FIRM-tree: a Multidimensional Index Structure for Reprogrammable Flash Memory

Shin-Ting Wu, Pin-Jung Chen, Po-Chun Huang, Wei-Kuan Shih, Yuan-Hao Chang, *Fellow, IEEE*

***Abstract*—For many emerging data-centric computing applications, it is a key capability to efficiently store, manage, and access multidimensional data. To achieve this, many multidimensional index data structures have been proposed. However, when existing multidimensional index data structures are maintained on modern nonvolatile memories such as NAND flash memory, they often face challenges in effective management of multidimensional data and handling of memory medium peculiarities, such as the write-once property and the need for block reclamation of NAND flash memory. Without appropriate management, these challenges often result in serious amplification of the read/write traffic, which degrades the performance of multidimensional data structures. Motivated by the urgent needs of efficient multidimensional index data structures on modern nonvolatile memories, we propose the *FIRM-tree*, a time-efficient and space-economic index data structure for multidimensional point data on NAND flash memory. Unique to the prior work, the FIRM-tree holistically utilizes RAM and flash memory space, and dedicatedly leverages the page reprogrammability of modern NAND flash memory, to enhance data access performance and flash management overheads. We then verify our proposal through analytical and experimental studies, where the results are quite encouraging.**

***Index Terms*—Multidimensional index structure, flash memory, reprogrammable flash, write amplification.**

I. Introduction

With the rapid development of data-centric computing scenarios, a high-performance data structure has become a must-have to handle massive data. Furthermore, many applications demand the capability to store, manage, and access the data of multiple or many dimensions. According to the different flavors of data, multidimensional index structures may be further classified as *point access methods (PAMs)* or *spatial access methods (SAMs)* [1]. While PAMs target data points, SAMs are able to handle region data of different geometries in the multidimensional space. Different from one-dimensional data, multidimensional data often exhibit sparsity in the multidimensional space. How to suppress the loss of performance and space utilization due to the data sparsity thus becomes a design focus on multidimensional index structures.

There have been a number of brilliant proposals of multidimensional index structures. Taking PAMs for example, there are four major classes of multidimensional index structures, such as the grid files [2], R-trees [3]–[5], kd-trees [6], [7], and quadtrees [1], [8]. The diversified designs of multidimensional index structures are driven by sophisticated trade-offs and joint design considerations over the data distributions and access operations. Specifically, how to strike a proper design trade-off between the latency and throughput of query operations in different flavors and those of update operations in different flavors is a key question. On the other hand, whether the data points scatter in a large key range or clustered in a small key range also affects the designs of multidimensional index structures. What is worse, the design complexity of multidimensional index structures exacerbates as the intrinsic characteristics of specific memory media—such as NAND flash memory—are also considered, which motivates this work.

When a multidimensional index structure is maintained on a NAND flash memory device, undesirable amplification in the read and write traffic might be encountered, resulting in unfortunate performance degradation. For instance, when a node of an R-tree overflow and is split, the data points in the node will be migrated to the child nodes. Consequently, some data points in the R-tree might be repetitively migrated, incurring serious write amplification and performance degradation. The reliance on NAND flash memory on garbage collection, which erases block to free up the space of invalid pages, exacerbating the issue of write amplification. This is because that, prior to the erasing of a block, all *its* valid pages must be moved to other blocks. On the other side, due to the page-based accesses of NAND flash memory, the space utilization of certain pages might be unacceptably low, which wastes the space of flash memory and amplifies the read/write traffic on query/update operations of the R-tree. This urgently motivates the designs of a new multidimensional index structure, which takes full advantages of NAND flash memory to satisfy the performance criteria of modern index structures. In this work, we present the *FIRM-tree*, an index data structure for multidimensional point data on NAND flash memory. Our technical contributions:

- A *selective data migration* strategy is proposed to avoid writing the few scattered data points into a flash page, thereby improving the space utilization of flash memory and mitigating the performance overheads due to the read/write amplifications on query/insert operations (Section III-B). To avoid aggressively collecting and rewriting the few scattered data points in a FIRM-tree node, these data points are simply left in situ, until eventually rearranged by the incremental compaction (Section III-D).
- A *level skipping* strategy is proposed to further mitigate the long-term write amplification for redistributing the data points to lower-level nodes in the FIRM-tree (Section III-C). The strategy leverages the non-negligible RAM space to maintain a *mega-root node* which keeps the topmost levels of the FIRM-tree. Due to the aligned key ranges of nodes with the *uniform partitioning* policy (Section II-B), the data points may be directly flushed to lower-level nodes in the FIRM-tree, effectively skipping some levels and mitigating the write amplification
- A *node compaction* strategy is proposed to exploit the *page reprogramming* capability of modern TLC flash memory to reduce the frequency and overhead of garbage

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2024 and appears as part of the ESWEEK-TCAD special issue. (Corresponding authors: Y.-H. Chang and P.-C. Huang).

S.-T. Wu, P.-J. Chen, and W.-K. Shih are with National Tsing Hua University, Taiwan.

P.-C. Huang is with Taiwan Tech, Taiwan. (Email: pch.ntust@gmail.com)

Y.-H. Chang is with Institute of Information Science, Academic Sinica, Taiwan. (Email: johnson@iis.sinica.edu.tw)

collection of flash blocks (Section III-D). With the help of page reprogramming, the data compaction can be realized in an *incremental* manner to reduce the instantaneous latency of garbage collection, which is a serious issue on future flash memory with larger block sizes.

The rest of this paper is organized as follows. First of all, Section II presents the background and considered system architecture, so as to motivate this work. Afterward, Section III presents the proposed FIRM-tree, a bucketed tree structure for multidimensional point data on modern NAND flash memory with page reprogramming supports. To evaluate the efficacy of the FIRM-tree, a series of analytical and experimental studies are performed in Sections IV and V, respectively. In Section VI, we discuss the important prior work, followed by Section VII which concludes this work.

II. System Architecture and Motivations

A. Background

1) Nonvolatile Memories (NVMs): Recently, various high-performance and energy-economic *nonvolatile memories (NVMs)* have become powerful competitors of classical memory and storage media in diversified computing systems. Depending on the unit of data accesses, NVMs may be further classified into byte-addressable *persistent memories (PMs)*, e.g, phase-change memory (PCM) [9], [10] or magnetoresistive random-access memory (MRAM) [9], [11], and *block-based NVMs*, e.g., NAND flash memory [12], [13]. While various PMs can replace RAM as the main-memory media, NAND flash memory is a promising alternative to mechanical hard disks as the secondary storage devices.

With the unique characteristics of NVMs, the existing hardware/software components of a computer system need to be redesigned to fully optimize the access performance and energy efficiency. For example, index data structures such as red–black tree and B-tree [14] are a core component in modern databases like *key–value (KV) stores*. When these data structures are maintained on NAND flash memory, they must consider the intrinsic characteristics of NAND flash memory, such as the *page-based access* (i.e., each flash page is read or written at once, prohibiting random accesses) and *write-once property* (i.e., each page can be written only once before its residing block is entirely erased and reclaimed by *garbage collection*). Thus, the space utilization of NAND flash memory is directly related to the read and write amplification of the data structures, because writing a nearly empty page might seriously amplify the write traffic and subsequent read traffic.

2) Multidimensional Index Data Structures: Classical index data structures, such as the red–black tree and B-tree [14], often rely on scalar keys, which are simply integers or real numbers. However, some applications might rely on the data in multidimensional space, which motivates the proposal of diversified *multidimensional index data structures*. According to the data formats, multidimensional index data structures may be classified as *point access methods (PAMs)* for managing *point data*, or *spatial access methods (SAMs)* for managing *regional data* [1]. Up to now, there are four major clans of PAMs, namely the *k-dimensional trees (kd-trees)* [6], *R-trees* [3]–[5], *grid files* [2], and *quadtrees* [1], [8]. However, although different multidimensional index data structures are proposed for different application scenarios with different design trade-offs, the *curse of dimensionality* is a common problem that might lead to abrupt degradation of space efficiency or access performance, which must be solved [1].

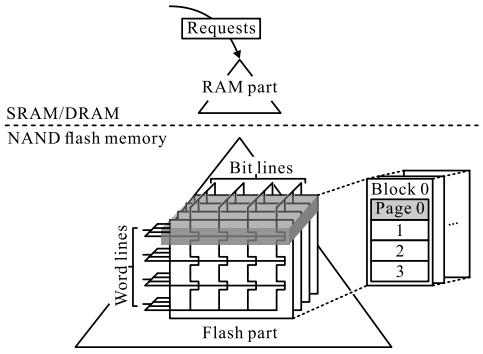

Fig. 1. Generic system architecture of a hybrid flash device.

3) Multidimensional Index Data Structures on NVMs: The access granularity is a major difference between the main memory and secondary storage. That is, while the media for main memory must be byte-addressable to allow in-place code execution, those for secondary storage are often accessed in large *blocks*, such as hard disk *sectors* or flash memory *pages*, to reduce management costs. For such block devices, there have been some *bucketed* multidimensional index structures, such as R*-tree [4] and bucket PR quadtree [1], where each node can accommodate more than one multidimensional data items and occupies one or more blocks. Unfortunately, NAND flash memory is not an ideal block device, and its unique characteristics (Section II-A1) often demand dedicated designs of index data structures to prevent performance degradation or space waste on common tree operations such as insertion and querying of data points. There are two promising approaches to such designs: leveraging the unique features of flash memory or integrating the strengths of other media, such as RAM.

To manage multidimensional point data on NAND flash memory, several flash-friendly index data structures have been proposed, such as the F-KDB [15]–[18] and LB-Grid [19]. Both F-KDB and LB-Grid perform log-structured writes to buffer new data points until there are sufficiently many of them to be flushed together into one or more pages, alleviating the write amplification effect. On the other hand, proposals for the hybrid of flash memory and 3D XPoint memory [10] also exist, such as the HyR-tree [20], which utilizes unsupervised learning to identify the hot nodes, and allocates the hot (/cold) nodes in the 3D XPoint memory (/NAND flash memory) to enhance access performance. However, prior work neither takes advantage of the unique features of modern flash memory nor holistically manages both RAM and flash memory, leaving much room for further optimization and motivating this work.

B. System Architecture & Motivation

Figure 1 shows the generic system architecture of a hybrid device with NAND flash memory and SRAM/DRAM. As compared to the flash memory, the SRAM/DRAM is faster in accesses, smaller in capacity, and can be randomly accessed in the unit of a byte. In contrast, the NAND flash memory comprises a number of fixed-sized *blocks*, each of which has a number of consecutive fixed-sized *pages*. Typically, the size of a page and that of a block are 4 KB–16 KB and 256 KB–8 MB, respectively [13]. While a page is the

unit of *read* or *write* (*/program*) operations, a block is the unit of *erase* operations. Classical NAND flash memory is limited by the *write-once property*, with which a page can be written/programmed only once unless its residing block is entirely erased and the contents of all pages are cleared.

To avoid performing a time-consuming block erase operation for updating a page, the updated data should be *out-place updated* to other free pages. Due to the out-place updates, the pages with the latest and obsolete data are referred to as the *live* and *dead pages*, respectively. To reclaim the space occupied by dead pages, the *garbage collection* activity is triggered to select and erase one or more *victim blocks*. Before a block can be erased, the contents of its valid pages must be migrated to other blocks to prevent data losses, referred to as *live data copying*. As a block typically has hundreds of pages, the overheads of live data copying might be high. Thus, it is important to avoid unnecessary garbage collection and select the appropriate victim blocks for reclamation.

This work is motivated by the design challenges of maintaining existing multidimensional index data structures on the hybrid of NAND flash memory and SRAM/DRAM. Specifically, when a node bucket has too many data points and overflows, how to partition the data points and redistribute them to appropriate child nodes is a key question.

- With the *balanced partitioning* policy such as in the R*-tree [5], the data points of the node will be evenly partitioned into the child nodes, which guarantees balanced structure of the tree but leads to misaligned key ranges of different nodes, which exacerbates the management complexity and access overheads of the data structures.
- In contrast, with the *uniform partitioning* policy such as in the bucket PR quadtree [1], the key range of the overflowing node will be divided into equal-sized subranges, and the data points of the node will be redistributed into the child nodes according to the subranges. However, due to the *dimensionality curse* [1], multidimensional data points are often unevenly distributed in the key space. Consequently, few scattered data points might be written as a whole page, which degrades the space utilization of flash memory, exacerbates the write amplification on insert operations, increases the tree height, and exacerbates the read amplification on subsequent query operations.
- Specifically, for d-dimensional data, each multidimensional tree node has up to 2^d child nodes, each uses exactly a page of space. Suppose that a page can keep up to f data points, the flash space utilization could be as low as $\frac{1}{2^d} + \frac{2^d-1}{2^d} \cdot \frac{1}{f}$, when the distribution of the data points are extremely skewed. This result suggests that, with a larger size gap between a page and a data point, or higher dimensionality of data, the worst-case flash space utilization will be lower. With a typical setting of $f = 100$ and $d = 4$, the worst-case space utilization will be only $\frac{115}{1600} \sim 7.19\%$. Although the space utilization would be higher in practice, it might still impact the space economicity and access performance of the tree.

Modern NAND flash memory devices are often equipped with some RAM space, which can be randomly accessed and leveraged to resolve the dilemma between the balanced and uniform partitioning policies. This motivates us in seeking for novel approaches to optimally leverage the RAM space in the designs of multidimensional index data structure on modern flash memory devices. Unlike existing buffering/caching approaches, the RAM space should be holistically utilized to maintain the multidimensional index data structures.

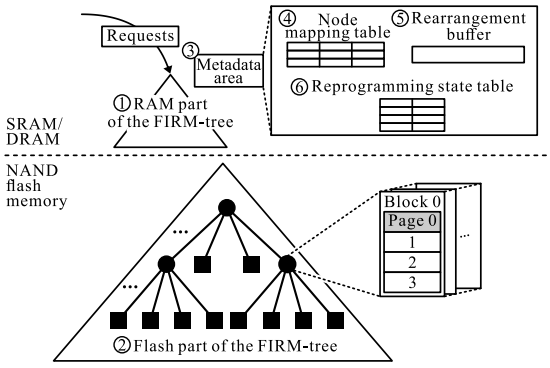

Fig. 2. Architecture of the FIRM-tree.

Garbage collection is a major source of performance overheads for NAND flash memory. When a multidimensional index data structure is maintained on flash memory, frequent insertion of data points will lead to frequent node splitting. As data points are repetitively redistributed across different nodes, a lot of obsolete data will be generated, which magnifies the garbage collection overheads. Fortunately, modern 3D TLC NAND flash memory supports *page reprogramming* operations [21] and allows each page to be programmed up to thrice before the erasure of its residing block, which is hopeful to suppress the garbage collection overheads. However, how to leverage the page reprogramming capability in efficient designs of multidimensional index data structures remain an unanswered question, which also motivates this work.

III. Designs of the FIRM-tree

A. Basic Structure & Working Principles of the FIRM-tree

The FIRM-tree is proposed for reprogrammable NAND flash memory with extra RAM space (Fig. 2). Unlike the prior work that treats RAM as a log buffer of flash memory [15], [19], the FIRM-tree holistically utilizes the RAM and flash memory space to mitigate the write amplification due to the flushing of scattered multidimensional data points. Specifically, the FIRM-tree is maintained simultaneously on RAM and flash memory, where the part on RAM and that on flash memory are called the *RAM part* and *flash part*, respectively. Each part is an augmented bucketed quadtree managed by *uniform partitioning policy* [1], where the up to 2^d children of each node is associated with an equal-sized subrange of the key range of the node (d is the dimensionality of data). With uniform partitioning policy, the key range of the nodes in the two parts are aligned, which simplifies the coordination of the two parts. In addition, the aligned key ranges help reduce the interference among different nodes and potentially benefit concurrent accesses of the tree. To close the size gap of the access units of RAM and flash, the RAM space is managed in the unit of a *frame*, which is as large as a flash page. From the database system view, the FIRM-tree is a flash-friendly drop-in replacement for existing multidimensional data structures. However, the FIRM-tree must work on open-channel solid-state disks (OCSSDs) or collaborate with flash translation layers (FTLs) to exploit the unique features of modern flash memory, such as page reprogramming capability.

When a new data point is inserted into the FIRM-tree, it will be inserted into the deepest *compatible node* in the RAM part, whose *key range covers the coordinate of the point* (① of Fig. 2). Besides the coordinate, a data point may have other satellite data, which will be inserted together into the node. When a node exhausted its space, a new frame will be allocated for the node, until the node becomes too large and must be split. When that happens, all or a part of the data points in the node will be selectively migrated to their deepest compatible offspring nodes (Section III-B). When the RAM has exhausted its frames, no more frames can be allocated and each frame of the deepest leaf node and its direct ancestors will be flushed to a flash page (②). With aligned key ranges of the nodes in the RAM and flash parts, these frames may be flushed to some non-root nodes of the flash part, effectively skipping some levels of the flash part and alleviating long-term data migration overheads (Section III-C).

To facilitate efficient management of the FIRM-tree, a separate *metadata area* (③) is maintained in RAM. In the metadata area, the mapping between each RAM or flash part node and the corresponding RAM frames or flash blocks will be kept in a *node mapping table* (④). Like in the RAM part, when a node in the flash part becomes too large and overflows, it will be split with its data points redistributed to appropriate offspring nodes. To achieve that, the data points in the overflowing node will first be read into a *rearrangement buffer* (⑤), rearranged in the buffer, and written back to the block of the overflowing node or that of the deepest compatible offspring nodes. With the reprogramming capability, a flash page may be written up to thrice after its residing block is erased [21]. After all pages in a block have been programmed in order, they can be reprogrammed from the first page of the block over again. Thus, to keep track of the reprogramming status of flash blocks, the FIRM-tree also keeps a *reprogramming state table* (⑥) to keep the number of reprogramming operations done since the last erasure of each block. Besides the metadata introduced by the FIRM-tree, other metadata needed by flash management are also kept in the metadata area.

B. Selective Data Migration Strategy

When a flash page is considerably larger than a data point, writing few data points into the entire page might reduce the space utilization of the page. This amplifies not only the write traffic of insert operations but also the read traffic of subsequent read operations, which affects the access performance. The problem is especially serious on multidimensional data points, which are often distributed in a highly biased manner in the multidimensional key space. To solve the problem, the FIRM-tree is equipped with a *selective data migration strategy*, which migrates the data points of an overflowing node in a certain key subrange (determined by the uniform partitioning policy) to the corresponding child node, only when there are enough of them to do so. In this way, the leaf nodes of the quadrants with only few scattered data points in the RAM or flash part will not be created at all. Instead, these data points will remain in the overflowing node, awaiting to be migrated together with subsequently inserted data points. As the space utilization of newly created nodes is guaranteed, read/write amplification can be alleviated.

When there are enough data points that are migrated together to the same child of the overflowing node, the space utilization of the written frames/pages is guaranteed at least as high as a threshold μ^* ($0 \leq \mu^* < 1$) determined by the applications. To determine the corresponding lower bound of the number s^* of data points that provides such a guarantee (Point x in Fig. 3), we consider the resulting space utilization when different numbers of data points are written together into one or more frames/pages. For simplicity of discussion, we assume that a frame/page can accommodate exactly f ($\in \mathbb{Z}^+$) data points without internal fragmentation. Also, we assume that $kf < s^* \leq (k+1)f$ for some $k \in \mathbb{Z}^+$, i.e., we need to write at least s^* data points together into $(k+1)$ frames/pages to guarantee the space utilization of at least μ^* of the written frames/pages. According to Fig. 3, we have that $\frac{k-1}{k} < \mu^* \leq \frac{k}{k+1}$, which gives that $\frac{\mu^*}{1-\mu^*} \leq k < \frac{1}{1-\mu^*}$. Since $k \in \mathbb{Z}^+$, we have $k = \left\lceil \frac{\mu^*}{1-\mu^*} \right\rceil$ (Eq. 1). In addition, to derive s^*, we know that $(s^* - kf) : \left(\mu^* - \frac{k-1}{k}\right) = ((k+1)f - s^*) : \left(\frac{k}{k+1} - \mu^*\right)$, which gives that $s^* = (1 + \mu^* k) f$ (Eq. 2). Combining Eqs. 1 and 2 then gives $s^* = \left(1 + \mu^* \left\lceil \frac{\mu^*}{1-\mu^*} \right\rceil\right) f$ (Eq. 3).

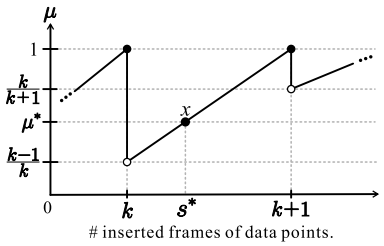

Fig. 3. Deriving the lower bound s^* of the # of data points for migration.

With an appropriate setting of s^*, the node split operation works as follows. When a node n overflows and is to be split: (1) If n is in the flash part, the existing data points of n and the to-be-inserted data point will be read into the rearrangement buffer first, due to the write-once property of flash memory. (2) All data points in the buffer will be classified according to the key subranges obtained by the uniform partitioning policy. (3) Finally, if there exist any key subranges with s^* or more data points, the child nodes of such key subranges will be created to store their corresponding data points. To prevent interference between the management of different nodes, each node may be allocated an integer number (typ. 1) of flash blocks (for flash-part nodes) or the same size of RAM space (for RAM-part nodes). When a block has b pages, it can accommodate up to bf data points. For sufficiently large blocks such that $bf \geq 2^d \cdot s^*$, there will be at least one among the 2^d quadrants where sufficient data points may be written, according to the pigeonhole principle. With the node splitting operation, any leaf node in either part of the FIRM-tree always has at least s^* data points; however, some internal nodes in either part might still have few scattered data points that still cannot have their dedicated nodes.

When the RAM space is exhausted, the entire RAM part of the FIRM-tree will be flushed into the flash memory to release the RAM space for subsequently inserted data points. Like the migration of data points, the flushing of data points also adheres to the same limitation. That is, the flushing will be done with one or more write operations, each writing at least s^* data points into at least $k = \left\lceil \frac{\mu^*}{1-\mu^*} \right\rceil$ consecutive pages in the block of the same deepest compatible node in the flash part. Specifically, we will repetitively (1) select a leaf node

in the RAM part, (2) flush all its data points together with a write operation, and (3) remove the node from the RAM-part quadtree, until the entire RAM part has been flushed. (Note that any non-root leaf node in the RAM part has at least s^* data points and can be flushed. The root node in the RAM part does not seriously amplify the read/write traffic.) Note that, once all children of an internal node have been removed, the internal node will become a leaf node. At this time, if the node does not have sufficient ($\geq s^*$) data points, (1) its data points shall be merged back to its parent node, and (2) it will be removed from the RAM-part quadtree, until the resulting leaf node has at least s^* data points after merging and may be flushed.

C. Level Skipping Strategy

Due to the write-once property of flash memory, the data structures maintained on flash memory often suffer from the write amplification due to repetitive migration of data points across different nodes. With the help of the RAM part, newly inserted data points may be preclassified, and directly migrated to their deepest compatible nodes in the flash part, effectively skipping some levels and mitigating write amplification. With larger RAM space, the RAM-part quadtree can grow higher, and more levels may be skipped when the RAM part is flushed. As compared to the existing approaches that leverage the RAM as buffers/caches, the FIRM-tree further considers the distribution of data points and more effectively reduces the write traffic of data point migration. However, how many levels of the FIRM-tree may be skipped depends on the actual distribution of the data points in the multidimensional space.

When there is abundant RAM space, one may guarantee the minimum number of levels of the FIRM-tree that can be skipped when the RAM part is flushed. By recursively prepartitioning the key universe into $2^{d\cdot\ell}$ equal-sized quadrants for some predetermined *skipping parameter* $\ell \in \mathbb{Z}^+$, the RAM part now has $2^{d\cdot\ell}$ quadtrees, each handling the data points in a quadrant. The root node of each quadtree is actually a level-ℓ node of the FIRM-tree, and all such root nodes are maintained by a RAM-resident *root node table* with $2^{d\cdot\ell}$ entries, effectively skipping the topmost ℓ levels of the FIRM-tree (Fig. 4). (Note that the quadtrees with no data points will still be created; however, they will not be allocated any frames to save the RAM space.) In the eye of the flash part, the $2^{d\cdot\ell}$ roots of the quadtrees in the RAM part form a much larger root node, termed as the *mega-root* of the FIRM-tree.

Everything has its price. Since the RAM part does not guarantee sufficient data points in the root node, when there are more quadtrees, the more root nodes might degrade the space utilization of flash memory and exacerbate future read amplification on query operations. Fortunately, the problem is temporary and will be implicitly resolved as the data points are reclassified and migrated to deeper nodes in the flash part. As the mega root is larger, the finer-grained prepartitioning is also potentially beneficial for query performance, as more irrelevant data points may be quickly excluded. The optimal size of the mega root is a design trade-off among space utilization of the flash memory, write amplification of insert operations, and read amplification of query operations; how to determine it from the distributions of data points and mixture of insert/query operations will be an interesting future work.

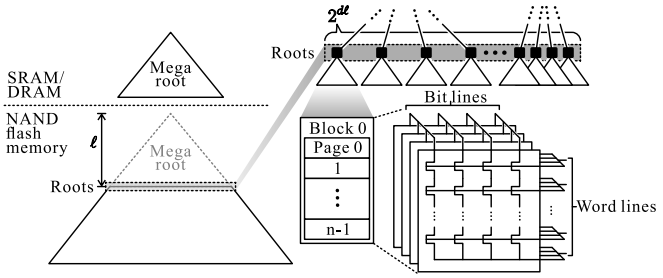

Fig. 4. With the mega root design, the topmost levels of the flash-part quadtrees are skipped for write reduction.

D. Full/Incremental Data Compaction Strategies

When a node in the flash part becomes too large, it overflows and must be split. That is, all pages in the block(s) allocated for the node have been written, and there are already too many such blocks so that no more may be allocated. At this time, the data points of the to-be-split node will be read to the rearrangement buffer and rearranged. The data points that belong to each quadrant of the to-be-split node will be sequentially written into the free pages in the block(s) of the corresponding child node, if there are sufficiently many ($\geq s^*$) of them to do so. (If the child node does not yet exist, it will be created and allocated a free block first.) In prior work, the blocks of the to-be-split node must be reclaimed, and the remaining scattered data points that cannot be migrated to the child nodes must be synchronously rewritten back into the to-be-split node. *In contrast, with the page reprogrammability of modern NAND flash memory [21], the FIRM-tree allows the scattered data points to remain in situ, so that the to-be-split node may be* lazily *compacted until it is necessary to do so.*

With page reprogrammability of modern flash memory, the FIRM-tree does not need to erase the block(s) of a node every time when the node is being split. Instead, the pages in a block of the node can be rewritten with *the node's scattered data points that cannot be migrated* and *the data points migrated to the node*, for up to thrice [21]. There are two possible approaches to node compaction in the FIRM-tree, namely, *full* and *incremental compaction*, with different design trade-offs. With *full compaction*, all scattered data points that cannot be migrated will be identified, compacted in RAM, and synchronously written with page reprogramming operations from the beginning of the block. When a block has reached its maximum number of reprogramming operations, it will be erased as normal. In contrast, to reduce the long instantaneous latency of the possibly many page reprogramming operations, *incremental compaction* temporarily leaves all such scattered data points in situ. Later, when new data points are inserted into the node, the minimum necessary number of the beginning pages of the block will be read, and the scattered data points in the pages will be identified, compacted, and written back along with the new data points using page reprogramming operations. Note that incremental compaction requires extra metadata to distinguish the scattered data points that have not yet been migrated to the child nodes.

To clarify how incremental compaction works, a working example would be useful (Fig. 5). When some node $n_\times$ is being split (Steps ① and ②), its block(s) will not be immediately reclaimed. Instead, the data points in each quadrant of $n_\times$ are classified in the rearrangement buffer (Step ③), and either

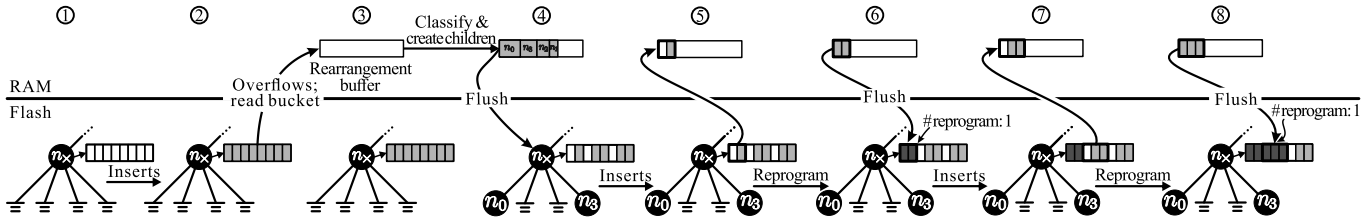

Fig. 5. Incremental compaction of data points in the FIRM-tree.

migrated to the corresponding child nodes, such as n_0 and n_3 (if there are sufficiently many of them), or stay in situ in $n_\times$ (if there are only few scattered data points in the quadrants of a child node, say n_1 or n_2) (Step ④). Such a node splitting strategy is considered *conservative* because only when there are sufficient data points in a quadrant will they be migrated into the corresponding child node. Later on, when some new data points are to be inserted into $n_\times$, we shall search it for any remaining scattered data points and write them along with the new data points through page reprogramming operations (Steps ⑤ and ⑥, and then Steps ⑦ and ⑧). Once a block has reached its maximum reprogramming threshold, it will be erased by the garbage collector as normal. Note that the exploitation of page reprogrammability of flash memory is applicable to both full and incremental compaction approaches.

E. Collaboration of Different FIRM-tree Components

As compared to the existing approach, our FIRM-tree design offers several unique technological advantages. First of all, the FIRM-tree exploits the page reprogrammability to reduce garbage collection activities of used blocks. While page reprogrammability has potentials in different application scenarios, however, it is especially suitable for the maintenance of metadata of data structures, which often undergo more frequent updates than data, as in the case of the FIRM-tree. In the high-dimensional space, the curse of dimensionality further worsens the issue of frequent updates, demanding technologies such as page reprogrammability to solve the problem. To our best knowledge, the FIRM-tree is an early effort that exhibits how to exploit page reprogrammability for efficient (multidimensional) search tree maintenance.

Next, the FIRM-tree provides a conservative split strategy to address the read/write amplification due to the mismatched sizes of a data point, tree node, flash page, and flash block. Concerning the scattered distribution of data points in a multidimensional space, the FIRM-tree allows a small number of scattered data points to stay in situ in the original block of a split node, deferring their demotion to the child nodes in the tree. In this way, the considerable overheads of node splitting and flash block reclamation may be effectively amortized over multiple insertion of data points, which reduces the worst-case latency. Moreover, with the deferred demotion of data points, some nodes might never be compacted, which also reduces the long-term management overheads of the FIRM-tree.

As astute readers might point out, different components of the FIRM-tree bear different responsibilities for enhancing the tree performance. Specifically, selective data migration avoids the write amplification (and subsequent read amplification) to demote the data points to the buckets of the child nodes at the next lower level of the FIRM-tree. Thus, selective data migration suppresses the read/write amplification occurring in individual flash pages at every level of the tree, which is similar to the internal fragmentation problem in memory management in operating system designs. However, when the tree becomes sufficiently tall, the level-by-level data migration might still incur read/write amplification across multiple levels of the FIRM-tree. Fortunately, modern computer systems are often equipped with quite some RAM space (albeit still smaller than the flash memory itself) that can accommodate a large number of data points until the data points are flushed to the flash. When we can buffer decently many data points, with the page and block sizes fixed, we can skip multiple levels of the flash part of the FIRM-tree, directly flushing the data points to the deepest node while controlling the space utilization, write amplification, and subsequent read amplification.

From the discussions, level skipping regards the RAM part of the FIRM-tree as a single large mega-root node, which internally performs data point classification in RAM without incurring any flash writes. When the RAM is sufficiently utilized, data points may skip multiple levels of the FIRM-tree, compensating the extra write amplification due to the imbalanced tree structure due to the uniform partitioning. Last but not least, the incremental compaction intends to reduce the worst-case latency of insert operations, instead of addressing the read/write amplifications. It collects the leftover data points in a node, combines them with the data points demoted from the parent, and rewrites the combined data points back to the node with page reprogramming operations. By postponing the rewriting of written data points, the maintenance overheads of the FIRM-tree can be remarkably reduced.

F. Implementation Remarks

1) Multi-channel Architecture Supports: To enhance access performance, modern flash devices often have multiple channels that can be accessed in parallel. By default, a node in either part of the FIRM-tree can grow up to the size of a block, and the space utilization is guaranteed on a page basis. To take the performance advantages of multi-channel accesses, the FIRM-tree can be slightly modified by increasing its node size to as large as a *block set*, which comprises all blocks with the same offset within their residing channels. Moreover, the guaranteeing of space utilization is now provided on the basis of a *page set*, which comprises all pages with the same offset within their residing blocks, which also come with the same offset in the residing channels. Note that a page set (/block set) is the unit of multi-channel read and write (/erase) operations; all pages (/blocks) in the set may be read or written (/erased) in parallel to significantly increase throughput and alleviate the impacts of read and write amplifications.

The augmentation of the FIRM-tree also comes with the corresponding increase in the RAM demands. When the flash

device has c channels[1], a page set (/block set) is also c times larger than a page (/block). As a FIRM-tree node takes a fixed number of flash blocks, it is also c times larger. Since the rearrangement buffer is always as large as a node, it must be c times larger as well. However, the size of the other two tables, node mapping table and reprogramming state table, are unaffected because their sizes are determined by the total number of nodes in (both parts of) the FIRM-tree and the flash memory capacity, instead of the node size. On the other hand, the RAM space taken up by the RAM part should also be larger on multi-channel flash devices, so as to provide the same level of effects for read/write reduction.

2) Support of Delete Operations: Besides insert and query operations, delete operations are also common in application scenarios. Unlike RAM or mechanical hard disks, NAND flash memory must be read or written in the unit of a page, and a written page cannot be updated before its residing block is entirely erased. Thus, unlike the data points in the RAM part that can be in-place deleted at very low overheads, those in the flash part must adopt alternative approaches such as to insert *deletion records*, a.k.a., *tombstones* [22], into the FIRM-tree. When the RAM part is being flushed to the flash memory or the high-level node buckets are merged into lower-level ones, the tombstones in the RAM part and the corresponding obsolete data points in the flash part will be merged, thereby removing the data points from the FIRM-tree.

The default design of the FIRM-tree is *ephemeral* in the sense that old versions of the updated or deleted data will disappear [23]. However, specific applications demand the capability to store and manage multiple versions of the same data. To make the FIRM-tree *persistent*, we keep all historical data when the data are updated or deleted. Upon receiving a *purge operation*, which requests to remove all versions prior to a specified purged version, only the number of the latest purged version will be globally kept. During the incremental compaction of data points, all data points older than the latest purged version can be removed, thereby maximally postponing purge overheads. However, there are still missing pieces in the design of multiversioned and multidimensional data structures on flash memory, leaving opportunities for future studies.

From a macroscopic point of view, the insertion and deletion of data points often become a performance trade-off of index data structures, especially on NAND flash memory with the write-once property. In practice, the optimal designs of index data structures of multidimensional data should be determined by the different distributions of inserted keys in the multidimensional space. The optimization of multidimensional index data structures for applications with different access patterns is yet another interesting topic of future studies.

3) Page Reprogrammability on Higher-level-cell Flash Memories: In the existing work of page reprogrammability, a page of TLC flash memory is programmed in the MLC mode, allowing up to two times of reprogramming per page [21]. However, page reprogramming is enabled by reusing cell states, which also has potentials on QLC and higher-level-cell flash memories [24]. As the designs of page reprogramming are orthogonal to those of the FIRM-tree, if page reprogramming is feasible on QLC and high-level-cell flash memory, it can also work with the FIRM-tree there. However, the adoption of page reprogrammability also confront extra challenges, such as the reliability of data.

As reported in [21], page reprogramming might considerably increase the bit error rates. Concerning the already higher bit error rates of high-level-cell flash memory such as QLC and beyond, stronger error-correcting code (ECC) is needed to guarantee data integrity. Meanwhile, when QLC flash memory is programmed, it may be programmed in the MLC or TLC mode, which exhibits different design trade-offs between the maximum allowed number of reprogramming operations and the reliability of written data. However, currently there are few results of page programmability on higher-level-cell (QLC and beyond) flash memory. Once page reprogrammability is available on higher-level-cell flash memory, how it integrates into the designs of the FIRM-tree and other flash-resident data structures would be an interesting topic of investigation.

4) Supports for Regional Data: While the FIRM-tree is suitable for serving intensive insert operations of multidimensional point data, its designs also make it a fit for more sophisticated data formats, such as regional data in the multidimensional space. Specifically, to augment the FIRM-tree for regional data, we allow a regional object to be stored in the bucket of only the tree nodes whose quadrants completely cover the regional object in the multidimensional space. Thus, unlike point data, which can be repetitively demoted in the FIRM-tree structure, regional data may be demoted only to a certain level in the tree structure, according to its size and position in the multidimensional space. Since the quadrant of every node at each level of the tree structure may be statically determined, it is more efficient to search regional data in the augmented FIRM-tree. However, in extreme cases where too many large data regions are stored in the bucket of the same node, one might need to allow larger node buckets, which results in the degradation of access operations of the FIRM-tree. As the optimal management of regional data is more challenging than that of point data, we leave the augmentation details as a key future work.

[1]For commodity flash storage devices, we often have that $1 \leq c \leq 16$.

TABLE I
PARAMETERS USED IN ANALYTICAL STUDIES

Parameter	Symbol	Unit
RAM capacity	R	Frames
Flash capacity	F	Blocks
Flash block size	B	Pages
Flash page size / RAM frame size	P	Bytes
Length of a frame or block address	δ	Bytes
Maximum node size	N	Blocks
Length of a programming/reprogramming counter	2	Bits

IV. ANALYTICAL STUDIES

In this section, we analyze the minimum RAM usage of the FIRM-tree. The parameters used in our analytical studies are listed in TABLE I for convenience. The size of the *RAM part* of the FIRM-tree determines the effectiveness of data point classification, but may be adjusted according to the available RAM space in the target application scenarios. However, the smallest allowed size of the RAM part should be as large as the maximum size of a RAM- or flash-part node, which is as large as a block. Thus, the minimum RAM demands of the RAM part would be $S_{rmpa} = B \cdot P$ bytes, where B is the number of pages per block and P is the page size in bytes.

Next, the *node mapping table* is responsible for mapping every node to its corresponding frames or blocks. In the worst case, every frame of RAM and all blocks of flash memory[2] are allocated. Suppose that the address of a frame or a block is $\delta = 4$ bytes, the node mapping table takes up to $S_{nmap} = (R + F \cdot B) \cdot \delta$ bytes of RAM space in total. On the other hand, the *rearrangement buffer* must be able to accommodate all data points in a node, and is thus made as large as $S_{rebu} = N \cdot F \cdot B$ bytes, where N is the maximum number of blocks that may be allocated to a node in the flash part. (Recall that we often let $N = 1$ in practice because a block is often sufficiently large, as compared to a data point.) Moreover, the *reprogramming state table* keeps the number of program/reprogram operations and the offset of the last written page of every block, and takes $S_{repg} = F \cdot (\lceil \lg(B+1) \rceil + 2)$ bits of RAM because each block can be programmed/reprogrammed up to thrice before being erased and 2 bits suffice to keep a program/reprogram counter. On a generic flash storage device with $(R, F, B, P, N) = (2^{14}, 2^{17}, 2^8, 2^{12}, 1)$, the total size of the RAM part and the size of flash memory are 64 MB and 128 GB, respectively. The total RAM usage S^* of the metadata area is thus given as $S^* = S_{rmpa} + S_{nmap} + S_{rebu} + S_{repg} = (2^{12} \cdot 2^8) + ((2^{14} + 2^{17}) \cdot 2^2) + (2^{12} \cdot 2^8) + (2^{17} \cdot 2) \sim (2^{20}) + (2^{16} + 2^{19}) + (2^{20}) + (2^{18}) <$ 3 MB, which is reasonable for modern flash memory devices.

V. Experimental Studies

A. Experimental Settings

In this section, we performed experimental studies to compare the FIRM-tree to two representative multidimensional tree structures, namely the bucket PR quadtree [1] and R*-tree [5], for block devices such as NAND flash memory. The bucket PR quadtree and R*-tree are selected because of their distinct choices for data point classification, i.e., uniform and balanced partitioning. By addressing the shortcomings of uniform partitioning, the FIRM-tree can get out of both approaches in terms of insert/query performance. To ensure a fair comparison, the control groups are equipped with the same size of RAM space managed as LRU buffers/caches.

Our experiments have four parts, namely *Part D* (w.r.t. different datasets of different data dimensionality and distributions), *Part U* (w.r.t. different space utilization criteria), *Part B* (w.r.t. different block sizes or node capacities), and *Part L* (w.r.t. different levels of the mega root). First of all, Part D, we evaluate the total erase counts, total write counts, total read counts, overall space utilization, and total number of node splitting operations of FIRM-tree, bucket PR quadtree, and R*-tree under nine realistic/synthetic workloads with different dimensionality (2, 4, and 8) and distributions of data points in the multidimensional space. The basic statistics of the considered datasets are listed in TABLE II for the convenience of discussion. Afterward, Part U compares the same performance metrics of the three trees, with respect to different space utilization criteria $\frac{m-1}{m}$, where $m = 2, \ldots, 5$. (The case with $m = 1$ has no criteria on the space utilization and is omitted.) Next, in Part B, we repeat the experiments with respect to different block sizes. While we assume that each node is as large as a flash block, Part B of the experiments observes the performance impacts of different page and block sizes. The number of skipped levels, ℓ, is fixed at 2 in Parts D, U, and B of the experiments. In the last part, Part L, we consider the FIRM-tree with different ℓ $(= 1, \ldots, 4)$ to evaluate the impacts of level skipping of the mega root. All the experiments are done on the SSDsim trace-driven simulator [25], with the environmental settings in TABLEs III and IV.

[2]The spare blocks are reserved for garbage collection and not counted in.

TABLE II
Basic Statistics of the Workloads in the Experimental Studies

Workload	Type	d	# data points	Source
Real1	Realistic	2	6,429,191	SpatialHadoop (TIGER 2015—AREALM) [26]
Real2	Realistic	2	360,177	UCI Machine Learning Repository (UrbanGB Accidents) [27]
Real3	Realistic	2	1,048,575	Kaggle (GeoNames Database) [28]
Real4	Realistic	2	303,735	Urban Institute (311 Calls in New Orleans) [29]
Real5	Realistic	4	110,204	UCI Machine Learning Repository (Sepsis Survival) [27]
Real6	Realistic	4	434,874	UCI Machine Learning Repository (3D Road Network) [27]
Syn1	Synthetic	4	10,000,000	Synthesized w/ uniform distribution
Real7	Realistic	8	199,835	UCI Machine Learning Repository (Query Analytics) [27]
Syn2	Synthetic	8	10,000,000	Synthesized w/ uniform distribution

B. Experimental Results

1) Part D: Impacts of Different Dimensionality and Distributions of Data Points: In this section, the FIRM-tree is compared to the bucket PR quadtree and R*-tree under different workloads, so as to quantize the impacts of the dimensionality and distributions of data on different performance metrics. The space utilization criterion μ^* of the FIRM-tree is fixed at $\frac{1}{2}$. The experimental results of the total erase counts, write counts, read counts, space utilization, and node split counts are then shown as in Figs. 7–10. Regarding to write amplification, the FIRM-tree effectively reduces the number of data points rewritten *due to tree maintenance* by averagely 33.77% and 47.43% as compared to the bucket PR quadtree and R*-tree, respectively. However, due to the working principle of reprogrammable flash, the effective capacity of a page is smaller for the FIRM-tree than that for the bucket PR quadtree or R*-tree. Such a difference in storage density cancels out a part of the advantages of the FIRM-tree in terms of the number of pages written (Fig. 6). Fortunately, as reported in [21], the page programming/reprogramming latency of reprogrammable flash is slightly better than that of normal TLC flash ($\sim$ 2,700 μs vs 3,000 μs), which also enhances the relative performance of the FIRM-tree accordingly.

On the other hand, in different workloads, the FIRM-tree remarkably reduces the number of block erases by averagely 78.48% and 71.13% than the bucket PR quadtree and R*-tree, respectively, which implies slower device wearing and reduced garbage collection overheads (Fig. 7). Next, the read traffic and space utilization of the FIRM-tree are similar to those of the control groups (Figs. 8 and 9). Last but not least, due to its conservative node splitting strategy, the FIRM-tree occasionally incurs slightly more node splitting than the control groups in some cases (Fig. 10). Overall, with the flash performance statistics of [21], the page read latencies are 53 μs and 66 μs for reprogrammable flash and normal flash, respectively, while the write latencies are the same as above. Based on these statistics, in terms of the extra read and write time (other than the necessary time to write the data points themselves), the FIRM-tree outperforms the bucket PR

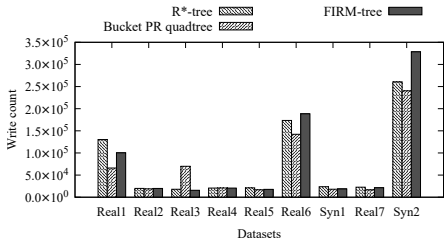

Fig. 6. Part D: total write counts.

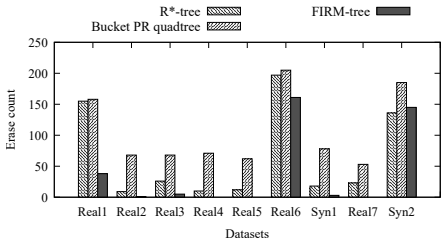

Fig. 7. Part D: total erase counts.

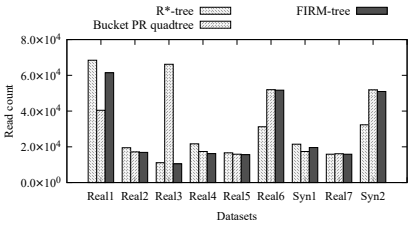

Fig. 8. Part D: total read counts.

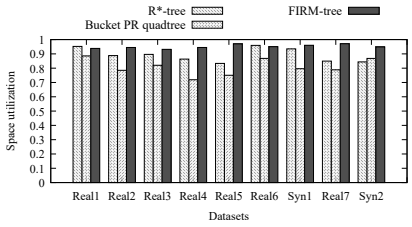

Fig. 9. Part D: space utilization.

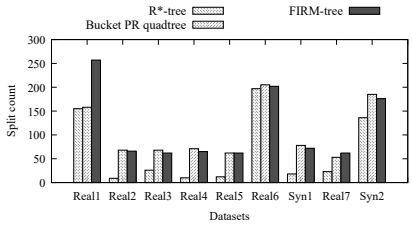

Fig. 10. Part D: total node split counts.

TABLE III
ENVIRONMENTAL SETTINGS OF THE TRACE-DRIVEN SIMULATION

Processor	Intel i5-11400
Memory	16 GiB
Operating system	Linux 5.15.0-88-generic (Ubuntu 20.04.3LTS)
Programming language & compiler	C++17, GCC version 9.4.0
SSDsim version [25]	2.x

TABLE IV
SSD SIMULATOR CONFIGURATIONS

1 SSD=4 channels	1 channel=4 chips	1 chip=2 dies	1 die=2 planes
1 plane=2^{11} blocks	1 block=2^8 pages	1 page=4 KB	

quadtree and R*-tree by averagely 20.75% and 15.81% among all workloads, respectively.

2) Part U: Impacts of the Space Utilization of New Writes: In the second part of our experiments, namely Part U, we compare the FIRM-tree to the control groups with different space utilization criteria of the FIRM-tree, which is controlled by the parameter $m = 2, \ldots, 5$ and the corresponding space utilization criteria $\mu^* = \frac{1}{2}, \ldots, \frac{4}{5}$. (The FIRM-tree can guarantee arbitrary space utilization criteria between 0 and 1. However, only the most common space utilization criteria are used in the experiments.) Throughout this part of experiments, we used a realistic workload `Real6` and a synthetic workload `Syn1`, both with 4D data points. As can be observed, the five performance metrics of the FIRM-tree are only slightly affected by μ^* (Figs. 12–15). Interestingly, although the resulted average space utilization of the FIRM-tree increases accordingly with m, that of the bucket PR quadtree and R*-tree also provide good space utilization (Fig. 14). Notably, the number of block erases of the FIRM-tree is considerably lower than that of the control groups, as in Part D.

3) Part B: Impacts of the Flash Block Size and Node Capacity: In Part B of the experiments, the impacts of the flash block size on the performance of the FIRM-tree and the control groups are measured. While the size of a node is fixed and is as large as a block, we adjust the number of fixed-sized pages in a block from 64 to 512. Two realistic workloads, `Real5` and `Real6`, as well as a synthetic workload `Syn1`, are used in this part. All three workloads are with 4D point data. In the bucket PR quadtree and R*-tree, each 4 KB page can keep up to $\left\lfloor \frac{4\text{ KB}}{4\cdot 8\text{ B}} \right\rfloor = 128$ data points; however, in the FIRM-tree, each page can only keep $\left\lfloor 128 \cdot \frac{2}{3} \right\rfloor = 85$ data points, due to the enabling of page reprogramming capability. The space utilization criterion is fixed at $\frac{1}{2}$.

As Figs. 16 and 18 show, the write and read traffic of the FIRM-tree is close to those of the bucket PR quadtree and R*-tree, due to the joint effects of skewness in data point distributions, frequency of garbage collection, and different ways to handle the data points in the to-be-split node. Due to the the exploitation of page reprogramming operations, in average, the FIRM-tree yields relatively 67.20% and 65.78% fewer block erases than the bucket PR quadtree and R*-tree do, respectively (Fig. 17). Furthermore, due to the conservative node splitting strategy, the space utilization of the FIRM-tree is higher than that of the bucket PR quadtree and R*-tree by averagely 15.82% and 5.43%, respectively (Fig. 19). Last but not least, the number of node splitting of the FIRM-tree is close to the bucket PR quadtree, as shown in Fig. 20.

4) Part L: Impacts of the Number of Levels Skipped by the Mega Root: In the last part of the experimental studies, Part L, we consider the effects of level skipping of the mega root design of the FIRM-tree. Intuitively, a larger size of the mega root implies that more levels are skipped by the flash-part of the FIRM-tree. As reported by Figs 22–25, the size of the mega root only has a minor effect on all performance metrics of the FIRM-tree, and can be scaled freely according to the available RAM buffer size.

VI. RELATED WORK

A. Emerging Nonvolatile Memories (NVMs)

Classical memory and storage media, such as SRAM, DRAM, and mechanical hard disks (HDDs), face greater challenges in fulfilling the requirements of next-generation applications, e.g., IO performance, energy efficiency, and storage density. To address this problem, much research effort has been dedicated to developing diverse types of NVMs, such as block-based non-volatile memories (NVMs) and byte-addressable persistent memories (PMs). However, considering the increasing number of data-centric computing scenarios, innovative NVMs have widely become comparative candidates,

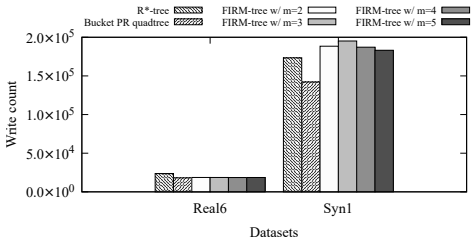

Fig. 11. Part U: total write counts.

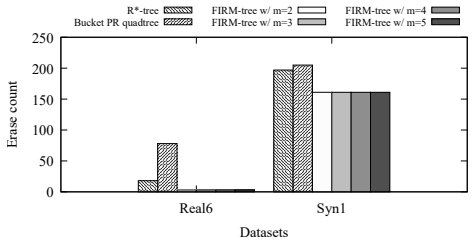

Fig. 12. Part U: total erase counts.

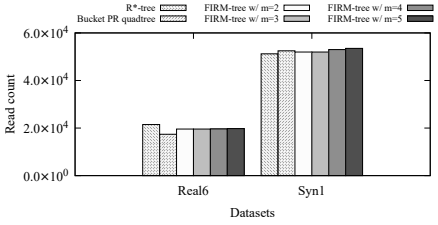

Fig. 13. Part U: total read counts.

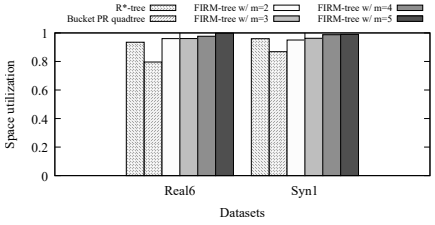

Fig. 14. Part U: space utilization.

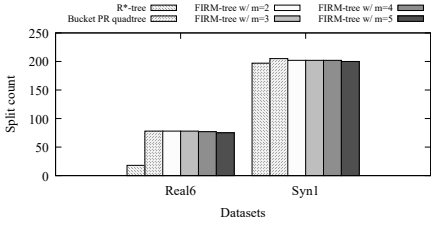

Fig. 15. Part U: total node split counts.

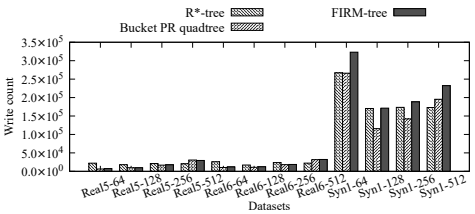

Fig. 16. Part B: total write counts.

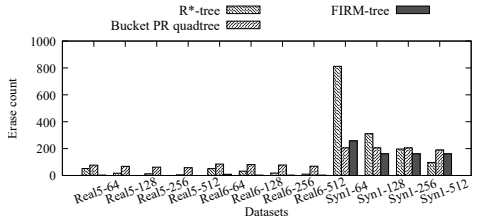

Fig. 17. Part B: total erase counts.

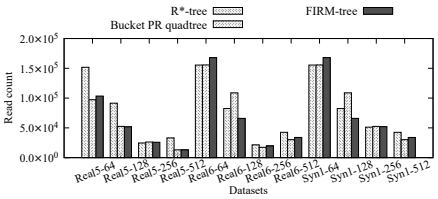

Fig. 18. Part B: total read counts.

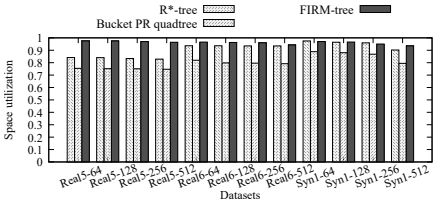

Fig. 19. Part B: space utilization.

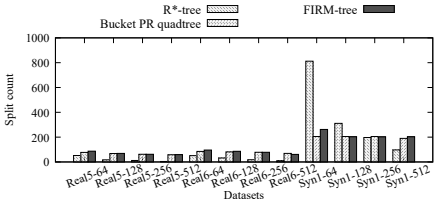

Fig. 20. Part B: total node split counts.

including flash memory [30], resistive random-access memory (ReRAM) [9], spin-torque transfer magnetic random-access memory (STT-RAM) [31], phase-change memory (PCM) [9], and skyrmion racetrack memory (SK-RM) [32]. Among various media choices, NAND flash memory is the dominant choice of storage media in different devices, e.g., memory cards, USB flash disks, and solid-state disks (SSDs) [13], [30].

To optimize the access performance and memory lifetime, dedicated management facilities are indispensable for addressing the intrinsic characteristics of flash memory, such as the page-based data accesses, write-once property, bulk erase property, and worn-out effect [13]. These management facilities are often implemented as a firmware layer called the *flash translation layer (FTL)*, which comprises an *address translator* [33], [34], *garbage collector* [33], [35], and *wear-leveler* [33], [35]. The address translator keeps track of the current physical page that stores each logical block of data, so as to efficiently serve read requests even when the data are *out-place updated* and moved in the flash space from time to time. To address the write-once and bulk erase properties, the garbage collector is responsible for selecting and erasing certain physical blocks to release the invalid pages within. Last but not least, the wear-leveler equalizes the wearing of different physical blocks to prevent early bad blocks and extend the flash lifetime. Due to their tremendous impacts on the performance and endurance of the flash storage devices [13], these facilities should be carefully designed, possibly with the knowledge of operating systems or user applications.

To economically store a growing volume of data on flash storage devices, the storage density is a factor that must be considered. There are two common approaches to enhancing the storage density, shifting from 2D planar flash memories to 3D flash memories or increasing the cell levels of flash memory. Conceptually, 3D flash memory stacks multiple layers of 2D flash cell arrays to improve storage density and access performance, at the price of degraded data reliability due to the interference across different layers [12], [13], [30], [36]. On the other hand, storing more than one bit of data in each cell can also effectively extend the storage density by several times. While the classical *single-level-cell (SLC)* flash memory stores only 1 b of data per cell, each cell of the *multi-level-cell (MLC)*, *triple-level-cell (TLC)*, and *quad-level-cell (QLC)* flash memory can store 2, 3, and 4 b per cell, thereby enhancing the storage density by 2×, 3×, and 4×, respectively [12],

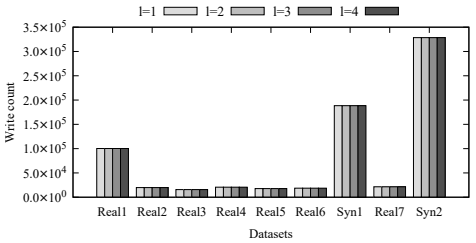

Fig. 21. Part L: total write counts.

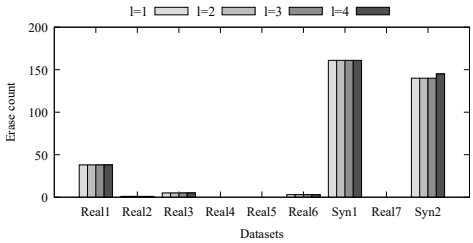

Fig. 22. Part L: total erase counts.

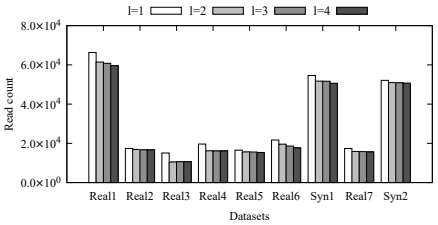

Fig. 23. Part L: total read counts.

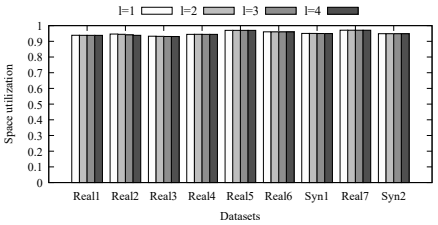

Fig. 24. Part L: space utilization.

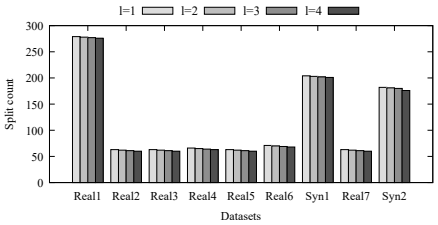

Fig. 25. Part L: total node split counts.

[37]. Like cross-layer interference of 3D flash memories, the narrowing gap between adjacent reference voltages in higher-level-cell flash memories also leads to potential reliability problems that must be treated with care [13].

The choice of different levels of flash memory cells is not just a trade-off among the storage density, access performance, and data reliability [21]. Instead, it introduces new opportunities for acrobatic optimization techniques, such as the *SLC-mode write buffering* [38] and *reprogramming capability* [21]. The SLC-mode write buffering allows a part of MLC/TLC/QLC-mode flash space to be written as in the SLC mode, which trades some storage capacity for enhanced write performance [38]. When the residual space is used up, the SLC-mode write buffer will be converted back into MLC/TLC/QLC mode to release more residual space for serving subsequent writes. On the other hand, a higher-level flash cell may also be programmed in a lower-level mode, thereby trading storage capacity for the capability to relax the write-once property and allow reprogramming a previously written page for a limited number of times [21]. For specific application scenarios like the FIRM-tree, the reprogramming capability might be helpful for reducing the garbage collection overheads and enhancing the overall system performance.

B. From Unidimensional to Multidimensional Data Structures

To efficiently store and access data, index data structures play a key role in diversified application scenarios, such as real-time systems, database systems, geographic information systems, data mining, and image processing. Driven by the trade-off among the performance of various access operations, such as insert, update, query, and delete operations, the design of different index data structures often vary a lot. While many common data structures exist, such as the stacks, queues, heaps, hash tables, skip lists, and search trees [14], they may be classified as *unidimensional* or *multidimensional*, according to the dimensionality of data [1]. Taking search trees for example, the binary search tree, red–black tree, and B-tree are representative unidimensional data structures [14], while the quadtree [1], [8], kd-tree [6], octree [39], R-tree [3], and SR-tree [40] are all milestones of multidimensional data structures.

Now let us take a closer look at multidimensional index data structures, which may be further classified into *kd-trees* [6], [7], [15], [16], *quadtrees* [8], [41], *grid files* [2], [19], and *R-trees* [3]–[5], according to the different ways that data points are organized. First of all, the kd-tree alternately selects a dimension from the k dimensions of the keys at each level to partition the data points [6]. Based on the kd-tree, the KDB-tree incorporates the idea of B-tree to allow multiple data points to be stored in the same node [7]. In contrast to the kd-tree, the quadtree partitions its data points from all k dimensions at every level [8], and each internal node of a quadtree will have up to 2^k children[3]. Based on the quadtree, the skip quadtree integrates a deterministic skipped list into a quadtree to enhance the search performance [41]. To improve scalability, the bucket PR quadtree uses large buckets and allows storing multiple data points in the same bucket [1]. As a variant of both kd-tree and quadtree, the PK-tree adaptively creates new parent nodes only when there are too many child nodes, so as to facilitate efficient range and kNN queries [42].

Next, the grid file is a pioneering work that extends conventional sequential or hashed file organizations to improve the performance of multidimensional data operations, such as inserts, queries, or deletes, on hard disks [2]. To adapt to dynamically shifting access patterns, the *balanced and nested grid (BANG) file* introduces the self-balancing idea of B-tree into the grid file, to maintain a balanced tree structure for the performance enhancement of queries [43]. Last but not least, R-tree and its variants have been recognized as an efficient multidimensional index data structure for range and kNN queries [1], [3]–[5]. By partitioning the data points of an overflowing node into subsets of approximately the same sizes, the R-tree guarantees the absence of nodes with few data points, thus ensuring satisfactory space utilization [3]. Based on the R-tree, the R$^+$-tree [4] and R*-tree [5] strives to optimize the geometries of the minimum bounding rectangles (MBRs) of nodes in terms of different metrics, such as the overlapping area of the MBRs.

Recently, the emergence of novel memory and storage media has driven the development of medium-specific index data structures. For example, as an extension of the KDB tree, F-KDB utilizes a logging buffer in the main memory to alleviate the write amplification due to random insertion

[3]In this paper, we use the more intuitive symbol d to denote the dimensionality of data, instead of k used by the original literature of the kd-tree [6].

of data points on NAND flash memory [15], [16]. On the other hand, as a variant of the grid file, the log bucket grid file (LB-Grid), has been proposed for flash memory based SSDs [19]. To suppress inefficient random writes to flash memory, LB-Grid exploits a log buffer to make reads and writes parallel across multiple channels of the SSDs, significantly improving the throughput of inserts and queries. Moreover, there exist index data structures for hybrid storage, such as the HyR-tree [20] for a hybrid of Intel 3D XPoint memory [44], [45] and NAND flash memory. HyR-tree makes use of an unsupervised learning approach to identify the hotness of all nodes, and arbitrate which nodes should be kept in the faster, byte-addressable 3D XPoint memory and which should be stored in the NAND flash memory. While many brilliant designs of medium-specific multidimensional index data structures have been proposed, however, there are still missing pieces in exploiting the unique access operations of NAND flash memory, such as the reprogramming capability of multilevel cell flash memory [21], which motivates our work.

VII. Conclusion and Future Work

The widespread use of multidimensional data highlights the significance of multidimensional data structures. While there have been many work of multidimensional data structures, they do not thoroughly consider the characteristics of modern storage media, e.g., NAND flash memory. In this work, we present the FIRM-tree, a novel data structure for multidimensional point data on NAND flash memory. By holistic management of RAM/flash space and dedicated exploitation of page reprogrammability of modern flash memory, the FIRM-tree alleviates read/write amplification and garbage collection overheads for tree maintenance and data point reclassification. Moreover, the FIRM-tree presents a conservative node splitting strategy to improve flash space utilization. Experimental results show the performance superiority of the FIRM-tree.

References

[1] H. Samet, *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[2] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 1, pp. 38–71, 1984.

[3] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, 1984, pp. 47–57.

[4] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects." in *Intl. Conf. on Very Large Data Bases (VLDB)*, 1987, pp. 507–518.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R^*-tree: An efficient and robust access method for points and rectangles," in *ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, 1990, pp. 322–331.

[6] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[7] J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," in *ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, 1981, pp. 10–18.

[8] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, pp. 1–9, 1974.

[9] S. Hong, O. Auciello, and D. Wouters, *Emerging non-volatile memories*. Springer, 2014.

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

[11] Y. Huai *et al.*, "Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects," *AAPPS bulletin*, vol. 18, no. 6, pp. 33–40, 2008.

[12] S. Aritome, *NAND flash memory technologies*. Wiley-IEEE Press, 2015.

[13] R. Micheloni *et al.*, *3D Flash Memories*. Springer, 2016.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 4th Edition*. MIT press, 2022.

[15] G. Li, P. Zhao, S. Gao, and J. Du, "F-KDB: An K-D-B Tree Implementation over Flash Memory," in *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE, 2010, pp. 635–642.

[16] G. Li, P. Zhao, L. Yuan *et al.*, "Efficient implementation of a multi-dimensional index structure over flash memory storage systems," *Journal of Supercomputing*, vol. 64, pp. 1055–1074, 2013.

[17] A. Fevgas, L. Akritidis, P. Bozanis, and Y. Manolopoulos, "Indexing in flash storage devices: a survey on challenges, current approaches, and future trends," *The VLDB Journal*, vol. 29, pp. 273–311, 2020.

[18] A. C. Carniel and C. D. de Aguiar, "Spatial index structures for modern storage devices: A survey," *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[19] A. Fevgas and P. Bozanis, "LB-Grid: an SSD efficient grid file," *Data & Knowledge Engineering*, vol. 121, pp. 18–41, 2019.

[20] A. Fevgas, L. Akritidis, M. Alamaniotis, P. Tsompanopoulou, and P. Bozanis, "HyR-tree: a spatial index for hybrid flash/3D XPoint storage," *Neural Computing and Applications*, pp. 1–13, 2021.

[21] C. Gao, M. Ye, C. J. Xue, Y. Zhang, L. Shi, J. Shu, and J. Yang, "Reprogramming 3D TLC Flash Memory based Solid State Drives," *ACM Transactions on Storage (TOS)*, vol. 18, no. 1, pp. 1–33, 2022.

[22] S. Sarkar *et al.*, "Lethe: A tunable delete-aware LSM engine," in *ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, 2020, pp. 893–908.

[23] A. Fiat and H. Kaplan, "Making data structures confluently persistent," *Journal of Algorithms*, vol. 48, no. 1, pp. 16–58, 2003, aCM-SIAM Symposium on Discrete Algorithms (SODA).

[24] B. Kim and M. Kim, "LazyRS: Improving the performance and reliability of high-capacity TLC/QLC flash-based storage systems using lazy reprogramming," *Electronics*, vol. 12, no. 4, 2023.

[25] Y. Hu *et al.*, "Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity," in *International Conference on Supercomputing*, 2011, pp. 96–107.

[26] A. Eldawy and M. F. Mokbel. (2015) SpatialHadoop: A Mapreduce Framework for Spatial Data. [Online]. Available: http://spatialhadoop.cs.umn.edu/datasets.html

[27] D. Dua and C. Graff, "UCI Machine Learning Repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[28] "Geonames database," https://www.kaggle.com/datasets/geonames/geonames-database/data, Accessed: 2024-05-28.

[29] "Urban Institute," https://datacatalog.urban.org/dataset/resource/4c9440ba-d711-46d9-8b85-8a11c3224966, Accessed: 2024-05-28.

[30] H. Maejima *et al.*, "A 512Gb 3b/Cell 3D flash memory on a 96-word-line-layer technology," in *IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2018, pp. 336–338.

[31] J.-G. Zhu, "Magnetoresistive random access memory: The path to competitiveness and scalability," *Proceedings of the IEEE*, vol. 96, no. 11, pp. 1786–1798, 2008.

[32] R. Tomasello, E. Martinez, R. Zivieri, L. Torres, M. Carpentieri, and G. Finocchio, "A strategy for the design of skyrmion racetrack memories," *Scientific reports*, vol. 4, no. 1, pp. 1–7, 2014.

[33] T.-S. Chung *et al.*, "A survey of flash translation layer," *Journal of Systems Architecture (JSA)*, vol. 55, no. 5, pp. 332–343, 2009.

[34] D. Ma, J. Feng, and G. Li, "A survey of address translation technologies for flash memories," *ACM Comput. Surv.*, vol. 46, no. 3, jan 2014.

[35] M.-C. Yang *et al.*, "Garbage collection and wear leveling for flash memory: Past and future," in *2014 International Conference on Smart Computing*. IEEE, 2014, pp. 66–73.

[36] K.-T. Park *et al.*, "Three-Dimensional 128 Gb MLC Vertical NAND Flash Memory With 24-WL Stacked Layers and 50 MB/s High-Speed Programming," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 204–213, 2014.

[37] Y. Takai *et al.*, "Analysis on heterogeneous SSD configuration with quadruple-level cell (QLC) NAND flash memory," in *IEEE International Memory Workshop (IMW)*. IEEE, 2019, pp. 1–4.

[38] K. Kwon, D. H. Kang, and Y. I. Eom, "An advanced SLC-buffering for TLC NAND flash-based storage," *IEEE Transactions on Consumer Electronics*, vol. 63, no. 4, pp. 459–466, 2017.

[39] D. Meagher, "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, vol. 19, no. 2, pp. 129–147, 1982.

[40] N. Katayama and S. Satoh, "The SR-tree: An index structure for high-dimensional nearest neighbor queries," *ACM Sigmod Record*, vol. 26, no. 2, pp. 369–380, 1997.

[41] D. Eppstein, M. T. Goodrich, and J. Z. Sun, "The skip quadtree: a simple dynamic data structure for multidimensional data," in *Annual Symposium on Computational Geometry*, 2005, pp. 296–305.

[42] W. Wang, J. Yang, and R. Muntz, "PK-tree: a dynamic spatial index structure for large data sets," *UCLA Computer Science Department Technical Report*, vol. 970039, 1997.

[43] M. Freeston, "The BANG file: a new kind of grid file," *ACM SIGMOD Record*, vol. 16, no. 3, pp. 260–269, 1987.

[44] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3D XPoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[45] I. Koltsidas and V. Hsu, "IBM storage and NVM express revolution," *IBM RedBooks*, 2017.