# Beyond Write-reduction Consideration: A Wear-leveling-enabled B$^+$-tree Indexing Scheme over an NVRAM-based Architecture

Dharamjeet, Tseng-Yi Chen, *Member, IEEE,* Yuan-Hao Chang, *Senior Member, IEEE,* Chun-Feng Wu, *Student Member, IEEE,* Chi-Heng Lee, and Wei-Kuan Shih, *Member, IEEE*

***Abstract*—Recently, non-volatile random-access memory (NVRAM) has been regarded as the most up-and-coming main memory technology in embedded and Internet-of-Thing (IoT) system due to its attractive features: Zero-static power consumption and high memory cell density. However, the endurance issue as a "nightmare" always haunts NVRAM system developers. Worse still, NVRAM's lifespan will wear out soon in embedded applications because their data management systems usually utilize an indexing scheme to maintain small data. Plus, an node structure within the indexing scheme will be frequently updated because of data creation and deletion. Therefore, many previous works rethink B$^+$-tree indexing scheme on an NVRAM-based system. The most previous studies focused on reducing the amount of write traffic to memory. Unfortunately, they are failed to extend NVRAM lifespan because their solution cannot evenly distribute the amount of write traffic to each memory cell. Additionally, prior solutions have not considered that all nodes within B$^+$-tree indexing structure have different update frequencies. Based on such the observation, this work proposes a wear-leveling-aware B$^+$-tree design, namely waB$^+$-tree, to consider the update frequency of each node within the B$^+$-tree structure, so as to evenly scatter the amount of write traffic to the NVRAM cells. According to our experiments, the proposed waB$^+$-tree shows the encouraging results of endurance improvement.**

***Index Terms*—NVRAM-based system, endurance, indexing scheme, B$^+$-tree structure**

## I. Introduction

As a B$^+$-tree indexing scheme is extensively applied to database and file systems [1, 9–11], it has been an efficient indexing scheme for data management on traditional storage devices. Nowadays, with the considerations of zero static power consumption and fast read/write performance, non-volatile random access memory (NVRAM) has been regarded as a promising candidate for replacing storage devices in embedded systems. A well-known critical issue with NVRAM is its short endurance. This issue will become fatal while a B$^+$-tree indexing scheme is adopted to NVRAM-based systems. This is because a B$^+$-tree indexing scheme will generate a massive amount of small write traffic to particular memory spaces when data are written, deleted, and updated to storage space. Although some prior works [1, 12–15] tried to decrease the amount of write traffic generated by the B$^+$-tree indexing scheme to NVRAM storage devices, they are failed to consider the wear-leveling issue in their design. This work is motivated by such observation to rethink the B$^+$-tree indexing scheme by jointly considering intranode and internode wear-leveling issues for exploiting the maximum utilization of NVRAM's endurance.

In the past decade, NVRAM has been a main-stream storage technology in embedded systems (e.g., Internet-of-Things and sensor devices) due to its fast I/O performance and non-volatility. Although NVRAM technology has abundant nice features, its Achilles's heel (i.e., endurance issue) restricts its developments and applications seriously. In order to prolong NVRAM's lifespan, many prior excellent studies proposed many write-reduction methodologies in file systems [5, 6] and database indexing scheme [1, 12, 13] to decrease the amount of write traffic to the NVRAM storage device. Unfortunately, these research works put their focus on write reduction but not wear-leveling issues on an NVRAM-based system. If the amount of write traffic is accumulated on some specific NVRAM memory cells, the NVRAM device will fail soon. Therefore, it is a significant problem: How to evenly distribute the amount of write traffic to memory cells within the NVRAM storage device.

The endurance problem will become severe in applying a B$^+$-tree indexing scheme is to the NVRAM-based system. This is because the B$^+$-tree indexing scheme generates many random and massive write requests to the NVRAM storage space during data insertion, deletion, and update operations. Worse still, the amount of write requests is unevenly distributed to NVRAM memory cells owing to the property of a B$^+$-tree structure. Speaking of the B$^+$-tree structure, all keys will be maintained in leafs, and the keys in internal nodes are just like a guide to indicate the final position of keys in the tree structure. As leaf nodes maintain all keys, the endurance of the leaf-node cells will get seriously hurt because of the amount of write traffic produced by all key operations (i.e., insertion, deletion, update, node split, and merge). This fact results in hot leafs and cold internal nodes. In this paper, we

Manuscript received July 2020.

Dharamjeet is with the Taiwan International Graduate Program in Social Networks and Human-Centered Computing, Institute of Information Science, Academia Sinica, Taipei, Taiwan. E-mail: dharamjeet@iis.sinica.edu.tw

T. Chen is with Department of Computer Science and Information Engineering, National Central University, Taoyuan, Taiwan. E-mail: tychen@g.ncu.edu.tw

Y.-H. Chang, C.-F. Wu, and C.-H. Lee are with the Institute of Information Science, Academia Sinica, Taipei, Taiwan. E-mail: {johnson, cfwu}@iis.sinica.edu.tw

W.-K. Shih is with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan.

Corresponding authors are Tseng-Yi Chen and Yuan-Hao Chang.

call such a phenomenon as the internode wear-leveling issue. Additionally, in order to increase the query efficiency of the $B^+$-tree structure, keys within a node should be kept in sorted order. However, keeping keys in the sorted sequence generates an intranode wear-leveling problem because of shifting keys in a node. Briefly, the lifespan of an NVRAM device would vanish soon due to internode and intranode wear-leveling issues while a $B^+$-tree indexing scheme is established on the NVRAM-based system. In other words, the whole lifespan of the NVRAM device will be dominated by the indexing area because all data operations (i.e., creation, deletion, and modification) generates key insertions, deletions, and data pointer updates to the indexing area. The size of the indexing area is much smaller than that of the data area on the NVRAM storage space.

Based on such observations, this study will be a pioneer research work to tackle the wear-leveling issue with the $B^+$-tree indexing scheme over an NVRAM-based system. In this work, we propose a wear-leveling-aware $B^+$-tree indexing scheme, namely waB$^+$-tree, to jointly consider the internode and intranode wear-leveling issues to extremely prolong the lifespan of NVRAM. The proposed waB$^+$-tree consists of a circular node structure, a node-based, and global wear-leveling strategies. In the waB$^+$-tree structure, the circular node can reduce the number of key shifts, and the node-based wear-leveling strategy can swap hot and cold entries within a node to resolve the intranode wear-leveling issue. Plus, the global wear-leveling design can adaptively replace hot with cold nodes during node allocation for resolving the internode wear-leveling issue. According to the experiments, the waB$^+$-tree can evenly distribute the amount of write traffic to each NVRAM's memory cell to thoroughly resolve the wear-leveling issue with a $B^+$-tree indexing scheme over NVRAM-based systems.

The contributions of this paper can be summarized as follows. First of all, this study originally discovers the internode and intranode wear-leveling issue on the NVRAM-based system because of a $B^+$-tree indexing scheme. Secondly, the proposed wa-$B^+$-tree not only evenly distributes but also reduces the amount of write traffic to NVRAM cells. Last and most importantly, according to the largest amount of write traffic to the memory cell, the waB$^+$-tree can prolong the NVRAM lifespan by 2-4 times, compared with the previous solutions.

The rest of this paper is organized as follows. Section II gives an introduction to the background and motivation. Section III details our waB$^+$-tree indexing scheme, and Section IV reveals the capability of the proposed indexing scheme. Finally, Section V concludes this work.

## II. Background and Motivation

### A. Low-power Memory Technology

As low power consumption is a necessary requirement for embedded and Internet-of-Thing (IoT) systems, non-volatile random access memory (NVRAM) is rising to prominence for low power system designs due to its zero-static power consumption. Unlike dynamic random access memory (DRAM), NVRAM does not need constant power for retaining data because its cell state will be permanently changed after the programming process. However, if an embedded (or IoT) hardware vendor applies the NVRAM technology to their devices, they need to conquer the natural defects (e.g., limited endurance and high write cost) in the NVRAM technology. As shown in Table I, the non-volatile technologies (e.g., phase-change memory (PCM) and Spin-Torque Transfer RAM (STT-RAM)) have comparable read performance but worse endurance and write performance, compared with DRAM. As a result, many system developers and researchers have extensively discussed and proposed solutions in reducing the amount of write traffic to NVRAM devices.

TABLE I
Memory Technology Comparison [1–4]

| Items | DRAM | STT-RAM | PCM |
|---|---|---|---|
| Density | 20 | 6 | 16 |
| Read Latency (ns) | 50 | 30 | 50 |
| Write Latency (ns) | 50 | 100 | 250 |
| Write Energy (J/GB) | 3.12 | 25.53 | 153.6 |
| Endurance | $>10^16$ | $10^16$ | $10^8$ |

As a write operation hurts an NVRAM-based system's endurance and performance, the most effective solution is to decrease the amount of write traffic to NVRAM devices. Some excellent research works [5, 6], for example, previously proposed novel write-reduction journaling mechanisms for avoiding writing duplicate data to NVRAM. In addition to the write-reduction journaling concept, data compression techniques were also leveraged to minimize the amount of written data to NVRAM in [7, 8]. Though the approaches of write-reduction design effectually reduce the write overhead on NVRAM-based systems, they are not the sovereign cure for NVRAM's endurance. This is because the amount of write traffic might be accumulated in some specific NVRAM cells. Thus, a wear-leveling strategy will be a unique solution for prolonging the lifetime of NVRAM. Additionally, it will have a synergistic effect in lengthening NVRAM's lifetime while a system developer simultaneously employs a write-reduction design and a wear-leveling strategy.

### B. $B^+$-tree Indexing Scheme

As aforementioned, a wear-leveling strategy will be the most effective solution to extend the lifetime of NVRAM. In the wear-leveling design, managing small and frequently-updated data is the sticky problem because such data will wear out the endurance of NVRAM soon. For example, indexing schemes, especially $B^+$-tree structures, will be frequently updated while data is inserted or deleted. Note that a $B^+$-tree structure will be more frequently updated than other indexing schemes because its tree structure will automatically split and merge for tree balance. Additionally, the $B^+$-tree indexing scheme has been widely applied to modern file systems (e.g., EXT-4 and BTRFS) and database systems [1, 9–11]. As a result, it will be an important an issue with how to design a wear-leveling

strategy in dealing with the update operations in a B$^+$-tree indexing scheme.

Speaking of B$^+$-tree, it is a balanced tree-based structure, in which the distance between all leaf nodes to the root node is the same. To maintain data records, a new index (or key) will be created and inserted into the indexing scheme while new data is needed to be kept in the memory/storage space. In other words, a data will be managed by its corresponding key in the B$^+$-tree structure. With the consideration of simple management, the B+-tree will keep all keys in its leaf nodes. That is to say, all data pointers are kept in leaf nodes. Unlike the leaf nodes, the root node and the internal nodes are working as a guide to point the next level of the tree; thus, in the root and internal nodes, it maintains the tree pointer for guiding search operations to the next level sub-tree or a leaf node. As the B$^+$-tree stores all data keys and pointers in leaf nodes, a leaf node will be more frequently updated than an internal node. Therefore, a B$^+$-tree structure has different update frequency at not only each node but each tree level. The different update frequency results in the critical wear-leveling issue while the B$^+$-tree indexing scheme is applied to database or file systems on an NVRAM-based main memory architecture.

### C. Related Works

In the past years, an efficient B$^+$-tree indexing scheme for NVRAM has been studied broadly [1, 12–15]. More specifically, Chi *et al.* [1] proposed an overflow node structure to reduce the write traffic over phase-change memory. Each leaf node can have one or more overflow sorted nodes, and the overflow node can postpone the timing of splitting nodes. In the same direction, Hu *et al.* [16] proposed a predictive (B$^p$) tree to improve the overflow and underflow node structure. On the write-reduction concept, Shimin Chen *et al.* [17] proposed two simple but efficient B+-tree node structures: Unsorted node with counter and unsorted node with the bitmap. The proposed structures reduced the amount of write traffic to NVRAM because they do not sort the elements in a node and shift the element while a key is deleted from B+-tree nodes. Unlike these solutions, the PB$^+$-tree proposed by Choi *et al.* [13] noticed the wear-leveling issue with the B+-tree structure over NVRAM main memory. They divided a node space into two different areas: Insertion and sorted area. All new keys will be inserted into the insertion area first. While the insertion area is full, the inserted keys will be sorted and moved to the sorted area. Although the amount of write traffic to the sorted area will be reduced, the insertion area still received a large amount of write traffic. Plus, the different update frequency at each B$^+$-tree layer has not been considered in their solution. Therefore, they still have not tackled the wear-leveling issue with B$^+$-tree indexing scheme.

Although the above solutions effectively reduced the amount of write traffic to NVRAM, they are failed to resolve the wear-leveling issue for prolonging the lifetime of NVRAM due to two major facts. First of all, the B$^+$-tree proposed by previous works cannot balance the amount of write traffic within a tree node because indexing keys will be frequently shifted for keeping the order of indexing keys in the node. Secondly, the prior solutions have not considered the different update frequency at each tree level. For example, interior nodes can be regarded as cold data for data management, compared with leaf nodes. This is because all key operations (i.e., insertion, deletion, and data pointer update) will be conducted upon leaf nodes. Unfortunately, to the best of our knowledge, none research work has addressed this wear-leveling issue.

### D. Motivation

In this work, we observe that a B$^+$-tree structure will shorten the lifespan of NVRAM because indexing keys will be shifted within a node for keeping the sorted sequence of keys during key insertion. Though the unsorted node structure can avoid shifting the keys while a new key is inserted, it has poor read performance and needs to sort the keys before node splitting as well. Some readers might point out that wB$^+$-Tree proposed by [12] can keep keys in the sorted sequence without shifting the keys. However, wB$^+$-Tree needs to maintain an extra structure, namely slot array, for recording the sorted sequence of keys. In other words, the slot array will be a hot-spot for writing because each new key is inserted into a node and then the slot array within the written node will be updated to keep the sorted sequence of inserted keys. Additionally, the different update frequency at each level of a B$^+$-tree indexing scheme is also an important wear-leveling issue while the B$^+$-tree indexing scheme is applied to NVRAM main memory. Figure 1 shows the wear-leveling issues with a B$^+$-tree indexing scheme over NVRAM's main memory.

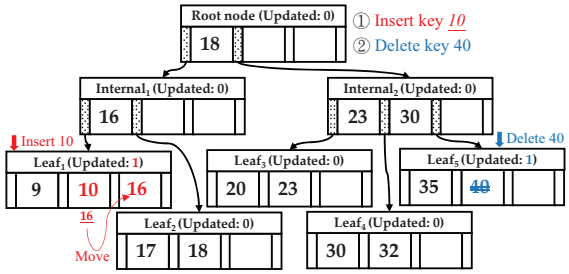


Fig. 1. The wear-leveling issues with a B$^+$-tree indexing scheme.

As illustrated in Figure 1, key 10 (resp. 40) is inserted into (resp. deleted from) the B$^+$-tree indexing scheme. In both operations, all updates are issue to leaf nodes. Key 10, for example, is inserted into the leaf node 1 (i.e., Leaf$_1$). Similarly, key 40 is deleted from leaf node 5 (i.e., Leaf$_5$). As all key updates will be issued to leafs, interior nodes and leafs will result in different damage states to NVRAM's endurance. Plus, a key insertion might move keys within a node. In Figure 1, when key 10 is inserted into the B$^+$-tree indexing scheme, key 16 is shifted to next entry for keeping the sorted sequence in the leaf node 1 (i.e., Leaf$_1$). Nevertheless, this shifting operation will result in uneven worn out in the memory space of Leaf$_1$.

Briefly, two major wear-leveling issues with the B$^+$-tree indexing schemes are *different update frequency between nodes* and *uneven worn-out state within a node space*. Based on

such observation, this work aims to develop a wear-leveling B$^+$-tree indexing scheme *that not only balances the amount of write traffic within a node space but considers the global wear-leveling issue over the whole NVRAM space as well*. To achieve our design goal, we need to face two major technical challenges: (1) *how to evenly distribute the write requests in a "sorted" node structure* and (2) *how to replace old*[1] *with young nodes in the B$^+$-tree indexing scheme for extending NVRAM's lifespan.*

## III. Wear-leveling-aware B$^+$-tree Scheme

### A. The Overview of waB$^+$-tree Scheme

To accomplish our design goal, this work proposes a neoteric B$^+$-tree indexing scheme, namely wear-leveling-aware B$^+$-tree (also referred to waB$^+$-tree for short). The waB$^+$-tree consists of three major components: A circular node structure for reducing the number of write requests resulted from shifting keys, a node-based wear-leveling policy for evenly distributing the write requests in a node, and a global wear-leveling strategy for prolonging the lifespan of NVRAM. Figure 2 illustrates the system diagram of the proposed waB$^+$-tree.

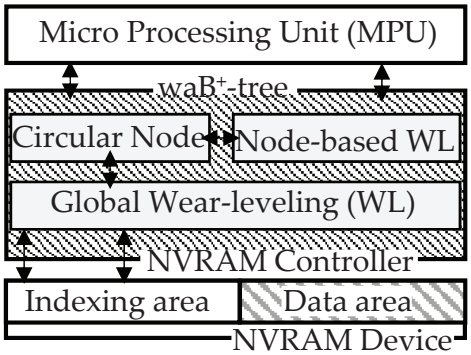


Fig. 2. The proposed system architecture.

*Note that this work only focuses on the wear-leveling issue with the indexing area because the endurance of the indexing area dominates the whole NVRAM's lifespan.* For example, data in the data area is rewritten and then its corresponding indexing information will also be updated for keeping the data pointer of the indexing information up-to-date. In other words, the endurance of the indexing area will be worn out sooner than that of the data area. a.

As the wear-leveling issue with the indexing area is the critical problem for NVRAM's endurance, the proposed waB$^+$-tree thoroughly rethink the current B$^+$-tree design by jointly considering internode and intranode wear-leveling issues. As presented in Figure 2, the proposed waB$^+$-tree will be integrated into the memory controller. In the waB$^+$-tree, all nodes will be managed as a circular node structure to reduce the amount of write traffic caused by shifting keys. In traditional a B$^+$-tree node, the key will be sequentially inserted from the leftmost to rightmost entry. Thus, the first half entries will receive heavy write traffic while the key needs to be shifted for the next insertion key. Such observation motivates us to rethink the position of the first insertion key. This work does not follow the traditional design and proposes a novel circular node structure. In the circular node structure, the first insertion key will be placed in the special position, called *pivot*, in a node. While the next key is inserted, the waB$^+$-tree will compare the pivot with the new key. If the new key is larger than the pivot, the key will be inserted to the right side of the pivot. Otherwise, the key will be inserted to the left side of the pivot. As a result, the waB$^+$-tree can decrease the amount of write traffic generated by shifting keys (please refer to Section III-B for details).

Although the circular node structure can subdue the frequency of shifting keys, the wear-leveling issue is still wide open because a node contains some hot entries (i.e., metadata entry) which receive a large amount of write traffic. To consider the wear-leveling issue, this work proposes two-phase wear-leveling designs: Node-based and global wear-leveling strategies. The node-based strategy will evenly distribute the amount of write traffic to each entry within a node by moving the positions of the metadata and pivot. According to our circular node structure, as the pivot entry will be updated only while its key is deleted, the pivot entry will receive the lightest write traffic. On the other hand, because all key insertions and deletions will update the information of the bitmap in the metadata entry, the metadata entry will suffer from the heaviest write traffic. Based on such observation, the node-based wear-leveling policy will adaptively select the right position for the metadata and pivot in order to balance the number of write requests within a node space (see Section III-C). Additionally, we observe that each tree node will have different update frequency. The root node, for example, receives the fewest update requests because it is at the topmost position. On the contrary, leafs receive the largest number of update requests because all keys and data pointers will be maintained in leaf nodes. Thus, the waB$^+$-tree enlists a global wear-leveling strategy for considering the different update frequency of each node (see Section III-D).

### B. Circular Node Structure

In a classical B$^+$-tree structure, an insertion operation will always place the first key from the leftmost entry within a tree node. However, such design has a fatal drawback for NVRAM's lifespan. As new insertion keys are inserted from the leftmost to rightmost entry, the first half entries within a tree node will accumulate the larger amount of write traffic than the second half entries within the tree node. This is because keys stored in the first half entries will be frequently shifted for keeping the sorted sequence. Worse still, the large amount of write traffic to the first half entries within a node cannot be distributed to other places. For tackling this problem, our proposed waB$^+$-tree redesign the node structure as a circular node.

The goal of our circular node structure is to move the start point of insertion in a tree node for evenly distributing the

[1]A node space receives a large amount of write traffic.

amount of write traffic generated by key movements to each entry within a node. In order to achieve this design goal, the waB$^+$-tree sets a pivot position and a boundary indicator in each node. The pivot is the position of the first insertion key in a node and the boundary is a pointer to indicate the position of the largest or smallest key in a node. As illustrated in Figure 3, the circular node structure is composed of a pivot indicator and a boundary pointer.

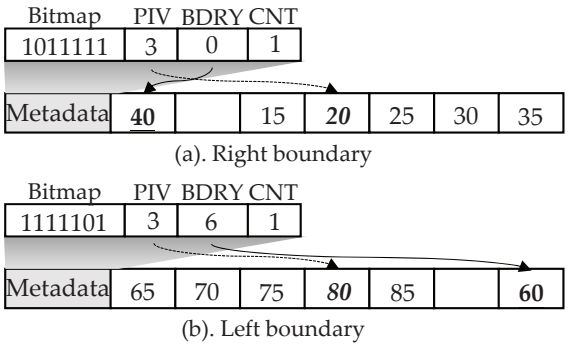


Fig. 3. The circular node structure.

As presented in Figures 3 a and b, a circular node structure consists of a bitmap, a pivot denoted (i.e., PIV), a boundary indicator (i.e., BDRY), and a counter (i.e., CNT). In the circular node structure, the bitmap is to record the state of each entry. Specifically, bit 1 represents an occupied entry, and bit 0 denotes an empty entry. In addition to the bitmap, the pivot information (i.e., PIV) indicates the pivot position which is the start point of insertion in a node. In Figures 3 a and b, keys 20 and 80, for example, are the first insertion key in these two nodes, respectively. After the first insertion key, each incoming key will be compared with pivot. If the value of the incoming key is larger than that of the pivot, the key will be inserted into the right-side of the pivot. Otherwise, it will be inserted into the left side of the pivot. Although the key shifting problem is still in the circular node structure, the pivot point will reduce the number of shifting positions from $n$ to $\frac{n}{2}$ on average because the pivot divides a node space into two insertion area. When the incoming key is smaller than the pivot value, it will be inserted into the left-side of the pivot position. Otherwise, it will be inserted into the right-side of the pivot position. Because the number of entries in the right-side and left-side of the pivot is $\frac{n}{2}$, the number of key shifts within a node can be reduced to $\frac{n}{2}$ on average.

Although the pivot position can decrease the number of key shifts within a node, some astute readers might point out that the node structure design will ruin the utilization of a node space. For example, when the key in the pivot position is not a middle key of the input sequence, one-sided entries within a node will be occupied when the node still have free entries on another side. As illustrated in Figure 3.a, when the key 40 is inserted into the node, our waB$^+$-tree indexing scheme will check the bitmap information for finding the free entry in the right side of the pivot. In this case, we cannot find the empty entry in the right side of the pivot; however, there are free entries within the left side of the pivot. In order to fully utilize the free entries within the node and keep the sorted sequence of keys, we need an indicator to borrow the free space from another side of the pivot. In the circular node structure, the boundary pointer (i.e., BDRY) is the indicator to borrow the free entries. In Figure 3.a, when the key 40 is waiting for insertion, the waB$^+$-tree scheme finds that all entries in the right side of the pivot are occupied but those in the left side of the pivot are still empty. Consequently, the waB$^+$-tree scheme will borrow half of the free entries from another side. To take the free space, the waB$^+$-tree scheme set the boundary pointer to the last (resp. first) position of borrowed free entries in the left-side (resp. right-side) of the pivot. In Figure 3.a, as the insertion of key 40 needs to borrow the entry spaces from the left-side of the pivot, the waB$^+$-tree will set the boundary position to zero for taking one free entry from the left-side of the pivot because the number of free entries in the left-side is two. Similarly, in Figure 3.b, all entries in the left-side of the pivot are occupied. The waB$^+$-tree scheme will borrow one free entry from another side by setting the boundary pointer to six. Note that the waB$^+$-tree scheme can adaptively adjust the boundary pointer for borrowing more free entries or returning the free entries. Plus, the waB$^+$-tree scheme always sets half of the free entries as a basic unit in the borrowing and returning processes.

### C. Node-based Wear-leveling Strategy

Although the circular node structure can reduce the number of key shifts in a node on average, each entry within the circular node still receives the different amount of write traffics. In the circular node structure, the key stored in the pivot position is only altered when the node is allocated or the pivot key is deleted. Conversely, the entry storing metadata will be frequently updated because the bitmap will be modified when a key is inserted/deleted into/from the node. In order to resolve such intranode wear-leveling issue, we need to shit the positions of pivot and metadata within a node when the node space has been repeatedly allocated. To accomplish this design goal, the node-based wear-leveling strategy will move these two positions by referring to how many times a node has been allocated, so as to evenly distribute the amount of write traffic to each entry within a node. In other words, the principle of our node-based wear-leveling is to make each entry within a node can be a pivot or metadata entry finally. Figure 4 presents the spirit of our node-based wear-leveling strategy. Note that **C** is how many times the node has been allocated.

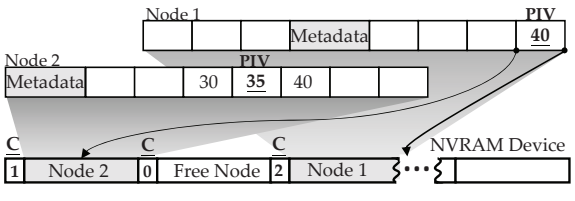


Fig. 4. The node-based wear-leveling strategy.

As illustrated in Figure 4, each node will set the middle entry as the pivot position and the first entry as the metadta area at the first allocation (e.g., *Node 2*). However, a node

space will be reclaimed and reallocated because of delete, split, and merge operations. Our node-based wear-leveling strategy will shift the metadata and pivot to a new position (which can be derived from Equation 1) for evenly distributing the amount of write traffic to each entry when the node is reallocated. For example, *Node 1* has been allocated twice in Figure 4; therefore, our node-based wear-leveling strategy shifts the metadata and pivot areas the predefined number of positions to the right. That is to say, the node-based wear-leveling strategy can replace hot with cold entries inside a node by the shifting design. However, two major problems in the node-based wear-leveling strategy are (1) how to determine the predefined number of shifting positions and (2) how to know the positions of pivot and metadata at each allocation.

For the first problem, the predefined number of shifting positions should be not a factor of the total number of entries within a node because the pivot and metadata will not be placed at some entries within the node if it is a factor. Additionally, the predefined number should push the hot area (i.e., metadata) to the nearby cold entry (i.e., the pivot entry) to achieve intranode wear-leveling state soon. Based on such observation, the predefined number should be a prime which is smaller than but close to half of the number of entries within a node. For example, in Figure 4, the number of entries within a node is seven. As a result, the predefined number of shifting positions will be three according to our rules for the predefined number.

After the predefined number is determined, we can utilize Equation 1 to calculate the new positions of metadata and pivot at each allocation.

$$P_n = (P_s + T_{allocated} * P_{predefined}) \mod N_E, \quad (1)$$

where $P_n$ denotes the new positions of pivot or metadata at the node allocation and $P_s$ represents the start positions of pivot and metadata at the first allocation. Generally, the start positions of pivot and metadata are half of the number of entries within a node and one, respectively. In Equation 1, $T_{allocated}$ is how many times the node has been allocated and $N_E$ is the total number of entries within a node. Therefore, if we obtain the predefined number of shifting positions (i.e., $P_{predefined}$), the new shifted positions can be derived from Equation 1. Briefly, the node-based wear-leveling strategy achieves intranode wear-leveling state via shifting the positions of metadata and pivot.

### *D. Global Wear-leveling Strategy*

Though the node-based wear-leveling strategy can evenly distribute the amount of write traffic to each entry within a node, it still does not consider the different update frequency at each node. In other words, the internode wear-leveling issue is still an unsolved problem for our node-based wear-leveling strategy. To cope with this issue, our waB$^+$-tree scheme includes another design, namely a global wear-leveling strategy, to swap hot with cold nodes during node allocation.

As the global wear-leveling strategy needs to be simple and efficient, we only need to set one cursor to point a candidate node for swapping in the current B$^+$-tree structure and know one information about the average amount of write traffic. In the beginning, the cursor will be set to the root node and it will be moved to the next node when a swapping process is executed or the node pointed by the cursor is not a cold node. The global strategy will trigger the swapping process when the amount of write traffic of the (to-be-allocated) free node is larger than the threshold value derived from Equation 2.

$$Threshold = AVG_{writes} + \frac{0.5 \times L_{NVRAM}}{AVG_{writes}}, \quad (2)$$

where $AVG_{writes}$ denotes the average amount of write traffic to a memory cell and $L_{NVRAM}$ represents the lifespan (i.e., write cycles) of the NVRAM technology. In addition to $AVG_{writes}$, the ratio of NVRAM lifespan and the average amount of write traffic to a memory cell should be considered in the threshold because it can be a constraint for frequently swapping at an early stage. Generally, the average amount of write traffic to a memory cell is very small at an early stage, and the small value will result in frequently swapping if we do not add the ratio to the threshold. Therefore, while the threshold includes the ratio as a regulator, our global strategy can avoid unnecessary key movements between nodes for achieving the wear-leveling state of the NVRAM device. After a while, the regulator will become invalid at the middle stage when the average amount of write traffic to a memory cell is close to half of NVRAM's lifespan. This is because we need an aggressive swapping process to ensure that each node space within the NVRAM space should suffer from the same or similar amount of write traffic.

Briefly, our global wear-leveling strategy will obey the following instructions to swap the to-be-allocated free node with the allocated node in the B$^+$-tree structure. First of all, when a new node needs to be allocated, our global strategy will estimate the amount of write traffic of the new node space by multiplying the allocation frequency of the new node and the average amount of write traffic to an NVRAM cell. If its amount of write traffic is larger than the value derived from Equation 2, the swapping process will be triggered to swap the new node with the (old) node pointed by the cursor through copying all keys and information from the old node to the new node and modifying the pointer of the parent node of the old node to reconnect with the new node. After that, the old node will be allocated as a leaf for the upcoming keys and the global wear-leveling strategy will move the cursor to the next to-be-swapped node in the B$^+$-tree structure. During the cursor movement, the global strategy will check whether the old node has a right sibling. If the old node has a right sibling, the global strategy will check the amount of write traffic of the sibling node. When its amount of write traffic is less than the average amount of write traffic to a memory cell, the cursor will be moved to this node. Otherwise, the global strategy will keep checking the next right sibling node. If the node does not have the right sibling, the global strategy will check the leftmost child node in the one lower level. After the checking process, the cursor should be moved to the next to-be-swapped node in the B$^+$-tree structure. If the checking process cannot find any node for next swapping, the cursor

will be reset to the root node of $B^+$-tree structure. Based on this design, our waB$^+$-tree scheme can tackle the issue of the different update frequency between tree nodes.

## IV. Performance Evaluation

### A. Experimental Environment

The purpose of this section is to assess the capability of waB+-tree, in terms of the amount of write traffic to the NVRAM space, the distribution of write requests to each memory cell, and query latency. All experiments are conducted on a DRAM-based simulation [19]. Specifically, we established all experimental indexing schemes on a DRAM storage space and collected the size of read/write requests to the DRAM space. Then, the experimental results related to performance metrics (e.g., query and write latencies) can be derived from simulating the collection of read/write accesses to DRAM on the NVRAM-based storage environment, as presented in Table II. For emphasizing the effect of waB$^+$-tree, our experiments include two previous solutions (i.e., original B$^+$-tree and wB$^+$-tree [12]) for comparisons. In our experiments, two benchmarks were conducted on the simulation process: Normal distribution [19, 20] and Yahoo! Cloud Serving Benchmark (YCSB) [18]. Specifically, the normal distribution contains the set of indexing key requests collected from the randomly-generating mechanism, which produces the key request under the normal distribution. Additionally, the YCSB is a database testing tool to examine the performance of NoSQL databases (e.g., Cassandra [21]); therefore, it can generate a series of indexing key requests to a storage system. By running these two benchmarks, we can show that our waB$^+$-tree can effectively and evenly distribute the amount of write traffic to memory cells, compared with other solutions. The experimental information is summarized in Table II. This work utilized phase-change memory (PCM) as a case study to observe the effect of all experimental solutions because PCM is the most promising candidate for the next-generation memory technology [23].

TABLE II
Experimental settings [22].

| Phase-change Memory | | | |
|---|---|---|---|
| Energy | | Performance | |
| Read | 16 J/TB | Read | 50 ns/access |
| Write | 153.6 J/TB | Write | 1 us/access |
| Endurance | | $10^6$ Cycles | |
| Benchmarks | | | |
| Normal Distribution | | 50 million keys | |
| YCSB | | 50 million keys | |

### B. Experimental Results

As aforementioned, our proposed waB$^+$-tree scheme concerns the wear-leveling issue resulted from B$^+$-tree structure over the NVRAM storage space. However, we cannot only focus on the wear-leveling issue but ignore the NVRAM system performance. Consequently, the experimental results present the write-request distribution, the amount of write traffic to NVRAM, the latency of constructing the indexing scheme, and the query performance.

*1) Write Distribution:* First of all, the most important metric for our proposed waB$^+$-tree scheme is the write distribution on the NVRAM storage space. Our evaluation utilized median and standard deviation (SD) to estimate the write distribution. If the SD is a large value, it means that all values in the write distribution are far away from the mean value. On the other hand, all values in the write distribution are close to the mean value under the small SD. In this experiment, we included original B$^+$-tree, wB$^+$-tree [12], and our waB$^+$-tree in the comparison.

TABLE III
Normal Distribution: Median and standard deviation (SD) under different deletion ratios.

| Benchmark: Normal Distribution | | | |
|---|---|---|---|
| Deletion ratio: 20% | | | |
| Solutions | B$^+$-tree | wB$^+$-tree | waB$^+$-tree |
| Median | 19422 | 34427 | 10422 |
| SD | 5181.02 | 14396.41 | 2699.54 |
| Deletion ratio: 40% | | | |
| Solutions | B$^+$-tree | wB$^+$-tree | waB$^+$-tree |
| Median | 20360 | 40598 | 10941 |
| SD | 5637.67 | 17099.17 | 2961.30 |
| Deletion ratio: 60% | | | |
| Solutions | B$^+$-tree | wB$^+$-tree | waB$^+$-tree |
| Median | 22352 | 53838 | 11962 |
| SD | 6693.04 | 22572.57 | 3566.32 |

As presented in Table III, this experiment showed three write distributions under different deletion rates (i.e., 20%, 40%, and 60%) when the normal distribution benchmark was conducted. In this experimental result, the waB$^+$-tree scheme has the smallest value in the median and standard deviation estimations, because our solution can evenly distribute the amount of write traffic to all memory cells. On the contrary, B$^+$-tree and wB$^+$-tree structures have a large value in the median and standard deviation because they ignore the wear-leveling issue with the original B$^+$-tree design on the NVRAM storage space. In order to make the result convincible, we also collected the write distributions when the YCSB benchmark was conducted. As aforementioned, YCSB is an open-source and well-known benchmark to test storage performance, so the results collected from the YCSB experiments might be more solid for practical systems.

TABLE IV
YCSB: Median and standard deviation (SD) under different deletion ratios.

| Benchmark: Normal Distribution | | | |
|---|---|---|---|
| Deletion ratio: 20% | | | |
| Solutions | B$^+$-tree | wB$^+$-tree | waB$^+$-tree |
| Median | 17961 | 31554 | 9696 |
| SD | 3790.67 | 10110.01 | 1982.72 |
| Deletion ratio: 40% | | | |
| Solutions | B$^+$-tree | wB$^+$-tree | waB$^+$-tree |
| Median | 18730 | 36110 | 10102 |
| SD | 4166.61 | 12147.40 | 2191.96 |
| Deletion ratio: 60% | | | |
| Solutions | B$^+$-tree | wB$^+$-tree | waB$^+$-tree |
| Median | 20242 | 47345 | 10892 |
| SD | 5289.85 | 16586.51 | 2789.59 |

The results in Table IV are the write distributions under different deletion rates when the YCSB benchmark was conducted on the experiments. As shown in Table IV, among all experimental solutions, the waB$^+$-tree still has the smallest value in the estimation of median and standard deviation. Although the values of median and standard deviation are slightly increased when the deletion rate is raised, the waB$^+$-tree structure also keeps fair to evenly scatter the amount of write traffic to every memory cells. Therefore, according to the results of standard deviation, the waB$^+$-tree structure can effectively prolong the lifespan of NVRAM storage by distributing the amount of write traffic to the whole NVRAM storage space.

*2) Construction Latency:* In the previous evaluation, the waB$^+$-tree structure shows its ability to evenly distributes the amount of write traffic to memory cells. This evaluation will examine all experimental solutions in the latency of indexing scheme construction. This is because a solution will not be adopted when its performance is slow, even though the solution can address the wear-leveling issue with the NVRAM storage technology. In this evaluation, we also include three solutions and two benchmarks in our comparison. In order to emphasize the improvement achieved by the waB$^+$-tree structure, we normalized all results to one baseline solution (i.e., the original B$^+$-tree structure). In Figure 5, the x-axis shows the different deletion rates and the y-axis denotes the construction latency after normalization. As illustrated Figure 5, when the normal distribution benchmark is conducted on the experiment, the waB$^+$-tree structure has the best performance because our solution can reduce the frequency of shifting key positions for a new key insertion. In this comparison, the wB$^+$-tree structure has the worst performance because it needs to do a lot of metadata update in a node. For example, although the wB$^+$-tree structure does not need to shift key in a node for an upcoming insertion request, it still needs to shift the key pointer in its metadata area for keeping the keys in the sorted sequence. Because of the metadata operations, the wB$^+$-tree has the worse performance than the original B$^+$-tree structure.

In addition to the normal distribution benchmark, this evaluation also collects the results of experimental solutions by running the YCSB benchmark. As aforementioned, the x-axis and y-axis of Figure 8 are the different rates and the construction latency after normalization, respectively. Figures 5 and 8 have similar results. In Figure 8, our proposed waB$^+$-tree structure still has the best performance because of the reduction of key movements. According to Figure 8, our waB$^+$-tree structure can save time for constructing the indexing scheme by 46% on average, compared with the baseline solution. It is a worthing note that the performance of wB$^+$-tree gets worse when the deletion rate is increased. This is because the metadata update operation will be more frequent in the increasing deletion rate. However, the performance of our waB$^+$-tree is not affected by the increase of deletion rate. Briefly, our proposed waB$^+$-tree structure not only figure out the wear-leveling issue but increase the storage performance as well.

*3) Query Latency:* Although our proposed waB$^+$-tree structure has the best performance on constructing the indexing scheme, some astute readers might wonder the performance of key query. Querying an index in a database is a common operation. In other words, if an indexing scheme has poor query performance, the solution will not be commonly-used in systems and applications. In order to prove that the waB$^+$-tree structure has comparable query performance, this experiment randomly generates 5 million query requests to the indexing solutions, including the original B$^+$-tree, the wB$^+$-tree, and our waB$^+$-tree. Figures 6 and 9 illustrate the results of query performance when two benchmarks (i.e., the normal distribution and YCSB) are conducted on the experiments.

In Figure 6, the x-axis represents different deletion rates, and the y-axis shows query latency. According to the experimental results, the original B$^+$-tree has the best query performance because it does not need to look-up any metadata during a query operation. However, the wB$^+$-tree and our waB$^+$-tree need to look-up the metadata for a query operation. For example, our waB$^+$-tree scheme needs to read boundary information and pivot position, and the wB$^+$-tree structure needs to read bitmap information and key pointers in the metadata area. Compared with wB$^+$-tree structure, our solution still has the better query performance.

On the other hand, we also examine the query performance by running the YCSB benchmark. The x-axis and y-axis of Figure 9 are the same with that of Figure 6. According to Figure 9, the original B$^+$-tree scheme still has the best query performance because of no metadata operation. Our waB$^+$-tree structure is still in the second place when the YCSB benchmark is conducted. The wB$^+$-tree structure has the worst performance because of many metadata operations.

*4) Writing Statistics:* In addition to the median and standard deviation, we further prove that our waB$^+$-tree structure can have the best endurance by collecting the results of maximum and average write counts. Figures 7 and 10 illustrate the maximum and average write counts while two benchmarks are conducted on three different indexing schemes. In Figures 7 and 10, the x-axis is the different indexing schemes, and the y-axis is the number of write counts.

According to Figures 7 and 10, our proposed waB$^+$-tree structure has the smallest value in the maximum and average write counts. This is because our waB$^+$-tree structure tackles the intranode and internode wear-leveling issue with the original B$^+$-tree scheme. As shown in Figures 7 and 10, the maximum write counts of our waB$^+$-tree structure is half of that of the original B$^+$-tree scheme. In other words, our solution can prolong the lifespan of NVRAM storage at least 2 times, compared with the original B$^+$-tree structure.

### C. Advanced Analysis

*1) Internode Wear-leveling:* In order to show the effect of the proposed waB$^+$-tree structure on internode wear-leveling issue, we dump the number of write counts on each physical node space. In our experimental settings, a node size is 4KB, and the memory address of nodes is aligned to node size. Figure 11 illustrates the write access pattern on each 4KB physical node. The x-axis is the physical node address, and the y-axis is the number of write counts on each physical node. According to Figure 11, we can observe that the waB$^+$-tree

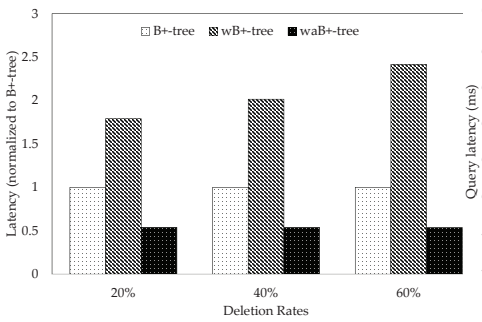


Fig. 5. Write latency under the normal distribution benchmark.

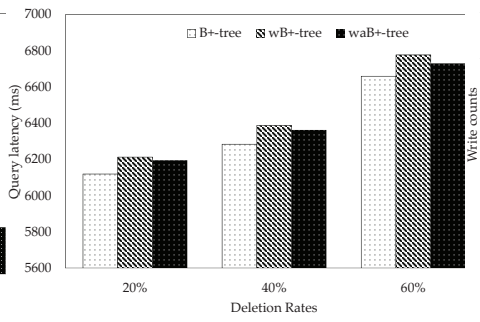


Fig. 6. Query latency under the normal distribution benchmark.

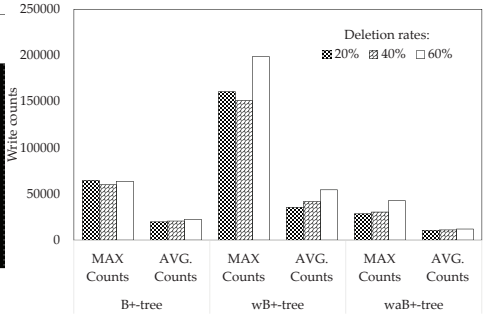


Fig. 7. Write counts under the normal distribution benchmark.

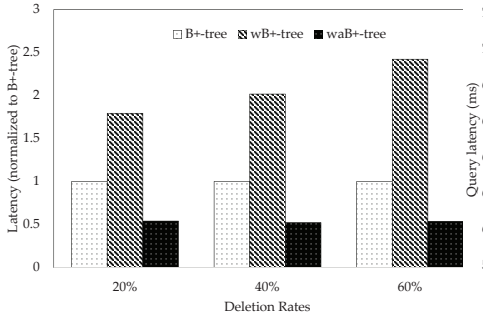


Fig. 8. Write latency under the YCSB benchmark.

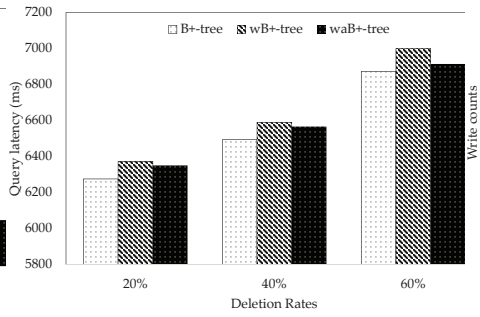


Fig. 9. Query latency under the YCSB benchmark.

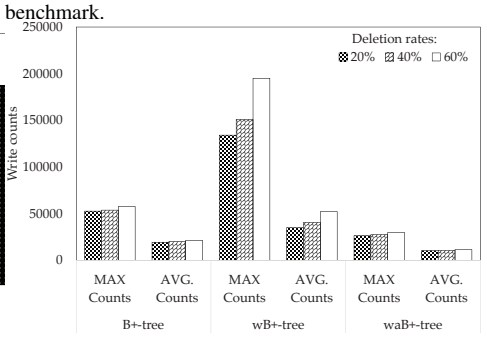


Fig. 10. Write counts under the YCSB benchmark.

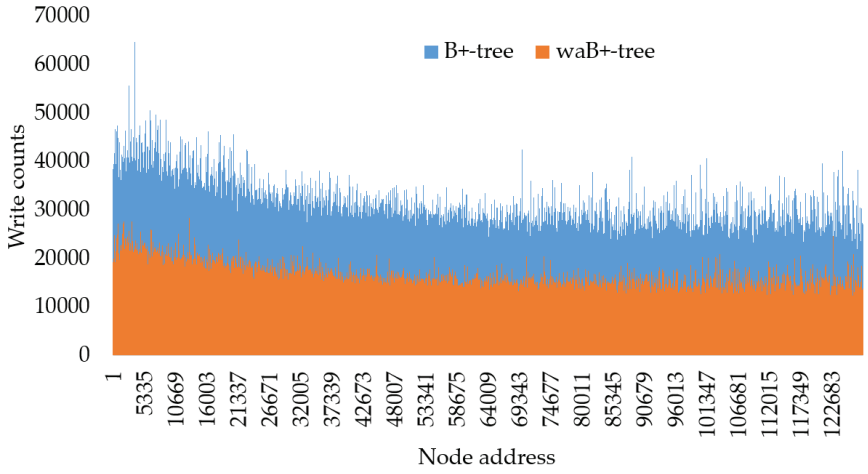


Fig. 11. The number of write counts on each physical node address ($B^+$-tree v.s. $waB^+$-tree).

structure significantly reduces the number of write counts on each 4KB physical node. Additionally, the $waB^+$-tree structure also has less peak value about the number of write counts on the 4KB node space, compared with the original $B^+$-tree.

The benefits of the proposed $waB^+$-tree structure on the internode wear-leveling issue is more obvious when the $waB^+$-tree structure is compared with the $wB^+$-tree scheme. In Figure 12, we can observe that the $waB^+$-tree structure can significantly decrease the number of write counts on each physical node space. Furthermore, the $waB^+$-tree evenly distributes the amount of write traffic to all physical node spaces.

*2) Intranode Wear-leveling:* According to the previous experiment, we can prove the effectiveness of the $waB^+$-tree on the internode wear-leveling issue. About the effect on the intranode wear-leveling issue, we compare the $waB^+$-tree scheme with the original $B^+$-tree structure because the original $B^+$-tree structure has good performance on the wear-leveling issue. Figures 13 and 14 presents the number of write counts on memory addresses within a tree node. The x-axis is the physical memory address in a node and the y-axis is the number of write counts. According to Figure 13, an original $B^+$-tree node has poor intranode wear-leveling because the keys in the first half of a node will be frequently shifted for new key insertion. On the contrary, because the $waB^+$-tree scheme has the pivot and circular node designs, it can evenly distribute write requests to all memory addresses.

## V. Concluding Remarks

As a $B^+$-tree indexing scheme is the most commonly used in file systems and database applications for index

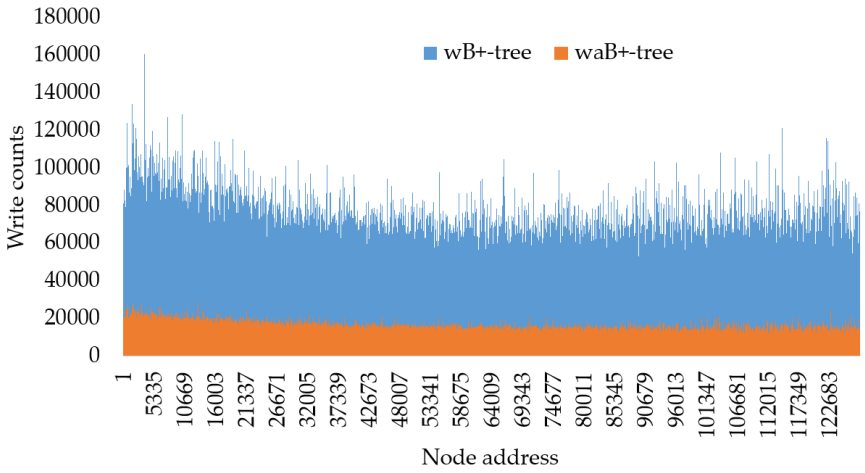


Fig. 12. The number of write counts on each physical node address (wB$^+$-tree v.s. waB$^+$-tree).

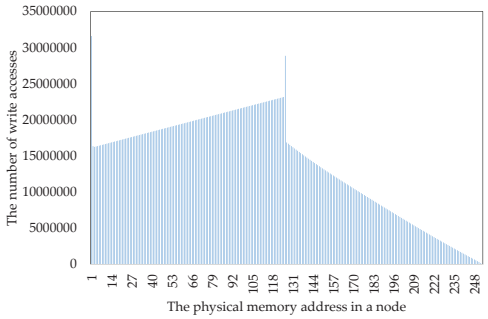


Fig. 13. The number of write counts in a physical node address (B$^+$-tree).

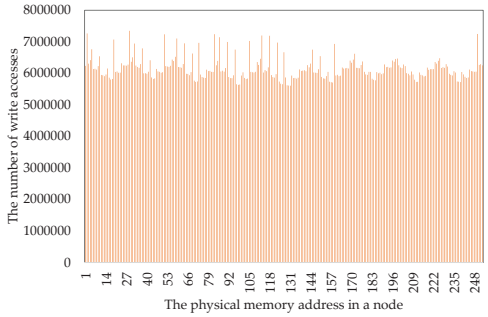


Fig. 14. The number of write counts in a physical node address (waB$^+$-tree).

management, and NVRAM-friendly B$^+$-tree structure should be developed. The endurance will be the most tricky problem when the B$^+$-tree indexing scheme is integrated into an NVRAM-based computer systems. In order to tackle the endurance issue of B$^+$-tree structure on an NVRAM storage space, this paper firstly address the internode and intranode wear-leveling issues. For resolving the wear-leveling issues, this work proposed a wear-leveling-aware B$^+$-tree indexing scheme (waB$^+$-tree). With the assistance of the circular node structure, pivot design, and global wear-leveling strategy, the proposed solution can significantly reduce the amount of write traffic to NVRAM storage and evenly distribute the amount of write traffic to all memory cells. According to the experimental results, the proposed waB$^+$-tree scheme can achieve even write distribution on an NVRAM storage space. Moreover, it can prolong the lifespan of the NVRAM storage device by at least two times.

## REFERENCES


[1] P. Chi, W.-C. Lee, and Y. Xie, "Making B+-tree efficient in PCM-based main memory," In Proceedings of the 2014 international symposium on Low power electronics and design (ISLPED '14), La Jolla, California, USA, pp. 69-74, August 11 - 13, 2014.

[2] P. Chi, S. Li, Y. Cheng, Yu Lu, S. H. Kang and Y. Xie, "Architecture design with STT-RAM: Opportunities and challenges," 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), Macau, 2016, pp. 109-114.

[3] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies. Nanoscale research letters", vol. 9, no. 526, 2014.

[4] H. -. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase Change Memory," in Proceedings of the IEEE, vol. 98, no. 12, pp. 2201-2227, Dec. 2010.

[5] T. Chen, Y. Chang, S. Chen, C. Kuo, M. Yang, H. Wei, and W. Shih, "wrJFS: A Write-Reduction Journaling File System for Byte-addressable NVRAM," in IEEE Transactions on Computers, vol. 67, no. 7, pp. 1023-1038, 1 July 2018.

[6] J. Kim, C. Min and Y. I. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," in IEEE Transactions on Consumer Electronics, vol. 60, no. 2, pp. 217-224, May 2014.

[7] D. B. Dgien, P. M. Palangappa, N. A. Hunter, J. Li and K. Mohanram, "Compression architecture for bit-write reduction in non-volatile memory technologies," 2014 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH), Paris, 2014, pp. 51-56.

[8] Y. Li, H. Xu, R. Melhem, and A. K. Jones. "Space Oblivious Compression: Power Reduction for Non-Volatile Main Memories," In Proceedings of the 25th edition on Great Lakes Symposium on VLSI (GLSVLSI '15), pp. 217-220, Pittsburgh, Pennsylvania, USA, May 20 - 22, 2015.

[9] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B+-tree based indexing method for nearest neighbor search", ACM Transactions on Database Systems (TODS), vol. 30, no. 2, pp. 364-397, June 2005.

[10] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," In Proceedings of the Linux Symposium, vol. 2, pages 21–33, 2007.

[11] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem", ACM Transactions on Storage (TOS), vol. 9, issue 3, no. 9, , August 2013.

[12] S. Chen and Q. Jin, "Persistent B+-trees in non-volatile main memory", Proceedings of the VLDB Endowment, vol. 8, issue 7, pp. 786-797, February 2015.

[13] G. S. Choi, B. On and I. Lee, "PB+-Tree: PCM-Aware B+-Tree," in IEEE Transactions on Knowledge and Data Engineering, vol. 27, no. 9, pp. 2466-2479, 1 Sept. 2015.

[14] L. Li, P. Jin, C. Yang, Z. Wu, and L. Yue, "Optimizing B+-tree for PCM-based Hybrid Memory. In EDBT, pages 662–663, 2016.

[15] L. Li, P. Jin, C. Yang and L. Yue, "Efficient Tree Indexing for PCM-Based Memory Systems," 2015 8th International Conference on Control and Automation (CA), Jeju, 2015, pp. 46-53.

[16] W. Hu, G. Li, J. Ni, D. Sun and K. Tan, "$\mathsf{B}^p$ - Tree : A Predictive $\mathsf{B}^+$ - Tree for Reducing Writes on Phase Change Memory," in IEEE Transactions on Knowledge and Data Engineering, vol. 26, no. 10, pp. 2368-2381, Oct. 2014.

[17] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking Database Algorithms for Phase Change Memory", In CIDR'11: 5th Biennial Conference on Innovative Data Systems Research, January 2011.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB", In Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10), June 2010.

[19] Y. Liang, T. Chen, Y. Chang, S. Chen, H. Wei and W. Shih, "B*-sort: Enabling Write-once Sorting for Non-volatile Memory," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), early access.

[20] Y. Liang, T. Chen, Y. Chang, S. Chen, K. Lam, W. Li, and W. Shih, "B*-sort: Enabling Write-once Sorting for Non-volatile Memory," in ACM Trans. Embed. Comput. Syst. 18, 5s, Article 66 (October 2019), 20 pages.

[21] A. Lakshman and P. Malik, "Cassandra: structured storage system on a P2P network," in Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC '09), August 2009.

[22] M. Yang, Y. Chang, and C. Tsao, "Byte-addressable update scheme to minimize the energy consumption of pcm-based storage systems." in ACM Trans. Embed. Comput. Syst., 15(3):55:1–55:20, June 2016.

[23] S. Fong, C. Neumann and H. Wong, "Phase-Change Memory—Towards a Storage-Class Memory," in IEEE Transactions on Electron Devices, vol. 64, no. 11, pp. 4374-4385, Nov. 2017.