

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

Saeed Kargar
UC Santa Cruz
skargar@ucsc.edu

Heiner Litz
UC Santa Cruz
hlitz@ucsc.edu

Faisal Nawab
UC Santa Cruz
fnawab@ucsc.edu

Abstract— Non-volatile memory (NVM) technologies suffer from limited write endurance. To address this challenge, 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 decreases the number of bit flips for PUT/UPDATE operations by determining the best memory location an updated value should be written to. PNW leverages the indirection level of K/V-stores to freely choose the target memory location for any given write based on its value. PNW organizes NVM addresses in a *dynamic address pool* clustered by the similarity of the data values they refer to. We show that, by choosing the right target memory location for a given PUT/UPDATE operation, the number of total bit flips and cache lines can be reduced by up to 85% and 56% over the state of the art.

Index Terms—hybrid DRAM-NVM, write endurance, K-means clustering, bit flips

I. INTRODUCTION

In recent years, there has been a growing interest in Non-Volatile Memory (NVM)—such as Phase-Change Memory (PCM)—due to their unique characteristics, including non-volatility, high density, high scalability, and byte addressability. However, these emerging NVMs also pose a number of challenges: They have limited write endurance and asymmetric read/write access properties, requiring special treatment when deployed in large scale computing systems [1]. While DRAM’s write endurance (the number of writes that can be applied to a block of storage media before it becomes unreliable) is on the order of 10^{15} writes, NVM technologies, such as PCM, can be written only up to 10^8 – 10^9 times [2].

The limited endurance of PCM means that cells can only be written a limited number of times before they “wear out”. Some recent technologies, such as Intel’s Optane DIMM [3], aim to increase the endurance of NVMs significantly. However, unlike other non-volatile technologies such as flash, PCM cells are written on the byte of cache line granularity instead of the page granularity leading to uneven wear-out even on the sub page-level. To ensure failure-atomicity for the data structures stored in NVMs, software schemes, such as logging [4] and shadowing [5], are used. This causes extra overheads in terms of write amplification due to writing log entries or creating additional copies of the data [6]. Even in modern NVM devices, such as Intel’s 3D XPoint—where it is claimed that performance is unaffected by the number of modified words in a cache line—it is beneficial to reduce

the number of write operations, to improve endurance and retention.

There exists many proposals to increase the write endurance of NVM storage. One promising approach is the Read-Before-Write (RBW) technique, in which the content of an old memory block is read before it is overwritten with the new data. This technique replaces each NVM 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. Other proposed methods, such as [6], [7], overcome the limitations of NVMs by designing data structures that decrease write amplification.

However, prior methods are either (1) *application-agnostic* without the ability to leverage the write and data patterns of applications, or (2) *specialized solutions* that are built for the write and data patterns of specific applications. Particularly, application-agnostic solutions such as FNW [8] and NVM data structures [6], [7], do not leverage the write and data patterns of the application and miss the opportunity to judiciously place writes on memory locations that would minimize bit flips. On the other hand, specialized solutions such as [9], try to minimize the number of bit flips via fixed bit masks that target specific predefined workloads. This renders these solutions limited to predefined applications limiting performance for applications with dynamically changing write patterns.

In this paper, we propose Predict and Write (PNW), 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 VI-A).

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

TABLE I: Comparison of memory technologies [10], [11]

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

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 V-A). 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 [12].

II. BACKGROUND

A. Non-Volatile Memory Technologies

Emerging Non-Volatile Memory (NVM) technologies, such as Phase-Change Random Access Memory (PCRAM, PRAM, or PCM) and Resistive Memory (ReRAM), provides fast persistent storage, significantly outperforming traditional Disk and Flash technologies. Table I shows the performance characteristics of some prevalent memory technologies. While NVM provides similar read latency to DRAM, its write latency is higher than DRAM and thus, minimizing write operations becomes critical for designing software systems on top of NVMs. For this work, we assume a hybrid memory architecture, where both DRAM and PCM exist on the same main memory level, managed under a single physical address space [13]. Although PNW can support other memory architectures, it is designed to work on hybrid memory systems, in which case NVM acts as a fast persistent memory directly connected to the memory bus.

A PCM write operation demands significantly more current and power than a read operation. This property is of great importance in systems like mobile systems, even requiring them to support “iterative writing” of data units of smaller sizes than memory words to limit the instantaneous current. For example, in [14] and [15], the write modes of $\times 2$, $\times 4$, and $\times 8$ are supported instead of faster modes like $\times 16$. As another example, in [16], the serial writing of even one bit at a time is supported. Integrating NVMs into existing computer systems requires to develop new NVM-friendly data structures [17] that focus on special properties such as reducing write amplification [7], or being lock-free [18], [19]. For instance, the techniques proposed in [7], [20], [21] target the reduction of write amplification, leading to the reduction of bit flips.

However, these methods lead to the increase in wear-out cost mostly because of overlooking the reduction of bit flips at the expense of the reduction of write amplification. We show in Section VI that this problem can also lead to the increase in the number of written cache lines compared to PNW, which focuses on minimizing bit flips.

III. RELATED WORK

Although recent methods have been able to address write endurance by reducing the number of bit flips through the RBW technique and specialized NVM data structures, they leave out significant opportunities for reducing additional bit flips: (1) application-agnostic methods that do not leverage the write and data patterns including RBW based techniques [8], [22], [23] and NVM-based structures that aim to reduce write amplification [6], [7] miss the opportunity of using existing patterns in the stored data to minimize bit flips. Specifically, writes are generally updated in place, whereas our proposed technique determines the target memory location based on the written data values (2) specialized methods that are designed for specific workloads, such as Captopril [9], decrease the number of bit flips via fixed bit masks. These methods only work on specific workloads and suffer from significant overheads. In particular, the bit-masks are storage space intensive themselves and, furthermore, as the bit masks are determined once, the approach cannot adapt to changing workloads.

In [23], the authors propose DCW to find common patterns and then compress data to reduce the number of bit flips in SCM. Like FNW, DCW replaces a write operation with an RBW process. MinShift [22] proposes a method to reduce the total number of updated 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. Captopril [9] is another method that reduces the total number of bit flips by masking specific hot locations that are flipped more than others. However, as we will see in section VI, this method suffers from relatively high overhead. More importantly, it is specialized and would only work on a predefined application.

Finally, there is a group of techniques that find the similarity between items through Locality Sensitive Hashing (LSH) [24], [25]. In this technique, each item is transformed to a hash fingerprint (usually using minhash), and later, LSH is applied to it. Since LSH does not preserve the bit-wise similarity among items, it cannot be efficiently used for bit-wise similarity clustering, which is the main purpose of PNW.

IV. 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

TABLE II: 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'

	Content
Old item	0xABCD23B, BCDABCD, BCDAB01A ... 1CDA23BA, CCDAAB00, ACCDAB01
New item	0xABCDAB01, BCDABCD, BCDAB01A ... 1CDA23BA, CCDAAB00, ABCDAB08

Fig. 1: An example of a memory content that is going to be replaced by a new item with close hamming distance in PNW.

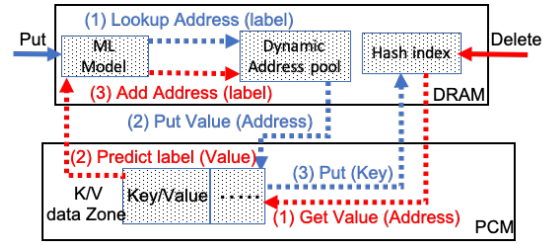
write operation in the right memory location that minimizes the hamming distance between the old and the new data, the number of bit flips can be significantly reduced. While promising for reducing bit flips, this technique introduces several challenges. First, it requires an indirection layer to map 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.

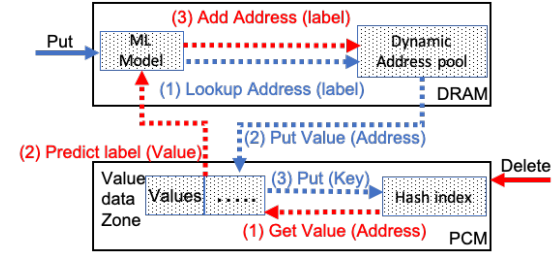
The intuition behind our clustering approach is that we cluster similar memory locations in terms of the bit patterns of their contents. Using this clustering, we can quickly retrieve a new memory location for a PUT operation such that the hamming distance between the new to-be-written data and the old 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 VI-F.

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 II). 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



(a) Proposed architecture for small keys



(b) Proposed architecture for large keys

Fig. 2: An example of procedures which serve K/V PUT and DELETE operations for a) small and b) large keys.

hamming distance between the new data and existing free memory locations maintained in the 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 II 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. 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

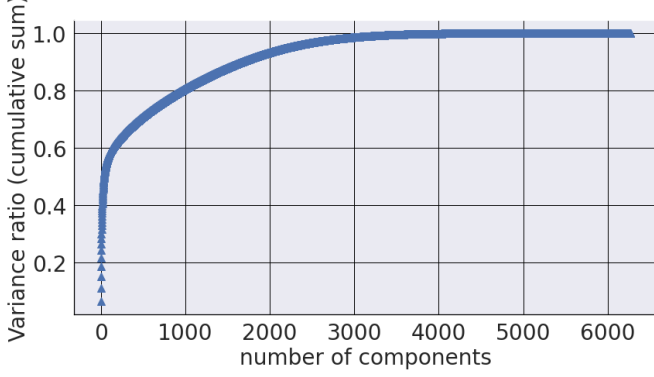


Fig. 3: PCA variance ratio according to the number of principal components.

example, suppose that the page size in our system is 4KB as shown in Figure. 1. 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 is a cache line). This enables PNW to decrease NVM word writes in addition to NVM line writes.

V. 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.

A. 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. 2, 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.

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 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.

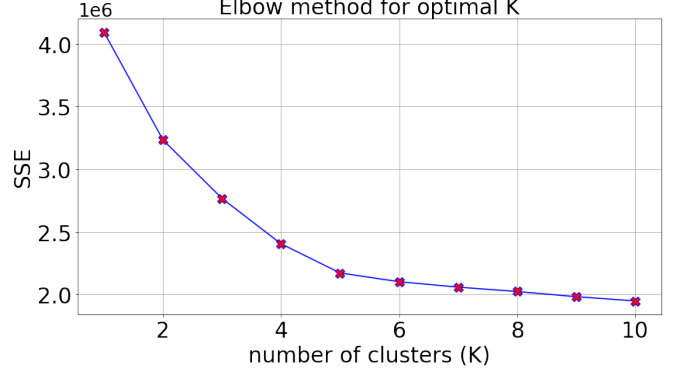


Fig. 4: Sum of Square Error graph to find the optimal K.

Addressing the Curse of Dimensionality To tackle the curse of dimensionality problem, we use Principal Component Analysis (PCA) on the data sets used in this paper 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 [26]–[29]. 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 [30]. Figure. 3 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 [31]. In this work, we use one of the most common techniques called the “elbow method” [32]–[34]. The elbow method is expressed as the following Sum of Squared Error (SSE) [33]:

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

where $\|\cdot\|_2$ 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 $|C_i|$, $\Pi = \{C_1, C_2, \dots, C_K\}$, and $X = \{x_1, \dots, x_i, \dots, x_N\}$ (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 known as the “elbow” or “knee”, which suggests the optimal value for K [33]–[35]. Figure. 4 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).

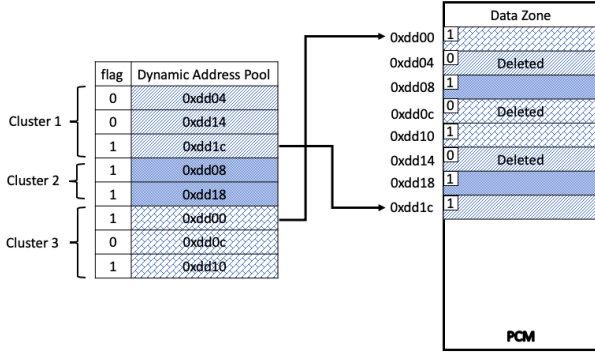


Fig. 5: Dynamic Address Pool.

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.

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. 5). 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 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. V-C.

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) *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 2b), 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 [20].
- Another design choice is to build the indexing structure on DRAM (Figure 2a). 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 [20]. 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.

B. Supported K/V Operations

1) *PUT Operation*: PUT and UPDATE operations are executed as follows. As shown in Figure 2, 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 2). 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 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).

2) *DELETE operation*: Algorithm 3 illustrates PNW’s delete operation (also see Figure 2). 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) *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

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}
```

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.

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.

C. 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.

VI. EVALUATION

A. 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 VI-F). 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 III, such as FNW [8], DCW [36], Captopril [9] and MinShift [22]. 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 [22]. With respect to Captopril, we

also considered its best case, which happens when the blocks are partitioned into $n = 16$ segments [9].

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 [37]–[40]. We assume an access latency of the latest 3D-XPoint of 600ns [41], [42].

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 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.

B. Real-world textual and numerical data sets

The first data set is called Amazon Access Samples Data Set [43], [44], 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 “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. 6a 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 [45], [46], contains information of road networks in North Jutland, Denmark (covering a region of $185 \times 135 \text{ km}^2$). 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. 6b. 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 [44], 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

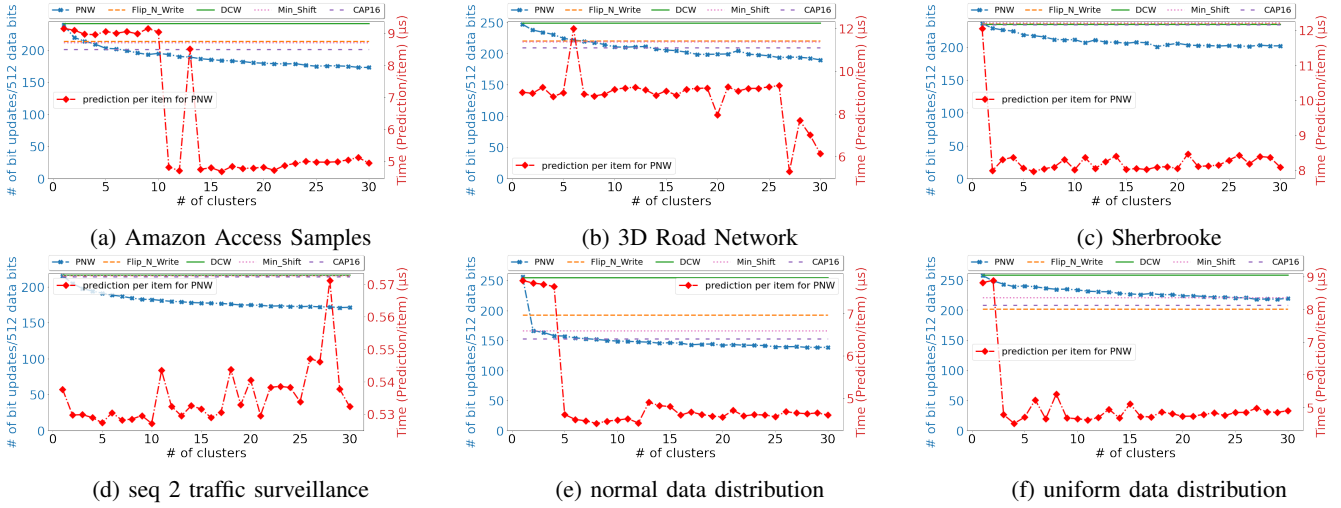


Fig. 6: The average number of actual bit updates per writing 512 bits as well as the latency of prediction per item in PNW for the real-world textual and numerical data sets (a-b), multimedia data sets (c-d), and synthetic data sets (e-f).

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.

C. 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 [47], representing a two-minute-long video (with resolution 800x600). 2) A Traffic Surveillance video [48], 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. 6c and 6d, 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 [49]. 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.

D. 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. 6e 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 $5\mu s$ to $6\mu s$ 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

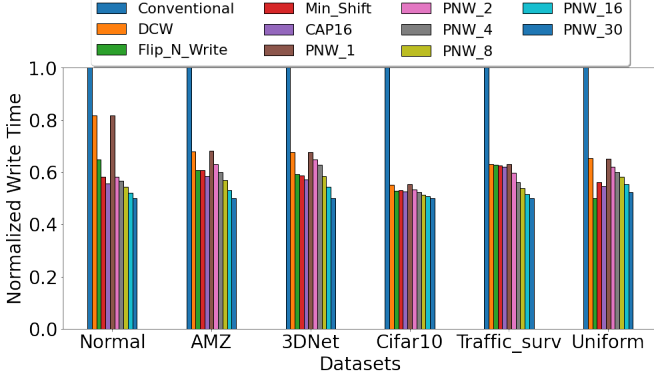


Fig. 7: End-to-end write latency comparison for various data sets.

learn using an ML model. The results are depicted in Figure. 6f 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.

E. 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. 7, we show the write latency comparisons for various data sets. In 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. 7 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

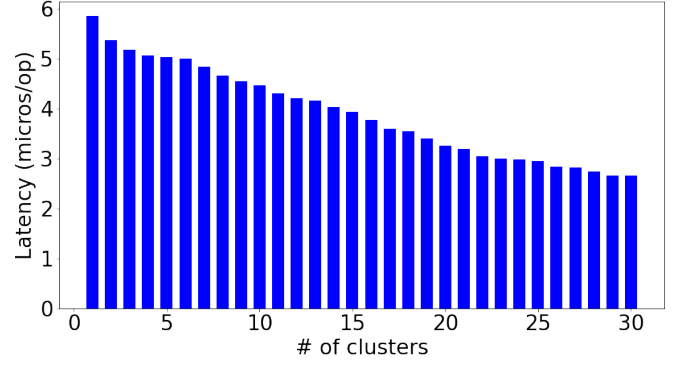


Fig. 8: The impact of choosing the number of clusters (K) on the average write latency for the PubMed abstracts data set.

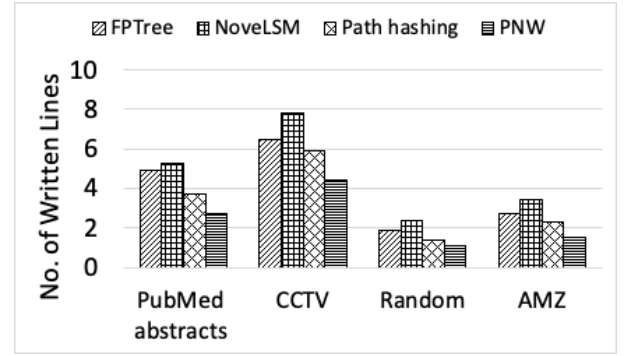


Fig. 9: The average number of written cache lines for each request.

to find a clear pattern among the data items to make up the extra steps.

Figure. 8 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 [21] is a hybrid SCM-DRAM persistent B+-Tree method that we implement and compared PNW with. The second persistent K/V store that we compare PNW with is NoveLSM [7], 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

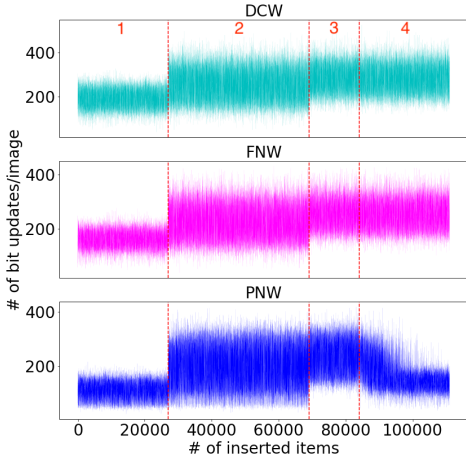


Fig. 10: The performance change by converting the workload from MNIST into Fashion-MNIST over time.

high throughput to applications. We also implement a hashing scheme that is designed for NVMs called Path hashing [20]. It is worth noting that for this test, we implement PNW as shown in Figure. 2a.

Figure. 9 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 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.

F. 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. 10. 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. 10, 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. 11). These experiments are performed on the traffic surveillance [48] and the Sherbrooke video data sets [47].

In this test (Figure. 11), 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. 11d), 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.

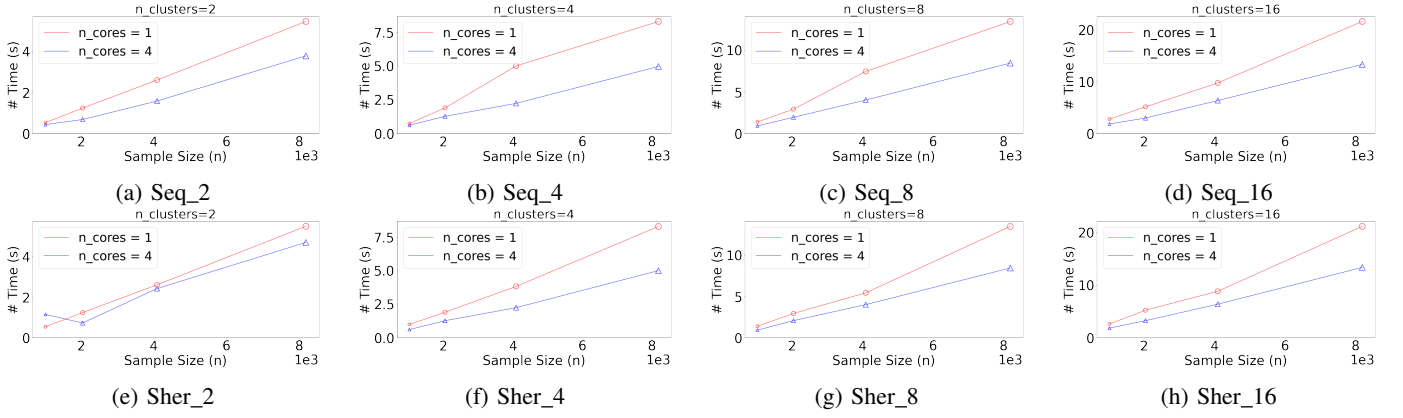


Fig. 11: PNW's average model training time for different data sets using single core versus multi-core processing.

G. 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 [2], [50]. 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. 12 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. 12a, 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% (Figure. 12a) and 86% (Figure. 12b) 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. 13 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. 13a), this likelihood rises to 98% when $k=30$ (Figure. 13b). This important observation shows an interesting fact about PNW: By increasing the number of

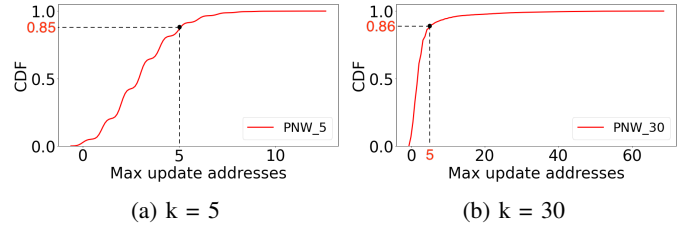


Fig. 12: The maximum update addresses as CDFs by applying PNW with a) $k=5$ and b) $k=30$ clusters.

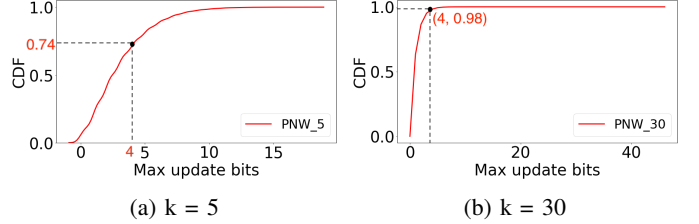


Fig. 13: Wear-leveling as CDFs by applying PNW with a) $k=5$ and b) $k=30$ clusters.

clusters, bit flips are distributed more evenly 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.

VII. CONCLUSION

In this paper, 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 approach to extend the lifetime of NVMs using machine learning. 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.

VIII. ACKNOWLEDGMENTS

This research is supported in part by the NSF under grants CCF-1942754 and CNS-1815212.

REFERENCES

- [1] B. C. Lee *et al.*, “Architecting phase change memory as a scalable dram alternative,” in *ISCA*, 2009, pp. 2–13.
- [2] S. Mittal and J. S. Vetter, “A survey of software techniques for using non-volatile memories for storage and main memory systems,” *TPDS* 2015, vol. 27, no. 5, pp. 1537–1550, 2015.
- [3] “optane-persistent-memory-200-series-brief,” <https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html>, Intel.
- [4] J. Coburn *et al.*, “Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.
- [5] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, “Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 836–848.
- [6] J. Liu *et al.*, “Lb+ trees: optimizing persistent index performance on 3dpoint memory,” *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.
- [7] S. Kannan *et al.*, “Redesigning lsms for nonvolatile memory with novelsm,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 993–1005.
- [8] S. Cho and H. Lee, “Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance,” in *MICRO* 2009, 2009, pp. 347–357.
- [9] M. Jalili and H. Sarbazi-Azad, “Captopril: Reducing the pressure of bit flips on hot locations in non-volatile main memories,” in *DATE 2016*. IEEE, 2016, pp. 1116–1119.
- [10] A. K. Kamath *et al.*, “Storage class memory: Principles, problems, and possibilities,” *arXiv preprint arXiv:1909.12221*, 2019.
- [11] A. van Renen *et al.*, “Persistent memory i/o primitives,” in *DaMoN’19*, 2019, pp. 1–7.
- [12] T. Kraska *et al.*, “The case for learned index structures,” in *SIGMOD 2018*, 2018, pp. 489–504.
- [13] G. Dhiman *et al.*, “P dram: A hybrid pram and dram main memory system,” in *2009 46th ACM/IEEE Design Automation Conference*. IEEE, 2009, pp. 664–669.
- [14] S. Kang, “A 0.1 μm 1.8 v 256mb 66mhz synchronous burst pram,” in *ISSCC 2006*, 2006.
- [15] W. Cho *et al.*, “A 90 nm 1.8 v 512 mb diode-switch pram with 266 mb/s read throughput,” in *ISSCC 2007*, 2007, pp. 26–1.
- [16] S. Hanzawa *et al.*, “A 512kb embedded phase change memory with 416kb/s write throughput at 100 μa cell write current,” in *ISSCC 2007*. IEEE, 2007, pp. 474–616.
- [17] F. Nawab *et al.*, “Dalí: A periodically persistent hash map,” in *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- [18] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. Morrey III, “Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience,” in *EDBT*, 2015, pp. 689–694.
- [19] F. Nawab, D. Chakrabarti, T. Kelly, and C. Morrey, “Zero-overhead nvram crash resilience,” in *Non-Volatile Memories Workshop*, 2015.
- [20] P. Zuo and Y. Hua, “A write-friendly and cache-optimized hashing scheme for non-volatile memory systems,” *TPDS*, vol. 29, no. 5, pp. 985–998, 2017.
- [21] I. Oukid *et al.*, “Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory,” in *SIGMOD ’16*, 2016, pp. 371–386.
- [22] X. Luo *et al.*, “Enhancing lifetime of nvram-based main memory with bit shifting and flipping,” in *RTCSA 2014*. IEEE, 2014, pp. 1–7.
- [23] D. B. Dgien *et al.*, “Compression architecture for bit-write reduction in non-volatile memory technologies,” in *NANOARCH 2014*. IEEE, 2014, pp. 51–56.
- [24] O. Chum *et al.*, “Near duplicate image detection: min-hash and tf-idf weighting,” in *BMVC*, vol. 810, 2008, pp. 812–815.
- [25] A. Ghasemazar, P. Nair, and M. Lis, “Thesaurus: Efficient cache compression via dynamic clustering,” in *ASPLOS ’20*, 2020, pp. 527–540.
- [26] H. Zha *et al.*, “Spectral relaxation for k-means clustering,” in *NIPS 2002*, 2002, pp. 1057–1064.
- [27] A. Y. Ng *et al.*, “On spectral clustering: Analysis and an algorithm,” in *NIPS 2002*, 2002, pp. 849–856.
- [28] I. T. Jolliffe and J. Cadima, “Principal component analysis: a review and recent developments,” *Philos. Trans. Royal Soc. A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, 2016.
- [29] C. Ding and X. He, “K-means clustering via principal component analysis,” in *ICML’04*, 2004, p. 29.
- [30] R. Cangelosi and A. Goriely, “Component retention in principal component analysis with application to cdna microarray data,” *Biology direct*, vol. 2, no. 1, p. 2, 2007.
- [31] M. E. Celebi *et al.*, “A comparative study of efficient initialization methods for the k-means clustering algorithm,” *Expert systems with applications*, vol. 40, no. 1, pp. 200–210, 2013.
- [32] K. D. Joshi and P. Nalwade, “Modified k-means for better initial cluster centres,” *IJCSMC 2013*, vol. 2, no. 7, pp. 219–223, 2013.
- [33] M. 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*, vol. 336, no. 1. IOP Publishing, 2018, p. 012017.
- [34] T. S. Madhulatha, “An overview on clustering methods,” *arXiv preprint arXiv:1205.1117*, 2012.
- [35] L. Vendramin *et al.*, “Relative clustering validity criteria: A comparative overview,” *Statistical analysis and data mining: the ASA data science journal*, vol. 3, no. 4, pp. 209–235, 2010.
- [36] B.-D. Yang *et al.*, “A low power phase-change random access memory using a data-comparison write scheme,” in *ISCAS 2007*. IEEE, 2007, pp. 3014–3017.
- [37] J. Ou *et al.*, “A high performance file system for non-volatile main memory,” in *EuroSys ’16*, 2016, pp. 1–16.
- [38] H. Volos *et al.*, “Mnemosyne: Lightweight persistent memory,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.
- [39] J. Huang *et al.*, “Nvram-aware logging in transaction systems,” *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 389–400, 2014.
- [40] F. Xia *et al.*, “Hikv: A hybrid index key-value store for dram-nvm memory systems,” in *USENIX ATC 17*, 2017, pp. 349–362.
- [41] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, “Basic performance measurements of the intel optane dc persistent memory module,” *arXiv preprint arXiv:1903.05714*, 2019.
- [42] Y. Ni, S. Chen, Q. Lu, H. Litz, Z. Pang, E. L. Miller, and J. Wu, “Closing the performance gap between dram and pm for in-memory index structures,” no. UCSC-SSRC-20-01, May 2020.
- [43] “Amazon access samples data set,” <https://archive.ics.uci.edu/ml/datasets/AmazonAccessSamples>.
- [44] D. Dua and C. Graff, “UCI machine learning repository,” 2017. [Online]. Available: <http://archive.ics.uci.edu/ml>
- [45] M. Kaul *et al.*, “Building accurate 3d spatial networks to enable next generation intelligent transportation systems,” in *MDM 2013*, vol. 1. IEEE, 2013, pp. 137–146.
- [46] C. Guo *et al.*, “Ecomark: evaluating models of vehicular environmental impact,” in *SIGSPATIAL ’12*, 2012, pp. 269–278.
- [47] J.-P. Jodoin *et al.*, “Urban tracker: Multiple object tracking in urban mixed traffic,” in *WACV 2014*. IEEE, 2014, pp. 885–892.
- [48] C. H. Bahnsen and T. B. Moeslund, “Rain removal in traffic surveillance: Does it matter?” *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–18, 2018.
- [49] A. Krizhevsky, G. Hinton *et al.*, “Learning multiple layers of features from tiny images,” 2009.
- [50] F. Xia *et al.*, “A survey of phase change memory systems,” *Journal of Computer Science and Technology*, vol. 30, no. 1, pp. 121–144, 2015.