Adapting B$^+$-Tree for Emerging Nonvolatile Memory-Based Main Memory

Ping Chi, *Student Member, IEEE*, Wang-Chien Lee, *Member, IEEE*, and Yuan Xie, *Fellow, IEEE*

***Abstract*—Among the emerging nonvolatile memory (NVM) technologies, some resistive memories, including phase change memory (PCM), spin-transfer torque magnetic random access memory (STT-RAM), and metal-oxide resistive RAM (ReRAM), have been considered as promising replacements of conventional dynamic RAM (DRAM) to build future main memory systems. Main memory databases can benefit from their nice features, such as their low leakage power and nonvolatility, the high density of PCM, the good read performance and low read energy consumption of STT-RAM, and the low cost of ReRAM's crossbar architecture. However, they also have some disadvantages, such as their long write latency, high write energy, and limited lifetime, which bring challenges to database algorithm design for NVM-based memory systems. In this paper, we focus on the design of the ubiquitous B$^+$-tree, aiming to make it NVM-friendly. We present a basic cost model for NVM-based memory systems which distinguishes writes from reads, and propose detailed CPU cost and memory access models for search, insert, and delete operations on a B$^+$-tree. Based on the proposed models, we analyze the CPU costs and memory behaviors of the existing NVM-friendly B$^+$-tree schemes, and find that they suffer from three issues. To address these issues we propose three different schemes. Experimental results show that our schemes can efficiently improve the performance, reduce the memory energy consumption, and extend the lifetime for NVM-based memory systems.**

***Index Terms*—Algorithm design, B$^+$-tree, cost model, main memory database, nonvolatile memory (NVM).**

I. Introduction

EMERGING nonvolatile memory (NVM) technologies have been developed and studied to be adopted in memory systems, since the conventional memory technologies, such as static random access memory (SRAM) and dynamic RAM (DRAM), are facing the scalability challenges, e.g., the increasing leakage power dissipation [1]. Among the emerging NVMs, some resistive memories, including phase change memory (PCM), spin-transfer torque magnetic RAM (STT-RAM), and metal-oxide resistive RAM (ReRAM), have been considered as promising replacements of DRAM for building future main memory systems [2]–[4].

Compared with the modern DRAM technology, the emerging NVMs consume near-zero idle power and have better scalability. Owing to their nonvolatility, they are free of refresh penalty, which has recently been found a big issue of DRAM as its density advances [5]. Additionally, they can support the durability property that traditional main memory databases (MMDBs) built upon volatile memory do not facilitate. Moreover, PCM and ReRAM offer a higher density than DRAM; the multilevel cell (MLC), multilayer structure, and 3-D stacking technologies can further enhance their density [4]. Therefore, under the same area constraint, they can provide a larger memory capacity, and can store most or all of the data in the main memory for many database applications. A previous study has shown that a 4× increase in the memory capacity reduces the number of page faults by 5× on average [2]. Although STT-RAM does not have a density advantage, it provides better read and write performance and energy as well as higher endurance than PCM and ReRAM [3]. With the continuous advance in these technologies, they are anticipated for use in building energy-efficient nonvolatile main memory systems of the near future.

Although the nice features of PCM, STT-RAM, and ReRAM bring great benefits to MMDBs, new challenges arise due to some of their characteristics. Different from DRAM, they have asymmetric read/write properties. Their read latencies are comparable to that of DRAM, but their write latencies are much slower than their corresponding read latencies [6]. In addition, their write operations consume much more energy than their read operations [6]. They also suffer from endurance issues, and thus we need to specifically consider their wear-out problems. These unique characteristics change the assumptions that have served as the basis in the designs of conventional database algorithms, making them suboptimal for the emerging NVM-based main memory systems. Aiming to tackle this issue, we re-examine the design of database algorithms. As writes are much more expensive than reads in these emerging NVMs, the new algorithm design goal is to reduce memory writes, even at the cost of increasing reads.

In this paper, we focus our attempt on B$^+$-tree, the widely used index structure in both MMDBs and disk-resident databases (DRDBs), to design an NVM-friendly variant of B$^+$-tree. Previously, Chen *et al.* [7] proposed unsorted node schemes in their redesign of B$^+$-tree algorithm for PCM. They showed that, using unsorted nodes instead of sorted nodes in

Manuscript received March 31, 2015; revised September 26, 2015; accepted December 2, 2015. Date of publication December 29, 2015; date of current version August 18, 2016. This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award DE-SC0013553 with disclaimer at http://seal.ece.ucsb.edu/doe/, in part by the National Science Foundation under Grant 1461698 and Grant 1500848, and in part by the Qualcomm. This paper was recommended by Associate Editor T.-Y. Ho.

P. Chi and Y. Xie are with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 USA (e-mail: pingchi@ece.ucsb.edu; yuanxie@ece.ucsb.edu).

W.-C. Lee is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802 USA (e-mail: wlee@cse.psu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2015.2512899

the tree can effectively reduce the write accesses required for keeping the nodes in a sorted order. However, we find that the unsorted node schemes suffer from three problems, according to the analytical results from our proposed cost model for the operations of B^+-tree on NVM-based memory systems. We then propose three techniques to address those problems.

In the evaluation, we compare our proposed sub-balanced unsorted node scheme, overflow node scheme, and merging factor scheme with the existing NVM-friendly unsorted node scheme and the original B^+-tree algorithm, for PCM, STT-RAM, and ReRAM-based main memories under the insert-, search-, and delete-only workloads. The experiment results show that, for insert operations, both the unsorted node scheme and the sub-balanced unsorted node scheme can reduce the write accesses and memory energy consumption efficiently for NVM-based systems when the node size ranges from 4 to 16 cache lines. For example, they reduce the write accesses by 19.3%–62.1%, and save the memory energy consumption by 15.0%–66.8% for the PCM-based main memory. However, the sub-balanced unsorted node scheme has better execution time than the unsorted node scheme by removing the computation-intensive sorting before node splits, for example, reducing the execution time by 1.1%–5.5% for STT-RAM and ReRAM-based systems when the node size ranges from 2 to 16 cache lines. For search operations, they have similar execution time and memory energy consumption; and for delete operations, the sub-balanced unsorted node scheme has better execution time and memory energy consumption than the unsorted node scheme. The experiment results also demonstrate that, when the node size is 2 cache lines, the unsorted node scheme as well as the sub-balanced unsorted node scheme cannot effectively reduce the write accesses in insert operations, because the tree reorganization causes a lot of writes and they are not able to reduce them. Then, we evaluate our proposed overflow node scheme for small branching factors and node sizes, and the experiment results show that it can reduce the write accesses efficiently in insert and delete operations. When the node size is 2 cache lines and the branching factor is 6, compared with the unsorted node scheme, it decreases the write accesses by 11.5%–18.2% as the overflow factor varies from 1 to 4 under the insert-only workload, and by 12.4% when the overflow factor is 2 under the delete-only workload. We also evaluate our merging factor scheme on the original B^+-tree, the unsorted node scheme, the sub-balanced unsorted node scheme, and the overflow node scheme, by varying the merging factor from 0.5 to 0 under a delete-only workload. The experiment results present that, for delete operations in all NVM-based systems, as the merging factor decreases, the execution time, the read and write accesses, and the memory energy consumption all decrease, with an increase in the space usage. Therefore, the merging factor scheme provides more tradeoff options.

In this paper, we make the following three contributions.

1) We develop a basic cost model for NVM-based memory systems, reflecting their asymmetric read/write properties. Accordingly, we also build detailed CPU computation and memory access models for search, insert, and delete operations on a B^+-tree.
2) We use our cost model to analyze the existing NVM-friendly schemes for B^+-tree, i.e., the unsorted node schemes proposed by Chen *et al.* [7]. The results show three problems.
   a) In insert operations the node splits are CPU-costly, because sorting the keys in an unsorted node before a split requires intensive computations.
   b) The write accesses in insert operations cannot be reduced effectively when branching factors and node sizes are both small, because in this case tree reorganization incurs a lot of writes.
   c) The delete operations may waste space significantly because a node is not deleted until it becomes empty.
3) To address the aforementioned problems, we propose three schemes to adapt B^+-tree for the emerging NVMs as follows.
   a) The sub-balanced unsorted node scheme which alleviates the computational overhead of sorting before a split is incurred in insert operations.
   b) The overflow node scheme which can efficiently reduce the write accesses in insert operations when the existing schemes do not work.
   c) The merging factor scheme which provides more effective tradeoffs among execution time, memory energy consumption, memory space usage, and NVM endurance in delete operations.

The remainder of this paper is organized as follows. In Section II, we introduce the background and related work on PCM, STT-RAM, and ReRAM technologies, NVM-based system design, MMDB index, and cost model for database algorithms. In Section III, we present a basic cost model for NVM-based memory systems, and design detailed CPU cost and memory access models for search, insert, and delete operations on a B^+-tree. Moreover, based on the models we build, we analyze existing NVM-friendly schemes for B^+-tree. In Section IV, we propose new schemes that effectively adapt B^+-tree to exploit the full potential of the emerging NVMs without suffering from the issues arising in the existing schemes. In Section V, we conduct experimental study to evaluate the proposed B^+-tree variants with the state-of-the-art. We conclude the paper in Section VI.

II. Background and Related Work

In this section, we introduce the basics of PCM, STT-RAM, and ReRAM, and related work on NVM-based system design, index for MMDB, and cost model for database algorithms.

A. PCM, STT-RAM, and ReRAM Basics

Among the emerging NVMs, resistive random access memory, is a type of NVM that uses the cell resistance to store the information by changing it between a high-resistance state (HRS) and a low-resistance state (LRS). It is known as ReRAM for short. However, ReRAM typically refers to metal-oxide ReRAM, a subset using metal oxides as resistive switches. In this paper, we use the term "ReRAM" in a typical way. PCM and STT-RAM are also resistive memories.

PCM exploits the unique behavior of phase change material, e.g., chalcogenide glass, which enters two different states under different heating temperatures and durations [2].

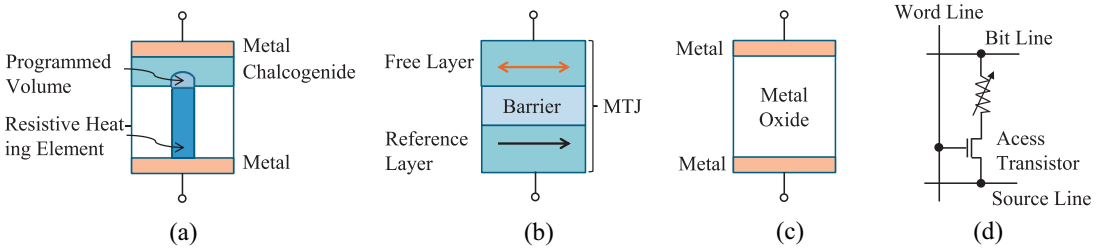


Fig. 1. Basics of NVM cell structures (not to scale). (a) PCM cell. (b) STT-RAM cell. (c) ReRAM cell. (d) 1T1R cell structure.

These two states, termed as amorphous state (HRS) and crystalline state (LRS), have significantly different electrical resistivities, and thus they can represent "0" and "1," respectively. Moreover, as this material can achieve several distinct intermediate states, it has the ability to represent multiple bits in a single cell. Fig. 1(a) depicts a PCM cell. PCM adopts the one-transistor–one-resistor (1T1R) cell structure, as shown in Fig. 1(d). Like a DRAM cell, each PCM cell has an access transistor to prevent the disturbance with other cells.

STT-RAM is a new generation of magnetic RAM, using spin-transfer torque to switch memory states. Fig. 1(b) shows a magnetic tunnel junction (MTJ), the key component to store the bit information in a STT-RAM cell. An MTJ consists of two ferromagnetic layers and one oxide barrier layer (e.g., MgO) that separates them. The magnetization direction of one ferromagnetic layer (i.e., reference layer) is fixed, called reference layer; while that of the other ferromagnetic layer (i.e., free layer) can be changed by applying a large enough spin polarized current through MTJ, either in parallel or anti-parallel with the fixed direction of the reference layer. "Parallel" indicates MTJ in an LRS; in contrast, "anti-parallel" means HRS. To improve the density of STT-RAM, MLC technology is a feasible way. Different from PCM and ReRAM, we cannot achieve intermediate resistance states for STT-RAM only by tailoring the switching current/voltage. To implement MLC STT-RAM, we either stack two MTJs of different sizes vertically, or make the free layer of one MTJ possess two domains of different sizes. STT-RAM also adopts the 1T1R cell structure as shown in Fig. 1(d). It is called one-transistor–one-MTJ (or 1T1J) cell structure for STT-RAM.

Fig. 1(c) presents the metal–insulator–metal structure of a ReRAM cell. It contains a top metal layer, a bottom metal layer, and a metal-oxide layer in the middle as resistive switches. There are various oxide materials and metal choices, and the cell switching behavior depends on their interfacial properties [8]. According to the widely accepted filamentary model, during a SET (writing "1") operation, nanoscale conductive filaments (CFs) are formed and the cell is in an LRS; during a RESET (writing "0") operation, the CFs are cut off and the cell is in a HRS. Like PCM, ReRAM can achieve the MLC characteristic by controlling the switching current/voltage to obtain several intermediate resistance levels. [9]. ReRAM has the 1T1R cell structure as shown in Fig. 1(d). To reduce the cost, ReRAM also supports a cross-point architecture, which eliminates the access transistor, using a bidirectional diode as a selector or no selector at all to achieve the smallest cell size of $4F^2$.

TABLE I
COMPARISONS OF DIFFERENT MEMORY TECHNOLOGIES [3], [4], [6], [10], [11]

	DRAM	PCM	STT-RAM	ReRAM
Retention	refresh	non-volatile	non-volatile	non-volatile
Density	$1\times$	$2-4\times$	$0.4-0.8\times$	$2-4\times$
Endurance	10^{15}	10^8-10^9	$10^{12}-10^{15}$	$10^{10}-10^{12}$
Idle power	high	low	low	low
Read latency	$1\times$	$2-8\times$	$0.5-2\times$	$0.5-10\times$
Write latency	$1\times$	$10-100\times$	$1-8\times$	$1-100\times$
Read energy	$1\times$	$2-6\times$	$0.5-2\times$	$0.5-5\times$
Write energy	$1\times$	$5-50\times$	$0.5-10\times$	$0.2-100\times$

Despite of different materials and technologies, PCM, STT-RAM, and ReRAM have some common characteristics. First, all have asymmetric read and write performance and energy. The reason is that the READ current/voltage they use is smaller and with a shorter duration than WRITE so as not to perturb the cell state. Table I compares PCM, STT-RAM, and ReRAM with DRAM. Their write latency is several times longer than their read latency, and their write energy consumption is several times higher than their read energy consumption. Second, they all have the write endurance problem, i.e., each cell will be worn out after a limited number of writes. PCM suffers from the most severe endurance issue. ReRAM is two or three orders of magnitude better than PCM in endurance; although STT-RAM has the best endurance among these three emerging NVMs, it is not free of this issue. Moreover, they are all nonvolatile and have low idle power.

From Table I, we also find that, compared with DRAM, PCM, STT-RAM, and ReRAM have longer write latency and higher write energy. In addition, PCM and ReRAM can provide a higher density than DRAM. Although STT-RAM does not have the density advantage, its read and write latency and energy are better than those of PCM and ReRAM, providing another tradeoff among memory capacity, performance, and energy consumption.

A lot of research studies has been conducted to tackle their challenges and make them more feasible memory/storage replacements. Some studies improve their lifetime by using wear leveling strategies [2], [12], [13] or reducing redundant writes [12], [14], [15]. Some deal with the long write latency by using a DRAM buffer at the architecture level [2] or by reducing redundant writes at the bit level [12], [14], [15]. In this paper, we explore to reduce the write accesses at the algorithm level in order to achieve performance gain, energy saving, and lifetime extension in NVM-based memory systems.

B. NVM-Based System Design

With their unique characteristics, the read/write costs of emerging NVMs are no longer aligned with the assumptions of the underlying memory systems that have motivated various system designs, including file system design, operating system (OS) design, as well as database system design. Condit *et al.* [16] proposed a new file system based on the properties of persistent, byte-addressable NVMs. Bailey *et al.* [17] examined the implications of fast and cheap NVMs on OS functions and mechanisms.

Chen *et al.* [7] proposed to rethink database algorithms for PCM, which inspires this paper. They present analytical metrics for PCM endurance, energy, and latency, and use them to improve two core databases techniques, B^+-tree index and hash joins, for PCM. Their goal for designing PCM-friendly algorithms is to reduce the number of writes while maintaining good cache performance. To improve B^+-tree index, they used unsorted nodes instead of sorted nodes in the tree, saving the writes incurred by sorting a node. The unsorted node schemes are simple and effective. However, they suffer from some issues that our proposed schemes avoid.

Hu [18] proposed a predictive B^+-tree, called B^p-tree, for PCM-based database systems. She uses a small DRAM buffer to maintain a small B^+-tree for current insertions, and predicts future data distribution based on the summary of previously inserted keys in a histogram. Space is preallocated in the memory for data to be accessed in the near future so as to reduce the data movements caused by node splits and merges. Our approach is different as we minimize the hardware overhead and extra design efforts by avoiding the usage of additional DRAM buffer.

C. Index for Main Memory Database

MMDB systems [19] maintain most of the database or all of the database in the main memory. Compared with conventional DRDB systems, access time is reduced by orders of magnitude in DRAM-based database systems. The design goals of data structures and algorithms for MMDB are to reduce overall computation time, minimize memory space consumption, reduce cache misses, and minimize costly memory accesses.

Indexing techniques play an essential role in database systems due to their strength in speeding up associative search and retrieval of data records. Various indexing techniques, roughly classified as hash-based indexes and tree-like indexes, have been developed. Hash-based indexes, such as chained bucket hashing, linear hashing, and extendible hashing, can locate target data very efficiently owing to hash function calculation. However, they are inherently constrained to support exact search only. Tree-like indexes, such as AVL tree, B-tree, T-tree [20] and their variants, by preserving the order of key values, are able to support range query efficiently and flexibly. B^+-tree, a variant of B-tree, keeps all data in leaf nodes and uses the internal nodes to guide the search. It is easy to maintain and scan while providing fast access to data. Thus, it is widely used in both DRDB and MMDB systems. Rao and Ross [21] proposed cache sensitive B^+-tree (CSB^+-tree) to make B^+-tree cache conscious in the main memory. To build NVM-based database systems, the design of index structures and algorithms need to be reconsidered due to the changes in the properties of the underlying memory systems.

D. Cost Model for Database Algorithms

Since the 1980s, great attention has been given to I/O-efficient algorithm design since disk I/O has been the performance bottleneck for many computer applications including conventional DRDBs. Aggarwal and Vitter [22], Nodine and Vitter [23], Vitter and Shriver [24] provided I/O models for sorting-related problems in single disk and parallel disk systems.

Later, Boncz *et al.* [25] and Manegold *et al.* [26] demonstrated that main-memory access is increasingly a performance bottleneck for many computer applications including database systems, and is becoming a more and more significant—if not the major—cost component of database operations in MMDB systems. They first focused on hash join and formulated detailed analytical cost models to make optimal use of hierarchical memory systems in modern computer hardware [25]. Later, they proposed a technique to build generic database cost models for hierarchical memory systems given the algorithms' data access patterns [26]. Accordingly, the miss numbers of each level cache can be well predicted from the data access characteristics of the algorithms.

All the cost models introduced above are designed for the systems with conventional memory and storage technologies. They do not distinguish read and write accesses to the memory/storage systems because they are equally costly. However, for NVM-based memory systems, writes are more costly than reads. The cost model proposed in this paper is designed for NVM-based main memory systems, which explicitly differentiates write accesses from read accesses in the model of the memory system.

III. Cost Model

In this section, we first introduce the basic cost model for NVM-based main memory systems. Then we analyze both the CPU costs and the memory behaviors of the original B^+-tree algorithm, formulating the memory access numbers incurred by each operation. Based on the cost models we have built, we analyze the existing PCM-friendly unsorted node schemes [7] to find out how they reduce the write accesses and what problems they have.

A. Basic Cost Model

For each operation of a B^+-tree in MMDB, either search, insert, or delete, the execution time T is comprised of three components, the CPU execution time T_{CPU} including the access time to the on-chip L1 cache, the access time to all the cache levels except L1 T_{Cache}, and the access time to the main memory T_{Mem}

$$T = T_{\text{CPU}} + T_{\text{Cache}} + T_{\text{Mem}}. \tag{1}$$

T_{CPU} depends on the computational complexity of the algorithm used to implement each operation and the performance of the processor. Let I denote the basic instruction number of the algorithm, CPI denote the average cycle per

instruction of the processor to execute basic instructions which does not include memory access latencies in loads and stores, and f denote the frequency of the processor. Then

$$T_{\text{CPU}} = I \times \text{CPI}/f. \tag{2}$$

For a memory system with l levels of cache, let M_i and L_i denote the miss count and the access latency of the ith level cache, respectively, for $i = 1, 2, \ldots, l$. Then

$$T_{\text{Cache}} = \sum_{i=1}^{l-1} M_i \times L_{i+1}. \tag{3}$$

The miss count of the ith level cache

$$M_i = A_{\text{total}} \times \prod_{k=1}^{i} Mr_k \tag{4}$$

in which A_{total} is the total number of memory accesses, and Mr_k is the average miss rate of the kth level cache. A_{total} simply depends on the algorithm to implement each operation. However, Mr_k is determined by a couple of factors, including both the memory access patterns of the algorithms and the characteristics of the cache hierarchy (e.g., capacity, associativity, and replacement policies of each level).

Till now, we have not distinguished read from write both as memory accesses. SRAM and DRAM have similar read and write latency. However, for an emerging NVM, its write latency is much longer than its read latency. We divide T_{Mem} into two parts, the latency of reading cache lines from the main memory to the last level cache and the latency of writing cache lines back to the main memory. The second part can be partially or even completely hidden since writing cache lines back is not in the critical path. Let R_{total} and W_{total} denote the total read and write access numbers, then $A_{\text{total}} = R_{\text{total}} + W_{\text{total}}$. Let Lr_{NVM} and Lw_{NVM} denote the read and write access latencies of an emerging NVM. Then

$$T_{\text{Mem}} = R_{\text{total}} \times \left(\prod_{i=1}^{l} Mr_i\right) \times Lr_{\text{NVM}} + \alpha \times W_{\text{total}} \times \left(\prod_{i=1}^{l} Mr_i\right) \times Lw_{\text{NVM}} \tag{5}$$

in which α describes the average impact of writing cache lines back on T_{Mem}, $0 \le \alpha \le 1$. Different from the conventional DRAM-based main memory, for an emerging NVM in which Lw_{NVM} can be several times larger than Lr_{NVM} (Table I), writing may significantly stall the front-end cache line fetches. Therefore, to reduce W_{total} might be beneficial to performance improvement.

In fact, memory access situations are much more complicated than (3) and (5) show. For example, a read request and a write request to the same bank will block each other, that is, the later request has to wait until the previous one gets completed. Several memory access scheduling schemes have been proposed to improve performance for DRAM and NVMs [27], [28].

Because emerging NVMs have high write energy and suffer from the endurance problem, the energy consumption of the NVM-based main memory, E_{Mem}, and the total wear of NVM, $\text{Wear}_{\text{total}}$, are another two concerns in algorithm design, besides the total execution time T in (1). To estimate E_{Mem}, we consider three parts: 1) the read dynamic energy; 2) the write dynamic energy; and 3) the background energy. Let Er_{NVM} and Ew_{NVM} denote the average energy consumption of a read access and a write access to NVM, and $E_{\text{background}}$ denote the background energy. Then

$$E_{\text{Mem}} = R_{\text{total}} \times \left(\prod_{i=1}^{l} Mr_i\right) \times Er_{\text{NVM}} + W_{\text{total}} \times \left(\prod_{i=1}^{l} Mr_i\right) \times Ew_{\text{NVM}} + E_{\text{background}}. \tag{6}$$

The background energy in NVM is typically much smaller than the read and write dynamic energy. From Table I, Ew_{NVM} can be several times larger than Er_{NVM}. Therefore, reducing W_{total} is helpful to save energy.

Each NVM cell has a limited lifetime. Let γ represent the average number of modified bits per modified cache line, then

$$\text{Wear}_{\text{total}} = \gamma \times W_{\text{total}} \times \left(\prod_{i=1}^{l} Mr_i\right). \tag{7}$$

Obviously, reducing W_{total} extends the lifetime of NVM.

B. B$^+$-Tree Parameters

B$^+$-tree is widely used for indexing in file systems and database management systems. It has several important parameters. It can have different key types for different applications, e.g., integer or string of a fixed length. Let key_size denote the length of a key in unit of bytes.

Node size (node_size) is an important parameter that affects the performance of B$^+$-tree. Previous work has suggested that the node size that results in the best performance should be a few times of the cache line size (cacheline_size) [29], [30], e.g., 2 cache lines or 4 cache lines. In modern computers, cacheline_size is usually 32, 64, or 128 bytes. The best node size also depends on the branching factor of the tree.

The branching factor (or the order) of a B$^+$-tree is the maximum number of children that each internal node can have, denoted by b. Then there are at most $b-1$ keys and b pointers in an internal node. The size of a pointer pointer_size is usually four bytes in a 32-bit machine and eight bytes in a 64-bit machine. For leaves, we assume that a record is a tuple of <key, pointer>, in which the pointer points to the data value. Therefore, the leaves have the same node structure with the internal nodes, and there are at most $b-1$ records in a leaf.

Each node also maintains some necessary information in their node structure, e.g., the number of keys and a flag indicating a leaf or nonleaf node. Let nodeinfo_size denote the space it takes a node to store the information. Then

$$b = \lfloor (\text{node\_size} - \text{nodeinfo\_size} + \text{key\_size}) / (\text{key\_size} + \text{pointer\_size}) \rfloor. \tag{8}$$

The height of a B$^+$-tree, denoted by h, indicates how many nodes in a search path, which means how many nodes we have to visit from the root to a leaf. Therefore, it is quite

TABLE II
ANALYTICAL COST MODEL FOR A SEARCH OPERATION

	T_{CPU_Search}	R_{total_Search}
L	$\alpha_L \times (f \cdot b/2) \times h$	$(1 + \lceil \frac{f \cdot node\_size}{cacheline\_size} \rceil)/2 \times h$
B	$\alpha_B \times \log_2 (f \cdot b) \times h$	$(1 + \lfloor \log_2 \frac{f \cdot node\_size}{cacheline\_size} \rfloor) \times h$

important to the search performance of a B$^+$-tree. Given the total number of records that a B$^+$-tree holds, say N, we can estimate the height of the tree with the branching factor of the tree and another parameter—the full factor of the tree.

Definition 1: The full factor of a B$^+$-tree is the ratio of the average number of children that each internal node has to the branching factor, denoted by f, $0.5 \le f \le 1$.

For a randomly inserted B$^+$-tree, our experimental results show that 0.75 is a good estimate of its full factor. It is reasonable to assume that the leaves are similarly full to the internal nodes. Then we can estimate the height of a B$^+$-tree

$$h = \lceil \log_{f \cdot b} N \rceil. \tag{9}$$

C. Search

To search a record (key), a path whose length is the height of the tree is followed, from the very root to the very leaf. Therefore, h nodes are accessed along the path. For searching a key within one node, we consider two algorithms: 1) linear search and 2) binary search.

To search the position of a key in an array of size n with linear search, the maximum number of iterations is n, and the average performance is $n/2$. However, with binary search, the maximum number of iterations is $\lfloor \log_2 n \rfloor + 1$, and the average performance is $\log_2 n$. Let α_L and α_B denote the CPU time of each iteration in linear search and in binary search, then T_{CPU} for the two search algorithms can be estimated as shown in Table II (in which "L" denotes linear and "B" denotes binary).

Ideally, there are no write involved in search operations. We analyze the read access numbers (in unit of cache lines) of the two algorithms as follows. The average usage of each node in unit of cache lines is

$$k = \left\lceil f \cdot \frac{\text{node\_size}}{\text{cacheline\_size}} \right\rceil. \tag{10}$$

For each linear search operation, we assume the probability to access $1 - k$ cache lines is the same, and then the average read access number is

$$\begin{aligned} R_{\text{linear}} &= \sum_{i=1}^{k} \frac{1}{k} \cdot i = (1 + k)/2 \\ &= \left(1 + \left\lceil \frac{f \cdot \text{node\_size}}{\text{cacheline\_size}} \right\rceil\right) \Big/ 2. \end{aligned} \tag{11}$$

For each binary search operation, the average read access number is estimated as

$$\begin{aligned} R_{\text{binary}} &= \lfloor \log_2 k \rfloor + 1 \\ &= \left\lfloor \log_2 \frac{f \cdot \text{node\_size}}{\text{cacheline\_size}} \right\rfloor + 1. \end{aligned} \tag{12}$$

We use "floor rounding plus 1" in (12), because we consider the case of $k = 1$ and the fact that the first cache line must be accessed as some necessary information (e.g., the number of keys in this node) is stored there.

Table II also summarizes R_{total} for the two search algorithms. Usually, α_B is larger than α_L, e.g., $\alpha_B = 2\alpha_L$. From Table II, we can find that when b or node_size is small, linear search might have better performance than binary search.

D. Insert

To insert a record to a B$^+$-tree, a search is performed first to determine in which leaf and to which position the record should be inserted. Within the target leaf, inserting the record would involve a lot of reads and writes since on average half the number of records in the node need to be shifted; and if the leaf is full, it needs to be split into two nodes, and a key needs to be inserted to the parent node; and a split can be propagated upward even to the root.

For an insertion to a nonfull node, called "a common insertion," either a leaf or nonleaf node, the average write access number involved is

$$W_{\text{comins}} = \begin{cases} 1, \text{ if node\_size} = \text{cacheline\_size} \\ 1 + \left\lceil \dfrac{f \cdot \text{node\_size}/2}{\text{cacheline\_size}} \right\rceil, \quad \text{otherwise.} \end{cases} \tag{13}$$

When node_size $>$ cacheline_size, 1 in (13) comes from updating the node length in the first cache line of the node.

For a split in a full node, either a leaf or nonleaf node, the write access number involved is

$$\begin{aligned} W_{\text{split}} &= \frac{1}{2}\left(\left\lceil \frac{\text{node\_size}/2}{\text{cacheline\_size}} \right\rceil + \left\lceil \frac{\text{node\_size}/4}{\text{cacheline\_size}} \right\rceil\right) \\ &\quad + \frac{1}{2}\left(\left\lceil \frac{\text{node\_size}/2}{\text{cacheline\_size}} \right\rceil + 1\right) \\ &= \left\lceil \frac{\text{node\_size}/2}{\text{cacheline\_size}} \right\rceil + \frac{1}{2}\left(\left\lceil \frac{\text{node\_size}/4}{\text{cacheline\_size}} \right\rceil + 1\right). \end{aligned} \tag{14}$$

Each insert operation (inserting a record to a B$^+$-tree) incurs exactly one common insertion, and zero or a few splits. The average number of splits that an insert operation incurs can be calculated by

$$\begin{aligned} P_{\text{split}} &= \frac{1}{f \cdot b - 1} + \frac{1}{(f \cdot b - 1)^2} + \cdots + \frac{1}{(f \cdot b - 1)^h} \\ &\approx \frac{1}{f \cdot b - 2}. \end{aligned} \tag{15}$$

Then the average W_{total} for an insert operation is

$$W_{\text{total\_Insert}} = W_{\text{comins}} + P_{\text{split}} \times W_{\text{split}}. \tag{16}$$

We find that all the data that are written are needed to be read first. Therefore, the estimate of R_{total} for an insert operation is

$$R_{\text{total\_Insert}} = R_{\text{total\_Search}} + W_{\text{total\_Insert}}. \tag{17}$$

The dominant component of the CPU time for an insert operation is the search part, so we can simply estimate T_{CPU} as

$$T_{\text{CPU\_Insert}} = T_{\text{CPU\_Search}}. \tag{18}$$

E. Delete

The cost analysis of a delete operation is a bit more complicated than that of insert. Deleting a record from a B$^+$-tree also executes a search first to determine in which leaf and from which position the record should be deleted. Deleting a key from a node may incur a borrow or a merge, and a merge can be propagated upward even to the root.

For "a common deletion," which does not incur a borrow or merge, from either a leaf or nonleaf node, half the number of keys in the node on average need to be shifted, so the average write access number involved is

$$W_{\text{comdel}} = \begin{cases} 1, \text{ if node\_size} = \text{cacheline\_size} \\ 1 + \left\lceil \dfrac{f \cdot \text{node\_size}/2}{\text{cacheline\_size}} \right\rceil, \quad \text{otherwise.} \end{cases} \tag{19}$$

When we delete a key from a half-full node, if one of its sibling nodes is more than half full, we will borrow a key from it; otherwise, we will merge the node with one sibling node. For borrowing a key from a right sibling node, we need to modify at most two cache lines in the node itself, one for the insertion of the borrowed key to the end, and the other for updating the node length if they are not in the same cache line. In the right sibling node we need to shift all the keys to the left by one position since the first key is lent. We also need to update the parent node by changing one key. We assume that the right sibling node is similar to half full, then the write access number incurred by borrowing a key from a right sibling node is

$$W_{\text{borrow\_}r} = \begin{cases} 3, \text{ if node\_size} = \text{cacheline\_size} \\ 3 + \left\lceil \dfrac{\text{node\_size}/2}{\text{cacheline\_size}} \right\rceil, \quad \text{otherwise.} \end{cases} \tag{20}$$

For borrowing a key from a left sibling node, the write access number incurred is the same with borrowing a key from a right sibling node

$$W_{\text{borrow\_}l} = W_{\text{borrow\_}r} = W_{\text{borrow}}. \tag{21}$$

Therefore, we use W_{borrow} to denote the write access number incurred by borrowing from either a right or a left sibling node.

For a merge of two half-full nodes, either leaves or nonleaf nodes, the write access number involved is

$$W_{\text{merge}} = \begin{cases} 2, \text{ if node\_size} = \text{cacheline\_size} \\ 2 + \left\lceil \dfrac{\text{node\_size}/2}{\text{cacheline\_size}} \right\rceil, \quad \text{otherwise.} \end{cases} \tag{22}$$

A merge will result in a common deletion or a borrow in the parent node, or propagate a merge upward.

Each delete operation (deleting a record from a B$^+$-tree) incurs 0 or a few merges, and one common deletion or one borrow. Let P_{borrow} denote the average number of borrows that a delete operation incurs and let P_{merge} denote the average number of merges that a delete operation incurs. Then the average W_{total} for a delete operation is

$$\begin{aligned} W_{\text{total\_Delete}} &= (1 - P_{\text{borrow}}) \times W_{\text{comdel}} \\ &\quad + P_{\text{borrow}} \times W_{\text{borrow}} + P_{\text{merge}} \times W_{\text{merge}}. \end{aligned} \tag{23}$$

All the data that are written are needed to be read first. Therefore, the estimate of R_{total} for a delete operation is

$$R_{\text{total\_Delete}} = R_{\text{total\_Search}} + W_{\text{total\_Delete}}. \tag{24}$$

The dominant component of the CPU time for a delete operation is still the search part, so we can simply estimate T_{CPU} as

$$T_{\text{CPU\_Delete}} = T_{\text{CPU\_Search}}. \tag{25}$$

F. Analysis of Existing Unsorted Node Schemes

In the insert and delete operations, keeping the keys of the involved node in order incurs a lot of write accesses. Let's look at the total write number in an insert operation. The first term, W_{comins} in (16), comes from making space for the inserted key (and pointer) by shifting those keys behind the inserted position. For the total write number in a delete operation, the first term, W_{comdel} in (23), comes from the similar way, filling the space of the deleted key by shifting those keys behind the deleted position. Therefore, the write costs of W_{comins} and W_{comdel} come from keeping all the keys in the node in order. To reduce these terms, one efficient solution is to leave the node unsorted.

Chen *et al.* [7] proposed three PCM-friendly variants of B$^+$-tree with unsorted node schemes: 1) unsorted with all the nonleaf and leaf nodes unsorted; 2) unsorted leaf with sorted nonleaf nodes but unsorted leaf nodes; and 3) unsorted leaf with bitmap in which each unsorted leaf node uses a bitmap to record valid locations. In the unsorted scheme, since all the nodes are unsorted, a search incurs a lot of computation overhead because we have to use linear search in every node of the search path. In the unsorted leaf scheme, we can choose the better one from binary search and linear search in the sorted nonleaf nodes and we use linear search in the only one unsorted target leaf. Therefore, it can achieve similar search performance to the original B$^+$-tree whose nodes are all sorted. Moreover, since most of the write accesses occur in the leaf nodes, the unsorted leaf scheme also captures the benefits to reduce the write accesses in the unsorted leaf nodes. Let us look at how this unsorted leaf scheme modifies our cost model.

For a search operation, as mentioned above, either linear search or binary search algorithm can be used in all the $h-1$ sorted nonleaf nodes along the search path of length h; at the end of the path, in the unsorted target leaf node, we have to search keys one by one. We can easily combine the terms in Table II to get the cost model for the search operation of the unsorted leaf scheme.

A "common insertion" to a leaf does not need to shift half the keys on average in the node; it only attaches the inserted key at the end as the last key. Therefore, at most two cache lines are needed to be written, one for the inserted key, and the other for updating the node length if they are not in the same cache line. Common insertions to nonleaf nodes have the same cost model with (13), and only splits incur insertions into nonleaf nodes. So the probability of a common insertion to a nonleaf node can be estimated by P_{split} in (15). The W_{comins} for the unsorted leaf scheme is

$$W_{\text{comins\_}usl} = \begin{cases} 1, \text{ if leaf and node\_size} = \text{cacheline\_size} \\ 2, \text{ if leaf and node\_size} > \text{cacheline\_size} \\ W_{\text{comins}} \text{ in eq.(13)}, \quad \text{if nonleaf.} \end{cases} \tag{26}$$

Although the unsorted leaf scheme reduces the cost of common insertions, it makes splits more costly, not only on CPU

execution time but also on memory accesses. To split a full unsorted leaf, we need to sort the keys first, and then split the sorted leaf to two nodes as the original sorted B$^+$-tree does. In-place Quicksort is used as the sorting algorithm, whose average time complexity is $O(n \log_2 n)$ (actually when the number of items to be sorted is small, say less than 6, a simple comparison sort is used). Then the CPU overhead due to a split of the unsorted leaf is

$$T_{\text{split\_leaf\_}usl} = \alpha_s \times (b-1)\log_2(b-1) \tag{27}$$

where α_s is the CPU time unit of the sorting algorithm.

The write number incurred by splitting an unsorted leaf can be estimated by

$$W_{\text{split\_leaf\_}usl} = \frac{\text{node\_size}}{\text{cacheline\_size}} + \left\lceil \frac{\text{node\_size}/2}{\text{cacheline\_size}} \right\rceil \tag{28}$$

in which the first term denotes the leaf node itself and the second term denotes the new allocated node. Additional space cost by the sorting algorithm is ignored.

$P_{\text{split\_leaf}}$ describes the weight how $T_{\text{split\_leaf\_}usl}$ and $W_{\text{split\_leaf\_}usl}$ affect the total CPU time and the total write access number. From (15)

$$P_{\text{split\_leaf}} = \frac{1}{f \cdot b - 1}. \tag{29}$$

It can be found that when the branching factor b is small, P_{split} and $P_{\text{split\_leaf}}$ might be large enough so that the split costs have considerable impacts on the overall CPU time and write access number.

We notice that $P_{\text{split\_leaf}} \times T_{\text{split\_leaf\_}usl}$ is $\sim \alpha_s \times \log_2(b-1)$, which indicates that for each insert operation, the CPU overhead is not small to sort a full unsorted node before its split. We propose the sub-balanced unsorted node scheme in the next section to reduce such costs in existing unsorted node schemes.

For small bs and small node_sizes, the unsorted leaf scheme might be inefficient to reduce the write number because the reduction on W_{comins} is not significant and splits might incur more write accesses than the original B$^+$-tree algorithm. Fig. 2 shows the relative write access number of the unsorted leaf scheme compared to the original B$^+$-tree algorithm, according to different branching factors and different node sizes, in which write number>1 means the total write access number of the unsorted leaf scheme is even larger than that of the original B$^+$-tree algorithm. Therefore, from Fig. 2, we can find that when the branching factor b and the node size node_size are small, e.g., $b \leq 10$ and node_size ≤ 4, the unsorted leaf scheme cannot efficiently reduce the total write number. We propose the overflow node scheme in the following section to cope with such cases with small bs and node_sizes.

For the delete operation in the unsorted leaf scheme, a "common deletion" from a leaf does not need to shift half the keys on average in the node either; it only puts the last key at the deleted position and decreases the node length. Therefore, at most two cache lines are modified in a common deletion from a leaf. Therefore, it can reduce the write accesses incurred by keeping the node in order. To reduce the write accesses incurred by the other source, tree reorganization, in unsorted node schemes, a node, either a leaf or nonleaf, will not be deleted unless there is no key in it. However, such delete algorithm may waste a lot of space when there are a lot of delete operations, and even degrade the search performance greatly. In the original B$^+$-tree algorithm, to delete a key from a half-full node, will either borrow a key from or merge with one of its sibling nodes, which incurs a lot of write accesses. We propose the merging factor scheme in the following section which makes a compromise with neither too much space nor too many write accesses.

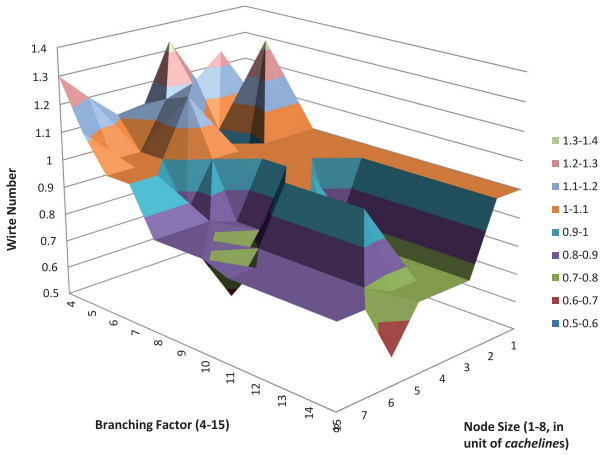


Fig. 2. Relative write access number of the unsorted leaf scheme compared to the original B$^+$-tree algorithm according to different branching factors and different node sizes.

IV. Algorithms

Our goal is to make B$^+$-tree NVM-friendly. Due to NVMs asymmetric characteristics between read and write and its limited lifetime, we aim to reduce the number of write accesses that are involved in insert and delete operations as well as to keep the new indexing method efficient for search operations. According to the issues found by our analysis in Section III-F that the existing unsorted node schemes suffer from, we introduce three schemes, i.e., the sub-balanced unsorted node scheme, the overflow node scheme, and the merging factor scheme. They can improve the performance and reduce the write accesses in the cases where the unsorted node schemes are inefficient.

A. Sub-Balanced Unsorted Node Scheme

The existing unsorted node schemes sort the keys in a full unsorted node before split it, and then the two consequent nodes have to be at least half full to keep the tree balanced. As shown above, the sorting overhead is considerable. In our sub-balanced unsorted node scheme, we propose that the nodes can be less than half full, i.e., unbalanced, after the split from a full unsorted node. With this scheme, during a split of a full unsorted node, we only need to choose a pivot to assign the keys into two nodes, of which one holds the keys greater than the pivot, and the other holds the rest keys. Since we do not need to sort all the keys in the full unsorted node before the split, the intensive computation of sorting is saved.

To choose a good pivot is very important to keep the tree balanced and hence efficient. However, to choose the perfect

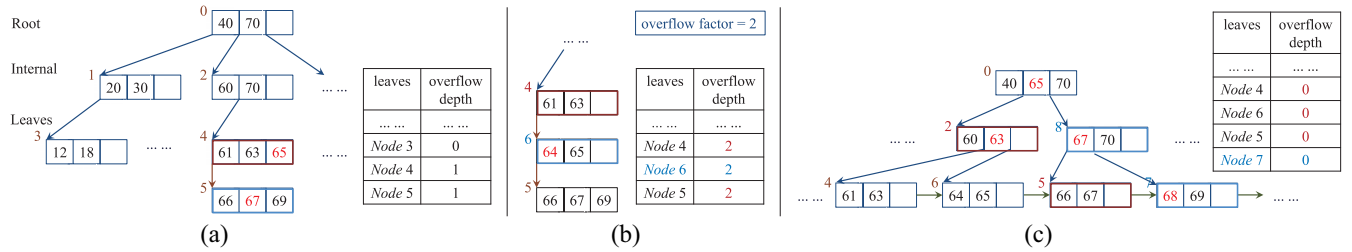


Fig. 3. Example of the overflow node scheme. (a) B$^+$-tree with overflow factor = 2; search a record with key 67. (b) Insert a record with key 64 to the tree; then Node 4 splits to a new overflow node Node 6 and its overflow depth becomes 2 reaching the overflow factor. (c) Insert a record with key 68 to the tree; then Node 5 splits to a new node Node 7, and Nodes 4–7 become independent leaves, and a set of keys are inserted to their parent node Node 2; then Node 2 splits to a new node Node 8, and a key is inserted to its parent node Node 0.

pivot, which results in two half-full nodes, is as complex as to sort the keys, i.e., $O(n \log_2 n)$. We may use the middle key value, $\text{key}_{\text{middle}} = (\text{key}_{\text{max}} + \text{key}_{\text{min}})/2$, as the pivot. We need to find the maximum key (key_{max}) and the minimum key (key_{min}) in the node first. Experiment results show that $\text{key}_{\text{middle}}$ is usually a very good choice. The time complexity of this scheme for a split is $O(n)$.

The sub-balanced unsorted node scheme also saves the additional space required by the sorting algorithm and reduces some memory accesses. In sub-balanced unsorted leaf scheme, the unbalanced leaf nodes have little impact on the search performance.

B. Overflow Node Scheme

When the branching factor b and the node size node_size are both small ($b \leq 10$ and node_size ≤ 4 as shown in Fig. 2), the unsorted leaf scheme cannot reduce the total write access number at all. The reason is that, when b and node_size are small, an insertion to or a deletion from a node will probably incur a write access to the whole node, no matter if the node is sorted or unsorted. Moreover, when b and node_size are small, a great portion of write accesses come from tree reorganizations, and the unsorted node schemes are not able to reduce such write accesses. The overflow node scheme is designed to reduce the write accesses incurred by tree reorganization operations, i.e., splits and merges.

In the original B$^+$-tree, a split of a leaf node would involve write accesses to at least three nodes: 1) the original leaf node which splits; 2) the new leaf node to split to; and 3) the parent node. The overflow node scheme postpones the update to the parent node by splitting the leaf node to an overflow node, and later execute several updates in batches in the parent node after many more splits occur in its child nodes. Because each update involves a certain number of cache line writes, to execute several updates together will save a lot of write accesses. Similarly, for delete operations, since a leaf node can merge with an overflow node which does not require an update to the parent node, the write accesses are also reduced.

In our overflow node scheme, a leaf node can have one or more overflow nodes, and all the nodes are sorted. The overflow nodes have the same structure with ordinary leaf nodes. Here are two important definitions in this scheme.

Definition 2: The overflow depth of a leaf node is the number of overflow nodes it attached; the overflow depth of the overflow nodes are the same with the first leaf node.

For example, if a leaf node has no overflow node, its overflow depth is 0; if a leaf node has one overflow node, the overflow depths of both the leaf node and the overflow node are 1. Only when the overflow depth of a leaf node reaches the overflow factor of the tree, the next split will cause a reorganization of the tree, in which all the overflow nodes of the leaf node become independent leaf nodes with an overflow depth of 0 and a set of keys are inserted into the parent node all at once.

In the implementation, we maintain the overflow factor of the tree as a global variable, and we modify the structure of leaf nodes to keep the overflow information for each leaf node. We add the overflow depth and an overflow pointer that points to the following overflow node to the original leaf node structure. The overflow depth costs 1 byte since 255 is large enough for a possible overflow factor. Because all the overflow nodes of a leaf node has the same overflow depth, we only update it in the 0th overflow node each time it changes to reduce the number of writes. The overflow pointer is a normal pointer which costs four bytes for each leaf node in a 32-bit machine. For nodesize = 2 cache lines, the space overhead to keep the overflow information in a leaf node is $(1+4)/(64\times2) \approx 4\%$, assuming the cache line size is 64 bytes. For nodesize = 4 cache lines, the overhead is about 2%. This overhead has little impact on the performance of the algorithm.

1) Insert: Fig. 3 shows an example of the insertion to a B$^+$-tree whose overflow factor is 2. Fig. 3(a) is the tree before insertion. It has six nodes labeled from 0 to 5, and each node is able to store up to three keys. Node 4 is a leaf node, and it has an overflow node, Node 5. Fig. 3(b) depicts inserting a record with key 64 to the tree. Node 4 splits to a new overflow node, Node 6, and the overflow depth of Node 4 becomes 2, reaching the overflow factor of the tree. Fig. 3(c) presents the result of the next insertion, inserting key 68 to Node 5. First, Node 5 has to split to a new node Node 7. Second, because the overflow depth of Node 5 has already reached the overflow factor, all the four leaf nodes, i.e., Nodes 4–7, become independent leaf nodes, and their overflow depths are reset to 0. Third, we need to send all the index information for each independent leaf node to the parent node, so we insert Node 2 with a set of three <key, pointer> tuples at once. Then Node 2 is full and has to split to a new node, Node 8, and at last we need to insert a key to its parent node, Node 0. From this example, we can find that the overflow node scheme has the ability to postpone the insertions to the parent node incurred by splits and deal with several updates in batches.

2) Search: It is easy to search a key with our overflow node scheme. Like in the original B$^+$-tree, we can locate the first leaf node, or the 0th overflow node, that may contain the key. Then we check its overflow depth to see if it has overflow nodes. If it has no overflow node, this leaf node is the only target leaf node that may contain the key, so we can execute either binary search or linear search within the node to find out the key. Otherwise, we first compare the key with the last key, also the greatest key, of the leaf node. If the key is no greater than the last key, this leaf node is the target leaf node; otherwise, we follow the overflow pointer to its overflow node, and then check if the overflow node is the target leaf node; and so on. For example, we search key 67 in the B$^+$-tree in Fig. 3(a). We first locate the first leaf node that may contain key 67, i.e., Node 4. As we find that Node 4 has an overflow node, we compare key 67 with the last key in Node 4, i.e., key 65. key 67 is greater, so we directly enter the overflow node, Node 5. Since Node 5 does not have an overflow node, we execute a binary or linear search in Node 5 to find key 67. With our overflow node scheme, the path to search a key might be longer, so the overflow node scheme may affect the search performance.

3) Delete: The delete operation of the overflow node scheme is similar to the original B$^+$-tree algorithm. To delete a key, We first find the key in the target leaf node, and then check if the node is half-full. If not, or the node is the root node, simply delete the key in the node; otherwise borrow a key from or merge with a neighbor node. In the original B$^+$-tree, the neighbor node must be a right or left sibling node. In the overflow node scheme, however, the neighbor node can be either a left or right overflow node, or a left or right sibling node. In our algorithm, an overflow node is preferred to a sibling node, because either to borrow a key from or to merge with a sibling node needs to write the parent node, which incurs an extra write access. Therefore, the overflow node scheme has the ability to postpone the updates in the parent node involved by a borrow or merge, thus reducing the write accesses incurred by a deletion from a half-full leaf node compared with the original B$^+$-tree.

C. Merging Factor Scheme

In the unsorted node schemes, a node will not be deleted unless there is no key in it. Such a delete algorithm removes most of the writes that may be involved by tree reorganizations due to deletions. However, for applications with a large amount of delete operations, this delete algorithm may waste too much space and even degrade the performance of the tree.

In the original B$^+$-tree, when a key is deleted from a node, it needs to merge with a sibling node if they are both half full. However, it seems too early to merge two nodes when they just become less than half full. On one hand, after a merge, the resulting node is 100% full, not a stable state since an insertion will cause it to split. On the other hand, after a split, the resulting two nodes are both half-full, also not a stable state since a deletion may cause them to borrow keys or to merge. Therefore, to reduce the write accesses that may be incurred by such unstable states, we propose the merging factor scheme which looses the merging conditions of the original B$^+$-tree. There are two important definitions in this scheme.

TABLE III
SIMULATION SETUP

Processor	1-core, x86-64, 2GHz, out of order
Cache	L1I, 32KB, 64B line, 4-way, 1 cycle latency L1D, 32KB, 64B line 4-way, 1 cycle latency L2, 2MB, 64B line, 16-way, 10 cycle latency
Memory	4GB NVM, 2 ranks, 8 banks PCM: 50 ns read latency, 500 ns write latency STT-RAM: 15 ns read latency, 20 ns write latency ReRAM: 15 ns read latency, 100 ns write latency

Definition 3: The filling degree of a node is the ratio of the number of keys in the node to the maximum number of keys that the node can contain.

Definition 4: The merging factor of a B$^+$-tree algorithm is the filling degree when two neighbor nodes need to merge with each other if a key is to be deleted from one of them.

The filling degree is in the range of [0–1], describing a node's full state, in which 0 means empty and 1 means full. The merging factor should be in range of [0–0.5], and the merging factor of the original B$^+$-tree algorithm is 0.5 and the merging factor of the unsorted node schemes is 0.

In the merging factor scheme, the merging factor can be defined less than 0.5, in order to reduce the write accesses caused by early merges. As the merging factor decreases, the merging condition is loosen, and merges occur less frequently so that the involved write accesses are saved. However, if the merging factor is too small, the leaves become sparse or even near empty and thus a lot of space is wasted.

V. EVALUATION

In this section, we first introduce our simulation setup, and then present the experiment results.

A. Simulation Setup

We build a pin-based simulator to model a 64-bit out-of-order processor and different NVM-based memory systems. Pin is a dynamic binary instrumentation framework from Intel, supporting computer architecture analysis for IA-32 and x86-64 instruction-set architectures [31]. For cache architecture, we model two levels of cache, including separate L1I and L1D cache each of size 32 kB, and a large L2 cache of size 2 MB. The cache line size is 64 bytes. For main memory, we tailor the simulator to support PCM, STT-RAM, and ReRAM models. The processor and memory system configurations are summarized in Table III.

B. Results for the Unsorted Node Schemes

We compare the unsorted leaf scheme and our sub-balanced unsorted leaf scheme with the original B$^+$-tree for PCM, STT-RAM, and ReRAM-based main memory systems under the insert-, search-, and delete-only workloads. The insert-only workload inserts one million records with random keys to an empty tree; the search-only workload searches every record of a tree with one million records in a random order; and the delete-only workload randomly deletes half the records from a tree holding one million records. We use binary search for searching in all the levels of sorted nonleaf nodes, and use

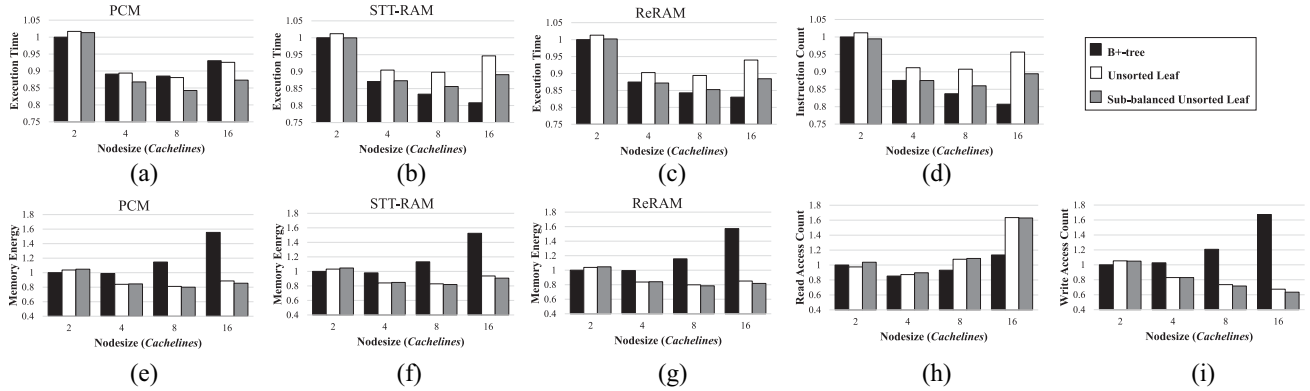


Fig. 4. Comparison among B$^+$-tree, the unsorted leaf scheme, and our sub-balanced unsorted leaf scheme in PCM, STT-RAM, and ReRAM-based systems for an insert-only workload (insert one million records with random keys to an empty tree). The results are normalized to those of B$^+$-tree with node_size = 2. (a) PCM execution time. (b) STT-RAM execution time. (c) ReRAM execution time. (d) Instruction count. (e) PCM memory energy consumption. (f) STT-RAM memory energy consumption. (g) ReRAM memory energy consumption. (h) Read access count. (i) Write access count.

linear search for searching in the unsorted leaves. The key type we use is 16-byte string. For the insert- and search-only workloads, we evaluate the node sizes from 2 to 16 cache lines. For the delete-only workload, we test the node size of 4 cache lines. A tree with one million records is large enough to cause a high last level (L2) cache miss rate. The L2 miss rate for the insert- or delete-only workload is 4%–10% with the increase of the node sizes; and for the search-only workload, it ranges from 20% to 60%.

Fig. 4 presents the comparison results for the insert-only workload, and all the results are normalized to the result of the original B$^+$-tree with node_size = 2. Fig. 4(a)–(c) shows the execution time in PCM, STT-RAM, and ReRAM-based systems, respectively. Compared with the original B$^+$-tree, the execution time of the unsorted leaf scheme is similar in PCM-based system; however, it is increased by 1.2%–13.8% in STT-RAM-based system, and by 1.3%–10.9% in ReRAM based system, among different node sizes. The reason is that, although the unsorted leaf scheme reduces the write access count significantly as the node size increases as shown in Fig. 4(i), the instruction count as well as the read access count rise greatly as shown in Fig. 4(d) and (h). Since the unsorted leaf scheme reduces the write access count by 19.8%–100% (19.3%–62.1% if not normalized to the result of the original B$^+$-tree with node_size = 2) when node_size ranges from 4 to 16 cacheline_size, it reduces the memory energy consumption a lot, 15.0%–66.8% for the PCM-based main memory, 13.8%–58.8% for the STT-RAM-based main memory, and 15.7%–72.2% for the ReRAM-based main memory.

Compared with the unsorted leaf scheme, our sub-balanced unsorted leaf scheme reduces the execution time by removing the CPU-intensive sorting before splits in all PCM, STT-RAM, and ReRAM systems. As shown in Fig. 4(a), the execution time is decreased to the extent even better than that of the original B$^+$-tree, in PCM-based system. In STT-RAM and ReRAM-based systems, the execution time is reduced by 1.1%–5.5% among different node sizes, as shown in Fig. 4(b) and (c). Fig. 4(d) shows that the instruction count is decreases in sub-balanced unsorted leaf scheme compared with the unsorted leaf scheme. It has similar read and write access counts and memory energy consumption in PCM, STT-RAM, and ReRAM systems to the unsorted leaf scheme, as shown in Fig. 4(c)–(e), (h), and (i).

When node_size = 2 ($b = 6$), from Fig. 4(i), we find that neither of the two unsorted leaf schemes can reduce the write access count. From Fig. 4(a)–(c) and (e)–(g), they cannot reduce the execution time and memory energy consumption either. That is why we propose the overflow node scheme for small node_sizes and small bs. As the node size increases, both the two unsorted leaf schemes demonstrate their efficiency in reducing the write access count and the total memory energy consumption compared with the original B$^+$-tree; however, in the meanwhile, the read access count increases.

Fig. 5 demonstrates how the two unsorted leaf schemes affect the search performance compared with original B$^+$-tree. In this experiment, we search every record of a tree with one million records in a random order. The results show that when node_size = $2, 4, 8$ ($b = 6, 13, 25$), the two unsorted leaf schemes incur little or affordable performance overhead or memory energy consumption overhead. However, when node_size = 16 ($b = 51$), compared with B$^+$-tree, they have 9.3% increase in execution time and 13.3% increase in total memory energy consumption for PCM-based system, 15.4% increase in execution time and 15.0% increase in total memory energy consumption for STT-RAM-based system, and 15.1% increase in execution time and 12.2% increase in total memory energy consumption for ReRAM-based system. It is because as the branching factor increases, the cost (instruction count and read access count) of linear search in the unsorted leaves becomes higher; when the branching factor is large enough, the cost of linear search becomes dominant, as shown in Fig. 5(d) and (h). The write access count stays stable for the search-only workload, among different node sizes and different B$^+$-tree schemes as shown in Fig. 5(i). These results indicate that, although the unsorted leaf scheme and the sub-balanced unsorted leaf scheme can reduce the write count more efficiently for larger branching factors in insert operations, they may degrade the search performance significantly.

For delete operations, the unsorted leaf scheme and the sub-balanced unsorted leaf scheme have shorter execution time than the original B$^+$-tree in PCM, STT-RAM, and ReRAM-based systems when node_size = 4, as shown in Fig. 7.

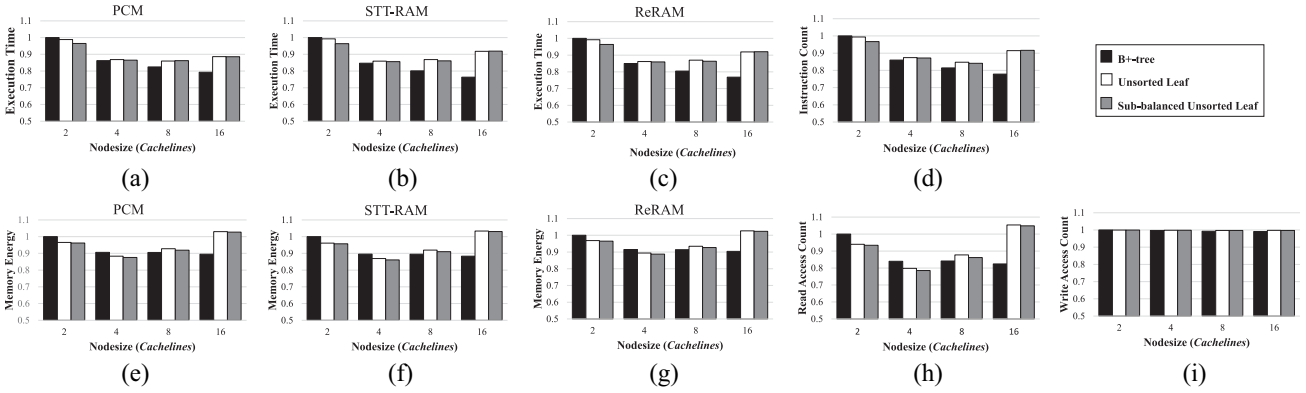


Fig. 5. Comparison among B^+-tree, the unsorted leaf scheme, and our sub-balanced unsorted leaf scheme in PCM, STT-RAM, and ReRAM-based systems for a search-only workload (search every record of a tree with one million records in random order). The results are normalized to those of B^+-tree with node_size = 2. (a) PCM execution time. (b) STT-RAM execution time. (c) ReRAM execution time. (d) Instruction count. (e) PCM memory energy consumption. (f) STT-RAM memory energy consumption. (g) ReRAM memory energy consumption. (h) Read access count. (i) Write access count.

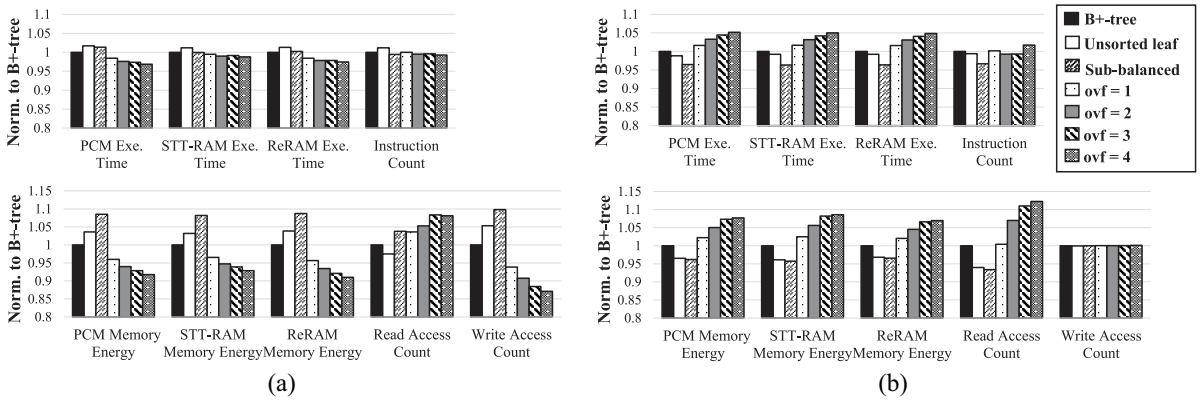


Fig. 6. Comparison results of the overflow node schemes for (a) insert-only and (b) search-only workloads in PCM, STT-RAM, and ReRAM-based systems when node_size = 2 and $b = 6$ (normalized to the B^+-tree results).

The reason is that, they remove the CPU-intensive sorting in each delete operation. For the delete-only workload, when the merging factor mgf = 0.5, the unsorted leaf scheme reduces the instruction count by 4.8%, and saves the execution time by 3.8%, 4.1%, and 4.1% in PCM, STT-RAM, and ReRAM-based systems, respectively, compared with the original B^+-tree; the sub-balanced unsorted leaf node scheme reduces the instruction count by 5.3%, and saves the execution time by 4.5%, 4.6%, and 4.6% in PCM, STT-RAM, and ReRAM-based systems, respectively. Their read and write access counts, and memory energy consumptions in PCM, STT-RAM, and ReRAM-based systems, and memory space usages are similar to those of the original B^+-tree for the delete-only workload when mgf = 0.5. We also find that, when node_size = 4, the unsorted leaf node scheme and the sub-balanced leaf node scheme cannot reduce the write accesses efficiently in delete operations.

C. Results for the Overflow Node Scheme

We evaluate the overflow node scheme for the cases of small node sizes and branching factors in PCM, STT-RAM, and ReRAM-based systems. Fig. 6 presents the results of the overflow node scheme with overflow factors (ovf) from 1 to 4 when node_size = 2 and $b = 6$. For the insert-only workload, as Fig. 6(a) shows, the overflow node scheme reduces the write count by 6.2%–12.9%, and saves memory energy by 4.0%–8.3% in PCM-based system, by 3.5%–7.1% in STT-RAM-based system, and by 4.3%–9.0% in ReRAM-based system, as the overflow factor increases from 1 to 4, without hurting the performance. This demonstrates the effectiveness of the overflow node scheme to reduce the write accesses in insert operations. However, for the search-only workload, the overflow node scheme increases the read access number by 0.4%–12.2%, as shown in Fig. 6(b), and thus incurs about 1.6%–5.2% performance overhead and increases the total memory energy consumption by 2.0%–8.6% in PCM, STT-RAM, and ReRAM-based systems, with the overflow factor from 1 to 4. Therefore, when considering the performance and memory energy consumption of both the insert and search operations, ovf = 1 and ovf = 2 are two good choices.

For delete operations, Fig. 7 shows the results of the overflow node scheme with node_size = 4 and ovf = 2. Compared with the original B^+-tree, it decreases the write accesses by 16.0% when mgf = 0.5, as shown in Fig. 7(j), because the write accesses due to merge and borrow are reduced since a leaf node can borrow keys from and merge with an overflow node without updating the parent node. In PCM-based system, it reduces the execution time by 3.2% when mgf = 0.5, as shown in Fig. 7(a). However, in STT-RAM

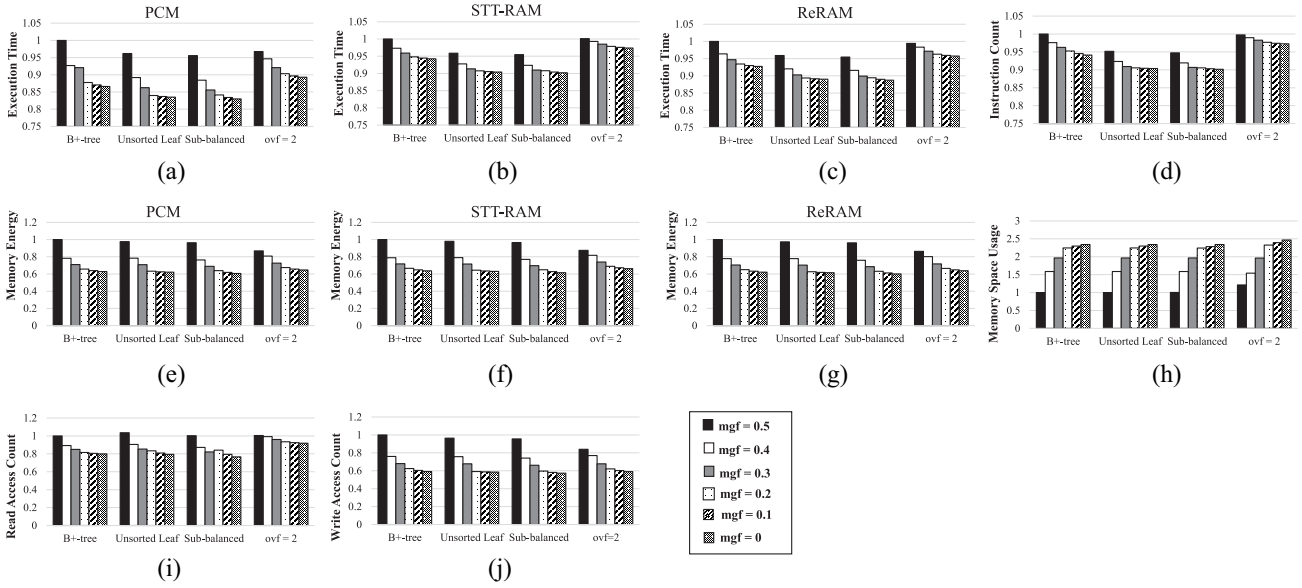


Fig. 7. Results of the merging factor scheme in PCM, STT-RAM, and ReRAM-based systems for a delete-only workload (randomly delete half the records from a tree holding one million records with random keys) with node_size = 4 (normalized to the B$^+$-tree results). (a) PCM execution time. (b) STT-RAM execution time. (c) ReRAM execution time. (d) Instruction count. (e) PCM memory energy consumption. (f) STT-RAM memory energy consumption. (g) ReRAM memory energy consumption. (h) Memory space usage. (i) Read access count. (j) Write access count.

and ReRAM-based systems, the execution time is similar to that of the original B$^+$-tree, as shown in Fig. 7(b) and (c). This is because the write latencies of STT-RAM and ReRAM are much shorter than that of PCM, and the reductions of write accesses in STT-RAM and ReRAM-based systems have less impact on the execution time than in PCM-based system. Moreover, Fig. 7(e)–(g) shows that the overflow node scheme saves memory energy consumption by 13.2%, 12.5%, and 13.7% in PCM, STT-RAM, and ReRAM-based systems, respectively. These benefits come at the cost of 21.2% more memory space, as shown in Fig. 7(h). The reason is that a leaf can be sparser before incurring a merge since it can borrow keys from either an overflow node or a neighbor node.

D. Results for the Merging Factor Scheme

The merging factor scheme is implemented with the merging factor from 0 to 0.5. We evaluate the original B$^+$-tree, the unsorted leaf node scheme, the sub-balanced unsorted leaf node scheme, and the overflow node scheme with ovf = 2, in PCM, STT-RAM, and ReRAM-based systems, for a delete-only workload. The results are shown in Fig. 7. It can be found that with the decrease of the merging factor from 0.5 to 0, the write accesses of all the four schemes are decreased (up to ∼40%) and so are their execution time, instruction count, read access count, and memory energy consumption (decreased by up to ∼35%). However, the space usage is increased dramatically (up to ∼2.5×). For MMDB applications, space efficiency is also an important goal for algorithm design. With the merging factor scheme, a proper merging factor can be chosen to make better tradeoffs among execution time, total wear, memory energy, and space usage.

VI. Conclusion

Among the emerging NVM technologies, PCM, STT-RAM, and ReRAM, are becoming promising to build future energy-efficient main memory systems. They will benefit MMDB systems with their nice features. This paper focuses on making B$^+$-tree NVM-friendly. A new algorithm design goal is to reduce the write accesses that have long latency, high energy consumption, and endurance issues. In this paper, we present a basic cost model for NVM-based memory systems, which distinguishes writes from reads according to NVM's asymmetric read/write characteristics, and also formulate the CPU costs and memory accesses for search, insert, and delete operations on a B$^+$-tree. We use the model to analyze the existing NVM-friendly B$^+$-tree schemes, i.e., the unsorted node schemes, and find that they suffer from three problems. Consequently, we propose three schemes to address these challenges: 1) the sub-balanced unsorted node scheme; 2) the overflow node scheme; and 3) the merging factor scheme. Experimental results show that they can provide more algorithm options for making tradeoffs among performance improvement, NVM lifetime extension, memory energy saving, and space usage reduction under different workloads.

References

[1] X. Wu *et al.*, "Hybrid cache architecture with disparate memory technologies," in *Proc. 36th Int. Symp. Comput. Archit. (ISCA)*, Austin, TX, USA, Jun. 2009, pp. 34–45.

[2] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Int. Symp. Comput. Archit. (ISCA)*, Austin, TX, USA, Jun. 2009, pp. 24–33.

[3] E. Kultursay, M. T. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Austin, TX, USA, Apr. 2013, pp. 256–267.

[4] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Burlingame, CA, USA, Feb. 2015, pp. 476–488.

[5] T. Zhang, M. Poremba, C. Xu, G. Sun, and Y. Xie, "CREAM: A concurrent-refresh-aware DRAM memory architecture," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Orlando, FL, USA, Feb. 2014, pp. 368–379.

[6] E. Doller. (2009). *Phase Change Memory and its Impacts on Memory Hierarchy*. [Online]. Available: http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf
[7] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *Proc. 5th Bienn. Conf. Innov. Data Syst. Res. (CIDR)*, Asilomar, CA, USA, Jan. 2011, pp. 21–31.
[8] H.-S. P. Wong *et al.*, "Metal-oxide RRAM," *Proc. IEEE*, vol. 100, no. 6, pp. 1951–1970, Jun. 2012.
[9] S. Yu, Y. Wu, and H.-S. P. Wong, "Investigating the switching dynamics and multilevel capability of bipolar metal oxide resistive switching memory," *Appl. Phys. Lett.*, vol. 98, 2011, Art. ID 103514.
[10] J. Meza, J. Li, and O. Mutlu, "Evaluating row buffer locality in future non-volatile main memories," Comput. Archit. Lab, Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. SAFARI 2012-002, Dec. 2012.
[11] P. Chi, C. Xu, T. Zhang, X. Dong, and Y. Xie, "Using multi-level cell STT-RAM for fast and energy-efficient local checkpointing," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Piscataway, NJ, USA, 2014, pp. 301–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=2691365.2691426
[12] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proc. 36th Int. Symp. Comput. Archit. (ISCA)*, Austin, TX, USA, Jun. 2009, pp. 14–23.
[13] M. K. Qureshi *et al.*, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, New York, NY, USA, 2009, pp. 14–23.
[14] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. 36th Int. Symp. Comput. Archit. (ISCA)*, Austin, TX, USA, Jun. 2009, pp. 2–13.
[15] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, New York, NY, USA, 2009, pp. 347–357.
[16] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Prin. (SOSP)*, Big Sky, MT, USA, 2009, pp. 133–146.
[17] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *Proc. 13th USENIX Conf. Hot Topics Oper. Syst. (HotOS)*, Napa, CA, USA, 2011, p. 2.
[18] W. Hu, "Redesign of database algorithms for next generation non-volatile memory technology," M.S. thesis, Dept. Comput. Sci., Nat. Univ. Singapore, Singapore, 2013.
[19] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. Knowl. Data Eng.*, vol. 4, no. 6, pp. 509–516, Dec. 1992.
[20] T. J. Lehman and M. J. Carey, "A study of index structures for main memory database management systems," in *Proc. 12th Int. Conf. Very Large Data Bases (VLDB)*, Kyoto, Japan, 1986, pp. 294–303.
[21] J. Rao and K. A. Ross, "Making B^+-tree cache conscious in main memory," in *Proc. ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, Dallas, TX, USA, 2000, pp. 475–486.
[22] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988.
[23] M. H. Nodine and J. S. Vitter, "Deterministic distribution sort in shared and distributed memory multiprocessors," in *Proc. 5th Annu. ACM Symp. Parallel Algorithms Archit. (SPAA)*, Velen, Germany, 1993, pp. 120–129.
[24] J. S. Vitter and E. A. M. Shriver, "Algorithms for parallel memory, I: Two-level memories," *Algorithmica*, vol. 12, nos. 2–3, pp. 110–147, 1994.
[25] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *Proc. 25th Int. Conf. Very Large Data Bases (VLDB)*, Edinburgh, U.K., 1999, pp. 54–65.
[26] S. Manegold, P. A. Boncz, and M. L. Kersten, "Generic database cost models for hierarchical memory systems," in *Proc. 28th Int. Conf. Very Large Data Bases (VLDB)*, Hong Kong, 2002, pp. 191–202.
[27] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proc. 27th Int. Symp. Comput. Archit. (ISCA)*, Vancouver, BC, Canada, 2000, pp. 128–138.
[28] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano, "Improving read performance of phase change memories via write cancellation and write pausing," in *Proc. IEEE 16th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Bengaluru, India, Jan. 2010, pp. 1–11.
[29] S. Chen, P. B. Gibbons, and T. C. Mowry, "Improving index performance through prefetching," in *Proc. ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, Santa Barbara, CA, USA, 2001, pp. 235–246.
[30] R. A. Hankins and J. M. Patel, "Effect of node size on the performance of cache-conscious B^+-tree," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst. (SIGMETRICS)*, New York, NY, USA, 2003, pp. 283–294.
[31] S. Berkowits. (2012). *Pin—A Dynamic Binary Instrumentation Tool*. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

Ping Chi (S'12) received the B.S. and M.S. degrees from Tsinghua University, Beijing, China, in 2008 and 2011, respectively. She is currently pursuing the Ph.D. degree with the Electrical and Computer Engineering Department, University of California at Santa Barbara, Santa Barbara, CA, USA.

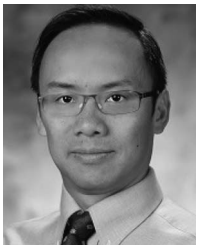
Her current research interests include emerging nonvolatile memory technologies, electronic design automation, and low power system design.

Ms. Chi was a recipient of the IEEE/ACM William J. McCalla ICCAD Best Paper Award—Front End in 2014 and was the second place winner of ACM Student Research Competition at ICCAD 2014.

Wang-Chien Lee (M'96) received the B.S. degree from National Chiao Tung University, Hsinchu, Taiwan, the M.S. degree from Indiana University, Bloomington, IN, USA, and the Ph.D. degree from Ohio State University, Columbus, OH, USA.

He was a Technical Staff Member with Verizon/GTE Laboratories, Inc., New York, NY, USA, for five years. He is currently an Associate Professor of Computer Science and Engineering with Pennsylvania State University, University Park, PA, USA. He leads the Intelligent Pervasive Data Access Research Group at Penn State to pursue cross-area research in data management, pervasive/mobile computing, and networking, with a special interest in spatial, temporal, and multidimensional aspects. His current interests include development of various techniques for supporting location-based services, recommendation services, social networking services, complex queries in a wide spectrum of networking and mobile computing environments, information retrieval, social computing, security, and Big Data. He has published over 250 technical papers in the above areas.

Dr. Lee currently serves on the Editorial Board of the IEEE TRANSACTION ON SERVICE COMPUTING.



Yuan Xie (F'15) received the B.S. degree from Tsinghua University, Beijing, China, in 1997, and the Ph.D. degree from Princeton University, Princeton, NJ, USA, in 2002.

He was an Advisory Engineer in world-wide design center with IBM, Armonk, NY, USA. From 2003 to 2014, he was an Assistant Professor, an Associate Professor and a Full Professor with Pennsylvania State University, University Park, PA, USA. He was also with AMD Research, Beijing, China, from 2012 to 2013. He is currently a Professor with the University of California at Santa Barbara, Santa Barbara, CA, USA. He has published over 200 papers in the IEEE/ACM publications. His current research interests include electronic design automation, computer architecture, and very-large-scale integration.