NVM is used to store memtables. In such a case, memtables are persistent and there is no need for a persistent log (or other associated overheads such as checksum calculations). This motivated many solutions to redesign LSM trees to utilize NVM [13, 79, 125].

5.2.5.2 Hamming Tree with LSM

The overall design starts from how LSM stores typically use NVM: memtables are placed on NVM and SSTables are placed on disk/flash. This structure is augmented with Hamming Tree by placing the mapping structure in DRAM. Hamming Tree is initialized using the content of free memory locations in NVM that are in the pool to be used for memtables. Then, every operation that is performed on memtables (since it is the structure on NVM) is intercepted by Hamming Tree to manage memory allocation and recycling. These are three main operations that can utilize Hamming Tree:

(1) when a data operation is appended to a (mutable) memtable: to insert the data operation information, memory in NVM needs to be allocated. This can be done through Hamming Tree to find a memory location that is similar to the data object information to be written. Because the memory location can be anywhere in the NVM memory space, there needs to be a level of indirection so that all the data objects belonging to the same memtable can be found. This can be a list of pointers that point to the data objects or can be a linked list of pointers, where each element contains the information of a data object.

(2) when the memtable is full and is transformed to an (immutable) memtable: Once a memtable is full, it is typical that the data objects in it are ordered and then rewritten as an immutable memtable consisting of an ordered list of data objects. Once such an immutable memtable is constructed in DRAM, Hamming Tree can be used to find a memory segment in NVM for the memtable to be written to.

(3) when a memtable is deleted: A (mutable) memtable is deleted when the (immutable) one is constructed and an (immutable) memtable is deleted when it's data objects are merged with SSTables in disk/flash. In both cases, the deleted memory locations with data objects or memtable segments need to be recycled. This is managed by Hamming Tree that re-inserts the memory locations into its mapping for future memory allocation.

5.3 Experiments

5.3.1 Methodology

There are many types of phase change material-based technologies, such as Intel® Optane™ Memory Series [71]. In this work, we used Intel® Optane™ Memory to get the results. The experiments are executed on (1) an Intel Core i7 processor running at 4.7 GHz with 4 cores, each of which has 1MB L2 Cache and 12MB L3 Cache using 32GB of Intel® Optane™ Memory Series 3D Xpoint™ and 16 GB of DRAM, and (2) an Intel Xeon® @2.6GHz with 16 cores, an Nvidia Tesla K20m GPU with 5GB memory using 32 GB DDR4 main memory and a 256 GB SSD hard drive. The machine has 32GB DDR3 main memory, 128GB of Intel® Optane™ Persistent Memory 200 Series (PMEM Module), and a 256 GB SSD hard drive. We use the latter machine to get the results in Figures 3.7, 5.13, 5.15, 5.16, 5.18, 5.19, and 5.20, . Although the amount of energy might be different for different setups, both machines showed similar behavior in terms of the relationship between the number of flipped bits and energy consumption. Also, we perform experiments with emulated Optane memory to measure bit flip reduction (which cannot be measured using the real device.)

There are two methods to measure energy consumption: (1) Power Monitors, which use hardware tools to measure the actual power of the device. Despite being very precise, they are extremely difficult to set up. (2) Energy Profilers, which are vastly used by researchers, do not require any special hardware, or power sensors, and estimate the power cost of different hardware using estimation models [35]. In this work, we use an energy profiler named Perf, which is a performance analysis tool and a part of the Intel's RAPL interface [56, 87]. we measured the energy and power consumption of the memory (both DRAM and PMEM), while running our tests using the perf [39] tool:

```
$ perf stat -a -r 5 -e power/energy-cores/, \
  power/energy-ram/, power/energy-gpu/, \
  power/energy-pkg/, power/energy-psys/ ./test
```

This tool provides the collection of energy measurements from various components of a computer system such as: cores, Intel's GPUs, package (all the core and un-core components), DRAM, total power consumption of a node, and so on. It is worth mentioning that, when we compare Hamming Tree with other software or hardware-based methods in terms of energy consumption, we always include the total overhead costs of Hamming Tree including the DRAM processing, cache line accesses, cores, and so on. Also, the sampling rate in our tests is 1000 samples per second.

5.3.2 Overview and setup

We compare with two main groups of solutions: 1) persistent K/V stores that use specialized data structures for NVM [79, 81, 106, 117, 166]. These methods generally focus on reducing write amplification. 2) hardware-based bit flip optimization methods that use the RBW technique [31, 73, 107, 154]. This group focuses directly on decreasing the number of bit flips. We compare Hamming Tree with two main groups of solutions: 1) persistent K/V stores that use specialized data structures to deal with the limitations of NVMs [79, 81, 106, 117, 166]. These methods generally focus on reducing write amplification. 2) hardware-based bit flip optimization methods that use the RBW technique to alleviate the limitations of NVMs [31, 73, 107, 154]. Unlike the previous category, this group focuses directly on decreasing the number of bit flips.

To compare Hamming Tree's results with other methods, we tune their parameters in a way so that they achieve their best performance. We allow MinShift to shift n times, where n is the size of the item instead of the size of the word. This means that it always results in its best performance in terms of the number of bit flips [107]. With respect to Captopril, we also considered its best case, which happens when the blocks are partitioned into n = 16 segments [73].

5.3.2.1 Workloads

We have used various types of real-world and synthetic workloads and data sets in our evaluations. We are using the words workload and dataset interchangeably. The workload (in terms of requests being made) is generated by drawing data from the datasets to form write operations.

Synthetic workloads. For synthetic data sets, our sample K/V store system has at least 10M buckets. When there are 10M buckets, for instance, we first warm-up KV stores with 10M K/V. This means that we store some items as "old data" before starting our tests. The data type and distribution of these items differ depending on the test. "old data" is used to initialize Hamming Tree. Also, for most of the tests with synthetic data, the size of the keys and values are 8 bytes each.

In the first synthetic workload, we run the widely-used YCSB benchmark [34], which provides a framework and a standard set of six Core Workloads for understanding the benefits and implications of cloud workloads, to evaluate Hamming Tree and compare the results with other methods. The six workloads in YCSB have different ratios, workload parameters and access patterns. Also, because in this section, we focus on comparing the performance of different methods in terms of bit flips, we first need to warm-up key-value stores with entries from the same data set before running the benchmark. YCSB has a warm-up (write-only) and a "transactions phase", and we show the transactions phase results when using 4-client threads. In this work, first, we load 10-GB data set into the database as the "old data" in the load phase. Then, we run the workloads one by one, and compare the results.

Real-world data sets. Amazon Access Samples [47] is the first data set that we used here. It contains 30K log entries of access that is provisioned within the company. The next real-world data set is a 3D Road Network Data Set [57,86], that contains 434874 entries of road networks information of North Jutland, Denmark. Finally, the last data

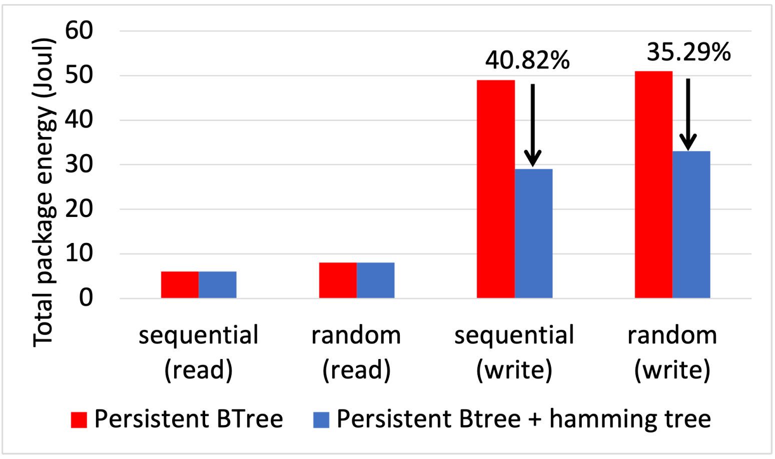


Figure 5.12: The energy consumption of persistent BTree before and after being augmented by Hamming Tree.

set is one of the collections of the DocWord database named “PubMed”, which consists of 730 million entries in the form of “bags-of-words”. This collection, which is called PubMed abstracts [47], consists of 730 million words in total.

5.3.2.2 Persistent Key-Value stores

Persistent key-value stores use special data structures to utilize NVMs. We implemented persistent stores which are designed for NVMs and analyzed them before and after plugging Hamming Tree to them. The following are the persistent key-value stores we compare with:

LSM-Trees (NoveLSM [79] and HiKV [152]). The first data structure is based on LSM-Trees. Due to the popularity of LSM-trees among modern data stores, a significant number of improvements have been proposed on LSM-trees [106]. The method in [79], which is called NoveLSM, is a persistent LSM-based K/V storage systems which

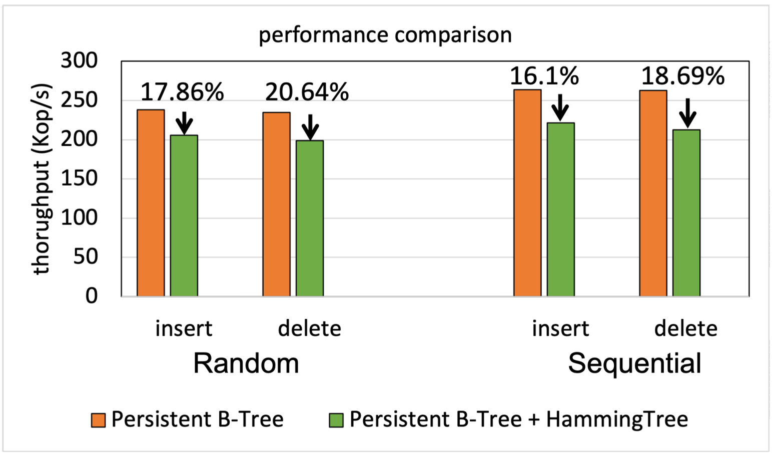


Figure 5.13: The throughput of persistent BTree before and after being augmented by Hamming Tree.

is designed to utilize NVM to provide low latency and high throughput. HiKV [152] is another persistent key-value store with the central idea of constructing a hybrid index in hybrid memory. We implemented both NoveLSM and HiKV and analyzed them in terms of the number of bit flips, throughput, and latency before and after plugging Hamming Tree.

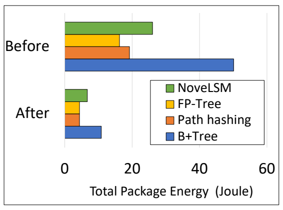
B+-Trees (FP-Tree [117] and wBTree [29]). The second data structure that is used widely in K/V data stores is B+-Trees [65]. FP-Tree [117] is one of the hybrid SCM-DRAM persistent and concurrent B+-Tree, named Fingerprinting Persistent Tree (FPTree) that is designed specifically for NVMs. Similarly, wBTree [29] is a specialized NVM-friendly write atomic B+-Tree to utilize the non-volatility of NVMs. Due to their high performance and low write amplification, we also implemented these methods alongside a regular B+Tree and compared the number of bit flips they cause before and after plugging Hamming Tree.

Hash indexing (Path Hashing [166]). Another type of data structures that are vastly used in various applications are hash-based indexing structures. A lot of effort has been made to improve hash-based indexing structures for NVMs, and almost all of them focus on decreasing the write amplification to reach their aims. We have also implemented one of the most recent hash-based indexing structures, named Path hashing [166], which is designed specifically for NVMs.

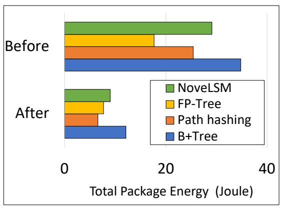
5.3.2.3 Experiments setup.

In this work, we evaluated latency and energy consumption on a real Intel Optane memory device. For this purpose, we used the Persistent Memory Development Kit (PMDK) [3]. For measuring bit flip reduction we emulated the NVM device and memory controller. This is because measuring bit flips is not possible on the real device. This is similar to previous work in this area [16, 81, 138, 139, 158]. Similar to these works, we have adopted previously used methods to emulate NVM [116, 146, 152] and we added an emulation of memory controller. The emulated memory controller utilizes a wear-leveling algorithm that is inspired from prior literature on wear leveling [138]. Specifically, the wear leveling algorithm swaps two NVM memory segments every ψ writes to the NVM device. This ψ value is called the *swapping period.* When a swap is triggered, the destination memory segment is swapped with another random memory segment. Like before, we set the value of ψ to 8 for all the emulation experiments.

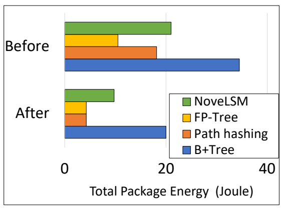
[3]Persistent Memory Development Kit https://pmem.io/pmdk/.



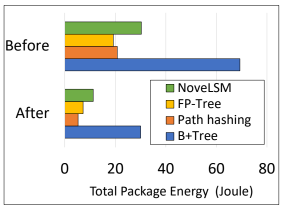
(a)Amazon dataset



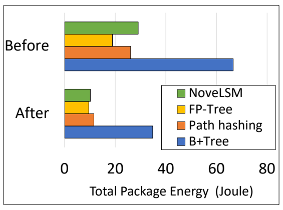
(b)3D Road Network



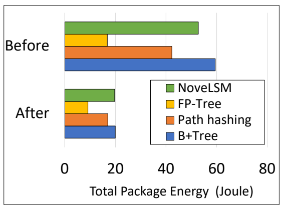
(c)PubMed



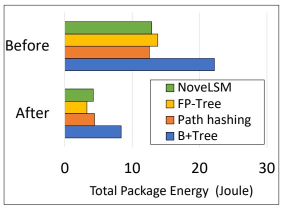
(d)normal dist



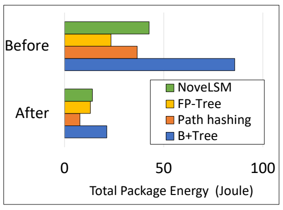
(e)uniform dist



(f)log-normal dist



(g)sequential



(h)Zipf

Figure 5.14: The average energy consumption for writing one memory segment, using various data structures and data sets, both before and after augmentation by the Hamming Tree.

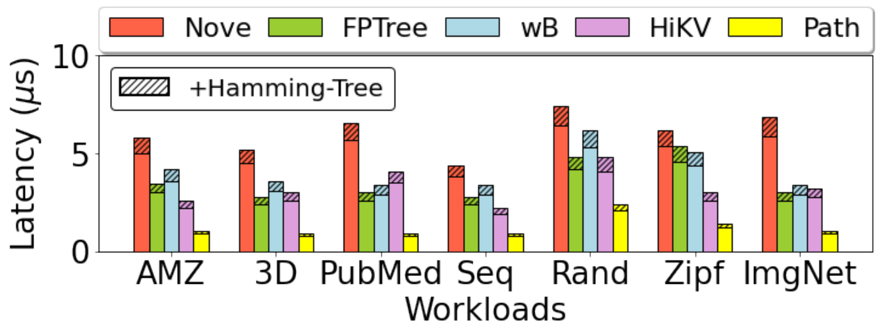


Figure 5.15: Write latency of different methods before and after being augmented by Hamming Tree.

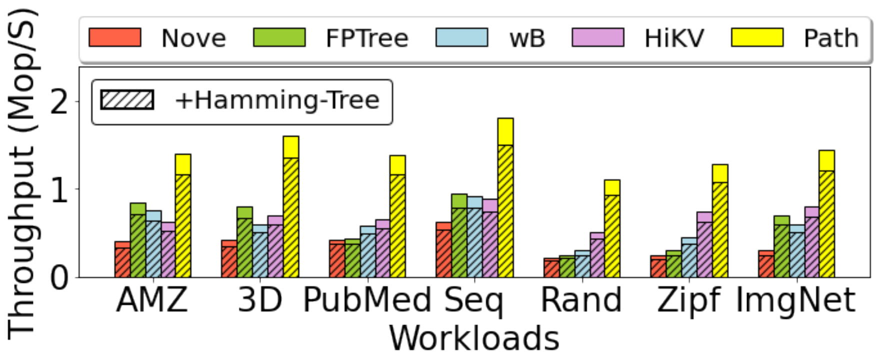


Figure 5.16: Throughput of different methods before and after being augmented by Hamming Tree.

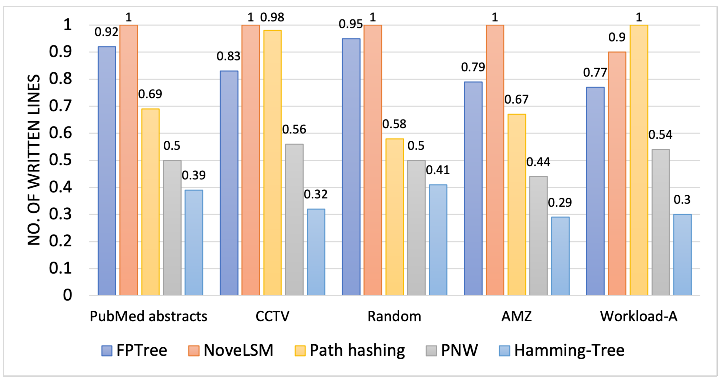


Figure 5.17: The normalized number of cache lines per request.

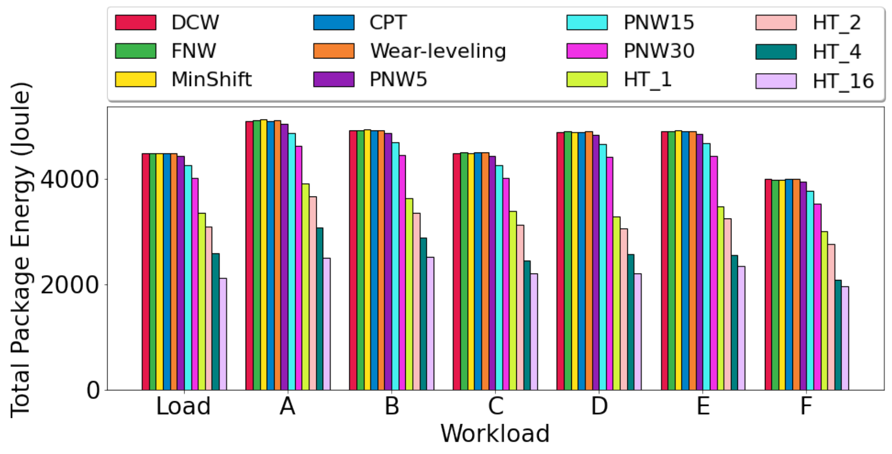


Figure 5.18: The average amount of energy consumed for various methods.

5.3.3 Results

In the first experiment, we tested a persistent B-Tree before and after augmenting with Hamming Tree on a real Intel Optane memory device. The results in Fig. 5.12 shows that by plugging Hamming Tree, the system consumes up to 41% less energy. This validates the objective of Hamming Tree and its potential to reduce energy consumption. Fig. 5.13 shows the performance overhead of Hamming Tree in terms of throughput with and without augmenting with Hamming Tree. The figure shows that the overhead is no more than 21% and can be as low as 16.1%. This overhead is due to the Hamming Tree operations and represents a trade-off between the benefits of bit flip reduction using Hamming Tree (improved energy efficiency and write endurance) and performance.

In the next experiment, we tested the energy consumption of the implemented methods in terms of the amount of energy they consume in two different ways: before plugging with Hamming Tree, and after plugging with Hamming Tree. The results are shown in Fig. 5.14. When not plugged to Hamming Tree, B+-Tree has the worst energy consumption because, in a regular B+-Tree, the items in leaf nodes need to be sorted, which increases the number of movements and bit flips. The improvement in energy efficiency after plugging Hamming Tree is up to 67.8%.

To analyze the performance overhead of Hamming Tree, we conducted an experiment and measured the throughput and latency of different methods before and after being augmented with Hamming Tree. Fig. 5.15 and 5.16 shows that results. As it is shown in the results, the performance overhead of Hamming Tree is up to 19.8%. Another important observation that we can make from this test is that the performance overhead of Hamming Tree depends on the ratio of the size of the memory segments to the size of the memory pool, which affects performance of Hamming Tree's search, insertion and deletion operations.

In the next experiment, the average number of written cache lines per request is normalized for different data sets, as shown in Fig. 5.17. Also, from the YCSB workloads, we chose Workload-A, which is an update heavy workload. As it is clear, the number of written cache lines per request in FPTree and NoveLSM is generally higher than Path-Hashing because they modify more items to process a request. PNW always has fewer written cache lines mostly because it chooses the memory location for writing the items from those clusters that have similar patterns to the item. However, it lags behind Hamming Tree due to the fact that the granularity of mapping memory location is the cluster, which leads to potential inaccuracies and not finding the most similar memory location of an incoming write. Fig. 5.18 shows the amount of energy that is consumed

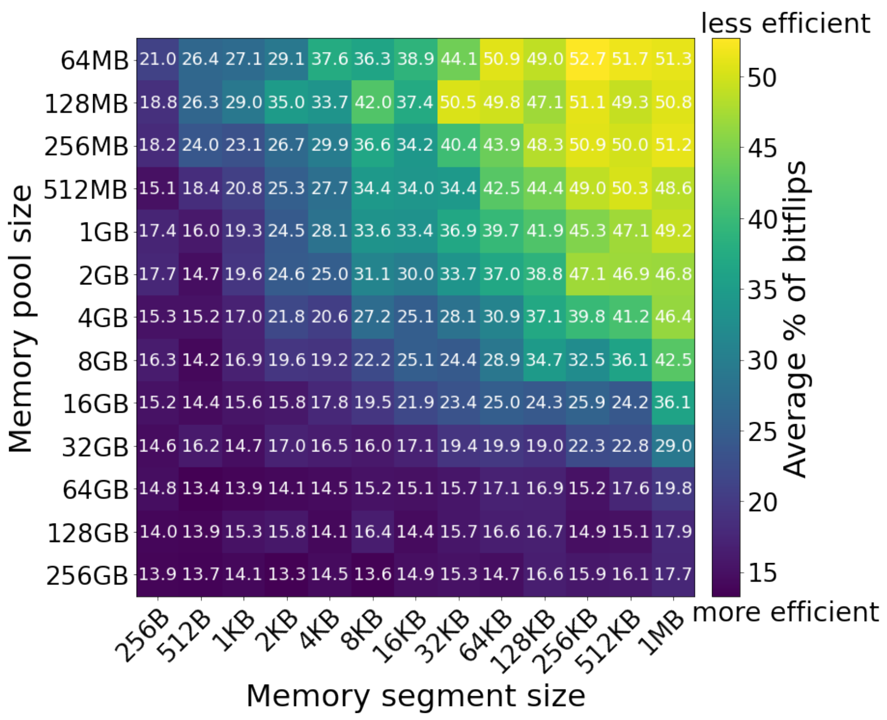


Figure 5.19: The average updated bits ratio for different memory segment and memory pool sizes in Hamming Tree.

in different hardware-based and software-based methods for writing the entire memory pool. The reported energy also includes energy being used to manage Hamming Tree in DRAM in addition to the energy consumed in PMEM. As it is shown in this figure, Hamming Tree can reduce the amount of consumed energy up to 41% compared to the other method due to its ability in finding the most similar memory contents.

Fig. 5.18 also has the results for the wear-leveling method, which distributes the writes evenly across the memory blocks of NVM. Based on the results, the wear-leveling method causes a similar number of bit flips as the other conventional methods that are not "memory-aware." The reason is that 1) in some data sets, the density of bits

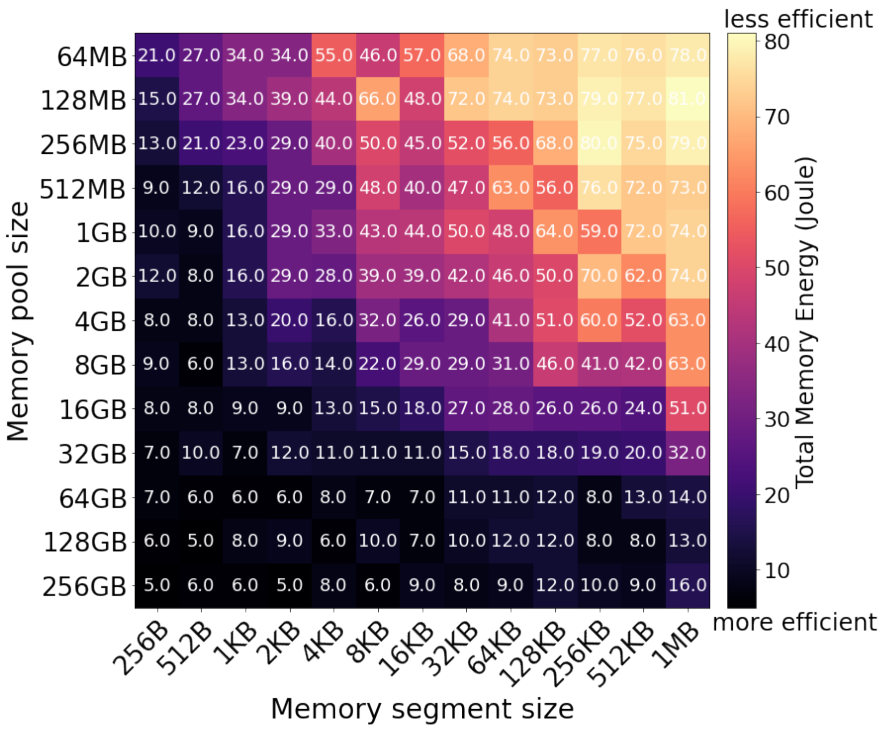


Figure 5.20: Total memory energy consumption (joule) for different memory segment and memory pool sizes in Hamming Tree.

in some regions—such as in the low-order bits—is more than the other parts and thus the most bit flipping is happening in this part, and 2) when the number of bit flips is less than half of the item size, most bit flipping optimized methods, such as FNW, behave similarly, which means that wear-leveling cannot improve the performance of the system in terms of the number of bit flips. This property of conventional methods also might have some serious consequences such as causing some parts of the memory locations, such as the lowest bits in this data set, to wear out faster than the middle and highest bits. However, this is not the case for Hamming Tree and PNW.

Fig. 5.19 and 5.20 illustrates the overall performance of Hamming Tree in terms

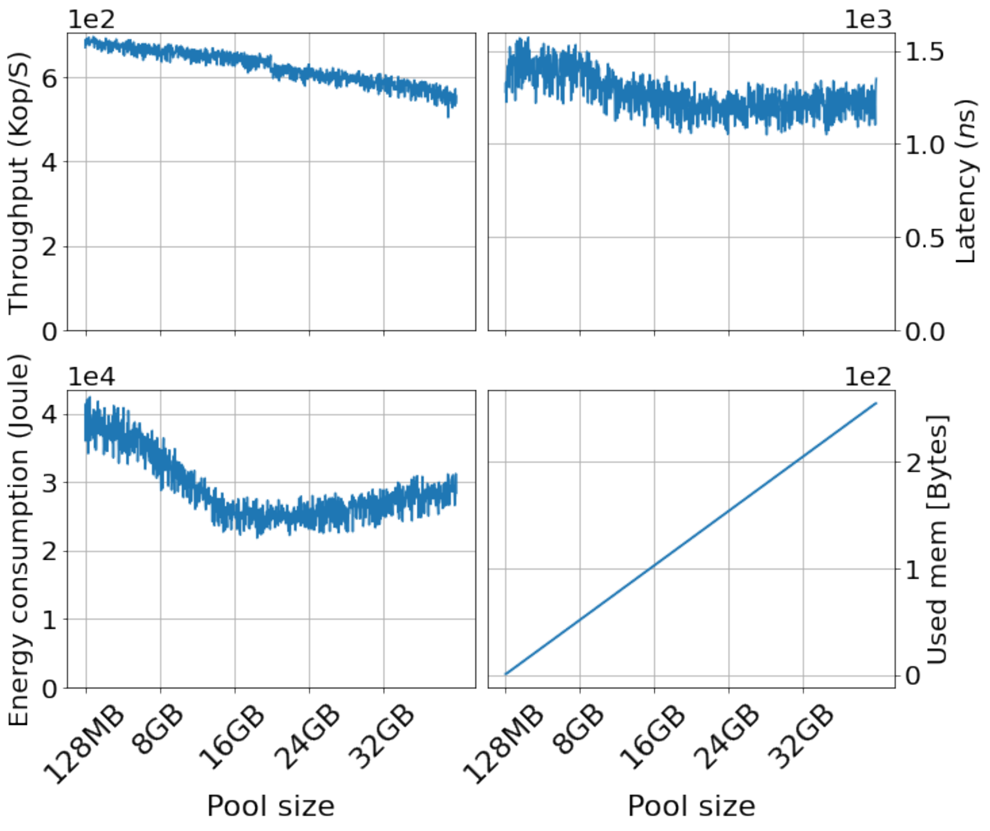


Figure 5.21: The impact of Hamming Tree on key performance metrics of the system when the memory pool size changes.

of energy efficiency and updated bits for different memory segment and pool size choices for the mixure of all the real-world data sets in this work. For the overall energy efficiency shown in Fig. 5.20, we observe that the Hamming Tree's power consumption increases as the ratio of the memory segment size to the memory pool size increases, which also aligns with results of the average bitflips ratio shown in Fig. 5.19. From this observation, we conclude that smaller memory segment sizes, compared to data zone (pool size), result in more available tree nodes for Hamming Tree, leading to fewer bit flips and lower energy consumption. However, as the ratio increases, Hamming Tree requires more DRAM memory to index all available memory segments, which hinders system performance. Therefore, there is a trade-off in choosing the optimal ratio, depending on the application's priorities—whether energy consumption or memory usage is more critical. Generally, a ratio of memory pool size to memory segment size between 1M to 16M yields better efficiency, though this may vary based on use cases and system resources.

In the next experiment, we conducted comprehensive tests to analyze the impact of Hamming Tree on key performance metrics of the system. To this aim, first, we set a fixed size memory segment size of 4KB, and then build Hamming Tree on different memory pool size. As it is shown in Fig. 5.21, given a fixed memory segment size, by increasing the size of memory pool, system's throughput decreases. That is the result of increasing the size of Hamming Tree, which not only increases the cost of search operations, but also makes the split or merge operations more expensive for the write/delete operations. For energy-consumption and latency, our experiments show a

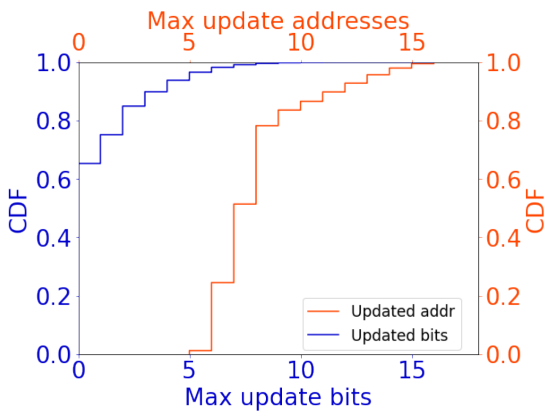


Figure 5.22: The maximum update addresses and wear-leveling as CDFs by applying Hamming Tree.

"valley" trend: the energy-consumption and latency is relatively high at both the biggest and smallest trees, and is relatively low at intermediate tree size. This is because, in small memory pools, the number of available addresses is too small and the contents in each part of the tree are not very similar to each other, which leads to high energy consumption in NVM. Conversely, increasing the ratio of pool size to the memory segment size beyond a certain level increases the total energy consumption while providing only a limited return on bit flip reduction, also leading to energy inefficiency in DRAM and CPU. This"energy-valley" trend indicates that choosing the best ratio of memory segments to the memory pool in an energy-aware fashion requires a good trade-off between the amount of energy that the system requires for writing on NVM and the energy that Hamming Tree needs to run Hamming Tree.

As stated earlier, many memory controllers are optimized to only flip bits when the value being written to a cell differs from the old value [16]. So, successful NVM-optimized systems will need to target not only wear-leveling but also bit flip reduction in a write operation. This not only can save a significant amount of energy but also delay wearing out of the device [16, 80, 81]. Therefore, to observe the performance of Hamming Tree in terms of the distribution of the maximum number of bit flips and the wear-leveling of PCM, we conduct two more tests. In these tests, we run Hamming Tree on the mixture of MNIST and Fashion-MNIST data sets. For this test, we first warm up the data zone with 28K items from the combination of both data sets. Then, we stream 112K writes from the same data sets to the system. During the test, we also perform delete actions to make space for incoming writes. In other words, each word in the data zone is updated 4 times on average.

Fig. 5.22 shows: (1) the estimation of the likelihood to observe an address in the data zone of PCM that is written less than or equal to a specific number of times, and (2) the estimation of the likelihood to observe a memory bit in the data zone of PCM that is written less than or equal to a specific number of times. For example, as we can see in Fig. 5.22, the estimated likelihood to observe an address in the PCM data zone to be written less than or equal to 8 (P $(X \leq 8)$) is 80% (red color). Similarly, we observe that while the estimated likelihood of a memory bit being written less than or equal to 6 times is almost 100% (blue color). This important observation shows that (1) Hamming Tree distributes write activities across the whole NVM chip uniformly, and (2) Hamming Tree distributes bit flips evenly across the whole data zone of the PCM chip.

Finally, Fig. 5.23 shows the impact of employing Hamming Tree on PCM

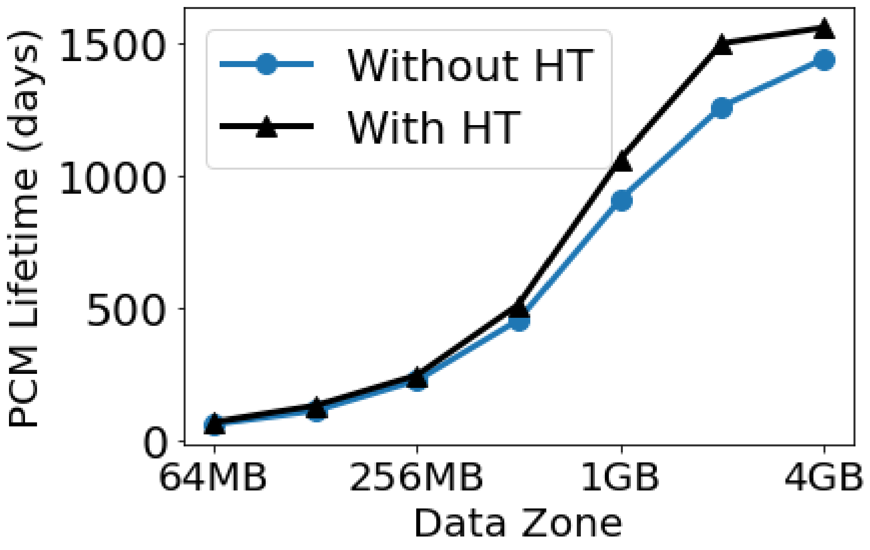


Figure 5.23: PCM lifetime improvement before and after utilizing Hamming Tree for different data zone sizes.

lifetime for the same mix workload that we used for the previous test. Based on this figure, first, we observe that the smaller the size of the data zone is for a specific application, the more a single PCM block is overwritten, and the shorter the PCM lifetime is. For instance, for a data zone of 256 MB, a single PCM block is overwritten as many as 4 times while executing 1 billion instructions. So, when assuming the system operation frequency of 2.6 GHz, PCM write endurance of 10^8, and 0.5 IPC, the lifetime of the PCM cell can be calculated as 227 days ($=10^8/4 \times 10^9$ instructions $\times$ 1/2IPC $\times$ 1/2.6GHz $\times$ 1/(24hours $\times$ 60minutes)) [135]. Furthermore, Fig. 5.23 shows that Hamming Tree can also have a positive effect on the lifetime of the PCM, especially in the systems that a smaller memory is available or a portion of the memory is excessively used by a write-intensive application.

5.4 Conclusion

Since non-volatile memory technologies are widely adopted into data storage solutions and battery powered mobile and IoT devices, wear-out and energy consumption have become two vital optimizations for these technologies. In this section, we presented the case for memory-awareness and showed that by judiciously selecting memory locations for new writes and updates we can reduce bit flipping and consequently improve the energy efficiency and write endurance of NVM devices. We take this concept and build Hamming Tree, with which existing data stores can be augmented, to make them memory-aware. Hamming Tree tackles the challenges associated with mapping free memory locations based on the hamming distance of their content. In our evaluation section, we augment various data stores with Hamming Tree and we performed experiments on a real Intel Optane memory device that show that Hamming Tree can achieve up to 67.8% improvement in energy efficiency.

Chapter 6

Conclusion and future directions

This thesis contributes to integrating NVM technologies into the memory hierarchy, addressing associated challenges through software-level solutions that facilitate the deployment of NVMs in database and storage systems. Through proposing software-level solutions, including advanced learning techniques and data structure-based methods, we achieve substantial improvements in energy efficiency and write endurance, thereby enhancing the practicality and longevity of NVM devices. The four main parts in this thesis are:

Part One: System study We conduct a thorough evaluation of real-world NVM devices, such as Intel Optane memory, to investigate the impact of memory awareness on performance, energy consumption, and lifespan. Our findings reveal that memory-aware strategies significantly extend device lifetime, reduce power consumption, and improve system latency. This section underscores the necessity of incorporating recent advancements from the NVM storage community into existing and future data management systems.

Part Two: Predict and Write (PNW) We introduce Predict and Write (PNW), a key-value store that employs a clustering-based machine learning approach to extend the lifetime of NVMs. PNW minimizes the number of bit flips during PUT/UPDATE operations by selecting optimal memory locations for updated values. By utilizing the indirection level of key-value stores, PNW dynamically organizes NVM addresses into clusters based on data value similarity. Our results show that PNW can reduce total bit flips by up to 85% and cache lines by 56% compared to current methods.

Part Three: E2-NVM We present E2-NVM, a software-level memory-aware storage layer designed to enhance the energy efficiency and write endurance (E2) of NVMs. E2-NVM uses a Variational Autoencoder (VAE) based design to intelligently direct write operations to memory segments that minimize bit flips. This innovative solution, which can be integrated with existing indexing and hardware-based methods, not only solves the existing problems in the previous methods but also demonstrates a reduction in energy consumption by up to 56% in real-world evaluations on an Optane memory device.

Part Four: Hamming Tree We introduce a software-level data storage layer solution employing an indexing data structure to improve the energy consumption and write endurance of NVMs. Hamming Tree, an indexing structure, enhances existing data stores to become memory-aware. Hamming Tree addresses the challenges of mapping free memory locations based on the hamming distance of their content. Our evaluations on an Intel Optane memory device show that Hamming Tree can achieve up to a 67.8% improvement in energy efficiency.

This thesis presents a comprehensive framework for enhancing the efficiency and longevity of NVM devices. By integrating advanced software solutions and innovative data structures, it addresses critical challenges related to energy consumption and write endurance, facilitating the broader adoption of NVM technologies in future data management systems.

Among the proposed methods, PNW is the simplest, suitable for systems with fixed memory segment sizes and basic hardware resources. It performs adequately even without advanced components like GPUs or large DRAM, though such resources can enhance its performance. E2-NVM, an advanced version of PNW, handles large, variable-sized memory segments and offers high flexibility but requires more than basic system resources, such as GPUs, to maximize the efficiency of its VAE and LSTM core models. Finally, the Hamming Tree, implemented on an indexing data structure, requires basic system resources without special capabilities. It is fast but sensitive to the number of indexed items, as this affects tree size, throughput, and latency.

It's important to note that while some NVM technologies, like Intel's Optane, have ceased development, existing NVM technologies such as PRAMs and SSDs still face challenges with energy consumption and longevity. The solutions introduced in this thesis, though tested on specific types like Optane and PCMs, are adaptable to other technologies. For instance, the concepts behind PNW and E2-NVM can be applied to other NVMs to extend their lifespan and reduce energy consumption by clustering similar writes and minimizing write amplification.

Future research could explore the applicability of our proposed methods in different applications used in IoT and mobile devices, where energy efficiency is crucial. Additionally, using NVMs in combination with other technologies, such as SSDs, is a prevalent use case where our methods could be beneficial. We are also beginning to explore reinforcement learning to enhance the efficiency and dynamism of our approaches, presenting further opportunities for researchers.

In conclusion, the proposed methods and future directions offer valuable opportunities for researchers in data management systems to develop solutions that overcome NVM limitations, which are essential for their adoption and success. This thesis presents practical approaches for augmenting existing techniques from the NVM storage community to be adopted in data management systems and outlines future opportunities in memory-awareness to improve the lifespan and energy efficiency of PCM devices.

Bibliography

[1] Sukarn Agarwal and Hemangee K Kapoor. Targeting inter set write variation to improve the lifetime of non-volatile cache using fellow sets. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2017.

[2] Sukarn Agarwal and Hemangee K Kapoor. Improving the lifetime of non-volatile cache by write restriction. *IEEE Transactions on Computers*, 68(9):1297–1312, 2019.

[3] Sukarn Agarwal and Hemangee K Kapoor. Improving the performance of hybrid caches using partitioned victim caching. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(1):1–27, 2020.

[4] Shoaib Akram. Performance evaluation of intel optane memory for managed workloads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(3):1–26, 2021.

[5] Shoaib Akram, Jennifer B Sartor, Kathryn S McKinley, and Lieven Eeckhout.

Write-rationing garbage collection for hybrid memories. *ACM SIGPLAN Notices*, 53(4):62–77, 2018.

[6] Alaa Alameldeen and David Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2004.

[7] Ahmed Izzat Alsalibi, Mohd Khaled Yousef Shambour, Muhannad A Abu-Hashem, Mohammad Shehab, Qusai Shambour, and Riham Muqat. Nonvolatile memory-based internet of things: A survey. In *Artificial Intelligence-based Internet of Things Systems*, pages 285–304. Springer, 2022.

[8] Jaan Altosaar. *Tutorial - What is a Variational Autoencoder?*, August 2016.

[9] Milad Andalibi, Mojtaba Hajihosseini, Sam Teymoori, Maryam Kargar, and Meysam Gheisarnejad. A time-varying deep reinforcement model predictive control for dc power converter systems. In *2021 IEEE 12th International Symposium on Power Electronics for Distributed Generation Systems (PEDG)*, pages 1–6. IEEE, 2021.

[10] Joy Arulraj et al. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722, 2015.

[11] Chris H. Bahnsen and Thomas B. Moeslund. Rain removal in traffic surveillance:

Does it matter? *IEEE Transactions on Intelligent Transportation Systems*, pages 1–18, 2018.

[12] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. {TRIAD}: Creating synergies between memory, disk and log in log structured {Key-Value} stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, 2017.

[13] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 80–94, 2017.

[14] Amir Ban. Wear leveling of static areas in flash memory, May 4 2004. US Patent 6,732,221.

[15] Sana Benhamaid, Abdelmadjid Bouabdallah, and Hicham Lakhlef. Recent advances in energy management for green-iot: An up-to-date and comprehensive survey. *Journal of Network and Computer Applications*, 198:103257, 2022.

[16] Daniel Bittman et al. Optimizing systems for byte-addressable {NVM} by reducing bit flipping. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 17–30, 2019.

[17] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging nvm:

A survey on architectural integration and research challenges. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(2):1–32, 2017.

[18] Hao Cai, You Wang, Lirida Alves de Barros Naviner, Jun Yang, and Weisheng Zhao. Exploring hybrid stt-mtj/cmos energy solution in near-/sub-threshold regime for iot applications. *IEEE Transactions on magnetics*, 54(2):1–9, 2017.

[19] Yi Cai, Yujun Lin, Lixue Xia, Xiaoming Chen, Song Han, Yu Wang, and Huazhong Yang. Long live time: Improving lifetime and security for nvm-based training-in-memory systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4707–4720, 2020.

[20] Richard Cangelosi and Alain Goriely. Component retention in principal component analysis with application to cdna microarray data. *Biology direct*, 2(1):2, 2007.

[21] Maurizio Capra, Riccardo Peloso, Guido Masera, Massimo Ruo Roch, and Maurizio Martina. Edge computing: A survey on the hardware requirements in the internet of things world. *Future Internet*, 11(4):100, 2019.

[22] Escuin Carlos, Ibañez Pablo, Monreal Teresa, Jose M Llaberia, and Victor Viñals. L2c2: Last-level compressed-cache nvm and a procedure to forecast performance and lifetime. *arXiv preprint arXiv:2204.09504*, 2022.

[23] Adrian M Caulfield, Arup De, Joel Coburn, Todor I Mollow, Rajesh K Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-

generation, non-volatile memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395. IEEE, 2010.

[24] M Emre Celebi et al. A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert systems with applications*, 40(1):200–210, 2013.

[25] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, 2021.

[26] Yuezhi Che, Yuanzhou Yang, Amro Awad, and Rujia Wang. A lightweight memory access pattern obfuscation framework for nvm. *IEEE Computer Architecture Letters*, 19(2):163–166, 2020.

[27] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. {SpanDB}: A fast,{Cost-Effective}{LSM-tree} based {KV} store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32, 2021.

[28] Lei Chen, Jiacheng Zhao, Chenxi Wang, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, Guoqing Harry Xu, et al. Unified holistic memory management supporting multiple big data processing frameworks over hybrid memories. *ACM Transactions on Computer Systems (TOCS)*, 2022.

[29] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[30] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, 13(12):2634–2648, 2020.

[31] Sangyeun Cho and Hyunjin Lee. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance. In *MICRO 2009*, pages 347–357, 2009.

[32] Zhaole Chu, Yongping Luo, and Peiquan Jin. An efficient sorting algorithm for non-volatile memory. *International Journal of Software Engineering and Knowledge Engineering*, 31(11n12):1603–1621, 2021.

[33] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. {SplinterDB}: Closing the bandwidth gap for {NVMe}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63, 2020.

[34] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[35] Luís Cruz. Tools to measure software energy consumption from your computer.

`http://luiscruz.github.io/2021/07/20/measuring-energy.html`, 2021. Blog post.

[36] Lixiao Cui, Kewen He, Yusen Li, Peng Li, Jiachen Zhang, Gang Wang, and Xiao-Guang Liu. Swapkv: A hotness aware in-memory key-value store for hybrid memory systems. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[37] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From wisckey to bourbon: A learned index for log-structured merge trees. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 155–171, 2020.

[38] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, Cambridge, United Kingdom, 2002.

[39] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[40] Yoshiaki Deguchi and Ken Takeuchi. 3d-nand flash solid-state drive (ssd) for deep neural network weight storage of iot edge devices with 700x data-retention lifetime extention. In *2018 IEEE international memory workshop (IMW)*, pages 1–4. IEEE, 2018.

[41] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet:

A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[42] David B Dgien et al. Compression architecture for bit-write reduction in non-volatile memory technologies. In *NANOARCH 2014*, pages 51–56. IEEE, 2014.

[43] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: A hybrid pram and dram main memory system. In *2009 46th ACM/IEEE Design Automation Conference*, pages 664–669. IEEE, 2009.

[44] Chris Ding and Xiaofeng He. K-means clustering via principal component analysis. In *ICML'04*, page 29, 2004.

[45] Krijn Doekemeijer and Animesh Trivedi. Key-value stores on flash storage devices: A survey. *arXiv preprint arXiv:2205.07975*, 2022.

[46] Wei Dong et al. Minimizing update bits of nvm-based main memory using bit flipping and cyclic shifting. In *HPCC 2015, CSS 2015, and ESS 2015*, pages 290–295. IEEE, 2015.

[47] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[48] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–15, 2014.

[49] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[50] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1218–1232, 2021.

[51] Amin Ashraf Gandomi, Maryam Kargar, Saeed Kargar, Leila Parsa, and Keith Corzine. Deep-learning-based fault detection and location method applied on isolated dc-dc converter. In *2023 IEEE Applied Power Electronics Conference and Exposition (APEC)*, pages 2245–2252. IEEE, 2023.

[52] Andrés Amaya García, René de Jong, William Wang, and Stephan Diestelhorst. Composing lifetime enhancing techniques for non-volatile main memories. In *Proceedings of the International Symposium on Memory Systems*, pages 363–373, 2017.

[53] Sanjay Ghemawat and Jeff Dean. Google leveldb, January 2022.

[54] Bob Gleixner, Fabio Pellizzer, and Roberto Bez. Reliability characterization of phase change memory. In *2009 10th Annual Non-Volatile Memory Technology Symposium (NVMTS)*, pages 7–11. IEEE, 2009.

[55] Binbin Gu, Saeed Kargar, and Faisal Nawab. Efficient dynamic clustering: Capturing patterns fromhistorical cluster evolution. *arXiv preprint arXiv:2203.00812*, 2022.

[56] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11), 2011.

[57] Chenjuan Guo et al. Ecomark: evaluating models of vehicular environmental impact. In *SIGSPATIAL '12*, pages 269–278, 2012.

[58] Xifeng Guo, Long Gao, Xinwang Liu, and Jianping Yin. Improved deep embedded clustering with local structure preservation. In *IJCAI*, pages 1753–1759, 2017.

[59] Yuncheng Guo, Yu Hua, and Pengfei Zuo. Dfpc: A dynamic frequent pattern compression scheme in nvm-based main memory. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1622–1627. IEEE, 2018.

[60] Yuncheng Guo, Yu Hua, and Pengfei Zuo. A latency-optimized and energy-efficient write scheme in nvm-based main memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(1):62–74, 2018.

[61] Christian Hakert, Roland Kühn, Kuan-Hsun Chen, Jian-Jia Chen, and Jens Teubner. Octo+: Optimized checkpointing of b+ trees for non-volatile main memory wear-leveling. In *2021 IEEE 10th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2021.

[62] Fazal Hameed et al. Efficient stt-ram last-level-cache architecture to replace dram cache. In *MEMSYS 2017*, pages 141–151, 2017.

[63] Chien-Chung Ho, Wei-Chen Wang, Te-Hao Hsu, Zhi-Duan Jiang, and Yung-Chun Li. Approximate programming design for enhancing energy, endurance and performance of neural network training on nvm-based systems. In *2021 IEEE 10th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2021.

[64] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[65] Jiangkun Hu et al. Understanding and analysis of b+ trees on nvm towards consistency and efficiency. *CCF Transactions on High Performance Computing*, pages 1–14, 2020.

[66] Fangting Huang, Dan Feng, Wen Xia, Wen Zhou, Yucheng Zhang, Min Fu, Chuntao Jiang, and Yukun Zhou. Security rbsg: Protecting phase change memory with security-level adjustable dynamic mapping. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1081–1090. IEEE, 2016.

[67] Jian Huang et al. Nvram-aware logging in transaction systems. *Proceedings of the VLDB Endowment*, 8(4):389–400, 2014.

[68] Jianming Huang, Yu Hua, Pengfei Zuo, Wen Zhou, and Fangting Huang.

An efficient wear-level architecture using self-adaptive wear leveling. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[69] Kaixin Huang, Yan Yan, and Linpeng Huang. Revisiting persistent hash table design for commercial non-volatile memory. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 708–713. IEEE, 2020.

[70] Tianhao Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Hyve: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 973–978. IEEE, 2018.

[71] Intel. Types of intel optane memory, January 2022.

[72] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[73] Majid Jalili and Hamid Sarbazi-Azad. Captopril: Reducing the pressure of bit flips on hot locations in non-volatile main memories. In *DATE 2016*, pages 1116–1119. IEEE, 2016.

[74] Jean-Philippe Jodoin et al. Urban tracker: Multiple object tracking in urban mixed traffic. In *WACV 2014*, pages 885–892. IEEE, 2014.

[75] Ian T Jolliffe and Jorge Cadima. Principal component analysis: a review and

recent developments. *Philos. Trans. Royal Soc. A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016.

[76] Kalpana D Joshi and PS Nalwade. Modified k-means for better initial cluster centres. *IJCSMC 2013*, 2(7):219–223, 2013.

[77] Aditya K Kamath et al. Storage class memory: Principles, problems, and possibilities. *arXiv preprint arXiv:1909.12221*, 2019.

[78] Wang Kang, Liuyang Zhang, Weisheng Zhao, Jacques-Olivier Klein, Youguang Zhang, Dafiné Ravelosona, and Claude Chappert. Yield and reliability improvement techniques for emerging nonvolatile stt-mram. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 5(1):28–39, 2014.

[79] Sudarsun Kannan et al. Redesigning lsms for nonvolatile memory with novelsm. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 993–1005, 2018.

[80] Saeed Kargar, Binbin Gu, Sangeetha Abdu Jyothi, and Faisal Nawab. E2-nvm: A memory-aware write scheme to improve energy efficiency and write endurance of nvms using variational autoencoders. 2023.

[81] Saeed Kargar, Heiner Litz, and Faisal Nawab. Predict and write: Using k-means clustering to extend the lifetime of nvm storage. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 768–779. IEEE, 2021.

[82] Saeed Kargar and Faisal Nawab. Extending the lifetime of nvm: challenges and opportunities. *Proceedings of the VLDB Endowment*, 14(12):3194–3197, 2021.

[83] Saeed Kargar and Faisal Nawab. Hamming tree: The case for memory-aware bit flipping reduction for nvm indexing. In *CIDR*, 2021.

[84] Saeed Kargar and Faisal Nawab. Challenges and future directions for energy, latency, and lifetime improvements in nvms. *Distributed and Parallel Databases*, pages 1–27, 2022.

[85] Saeed Kargar and Faisal Nawab. Hamming tree: The case for energy-aware indexing for nvms. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

[86] Manohar Kaul et al. Building accurate 3d spatial networks to enable next generation intelligent transportation systems. In *MDM 2013*, volume 1, pages 137–146. IEEE, 2013.

[87] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

[88] Sunggon Kim and Yongseok Son. Optimizing key-value stores for flash-based ssds via key reshaping. *IEEE Access*, 9:115135–115144, 2021.

[89] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. Pageseer: Using page

walks to trigger page swaps in hybrid memory systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 596–608. IEEE, 2019.

[90] Adarsh Kosta, Efstathia Soufleri, Indranil Chakraborty, Amogh Agrawal, Aayush Ankit, and Kaushik Roy. Hyperx: A hybrid rram-sram partitioned system for error recovery in memristive xbars. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 88–91. IEEE, 2022.

[91] Tim Kraska et al. The case for learned index structures. In *SIGMOD 2018*, pages 489–504, 2018.

[92] R Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. {TIPS}: Making volatile index structures persistent with {DRAM-NVMM} tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 773–787, 2021.

[93] Alex Krizhevsky et al. Learning multiple layers of features from tiny images. 2009.

[94] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267. IEEE, 2013.

[95] Jinzhi Lai, Jueping Cai, and Jie Chu. A congestion-aware hybrid sram and stt-

ram buffer design for network-on-chip router. *IEICE Electronics Express*, pages 19–20220078, 2022.

[96] Hoyoung Lee, Minho Lee, and Young Ik Eom. Partial tiering: A hybrid merge policy for log structured key-value stores. In *2021 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 20–23. IEEE, 2021.

[97] Cheng Li, Hao Chen, Chaoyi Ruan, Xiaosong Ma, and Yinlong Xu. Leveraging nvme ssds for building a fast, cost-effective, lsm-tree-based kv store. *ACM Transactions on Storage (TOS)*, 17(4):1–29, 2021.

[98] Jianhong Li, Andrew Pavlo, and Siying Dong. Nvmrocks: Rocksdb on non-volatile memory systems, 2017.

[99] Wenjie Li et al. Hilsm: an lsm-based key-value store for hybrid nvm-ssd storage systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages 208–216, 2020.

[100] Chun-Yi Liu, Yunju Lee, Myoungsoo Jung, Mahmut Taylan Kandemir, and Wonil Choi. Prolonging 3d nand ssd lifetime via read latency relaxation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 730–742, 2021.

[101] Jihang Liu et al. Lb+ trees: optimizing persistent index performance on 3dxpoint memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, 2020.

[102] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. Hierarchical hybrid memory

management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2223–2236, 2019.

[103] Zihao Liu, Tao Liu, Jie Guo, Nansong Wu, and Wujie Wen. An ecc-free mlc stt-ram based approximate memory design for multimedia applications. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 142–147. IEEE, 2018.

[104] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020.

[105] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[106] Chen Luo and Michael J Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[107] Xianlu Luo et al. Enhancing lifetime of nvm-based main memory with bit shifting and flipping. In *RTCSA 2014*, pages 1–7. IEEE, 2014.

[108] Zhulin Ma, Edwin H-M Sha, Qingfeng Zhuge, Weiwen Jiang, Runyu Zhang, and Shouzhen Gu. Towards the design of efficient hash-based indexing scheme for growing databases on non-volatile memory. *Future Generation Computer Systems*, 105:1–12, 2020.

[109] T Soni Madhulatha. An overview on clustering methods. *arXiv preprint arXiv:1205.1117*, 2012.

[110] Haiyu Mao, Xian Zhang, Guangyu Sun, and Jiwu Shu. Protect non-volatile memory from wear-out attack based on timing difference of row buffer hit/miss. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1623–1626. IEEE, 2017.

[111] Erxue Min, Xifeng Guo, Qiang Liu, Gen Zhang, Jianjing Cui, and Jun Long. A survey of clustering with deep learning: From the perspective of network architecture. *IEEE Access*, 6:39501–39514, 2018.

[112] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *TPDS 2015*, 27(5):1537–1550, 2015.

[113] Fargol Nematkhah, Farrokh Aminifar, Mohammad Shahidehpour, and Sasan Mokhtari. Evolution in computing paradigms for internet of things-enabled smart grid applications: Their contributions to power systems. *IEEE Systems, Man, and Cybernetics Magazine*, 8(3):8–20, 2022.

[114] Andrew Y Ng et al. On spectral clustering: Analysis and an algorithm. In *NIPS 2002*, pages 849–856, 2002.

[115] Yuanjiang Ni, Shuo Chen, Qingda Lu, Heiner Litz, Zhu Pang, Ethan L Miller, and

Jiesheng Wu. Closing the performance gap between dram and pm for in-memory index structures. (UCSC-SSRC-20-01), May 2020.

[116] Jiaxin Ou et al. A high performance file system for non-volatile main memory. In *EuroSys '16*, pages 1–16, 2016.

[117] Ismail Oukid et al. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386, 2016.

[118] Poovaiah M Palangappa and Kartik Mohanram. Flip-mirror-rotate: An architecture for bit-write reduction and wear leveling in non-volatile memories. In *GLSVLSI 2015*, pages 221–224, 2015.

[119] Poovaiah M Palangappa and Kartik Mohanram. Compex++ compression-expansion coding for energy, latency, and lifetime improvements in mlc/tlc nvms. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):1–30, 2017.

[120] Bahareh Pourshirazi, Majed Valad Beigi, Zhichun Zhu, and Gokhan Memik. Writeback-aware llc management for pcm-based main memory systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(2):1–19, 2019.

[121] R Sankara Prasad, Nitin Chaturvedi, S Gurunarayanan, et al. A low power high

speed mtj based non-volatile sram cell for energy harvesting based iot applications. *Integration*, 65:43–50, 2019.

[122] Gianlucca O Puglia et al. Non-volatile memory file systems: A survey. *IEEE Access*, 7:25836–25871, 2019.

[123] Moinuddin K Qureshi et al. Scalable high performance main memory system using phase-change memory technology. In *ISCA '09*, pages 24–33, 2009.

[124] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 14–23. IEEE, 2009.

[125] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.

[126] Khushboo Rani and Hemangee K Kapoor. Write variation aware buffer assignment for improved lifetime of non-volatile buffers in on-chip interconnects. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(9):2191–2204, 2019.

[127] Elizabeth Reed, Alaa R Alameldeen, Helia Naeimi, and Patrick Stolt. Probabilistic

replacement strategies for improving the lifetimes of nvm-based caches. In *Proceedings of the International Symposium on Memory Systems*, pages 166–176, 2017.

[128] Pooneh Safayenikoo, Arghavan Asad, Mahmood Fathy, and Farah Mohammadi. An energy efficient non-uniform last level cache architecture in 3d chip-multiprocessors. In *2017 18th International Symposium on Quality Electronic Design (ISQED)*, pages 373–378. IEEE, 2017.

[129] Pooneh Safayenikoo, Arghavan Asad, Mahmood Fathy, and Farah Mohammadi. Exploiting non-uniformity of write accesses for designing a high-endurance hybrid last level cache in 3d cmps. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–5. IEEE, 2017.

[130] Pooneh Safayenikoo, Arghavan Asad, Mahmood Fathy, and Farah Mohammadi. A new traffic compression method for end-to-end memory accesses in 3d chip-multiprocessors. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–4. IEEE, 2017.

[131] Gunther Schmidt and Thomas Ströhlein. *Relations and graphs: discrete mathematics for computer scientists.* Springer Science & Business Media, Berlin, 2012.

[132] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with

dynamically randomized address mapping. *ACM SIGARCH computer architecture news*, 38(3):383–394, 2010.

[133] Jie Shi, Hualu Zhang, Yang Bai, Guangjie Han, and Gangyong Jia. A novel data aggregation preprocessing algorithm in flash memory for iot based power grid storage system. *IEEE Access*, 6:57279–57290, 2018.

[134] Yuhan Shi, Sangheon Oh, Zhisheng Huang, Xiao Lu, Seung H Kang, and Duygu Kuzum. Performance prospects of deeply scaled spin-transfer torque magnetic random-access memory for in-memory computing. *IEEE Electron Device Letters*, 41(7):1126–1129, 2020.

[135] Dongsuk Shin, Hakbeom Jang, Kiseok Oh, and Jae W Lee. An energy-efficient dram cache architecture for mobile platforms with pcm-based main memory. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(1):1–22, 2022.

[136] Shihao Song, Adarsha Balaji, Anup Das, and Nagarajan Kandasamy. Design-technology co-optimization for nvm-based neuromorphic processing elements. *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[137] Shihao Song, Anup Das, and Nagarajan Kandasamy. Exploiting inter-and intra-memory asymmetries for data mapping in hybrid tiered-memories. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pages 100–114, 2020.

[138] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. Improving

phase change memory performance with data content aware access. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pages 30–47, 2020.

[139] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. Aging-aware request scheduling for non-volatile main memory. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 657–664, 2021.

[140] Baohua Sun, Daniel Liu, Leo Yu, Jay Li, Helen Liu, Wenhan Zhang, and Terry Torng. Mram co-designed processing-in-memory cnn accelerator for mobile and iot applications. *arXiv preprint arXiv:1811.12179*, 2018.

[141] Penghao Sun, Dongliang Xue, Litong You, Yan Yan, and Linpeng Huang. Hyperkv: A high performance concurrent key-value store for persistent memory. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 125–134. IEEE, 2021.

[142] MA Syakur et al. Integration k-means clustering method and elbow method for identification of the best customer profile cluster. In *IOP Conference Series: Materials Science and Engineering*, volume 336, page 012017. IOP Publishing, 2018.

[143] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B Brockman, and

Norman P Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. *ACM SIGARCH Computer Architecture News*, 36(3):51–62, 2008.

[144] Alexander van Renen et al. Persistent memory i/o primitives. In *DaMoN'19*, pages 1–7, 2019.

[145] Lucas Vendramin et al. Relative clustering validity criteria: A comparative overview. *Statistical analysis and data mining: the ASA data science journal*, 3(4):209–235, 2010.

[146] Haris Volos et al. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[147] Qiuping Wang, Jinhong Li, Patrick PC Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in logstructured storage. In *Proc. of USENIX FAST*, 2022.

[148] Yasi Wang, Hongxun Yao, and Sicheng Zhao. Auto-encoder based dimensionality reduction. *Neurocomputing*, 184:232–242, 2016.

[149] Jin Wu, Jian Dong, Ruili Fang, Wen Zhang, Wenwen Wang, and Decheng Zuo. Wdbt: Non-volatile memory wear characterization and mitigation for dbt systems. *Journal of Systems and Software*, 187:111247, 2022.

[150] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. Parallax: Hybrid key-value placement in lsm-based key-value

stores. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 305–318, 2021.

[151] Fei Xia et al. A survey of phase change memory systems. *Journal of Computer Science and Technology*, 30(1):121–144, 2015.

[152] Fei Xia et al. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *USENIX ATC 17*, pages 349–362, 2017.

[153] Jian Xu and Steven Swanson. {NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[154] Byung-Do Yang et al. A low power phase-change random access memory using a data-comparison write scheme. In *ISCAS 2007*, pages 3014–3017. IEEE, 2007.

[155] Jianguo Yang, Yinyin Lin, Yarong Fu, Xiaoyong Xue, and BA Chen. A small area and low power true random number generator using write speed variation of oxidebased rram for iot security application. In *2017 IEEE international symposium on circuits and systems (ISCAS)*, pages 1–4. IEEE, 2017.

[156] Vinson Young, Prashant J Nair, and Moinuddin K Qureshi. Deuce: Write-efficient encryption for non-volatile memories. *ACM SIGARCH Computer Architecture News*, 43(1):33–44, 2015.

[157] Furqan Zahoor, Tun Zainal Azni Zulkifli, and Farooq Ahmad Khanday. Resistive random access memory (rram): an overview of materials, switching mechanism,

performance, multilevel cell (mlc) storage, modeling, and applications. *Nanoscale research letters*, 15(1):1–26, 2020.

[158] Qi Zeng and Jih-Kwon Peir. Content-aware non-volatile cache replacement. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 92–101. IEEE, 2017.

[159] Hongyuan Zha et al. Spectral relaxation for k-means clustering. In *NIPS 2002*, pages 1057–1064, 2002.

[160] Baoquan Zhang and David HC Du. Nvlsm: A persistent memory key-value store using log-structured merge tree with accumulative compaction. *ACM Transactions on Storage (TOS)*, 17(3):1–26, 2021.

[161] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: a key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 194–209, 2021.

[162] Hengyu Zhao and Jishen Zhao. Leveraging mlc stt-ram for energy-efficient cnn training. In *Proceedings of the International Symposium on Memory Systems*, pages 279–290, 2018.

[163] Fang Zhou, Song Wu, Youchuang Jia, Xiang Gao, Hai Jin, Xiaofei Liao, and Pingpeng Yuan. Vail: A victim-aware cache policy to improve nvm lifetime for hybrid memory system. *Parallel Computing*, 87:70–76, 2019.

[164] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient

main memory using phase change memory technology. *ACM SIGARCH computer architecture news*, 37(3):14–23, 2009.

[165] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. Dptree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment*, 13(4):421–434, 2019.

[166] Pengfei Zuo and Yu Hua. A write-friendly hashing scheme for non-volatile memory systems. In *Proc. MSST*, 2017.