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

SAEED KARGAR, University of California, Santa Cruz, USA
FAISAL NAWAB, University of California, Irvine, USA

Non-volatile memory (NVM) technologies are widely adopted in data storage solutions and battery-powered mobile and IoT devices. Wear-out and energy efficiency are two vital challenges facing the use of NVM. In Hamming Tree, we propose a software-level memory-aware solution that picks the memory segment of where a write operation is applied judiciously to minimize bit flipping. It has been shown that reducing bit flips leads to reducing energy consumption and improving write endurance. We performed real evaluations on an Optane memory device that show that Hamming Tree can achieve up to 67.8% reduction in energy consumption.

CCS Concepts: • **Information systems → Phase change memory**; *Information storage technologies*; *Storage class memory*; Data access methods.

Additional Key Words and Phrases: Non-Volatile Memory, energy efficiency, bit flip reduction

ACM Reference Format:
Saeed Kargar and Faisal Nawab. 2023. Hamming Tree: The Case for Energy-Aware Indexing for NVMs. *Proc. ACM Manag. Data* 1, 2, Article 182 (June 2023), 27 pages. https://doi.org/10.1145/3589327

1 INTRODUCTION

Non-Volatile Memory (NVM) has the potential of transforming the memory architecture in data management systems due to their characteristics such as persistence, high density, and byte addressability [23]. However, NVM technologies suffer from two main challenges: (1) NVM write operations demand a significant amount of current and power. For flipping an individual bit in PCM, it requires around 50 pJ/b, compared to writing a whole DRAM page that only needs 1 pJ/b [6, 34]. (2) NVM has low write endurance (the number of writes that can be applied to a segment of storage media before it becomes unreliable.) NVM write endurance is in the order of 10^8–10^9 writes, which is significantly lower than DRAM write endurance in the order of 10^{15} writes [32, 34, 41].

To overcome the energy consumption and write endurance problems in NVM, two approaches were developed. The first approach develops hardware-based write optimization techniques that are mostly based on a *Read-Before-Write* (RBW) pattern [56]. In RBW, a write operation w to a memory location x is always preceded by 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 reduces energy consumption and increases write endurance [56]. Other approaches build on RBW as we discuss in the related work section [1, 10, 15, 28, 45].

The second approach tackles the problem of energy consumption and write endurance by minimizing Write Amplification via techniques such as caching [3, 9, 44], delayed merging [29, 37], or by designing specialized data structures that require fewer writes [59]. However, these methods conflate the problem of energy efficiency and write endurance with the problem of write amplification. Although in most cases a technique that leads to reducing write amplification has

Authors' addresses: Saeed Kargar, University of California, Santa Cruz, Santa Cruz, California, USA, skargar@ucsc.edu; Faisal Nawab, University of California, Irvine, Irvine, California, USA, nawabf@uci.edu.

2836-6573/2023/6-ART182
https://doi.org/10.1145/3589327

the side-effect of increasing energy efficiency and write endurance, they do not enable reaching the full potential of such improvements that can be attained with bit flip reduction because they focus on decreasing the number of written words instead of bit flips. For example, Path Hashing [59] and NoveLSM [29]—that reduce write amplification—do not reach the full potential of improving energy efficiency compared to bit flip-reduction techniques (Section 4).

These 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*. 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 (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 [26, 31–34, 51, 57].

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. We show this experimentally with evaluations with a high percentage of updates in the workload.

We augment existing data stores with our proposed software-level data storage layer called *Hamming Tree*. 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 [29], a cache optimized NVM index called FP-Tree [44], and a write-friendly hashing scheme [59]. We performed real evaluations on an Optane memory device that show that Hamming Tree can reduce energy consumption by up to 67.8%.

2 BACKGROUND

2.1 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 [26]. Some memory controllers also adopt bit flipping reduction technologies such as ones based on RBW [10, 15, 22]. 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.

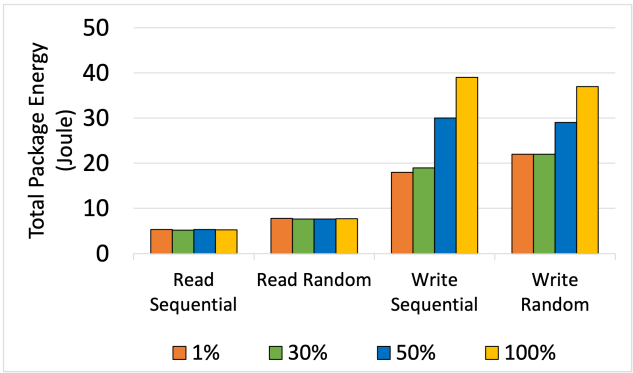


Fig. 1. Total memory energy consumption on a real Intel Optane memory device for read and write operations with different percentages of hamming distance.

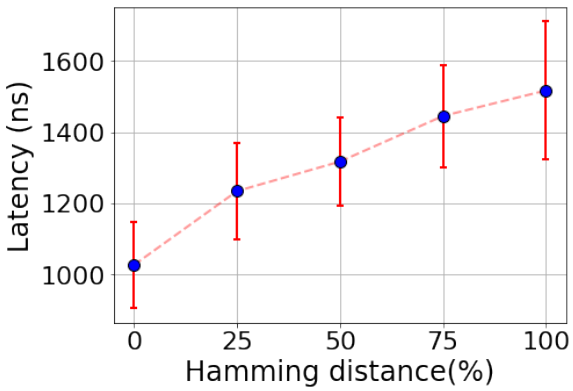


Fig. 2. The write latency in a real Intel Optane memory device for different percentages of hamming distance.

2.2 Motivation: 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) [46], 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 energy consumption of the socket for each round. Fig. 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. 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 control where a new write is applied in the NVM device is that the memory controller's wear leveling method might

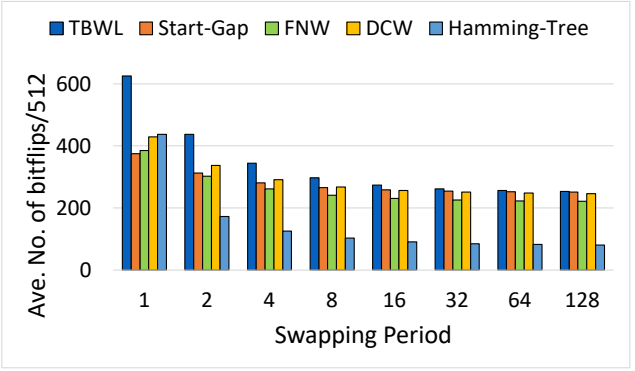


Fig. 3. The average number of bit updates for different wear-leveling techniques when the swapping period changes. (TBWL: Table-based Wear-Leveling [58], Start-Gap [47], FNW: Flip-n-Write [10], DCW: Data Comparison Write [15])

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 [26, 30]. 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 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. 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 [17], 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 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 [7], such as the Internet of Things (IoT) and mobile devices, in which conserving power is one of the main concerns [5, 42].

2.3 Related Work

Local write optimizations (Hardware-level techniques). Some methods focus on reducing the number of bit flips through the Read-Before-Write (RBW) technique [10, 15, 28]. For example, Flip-n-Write (FNW) [10] 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. CDE [15] is another example, which finds common patterns and then compresses data to reduce the number of bit flips in NVM. Like Flip-N-Write, CDE replaces a write operation with a read-modify-write process after comparing the new data and the old data bit by bit. Similarly, Captopril [28] masks "hot locations", where bits are flipped more, to reduce the number of bit flips.

Wear leveling. Wear leveling [10, 15, 26, 47, 51, 57] aims to extend the lifetime of NVM devices. It does so 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 [8]. This, however, means that wear leveling does not reduce the actual number of bit flips for writes—it only distributes them across the device. This means that benefits of reducing bit flips in terms of energy efficiency would not be observed with wear leveling. Additionally, combining wear leveling with bit flipping reduction can achieve further improvement in the lifespan of NVM devices. This is especially the case since distributing writes by itself is susceptible to rapid wear-out as previous work has shown [25, 40].

Local write optimizations (software techniques). There are methods that focus on reducing the number of bit flips at the software level without the need for special hardware.

Recently, many applications from power systems 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 [2, 19, 30, 54]. In [6], the authors designed new data structures based on the idea of pointer distance to reduce the number of bit flips. Predict and Write (PNW) [31] uses a clustering-based approach to extend the lifetime of NVMs. Like the previous method, PNW also targets bit flip reduction through software techniques. PNW suffers from two challenges that Hamming Tree overcomes. First, PNW relies on machine learning to cluster memory locations based on their content. This is an expensive process that incurs high overhead stemming from the data pre-processing step, which needs applying expensive dimension reduction techniques, and the training/prediction steps, which are all relatively expensive. Hamming Tree, in comparison, is faster due to building on a conventional tree-based structure. Second, PNW adopts a clustering technique which means that the granularity of mapping memory location is the cluster. This leads to potential inaccuracies. Hamming Tree, on the other hand, maps at the granularity of the memory location, which enables finding more similar candidates for new writes. We compare with PNW in the evaluation section.

E2-NVM [30] is another software-level memory-aware solution to improve energy efficiency and write endurance of NVMs using deep learning. In this method, memory locations are selected judiciously for new writes and updates, which results in reducing bit flipping and improving the energy efficiency and write endurance of NVM devices. Similar to Hamming Tree, E2-NVM can be augmented with existing data stores to make them memory-aware.

Reducing write amplification. Many data storage and indexing solutions target the reduction of write amplification to optimize the utilization of I/O bandwidth [3, 9, 29, 37, 44, 59]. With the introduction of NVM to the memory hierarchy, it turns out that reducing write amplification can have the positive side-effect of improving energy efficiency and write endurance since fewer data

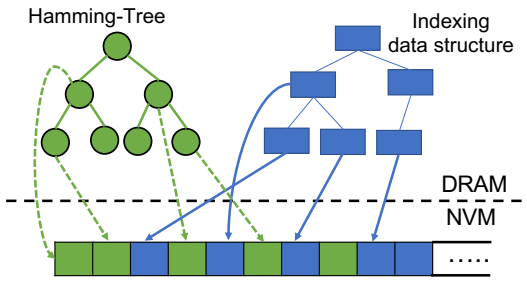


Fig. 4. The storage and memory layout of Hamming Tree.

is written. Nevertheless, reducing write amplification does not enable reaching the full potential of such improvements that can be attained with bit flip reduction [6, 31]. This is because—unlike flash—NVM cells are written individually, which means that the number of flipped bits is more critical to optimize than the total number of written words [6]. In the evaluation section, we show how Hamming Tree achieves better energy efficiency when compared to methods that rely on only reducing write amplification.

Missed opportunity. Although the mentioned techniques can reduce the bit flip rate in the word level, they miss an opportunity to significantly reduce bit flipping. This opportunity is to be *memory-aware*, which enables us to judiciously select memory locations for write operations that would minimize bit flipping and thus improve the energy efficiency and lifetime of NVM devices.

3 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 [10] 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 paper, 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.

3.1 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 [16]. Fig. 4 shows

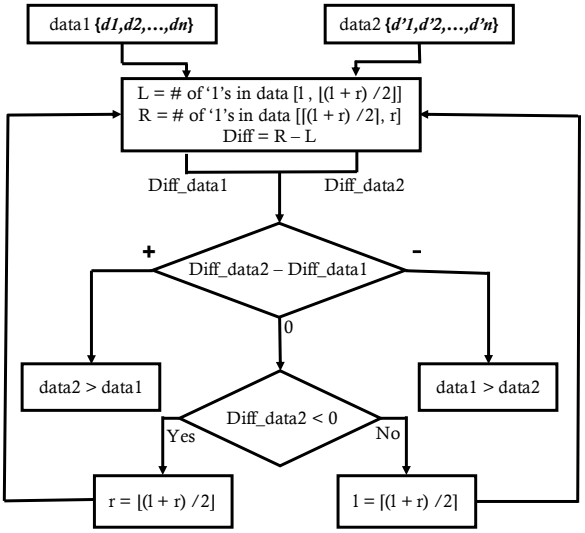


Fig. 5. The comparison method in Hamming Tree.

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 [31], 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.

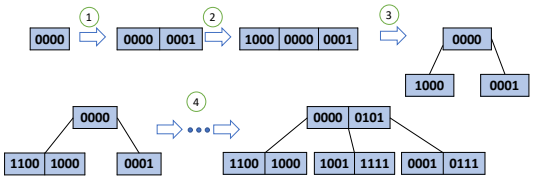


Fig. 6. An example of how Hamming Tree is formed.

Hamming-Distance-Based Comparison Function. Fig. 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, 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.

Hamming Tree Evolution. Fig. 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). 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, 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., $<_c$, $>_c$, $=_c$, $\geq_c$, $\leq_c$)[1], 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 [13, 49].

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

[1]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.

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 $\geq_c$, $\leq_c$, 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 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 [13, 49].

Hamming Tree Mapping Structure. Fig. 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 3.2.)

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

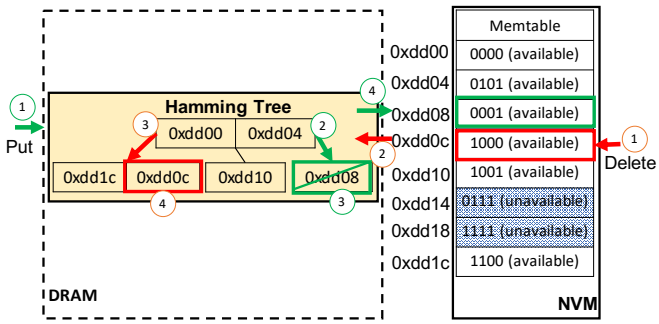


Fig. 7. An example of procedures which serve key-value PUT and DELETE operations in Hamming Tree.

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.

3.2 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. 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. 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. 7 shows an example of how a PUT request leads to removing an

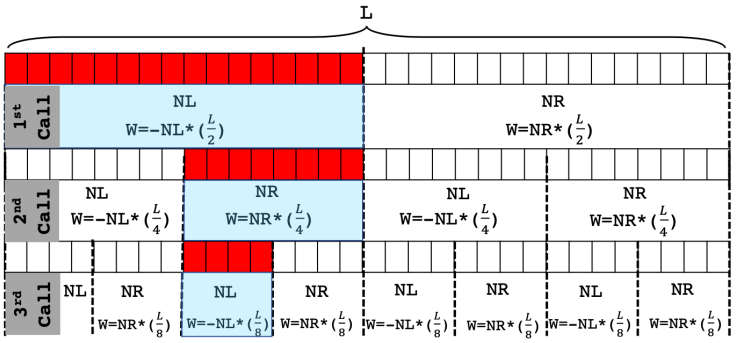


Fig. 8. Hamming Tree's compaction strategy with three calls (the highlighted areas indicate the half with higher density of 1's).

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. 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).

3.3 Compact Content Representation

3.3.1 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

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	
[1,0,0,0,1,0,0,0,0,0,1,0,1,1]	[1,0,0,0,1,0,0,0,0,0,1,0,1,1]			[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	

Fig. 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

Fig. 10. The compacted values for all the 4-bit inputs.

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.

3.3.2 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. 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. 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:

$$T(A, L, H, n) = W(A, L, H) \times \frac{n}{2} + \begin{cases} \text{T(A, L, } \lfloor \frac{L+H}{2} \rfloor, \frac{n}{2}) & \text{if W<0} \\ \text{T(A, } \lceil \frac{L+H}{2} \rceil \text{, H, } \frac{n}{2}) & \text{if W}\geq 0 \end{cases} \quad (1)$$

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:

$$W(A, L, H) = \begin{cases} N_R & \text{if } N_R \text{ - } N_L > 0 \\ -N_L & \text{if } N_R \text{ - } N_L < 0 \\ 0 & N_R \text{ - } N_L = 0 \end{cases} \quad (2)$$

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

Fig. 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. 9. Based on Equation. 2, W(A, 0, 16) = N_R - N_L = 1 - 6 = -5 < 0, so, in the first call, Equation. 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. 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.

3.3.3 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. 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).

$$T(n) = W \times \frac{n}{2} + T(\frac{n}{2}) \quad (3)$$

Now, based on Equation. 2, because the maximum value of W for an item of size n is $\frac{n}{2}$, Equation. 3 can be re-written as follows:

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

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

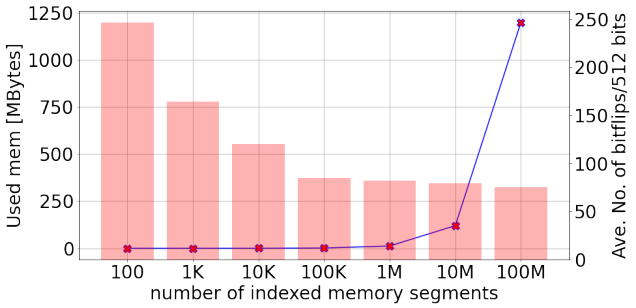


Fig. 11. Hamming Tree's memory consumption (line graph) and performance (bar graph) for indexing different numbers of memory segments.

$$T(n) \leq \frac{n^2}{4} + \frac{n^2}{16} + \frac{n^2}{64} + ... + T(1) = \sum_{i=1}^{\log_2 n} \frac{n^2}{4^i} \tag{5}$$

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

$$T(n) \leq \frac{n^2 - 1}{3} \tag{6}$$

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 $O(n^2)$. Such a value can be stored using only $O(\log n)$ 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.

3.3.4 Total order. In Section 3.1 (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 strategy, each memory segment is represented as a number that is calculated using the function T presented in Equation 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.

3.4 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. 11 illustrates the amount of memory that Hamming Tree uses for indexing different numbers of memory segments for PubMed data set [17]. 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. 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.

3.5 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 [18] augmented with Hamming Tree.

3.5.1 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 [4, 29, 48].

3.5.2 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.

4 EXPERIMENTS

4.1 Methodology

There are many types of phase change material-based technologies, such as Intel® Optane™ Memory Series [27]. In this paper, 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 Fig. 2, 13, 17, 18, and 19. 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 [12]. In this paper, we use an energy profiler named Perf, which is a performance analysis tool and a part of the Intel's RAPL interface [20, 36]. we measured the energy and power consumption of the memory (both DRAM and PMEM), while running our tests using the perf [14] 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.

4.2 Overview and setup

We compare with two main groups of solutions: 1) persistent K/V stores that use specialized data structures for NVM [29, 31, 38, 44, 59]. These methods generally focus on reducing write

amplification. 2) hardware-based bit flip optimization methods that use the RBW technique [10, 28, 39, 56]. 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 [29, 31, 38, 44, 59]. 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 [10, 28, 39, 56]. 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 [39]. With respect to Captopril, we also considered its best case, which happens when the blocks are partitioned into n = 16 segments [28].

4.2.1 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 [11], 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 paper, 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 [17] is the first data set that we used in this paper. 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 [21, 35], that contains 434874 entries of road networks information of North Jutland, Denmark. Finally, the last data 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 [17], consists of 730 million words in total.

4.2.2 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 [29] and HiKV [55]). 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 [38]. The method in [29], which is called NoveLSM, is a persistent LSM-based K/V storage systems which is designed to utilize NVM to provide low latency and high throughput. HiKV [55] is another persistent key-value store with the central idea of constructing a

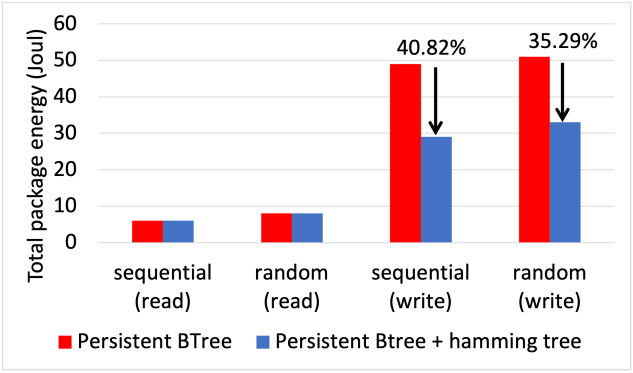


Fig. 12. The energy consumption of persistent BTree before and after being augmented by Hamming Tree.

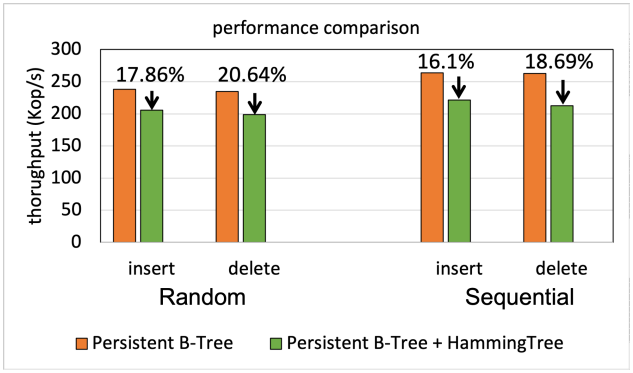


Fig. 13. The throughput of persistent BTree before and after being augmented by Hamming Tree.

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 [44] and wBTree [9]). The second data structure that is used widely in K/V data stores is B+-Trees [24]. FP-Tree [44] 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 [9] 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 [59]). 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 [59], which is designed specifically for NVMs.

4.2.3 Experiments setup. In this paper, we evaluated latency and energy consumption on a real Intel Optane memory device. For this purpose, we used the Persistent Memory Development Kit (PMDK) [46]. 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

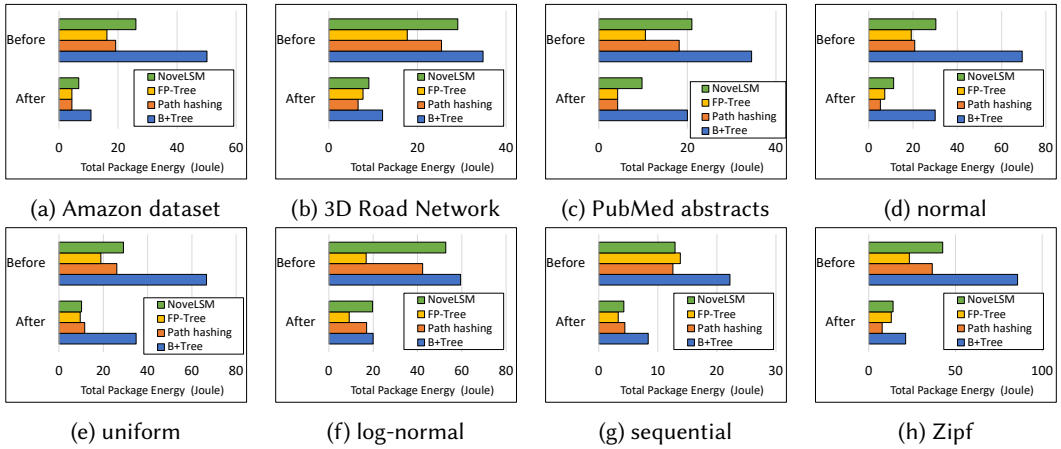


Fig. 14. The average amount of energy consumed for writing one memory segment for different data structures using different data sets before and after being augmented by Hamming Tree.

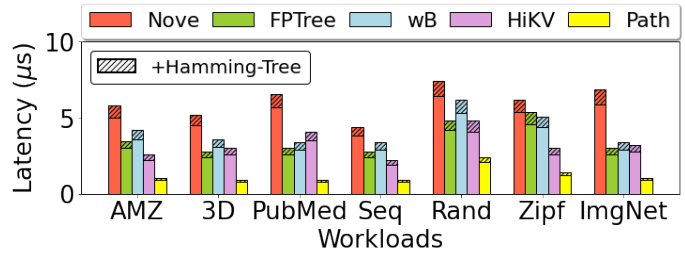
in this area [6, 31, 51, 52, 57]. Similar to these works, we have adopted previously used methods to emulate NVM [43, 53, 55] and we added an emulation of memory controller. The emulated memory controller utilizes a wear-leveling algorithm that is inspired from prior literature on wear leveling [51]. 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. In this paper, we set the value of ψ to 8 for all the emulation experiments.

4.3 Results

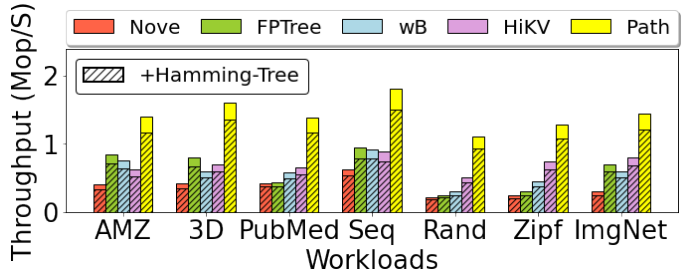
In the first experiment, we tested a persistent B-Tree before and after augmenting with Hamming Tree on a real Intel Optane memory device. The results in Fig. 12 shows that by plugging Hamming Tree, the system consumes up to 41% less energy. This validates the objective of Hamming Tree and its potential to reduce energy consumption. Fig. 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. 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. 15 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.



(a) latency



(b) throughput

Fig. 15. Write latency and throughput of different methods before and after being augmented by Hamming Tree.

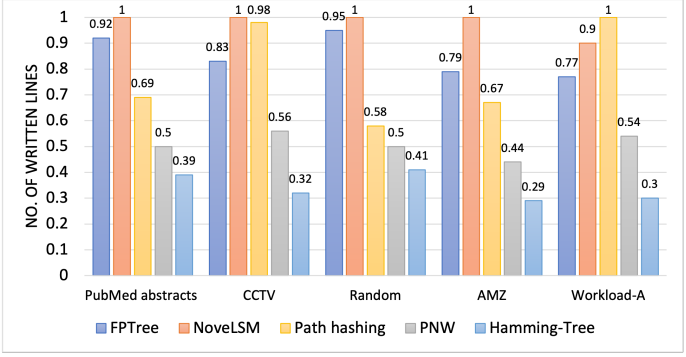


Fig. 16. The normalized number of cache lines per request.

In the next experiment, the average number of written cache lines per request is normalized for different data sets, as shown in Fig. 16. 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. 17 shows the amount of energy that is consumed in different

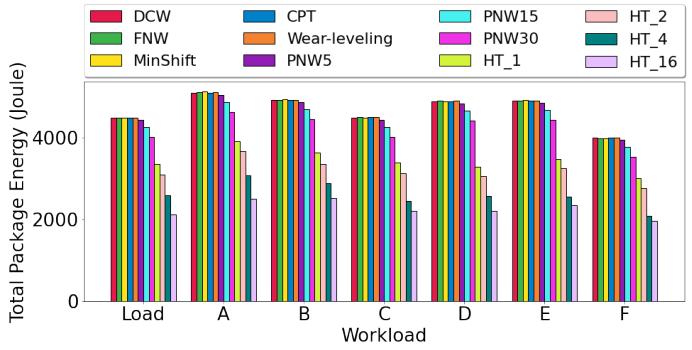
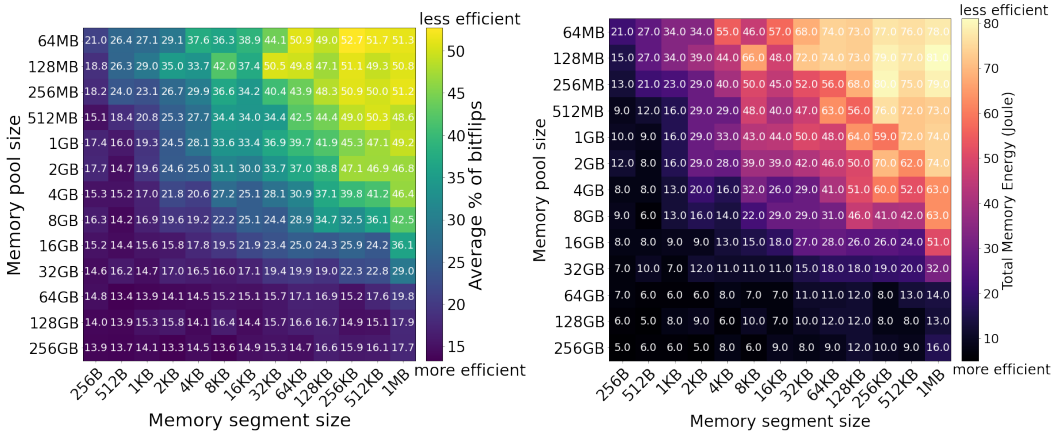


Fig. 17. The average amount of energy consumed for various methods.



(a) Average updated bits ratio

(b) Total memory energy (Joule)

Fig. 18. Hamming Tree's average updated bits ratio and total memory energy consumption (joule) for different memory segment and memory pool sizes.

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. 17 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 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

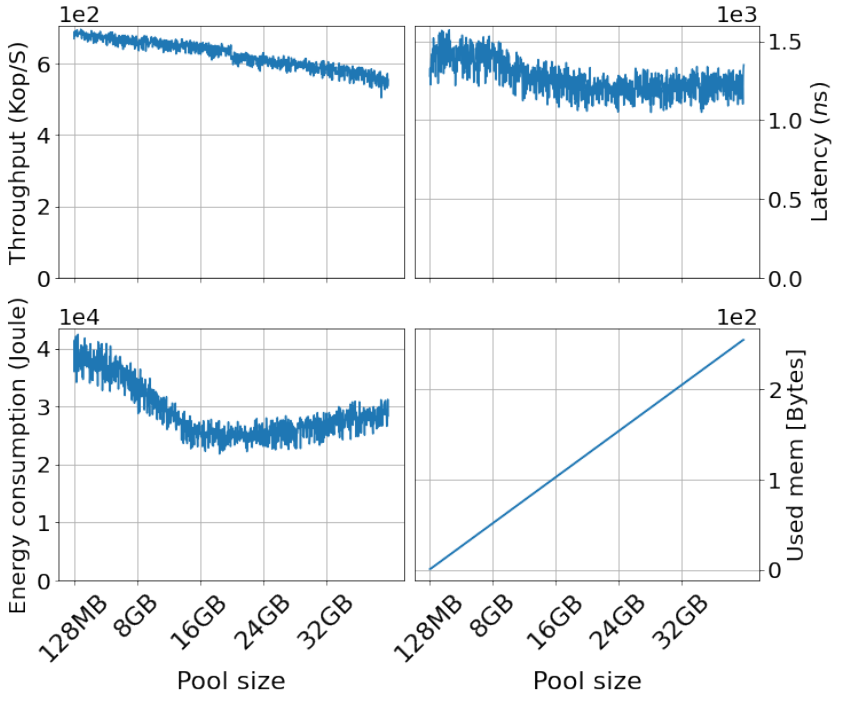


Fig. 19. The impact of Hamming Tree on key performance metrics of the system when the memory pool size changes.

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. 18 illustrates the overall performance of Hamming Tree in terms 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 paper. For the overall energy efficiency shown in Fig. 18b, 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. 18a. Based on this observation, we conclude that the smaller we choose the size of the memory segments, which we discussed in section 3.2, compared to the size of the data zone (pool size), the more tree nodes are available for Hamming Tree, which also results in fewer bit flips and less energy consumption.

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. 19, 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 "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

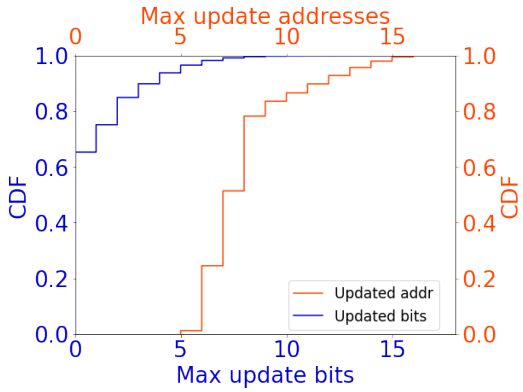


Fig. 20. The maximum update addresses and wear-leveling as CDFs by applying Hamming Tree.

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 [6]. 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 [6, 30, 31]. 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. 20 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. 20, 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. 21 shows the impact of employing Hamming Tree on PCM 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

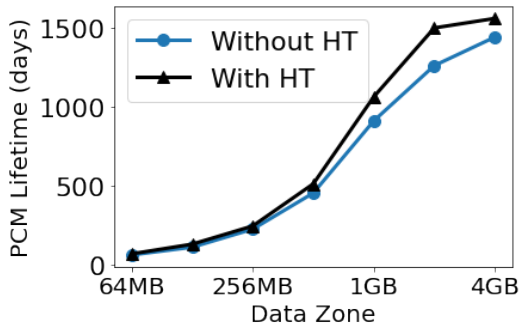


Fig. 21. PCM lifetime improvement before and after utilizing Hamming Tree for different data zone sizes.

PCM cell can be calculated as 227 days (=10^8/4 ×10^9 instructions × 1/2IPC × 1/2.6GHz × 1/(24hours × 60minutes))[50]. Furthermore, Fig. 21 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 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 paper, 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.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their insightful comments and feedback. This research is supported in part by the NSF under grant CNS-1815212.

REFERENCES

[1] Alaa Alameldeen and David Wood. 2004. *Frequent pattern compression: A significance-based compression scheme for L2 caches.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[2] Milad Andalibi, Mojtaba Hajihosseini, Sam Teymoori, Maryam Kargar, and Meysam Gheisarnejad. 2021. 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)*. IEEE, 1–6.

[3] Joy Arulraj et al. 2015. 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*. Association for Computing Machinery, Melbourne, Victoria, Australia, 707–722.

[4] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems*. Association for Computing Machinery, Rome , Italy, 80–94.

[5] Sana Benhamaid, Abdelmadjid Bouabdallah, and Hicham Lakhlef. 2022. Recent advances in energy management for Green-IoT: An up-to-date and comprehensive survey. *Journal of Network and Computer Applications* 198 (2022),

103257.
[6] Daniel Bittman et al. 2019. Optimizing Systems for Byte-Addressable {NVM} by Reducing Bit Flipping. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. USENIX, Boston, MA, United States, 17–30.
[7] Maurizio Capra, Riccardo Peloso, Guido Masera, Massimo Ruo Roch, and Maurizio Martina. 2019. Edge computing: A survey on the hardware requirements in the internet of things world. *Future Internet* 11, 4 (2019), 100.
[8] Yuezhi Che, Yuanzhou Yang, Amro Awad, and Rujia Wang. 2020. A Lightweight Memory Access Pattern Obfuscation Framework for NVM. *IEEE Computer Architecture Letters* 19, 2 (2020), 163–166.
[9] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.
[10] Sangyeun Cho and Hyunjin Lee. 2009. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *MICRO 2009*. Association for Computing Machinery, New York, USA, 347–357.
[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. Association for Computing Machinery, Indiana, USA, 143–154.
[12] Luís Cruz. 2021. Tools to Measure Software Energy Consumption from your Computer. http://luiscruz.github.io/2021/07/20/measuring-energy.html. https://doi.org/10.6084/m9.figshare.19145549.v1 Blog post..
[13] Brian A Davey and Hilary A Priestley. 2002. *Introduction to lattices and order*. Cambridge university press, Cambridge, United Kingdom.
[14] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. Linux Kongress, Nuremberg, Germany, 1–42.
[15] David B Dgien et al. 2014. Compression architecture for bit-write reduction in non-volatile memory technologies. In *NANOARCH 2014*. IEEE, Association for Computing Machinery, Paris, France, 51–56.
[16] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. 2009. PDRAM: A hybrid PRAM and DRAM main memory system. In *2009 46th ACM/IEEE Design Automation Conference*. IEEE, IEEE, San Francisco California, 664–669.
[17] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/Amazon+Access+Samples
[18] Sanjay Ghemawat and Jeff Dean. 2022. *Google LevelDB*. Google. Retrieved Jan 2, 2022 from https://github.com/google/leveldb
[19] Binbin Gu, Saeed Kargar, and Faisal Nawab. 2022. Efficient dynamic clustering: Capturing patterns fromhistorical cluster evolution. *arXiv preprint arXiv:2203.00812* (2022).
[20] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011), 0–40.
[21] Chenjuan Guo et al. 2012. Ecomark: evaluating models of vehicular environmental impact. In *SIGSPATIAL '12*. Association for Computing Machinery, Redondo Beach California, 269–278.
[22] Yuncheng Guo, Yu Hua, and Pengfei Zuo. 2018. DFPC: A dynamic frequent pattern compression scheme in NVM-based main memory. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, IEEE, DRESDEN, GERMANY, 1622–1627.
[23] Fazal Hameed et al. 2017. Efficient STT-RAM last-level-cache architecture to replace DRAM cache. In *MEMSYS 2017*. Association for Computing Machinery, Virginia, USA, 141–151.
[24] Jiangkun Hu et al. 2020. Understanding and analysis of B+ trees on NVM towards consistency and efficiency. *CCF Transactions on High Performance Computing* 2, 1 (2020), 1–14.
[25] Fangting Huang, Dan Feng, Wen Xia, Wen Zhou, Yucheng Zhang, Min Fu, Chuntao Jiang, and Yukun Zhou. 2016. Security RBSG: Protecting phase change memory with security-level adjustable dynamic mapping. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, Chicago, Illinois USA, 1081–1090.
[26] Jianming Huang, Yu Hua, Pengfei Zuo, Wen Zhou, and Fangting Huang. 2020. An Efficient Wear-level Architecture using Self-adaptive Wear Leveling. In *49th International Conference on Parallel Processing-ICPP*. Association for Computing Machinery, New York, USA, 1–11.
[27] Intel. 2022. *Types of Intel Optane memory*. Intel. Retrieved Jan 1, 2023 from https://www.intel.com/content/www/us/en/support/articles/000058286/memory-and-storage/intel-optane-memory.html
[28] Majid Jalili and Hamid Sarbazi-Azad. 2016. Captopril: Reducing the pressure of bit flips on hot locations in non-volatile main memories. In *DATE 2016*. IEEE, IEEE, Dresden, Germany, 1116–1119.
[29] Sudarsun Kannan et al. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. USENIX, Boston, MA, USA, 993–1005.
[30] Saeed Kargar, Binbin Gu, Sangeetha Abdu Jyothi, and Faisal Nawab. 2023. E2-NVM: A Memory-Aware Write Scheme to Improve Energy Efficiency and Write Endurance of NVMs using Variational Autoencoders. (2023).
[31] Saeed Kargar, Heiner Litz, and Faisal Nawab. 2021. Predict and write: Using k-means clustering to extend the lifetime of nvm storage. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 768–779.

[32] Saeed Kargar and Faisal Nawab. 2021. Extending the lifetime of NVM: challenges and opportunities. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3194–3197.
[33] Saeed Kargar and Faisal Nawab. 2021. Hamming Tree: The Case for Memory-Aware Bit Flipping Reduction for NVM Indexing.. In *CIDR*.
[34] Saeed Kargar and Faisal Nawab. 2022. Challenges and future directions for energy, latency, and lifetime improvements in NVMs. *Distributed and Parallel Databases* (2022), 1–27.
[35] Manohar Kaul et al. 2013. Building accurate 3d spatial networks to enable next generation intelligent transportation systems. In *MDM 2013*, Vol. 1. IEEE, IEEE, Milan, Italy, 137–146.
[36] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 2 (2018), 1–26.
[37] Wenjie Li et al. 2020. HiLSM: an LSM-based key-value store for hybrid NVM-SSD storage systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*. Association for Computing Machinery, Catania, Sicily, Italy, 208–216.
[38] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.
[39] Xianlu Luo et al. 2014. Enhancing lifetime of NVM-based main memory with bit shifting and flipping. In *RTCSA 2014*. IEEE, IEEE, Chongqing, China, 1–7.
[40] Haiyu Mao, Xian Zhang, Guangyu Sun, and Jiwu Shu. 2017. 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*. IEEE, IEEE, LAUSANNE, SWITZERLAND, 1623–1626.
[41] Sparsh Mittal and Jeffrey S Vetter. 2015. A survey of software techniques for using non-volatile memories for storage and main memory systems. *TPDS 2015* 27, 5 (2015), 1537–1550.
[42] Fargol Nematkhah, Farrokh Aminifar, Mohammad Shahidehpour, and Sasan Mokhtari. 2022. 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 (2022), 8–20.
[43] Jiaxin Ou et al. 2016. A high performance file system for non-volatile main memory. In *EuroSys '16*. Association for Computing Machinery, London, United Kingdom, 1–16.
[44] Ismail Oukid et al. 2016. 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*. Association for Computing Machinery, San Francisco, USA, 371–386.
[45] Poovaiah M Palangappa and Kartik Mohanram. 2015. Flip-Mirror-Rotate: An architecture for bit-write reduction and wear leveling in non-volatile memories. In *GLSVLSI 2015*. IEEE, Pennsylvania, USA, 221–224.
[46] PMDK. accessed: 2022-02. Persistent Memory Development Kit. https://pmem.io.
[47] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. 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)*. IEEE, IEEE, New York, USA, 14–23.
[48] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. USENIX, Shanghai, China, 497–514.
[49] Gunther Schmidt and Thomas Ströhlein. 2012. *Relations and graphs: discrete mathematics for computer scientists*. Springer Science & Business Media, Berlin.
[50] Dongsuk Shin, Hakbeom Jang, Kiseok Oh, and Jae W Lee. 2022. An Energy-Efficient DRAM Cache Architecture for Mobile Platforms With PCM-Based Main Memory. *ACM Transactions on Embedded Computing Systems (TECS)* 21, 1 (2022), 1–22.
[51] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. 2020. Improving phase change memory performance with data content aware access. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*. Association for Computing Machinery, New York, United States, 30–47.
[52] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. 2021. Aging-aware request scheduling for non-volatile main memory. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. IEEE, New York, United States, 657–664.
[53] Haris Volos et al. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.
[54] Yasi Wang, Hongxun Yao, and Sicheng Zhao. 2016. Auto-encoder based dimensionality reduction. *Neurocomputing* 184 (2016), 232–242.
[55] Fei Xia et al. 2017. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *USENIX ATC 17*. USENIX, Santa Clara, CA, USA, 349–362.

[56] Byung-Do Yang et al. 2007. A low power phase-change random access memory using a data-comparison write scheme. In *ISCAS 2007*. IEEE, IEEE, New Orleans, LA, USA, 3014–3017.
[57] Qi Zeng and Jih-Kwon Peir. 2017. Content-aware non-volatile cache replacement. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, Orlando, Florida USA, 92–101.
[58] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. *ACM SIGARCH computer architecture news* 37, 3 (2009), 14–23.
[59] Pengfei Zuo and Yu Hua. 2017. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2017), 985–998.

Received October 2022; revised January 2023; accepted February 2023