Persistent B+-Trees in Non-Volatile Main Memory

Shimin Chen
State Key Laboratory of Computer Architecture
Institute of Computing Technology
Chinese Academy of Sciences
chensm@ict.ac.cn

Qin Jin*
Computer Science Department
School of Information
Renmin University of China
qjin@ruc.edu.cn

ABSTRACT

Computer systems in the near future are expected to have Non-Volatile Main Memory (NVMM), enabled by a new generation of Non-Volatile Memory (NVM) technologies, such as Phase Change Memory (PCM), STT-MRAM, and Memristor. The non-volatility property has the promise to persist in-memory data structures for instantaneous failure recovery. However, realizing such promise requires a careful design to ensure that in-memory data structures are in known consistent states after failures.

This paper studies persistent in-memory B^+-Trees as B^+-Trees are widely used in database and data-intensive systems. While traditional techniques, such as undo-redo logging and shadowing, support persistent B^+-Trees, we find that they incur drastic performance overhead because of extensive NVM writes and CPU cache flush operations. PCM-friendly B^+-Trees with unsorted leaf nodes help mediate this issue, but the remaining overhead is still large. In this paper, we propose write atomic B^+-Trees (wB^+-Trees), a new type of main-memory B^+-Trees, that aim to reduce such overhead as much as possible. wB^+-Tree nodes employ a small indirect slot array and/or a bitmap so that most insertions and deletions do not require the movement of index entries. In this way, wB^+-Trees can achieve node consistency either through atomic writes in the nodes or by redo-only logging. We model fast NVM using DRAM on a real machine and model PCM using a cycle-accurate simulator. Experimental results show that compared with previous persistent B^+-Tree solutions, wB^+-Trees achieve up to 8.8x speedups on DRAM-like fast NVM and up to 27.1x speedups on PCM for insertions and deletions while maintaining good search performance. Moreover, we replaced Memcached's internal hash index with tree indices. Our real machine Memcached experiments show that wB^+-Trees achieve up to 3.8X improvements over previous persistent tree structures with undo-redo logging or shadowing.

1. INTRODUCTION

Two general trends motivate the investigation of persistent data structures in Non-Volatile Main Memory (NVMM). The first trend is the advancement of a new generation of Non-Volatile Memory (NVM) technologies, including Phase Change Memory (PCM) [4], Spin-Transfer Torque Magnetic Random Access Memory (STT-MRAM) [2], and Memristor [34]. There are significant challenges in scaling the current DRAM technology to even smaller feature sizes. For example, it would be more and more difficult to create reliable capacitance to store charges in a DRAM cell [14]. The emerging NVM technologies provide promising answers to this problem; their physical mechanisms are amenable to much smaller feature sizes. These NVM technologies all support byte-addressable reads and writes with performance close to that of DRAM, and require much lower power than DRAM due to non-volatility. As a result, there are widespread interests in the research community and in the industry to consider incorporating NVM technologies to substitute or complement DRAM as the main memory in future computer systems [5, 9, 36, 15, 26, 7, 31, 8, 33, 29, 20].

The second trend is the fast increasing capacity of main memory and its more and more significant role in database and data-intensive systems. Ten years ago, main memory database systems are mainly a research topic with products in niche markets. Today, major database vendors seriously consider main memory data processing. Examples include the IBM Blink project [3], the Hekaton OLTP engine in Microsoft SQL Server [10], and SAP HANA [24]. Moreover, in the broader data-intensive computing world, there are many proposals and products based on the principle of storing and processing data in main memory, such as Memcached [19], Pregel [17], Ramcloud [22], and Spark [35].

Combining the two trends, we see an important future need to take advantage of the non-volatility property provided by NVMM to support persistent data structures for instantaneous failure recovery. This paper focuses on persistent B^+-Trees because B^+-Trees represent an important class of tree-based structures, and are widely used in database and data-intensive systems.

At first glance, it seems straightforward to employ traditional techniques such as logging and shadowing. What we find is that the problem of supporting persistent B^+-Trees in NVMM has very different characteristics from that of disk-based B^+-Trees. First, software have only limited control of the CPU caches. There is generally no guarantee on when and in which order modified CPU cache lines are written back to main memory. While the ordering of write-backs can be forced by special CPU cache line flush and memory fence instructions, these instructions incur non-trivial overhead. Second, different NVM technologies have different characteristics. For example, PCM has slower writes with energy consumption and endurance issues [7]. In comparison, STT-MRAM and Memristor have both shown promise to have comparable performance to DRAM. Reducing writes is an important goal for PCM, but may not be as significant for fast DRAM-like NVM.

*Corresponding author, also affiliated with Key Lab of Data Engineering and Knowledge Engineering, Renmin University of China.

Proceedings of the VLDB Endowment, Vol. 8, No. 7

We find that undo-redo logging and shadowing can both incur drastic overhead because of extensive additional NVM writes and cache line flush instructions. We discuss other recent solutions related to persistent B$^+$-Trees, including Mnemosyne [31], NV-heaps [8], WSP [20], and CDDS Btree [29], but find they are insufficient for persistent B$^+$-Trees.

In this paper, we propose write atomic B$^+$-Trees (wB$^+$-Trees), a new type of main-memory B$^+$-Trees, that aim to reduce the overhead of extra NVM writes and cache line flush instructions as much as possible. wB$^+$-Tree nodes employ a small indirect slot array and/or a bitmap so that most insertions and deletions do not require the movement of index entries. In this way, wB$^+$-Trees can achieve node consistency either through atomic writes in the nodes or by redo-only logging. In performance evaluation, we use DRAM in a real machine to model DRAM-like fast NVM, and we model PCM with a cycle-accurate simulator. We perform experiments with both fixed-sized keys and variable sized keys. We also study the performance of our solution in a full system, Memcached, by replacing its internal hash index with our tree indices. Experimental results show that compared with previous persistent B$^+$-Tree solutions, wB$^+$-Trees achieve up to 8.8x speedups on DRAM-like fast NVM and up to 27.1x speedups on PCM for insertions and deletions while maintaining good search performance. Our real machine Memcached experiments show that wB$^+$-Trees achieve up to 3.8X improvements over previous persistent tree structures with undo-redo logging or shadowing.

The contributions of this paper are fivefold: (i) We characterize the mechanism of `clflush` and `mfence` to force the order to write back modified cache lines; (ii) We propose a set of metrics to analyze persistent data structures, including number of words written, and counts of `clflush` and `mfence`; (iii) Using these metrics, we analyze and compare a number of previous solutions; (iv) We propose wB$^+$-Tree structures for both fixed sized and variable sized keys, which achieve good insertion and deletion performance for persistent B$^+$-Trees; and (v) We present an extensive performance study on both a real machine modeling fast DRAM-like NVM and a cycle-accurate simulator modeling PCM.

The rest of the paper is organized as follows. Section 2 provides background on NVMM and main memory B$^+$-Trees, then drills down to understand the challenge of supporting persistent B$^+$-Trees. Section 3 analyzes existing solutions. Then Section 4 proposes our wB$^+$-Tree structures. Section 5 presents an extensive performance evaluation. Finally, Section 6 concludes the paper.

2. BACKGROUND AND CHALLENGES

We begin by describing NVMM in Section 2.1 and B$^+$-Trees in Section 2.2. Then we analyze the problem of data inconsistency in Section 2.3, and describe and characterize the mechanisms to address the problem in Section 2.4. Finally, we propose three metrics to analyze persistent data structures in NVMM in Section 2.5.

2.1 Non-Volatile Main Memory (NVMM)

There are wide-spread discussions on incorporating NVM technologies to substitute or complement DRAM as the main memory in future computer systems [5, 9, 36, 15, 26, 7, 31, 8, 33, 29, 20]. A distinctive property of NVMM is its non-volatility. Data written to NVMM will persist across power failures. Following previous work [9, 31, 8], we assume that an NVM chip can guarantee atomic writes to aligned 8-byte words. That is, the capacitance on the NVM chip is powerful enough to guarantee the completion of an 8-byte word write if a power failure occurs during the write[1].

[1]We do not assume atomic cache line sized writes because there is no way to "pin" a line in the cache. Suppose a program writes to word A and word B in the same cache line with two consecutive instructions. It is possible that the new value of A and the old value of B are written to main memory, e.g., if a context switch occurs in between the two writes.

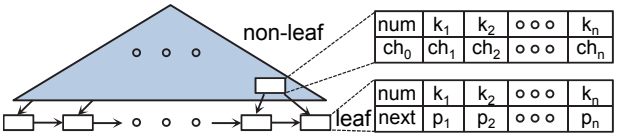


Figure 1: The main-memory B$^+$-Tree structure.

There are several competing NVM technologies based on different underlying physical mechanisms, e.g., the amorphous and crystalline states in phase change materials [4], the spin-transfer torque effect in a magnetic tunnel junction in STT-MRAM [2], or the impact of different directions of electrical currents on the resistance of Memristive devices [34]. The most mature NVM technology, PCM, sees much slower writes (e.g., $\sim$200 ns–1 us) than reads (e.g., $\sim$50ns), and a PCM cell can endure only a limited number (e.g., $\sim 10^8$) of writes [7, 11, 4]. In comparison, studies on STT-MRAM and Memristor show faster read and write performance and higher endurance [2, 34], while it still remains to be seen if production quality memory chips can achieve the same performance.

In this paper, we would like our design to support diverse NVM technologies, including both PCM and fast DRAM-like NVM (such as STT-MRAM and Memristor). In our experiments in Section 5, we model PCM with a cycle-accurate simulator and we use DRAM in a real machine to model fast DRAM-like NVM.

2.2 B$^+$-Trees in NVMM

Figure 1 illustrates the main-memory B$^+$-Tree structure. A B$^+$-Tree is a balanced tree structure with all the leaf nodes on the same level. Compared to disk-based B$^+$-Trees, the node size of main-memory B$^+$-Trees is typically a few cache lines large (e.g., 2–8 64-byte cache lines) [28, 6, 12], rather than disk pages (e.g., 4KB–256KB). Moreover, nodes store pointers instead of page IDs for fast accesses. (We will revisit this point later in Section 3.1).

As shown in Figure 1, a non-leaf node contains an array of index entries, i.e. n keys and $n+1$ child pointers. More concretely, suppose each tree node is eight 64-byte cache lines large. If the keys are 8-byte integers in a 64-bit system, then $n = 31$. A non-leaf node can hold 31 8-byte keys, 32 8-byte child pointers, and a number field. Similarly, a leaf node has space for 31 8-byte keys, 31 8-byte record pointers, a number field, and an 8-byte sibling pointer. Here, we would like to emphasize the fact that the leaf nodes of B$^+$-Trees are connected through sibling pointers for the purpose of efficient range scans. As we will discuss in Section 3.2, this complicates the use of shadowing for persistence.

Previous work proposed CPU cache optimized B$^+$-Tree solutions [28, 6, 12]. However, cache-optimized trees may incur extra NVM writes. For example, CSB$^+$-Trees require all the child nodes of a node to be contiguous in main memory [28]. To maintain this property when splitting a tree node A, a CSB$^+$-Tree has to copy and relocate many of A's sibling nodes, incurring a large number of NVM writes. In this paper, we choose prefetching B$^+$-Trees as our baseline main memory B$^+$-Trees. The basic idea is to issue CPU cache prefetch instructions for all cache lines of a node before accessing the node. The multiple prefetches will retrieve multiple lines from main memory in parallel, thereby overlapping a large portion of the cache miss latencies for all but the first line in the node. We choose this scheme because it does not incur extra NVM writes and achieves similar or better performance than CSB$^+$-Trees [6]. For fairness in performance comparison in Section 5, we employ the same principle of prefetching to improve the CPU cache performance for all the B$^+$-Tree variants.

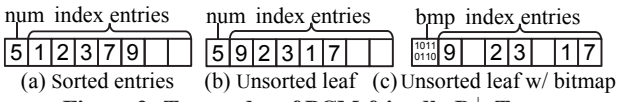


Figure 2: Tree nodes of PCM-friendly B$^+$-Trees.

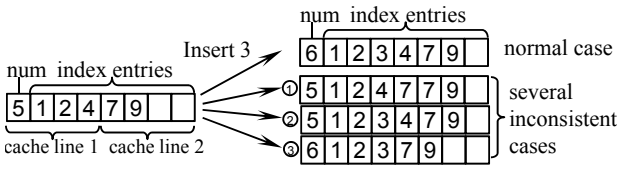


Figure 3: Potential inconsistencies upon failure.

Recently, we have proposed PCM-friendly B$^+$-Trees in a study of database algorithms for PCM [7]. The goal is to redesign B$^+$-Trees to reduce PCM writes in light of their significant performance penalty. As shown in Figure 2, PCM-friendly B$^+$-Trees intentionally allow leaf nodes to be unsorted. As shown in Figure 2(a), a normal B$^+$-Tree node contains a sorted array of index entries. To keep this sorted property, an insertion or a deletion will have to move half of the entries in the array on average, incurring a large number of PCM writes. Unsorted nodes avoid this penalty. Figure 2(b) shows an unsorted node with packed index entries, and Figure 2(c) shows an unsorted node with holes of unused entries. A bitmap replaces the number field to remember the locations of valid entries. A PCM-friendly B$^+$-Tree consists of sorted non-leaf nodes and unsorted leaf nodes. The former maintains good search performance, while the latter reduces PCM writes for updates. Our current work extends this previous work to support persistent B$^+$-Trees. As will be discussed, in order to ensure persistence, other performance factors besides PCM writes become significant, asking for new solutions.

2.3 Data Structure Inconsistency Problem

The non-volatility nature of NVMM makes it feasible to realize persistent data structures in the face of power failures and system crashes[2]. In the following, we will not distinguish the two types of failures and use the word "failure" to mean both.

Without careful designs, data structures may be in inconsistent non-recoverable states after failures. Figure 3 depicts what may happen if a failure occurs in the middle of an insertion to a sorted tree node J. Normally, the insertion algorithm will move 9, 7 and 4 one slot to the right, insert 3, and increment the number field, leading to the correct end state shown at the top in Figure 3. The figure shows three inconsistent cases. In the first case, the failure occurs before the algorithm moves 4. In the second case, the failure occurs before the algorithm increments the number field. While case 1 and case 2 are quite straightforward, case 3 is non-obvious. In the third case, the algorithm has actually completed the insertion but not all the data has been written back to NVMM. Suppose node J consists of two cache lines. The number field and the first three slots are in cache line 1, while the last four slots are in cache line 2. Case 3 occurs if cache line 1 has been written back to NVMM, but cache line 2 is still in the CPU cache when the failure occurs. Therefore, line 1 reflects the states after the insertion but line 2 reflects the states before the insertion. From the same start state, the above three cases lead to three different inconsistent states. Similarly, it is easy to show that the same inconsistent state can be reached from multiple start states with different failure cases. Consequently, using the information of an inconsistent tree node alone, it is impossible to recover from the failure because there is no way to tell which start state is the correct one.

From this example, we can see that the modern CPU cache hierarchy can be at odds with the goal of achieving persistent data structures in NVMM. Many of the design decisions of the cache hierarchy target system performance without consideration of non-volatility. The CPU considers a store instruction to have completed when it is sent to the caches. Dirty lines may be cached for a long time, and there are no guarantees on the order of dirty line write backs. Compared to the memory buffer pool of disk-based database systems, we do not have the same level of control for CPU caches.

Intel has recently introduced restricted hardware transactional memory (RTM) in its Haswell processors [13]. During a hardware transaction on a CPU core, RTM keeps a limited number of modified dirty cache lines in the CPU cache private to the core. When the hardware transaction commits, RTM makes the dirty cache lines in the transaction visible to the whole system. However, there is still no guarantee on the order and the timing of writing the dirty lines back to main memory. Therefore, RTM does not provide a solution to the data inconsistency problem.

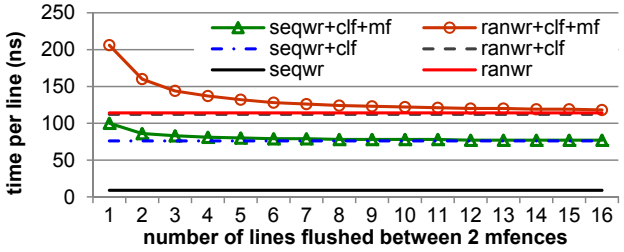


Figure 4: Performance impact of clflush and mfence instructions on sequential and random writes on a real machine.

2.4 Clflush and Mfence Instructions

X86 processors provide limited capability for software to control CPU caches [13]. In this paper, we exploit two x86 instructions, `clflush` and `mfence`, in order to protect NVM writes from failure[3]. Note that other processors (e.g., ARM) have similar support.

- *clflush*: Given a memory address, the `clflush` instruction invalidates the cache line that contains the address from all levels of caches, and broadcasts the invalidation to all CPU cores in the system. If the specified cache line is dirty, it will be written to memory before invalidation.
- *mfence*: The `mfence` instruction guarantees that all memory reads and memory writes issued before the `mfence` in program order become globally visible before any reads or writes that follow the `mfence` instruction. However, `mfence` alone does not write back any dirty cache lines.

Multiple `clflush` instructions may be executed in parallel. However, dirty cache lines may be wrtten back in arbitrary order. To ensure the write-back order of two dirty cache lines, we leverage the fact that `clflush` is also ordered by `mfence`. Therefore, we can insert `mfence` between `clflush` instructions to force the write-back order of dirty cache lines.

Figure 4 measures the impact of `clflush` and `mfence` instructions on an Intel Xeon x86-64 machine with 64-byte cache lines (cf. machine configuration in Section 5). We model the typical case in database and data intensive systems where the total data

[2]We focus on failures that can be recovered by utilizing data in NVMM. Other types of failures (e.g., hardware component failures, and software corruptions) require additional mechanisms (such as replication and error correction codes), which are orthogonal and beyond the scope of this paper.

[3]One can also use x86 non-temporal store instructions (e.g., `movntdq`) to write data back to memory. For space limitation, we focus on `clflush` and `mfence` as the underlying hardware mechanism in this paper. Microbenchmarks show similar performance for non-temporal stores especially for random writes, which are typical for B$^+$-Trees.

Table 1: Terms used in analyzing persistent data structures.

Term	Description
N_w	Number of words written to NVMM
N_{clf}	Number of cache line flush operations
N_{mf}	Number of memory fence operations
n	Total number of entries in a B$^+$-Tree node
n'	Total number of entries in a wB$^+$-Tree node
m	Number of valid entries in a tree node
l	Number of levels of nodes that are split in an insertion

size is much larger than the CPU cache size, by making the data in the experiments 10 times as large as the last level CPU cache. The data consists of cache line sized records aligned on cache line boudaries. We perform sequential write and random write experiments. The sequential write experiment writes an 8-byte integer in every consecutive record. For the random write experiment, we first randomize the order of the records and construct a linked list among the records. We then measure the elapsed time of writing an 8-byte integer in every record following the linked list. In this way, processor optimizations such as hardware prefetching are not effective. In Figure 4, there are three curves for sequential writes (lower), and three for random writes (upper). The Y-axis reports the average elapsed time for accessing each record. The three curves show the performance of (1) writes only, (2) issuing a `clflush` after writing every record, and (3) issuing a `mfence` every i records for (2) where i varies from 1 to 16 as shown on the X-axis.

From the figure, we see that `clflush` significantly slows down sequential writes, because `clflush` forces dirty cache lines to be written back. This disrupts the CPU and the memory controller's optimizations for sequential memory accesses (e.g., combining multiple accesses to the same open memory row). In comparison, since random writes are not amenable to such optimizations, `clflush` has neglegible impact on random writes. Moreover, inserting an `mfence` after every `clflush` incurs significant overhead for both sequential and random writes, because the `mfence` waits for the previous `clflush` to complete. This overhead is reduced as the number of `clflush`-ed records between two `mfence`s increases. Therefore, it is important to reduce the relative frequency of `mfence`.

Insertions and deletions in B$^+$-Trees contain a mix of random and sequential writes. The leaf nodes to be updated are usually randomly distributed (e.g., in OLTP workload). Depending on the node structure, writes in a leaf node can be sequential (e.g., for the sorted nodes) or not (e.g., for unsorted leaf nodes). Therefore, we may see a mix of the effect of Figure 4 for B$^+$-Tree updates.

2.5 Metrics for Persistent Data Structures

Summarizing the discussion in this section, we propose to use three metrics to analyze algorithms for achieving persistent data structures. We follow previous work [7] to analyze writes (N_w) because writes are bad for PCM [7], and NVM writes must be protected by `clflush` and `mfence` from failure[4]. Moreover, we pay special attention to `clflush` (N_{clf}) and `mfence` (N_{mf}) because they may result in significant overhead. The terms used in this paper are summarized in Table 1.

3. ANALYZING EXISTING SOLUTIONS

Existing solutions mainly follow two traditional principles, logging and shadowing, to achieve persistent data structures in NVMM. Both principles have been long studied in database systems [27]. In this section, we first analyze the two approaches in the context of NVMM and then discuss other related proposals.

[4]Other metrics proposed in previous work [7], e.g., number of lines fetched, do not vary much across different persistent B$^+$-Tree structures.

```
1:  procedure WRITEUNDOREDO(addr,newValue)
2:      log.write (addr, *addr, newValue);
3:      log.clflush_mfence ();
4:      *addr= newValue;
5:  end procedure
6:  procedure NEWREDO(addr,newValue)
7:      log.write (addr, newValue);
8:      *addr= newValue;
9:  end procedure
10: procedure COMMITNEWREDO
11:     log.clflush_mfence ();
12: end procedure
```

Figure 5: NVMM write protected by undo-redo logging.

3.1 Undo-Redo Logging

The first question that arises is whether the idea of a traditional buffer pool can be employed for persistent B$^+$-Trees. We could divide the main memory into two parts: a persistent part where persistent tree nodes are stored, and a volatile part that holds the buffer pool. The content of the buffer pool is deemed useless upon recovery. The buffer pool caches all the tree nodes to be accessed. Dirty nodes are held in the buffer pool, and write-ahead logging ensures data persistence. However, like disk-based database systems, tree nodes must be copied into the buffer pool, and be referenced by node IDs, which are mapped to the actual memory location of the buffered nodes through a hash table. The hash table will typically be larger than the CPU cache, and thus a (random) node ID dereference operation will often incur an expensive cache miss. In real-machine experiments, we see a 2.3x slowdown for search performance of main memory B$^+$-Trees with a buffer pool. Consequently, we mainly focus on in-place update in this subsection.

Protecting in-place NVM writes requires undo-redo logging. As shown in Figure 5, `WriteUndoRedo` first records the address, the old value ($*addr$), and the new value to the log in NVMM. Then it issues a `clflush` and a `mfence` to ensure the log content is stable before performing the actual write. In this way, failure recovery can use the logged information to undo the change if the tree update operation is to be aborted, or redo the change if the tree update has committed. This algorithm requires a `clflush` and a `mfence` per NVM write, and performs multiple NVM writes to the log (e.g., three extra 8-byte writes for each 8-byte update).

We can reduce the overhead of logging if a write W is to a previously unused location, using `NewRedo` and `CommitNewRedo`, as shown in Figure 5. Since the old value is meaningless, `NewRedo` logs the address and only the new value. Then it performs W without flushing the log. Later, a `CommitNewRedo` flushes the log and calls `mfence` once for all the previous `NewRedo`s. This approach performs fewer `clflush` and `mfence` operations, and writes less to the log. If a failure occurs before the log record for W is stable, we can safely ignore any change because the location is previously unused. If the log record is stable in the log, we can use the redo information to ensure W's correctness.

Figure 6 shows a more aggressive optimization. It is applicable only if a newly written value is not to be accessed again before commit. `WriteRedoOnly` does not perform the actual write. Instead, it only logs the intention of the write ($addr, newValue$). At commit time, `CommitRedoWrites` issues `clflush` instructions to flush all the redo log records to memory. A single `mfence` ensures redo log records are stable in NVMM. The impact of `mfence` will be amortized across all the cache line flushes. Then the algorithm reads the log records and performs the actual in-place writes. Note that this optimization can be applied only judiciously because re-reading the newly written value before commits will cause an error.

```
1: procedure WRITEREDOONLY(addr,newValue)
2:     log.write (addr, newValue);
3: end procedure
4: procedure COMMITREDOWRITES
5:     log.clflush_mfence ();
6:     for all (addr,newValue) in log do
7:         *addr= newValue;
8:     end for
9: end procedure
```

Figure 6: Redo-only logging.

```
1: procedure INSERTTOLEAF(leaf,newEntry,parent,ppos,sibling)
2:     copyLeaf= AllocNode();
3:     NodeCopy(copyLeaf, leaf);
4:     Insert(copyLeaf, newEntry);
5:     for i=0; i < copyLeaf.UsedSize(); i+=64 do
6:         clflush(&copyleaf + i);
7:     end for
8:     WriteRedoOnly(&parent.ch[ppos], copyLeaf);
9:     WriteRedoOnly(&sibling.next, copyLeaf);
10:    CommitRedoWrites();
11:    FreeNode(leaf);
12: end procedure
```

Figure 7: Shadowing for insertion when there is no node splits.

Let us consider an insertion into a B$^+$-Tree leaf node without node splits. Suppose the sorted leaf has m index entries, each containing an 8-byte key and an 8-byte pointer. The insertion moves an average $m/2$ entries, inserts a new entry, and increments the number field, writing $m+3$ words. For each word write, the undo-redo logging incurs 3 extra writes, a `clflush` and a `mfence`. Hence, $N_w = 4m + 12$, $N_{clf} = m + 3$, and $N_{mf} = m + 3$.

Table 3 in Section 4.7 shows the cost analysis for nine persistent B$^+$-Tree solutions. B$^+$-Tree with undo-redo logging is in the first row. From left to right, the table lists the costs of insertion without node splits, insertion with l node splits, and deletion without node merges. Compared to insertion, a deletion in a leaf node moves an average $(m-1)/2$ entries, and thus has a slightly lower cost. On the other hand, the cost of an insertion with node splits grows dramatically because of the node split operation.

If an insertion does not lead to node splits, a PCM-friendly B$^+$-Tree writes the new index entry to an unused location and updates the number/bitmap field. We use `NewRedo` for the former and `WriteUndoRedo` for the latter. $N_w = 2*3+1*4 = 10$, $N_{clf} = 2$, and $N_{mf} = 2$, as shown in row 2 and row 3 in Table 3. However, the two types of trees have different deletion costs. For a packed unsorted leaf node, a deletion needs to move the last entry to fill the hole of the deleted entry. This is an overwrite operation and must use `WriteUndoRedo`. Therefore, $N_w = 3 * 4 = 12$. In contrast, for an unsorted leaf node with a bitmap, only the bitmap needs to be overwritten. Therefore, it sees lower costs and $N_w = 4$. Finally, when an insertion causes node splits, PCM-friendly B$^+$-Trees behave the same as B$^+$-Trees except for the leaf node split. Therefore, they have similar costs to B$^+$-Trees.

3.2 Shadowing

The second method to achieve persistence is shadowing. To make changes to a node J, we first create a copy J' of J, then update the copy J'. We flush J' and commit it as the new update-to-date node. However, since J is pointed to by J's parent node, we will have to update J's parent to point to J'. Then we follow the same shadowing procedure to create a new copy of the parent node. This process will continue until the root node.

To avoid the extensive copy operations in a Btree-like structure, Condit et al. [9] proposed a technique called short-circuit shadowing. The idea is to take advantage of the 8-byte atomic write feature in NVM. For the above example, it will atomically modify the leaf node pointer in J's parent. In essence, when there is a single 8-byte pointer that points to the newly created node copies, short-circuit shadowing can avoid propagating the copy operation further. Unfortunately, the B$^+$-Tree structure introduces additional complication— the leaf sibling pointers. Both the pointer in the leaf parent and the pointer in its sibling leaf node need to be updated, which cannot be handled by short-circuit shadowing.

We employ `clflush` and `mfence` to solve this problem, as shown in Figure 7. The algorithm creates a copy of the leaf node, inserts the new entry, and flushes the shadow node. Then it uses two `WriteRedoOnly`s to log the update intentions to the two pointers in the parent and the sibling. Finally, it calls a `CommitRedoWrites` to commit the changes. Note that `CommitRedoWrites` will flush the log and perform a `mfence` before actually updating the two pointers. This sequence of operations guarantees that the modifications to both the parent and the sibling occur after the shadow node and the redo log records are stable in NVMM. In the case of a failure, if the `mfence` in `CommitRedoWrites` has not yet completed, then the original tree structure is kept intact. If the `mfence` has succeeded, then we can always use the redo log records to recover and complete the insertion operation.

This shadowing procedure requires copying the entire leaf node. Suppose that the leaf node contains m used entries and each entry consists of an 8-byte key and an 8-byte pointer. Shadowing incurs $2m + 4$ writes for copying the entries, the number field, and the sibling pointer field, and inserting the new entry. The two `WriteRedoOnly`s require 4 word writes, and the actual pointer updates require 2 writes. `AllocNode` will require an additional log write, `clflush`, and `mfence` to ensure persistence of the allocation operation. Therefore, $N_w = 2m + 11$. The algorithm flushes the shadow node and the redo log records. $N_{clf} = (2m+4)\frac{8}{64} + 1 + 1 = 0.25m + 2.5$. `CommitRedoWrites` and `AllocNode` both perform a `mfence`. Thus $N_{mf} = 2$.

Table 3 shows the cost analysis for three shadowing solutions. For each solution, it shows insertion without node splits, as well as deletion without node merges, and the complex case where an insertion triggers node splits. We see that the deletion cost is similar to the insertion cost except that $m - 1$ entries are copied to the shadow node. For PCM-friendly B$^+$-Trees, the cost of shadowing is the same as normal B$^+$-Trees because the node copy operation removes any benefits from reducing writes in a leaf node.

3.3 Other Related Work

In late 1980s, Agrawal and Jagadish [1] designed recovery algorithms for databases running on Ferroelectric Random Access Memories based NVMM. To our knowledge, this is the earliest study of NVMM for a database system. This work assumes that main data is on disks, and exploits NVMM with logging and shadowing to reduce the overhead of traditional disk-based recovery. In contrast, we assume that the data in NVMM is the primary copy. Hence, updates to persistent B$^+$-Trees must persist across failures.

In 1990s, several pieces of work studied storage systems, file systems, and database systems on NVMM [32, 16, 21]. eNVy is a Flash memory based storage system [32], which deals with the draw-backs of Flash (e.g., erase, wear-leveling). The Rio project [16, 21] designed a Rio file cache from NVMM and exploited the Rio file cache for databases. The work assumes that each individual store instruction immediately persists the data in NVMM. Unfortunately, in the present CPU cache hierarchy, this assumption can

be only achieved by issuing a `clflush` and a `mfence` after every write, which would result in significant performance degradation.

Several recent studies considered the same NVMM setting as this paper. Mnemosyne [31] and NV-heaps [8] aimed to provide general libraries and programming interfaces to use NVMM. Both studies exploited software transactional memory (STM) and redo-only logging at transaction commit time to support in-place updates to NVMM. However, STM may incur significant overhead (up to 20x slowdowns) [18]. WSP [20] proposed to take advantage of a small residual energy window provided by the power supply to flush cache and register states to NVMM after a power failure. However, a software routine is responsible for coordinating the saving of transient states to NVMM when a power failure is detected. Such design cannot cope with operating system crashes. Pelley et al. [23] performed trace-based analysis to understand the NVM design space in a traditional page organized OLTP system. The study indicates that disk-like organization incurs significant performance overheads. Our study considers main memory data structures in NVMM. As discussed previously, we studied index and hash join algorithms on PCM [7]. Viglas [30] studied sorting and join algorithms on NVMM. Both of these studies focus on the efficiency of algorithm designs without persistence.

Venkataraman et al. [29] proposed a consistent CDDS Btree structure for NVMM. This is the closest to our work. The proposal does not employ logging or shadowing. It enhances each index entry with a pair of start version and end version fields, and keeps a global version for the entire tree. All the updates are performed in place and flushed to memory. When an entire update operation succeeds, the global version is atomically incremented. In this way, the design can carefully use the global version to remove any on-going uncommitted updates during failure recovery. While this design supports persistence, there are several drawbacks. First, it requires a `clflush` and a `mfence` for every write, which can lead to significant time overhead compared to plain Btrees. Second, a version is an 8-byte integer in the CDDS Btree. This means that the size of an index entry is doubled if the key size is 8 byte. Basically, this leads to 50% space overhead. Third, the use of a single global version essentially serializes any update operations. Therefore, updates must be performed sequentially, which is undesirable.

4. WRITE-ATOMIC B$^+$-TREES

In this section, we propose a new persistent B$^+$-Tree structure, Write-Atomic B$^+$-Trees (a.k.a. wB$^+$-Trees). We first describe the design goals, then present wB$^+$-Tree structures and operations with fixed-sized keys and with variable sized keys, such as strings.

4.1 Design Goals

We would like our design to achieve the following three goals:

- *Atomic write to commit all changes:* Since logging incurs extra NVM writes and cache line flushes, it is desirable to optimize away the need for logging in the common cases, where an insertion or a deletion is handled within a node without node splits or merges. It would be nice to be able to apply the insertion or the deletion while maintaining the original state of the node, and then allow a single atomic write to commit all changes.
- *Minimize the movement of index entries:* In a sorted node, half of the entries have to be moved on average for an insertion or a deletion. As discussed in Section 2.2, unsorted nodes significantly reduce the movement of index entries. An insertion writes to an unused entry without moving existing entries. The best deletion scheme updates only the bitmap field in a node. We would like our design to achieve similar update efficiency.

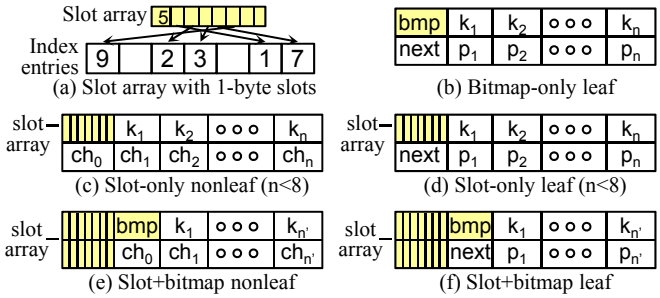


Figure 8: wB$^+$-Tree node structures. (For a slot+bitmap node, the lowest bit in the bitmap indicates whether the slot array is valid. Slot 0 records the number of valid entries in a node.)

Table 2: wB$^+$-Tree structures considered in this paper.

Structure	Leaf Node	Non-leaf Node
wB$^+$-Tree	slot+bitmap leaf	slot+bitmap non-leaf
wB$^+$-Tree w/ bitmap-only leaf	bitmap-only leaf	slot+bitmap non-leaf
wB$^+$-Tree w/ slot-only nodes	slot-only leaf	slot-only non-leaf

- *Good search performance:* The structure of non-leaf nodes is important for search performance. Since the node size of main-memory B$^+$-Trees is relatively small compared to disk-based B$^+$-Trees, main-memory trees are much taller. For example, a B$^+$-Tree with 512-byte sized nodes will have 11 levels after being bulkloaded with 50 million index entries to be 70% full. If a scheme incurs a delay of f to search a non-leaf node, then the overall search time will see a delay of $10f$. Previous work [7] observes this effect and recommends to keep sorted nodes for non-leaf nodes. However, sorted nodes would have very poor update performance when there are node splits or node merges. We would like our design to achieve good update performance while maintaining good search performance.

4.2 wB$^+$-Tree Structures

Figure 8(b) shows the previous proposal of bitmap-only unsorted leaf nodes. If the bitmap size is bounded by an 8-byte word, then this structure can achieve the goal of write atomicity. However, the unsorted nature of the node makes binary search impossible. Can we achieve both write atomicity and good search performance?

We introduce a small indirection array to a bitmap-only unsorted node, as shown in Figure 8(e) and (f). This solution is inspired by the slotted page structure of NSM disk pages. The indirection slot array remembers the sorted order of the index entries, as shown in Figure 8(a). A slot contains the array offset for the corresponding index entry. Slot 0 records the number of valid entries in the node.

The resulting slot+bitmap nodes contain both a bitmap and an indirection slot array. The bitmap is used to achieve write atomicity. Like bitmap-only nodes, the bitmap always contains the locations of valid index entries. In addition, we use the lowest bit of the bitmap to indicate whether the slot array is valid. Normally, the slot array is valid and a binary search can take advantage of the slot array to obtain the sorted order of index entries. However, after a failure, the slot array may be invalid. Then the bitmap is used to do the search as in bitmap-only nodes. We will describe failure recovery in depth in Section 4.3.

If the tree node size is small so that the maximum number of index entries (n) in a node is less than 8, then the entire slot array can fit into an 8-byte word. The slot array itself can achieve write atomicity without the help of the bitmap. For such cases, we propose slot-only nodes, as shown in Figure 8 (c) and (d).

We combine slot-only nodes, slot+bitmap nodes, and bitmap-only leaf nodes to form three types of wB$^+$-Tree structures, as

```
1: procedure INSERT2SLOTONLY_ATOMIC(leaf, newEntry)
2:     /* Slot array is valid */
3:     pos= leaf.GetInsertPosWithBinarySearch(newEntry);
4:     /* Write and flush newEntry */
5:     u= leaf.GetUnusedEntryWithSlotArray();
6:     leaf.entry[u]= newEntry;
7:     clflush(&leaf.entry[u]); mfence();
8:     /* Generate an up-to-date slot array on the stack */
9:     for (j=leaf.slot[0]; j≥pos; j - -) do
10:        tempslot[j+1]= leaf.slot[j];
11:    end for
12:    tempslot[pos]=u;
13:    for (j=pos-1; j≥1; j - -) do
14:        tempslot[j]= leaf.slot[j];
15:    end for
16:    tempslot[0]=leaf.slot[0]+1;
17:    /* Atomic write to update the slot array */
18:    *((UInt64 *)leaf.slot)= *((UInt64 *)tempslot);
19:    clflush(leaf.slot); mfence();
20: end procedure
```

Figure 9: Insertion to a slot-only node with atomic writes.

listed in Table 2. When the node size is small, slot-only nodes are the best node organization. When the node size is large, we consider both wB$^+$-Tree and wB$^+$-Tree with bitmap-only leaf nodes. A non-leaf node is always a slot+bitmap node, while a leaf node contains a bitmap with or without a slot array.

It is easy to see that the above wB$^+$-Tree structures support any fixed sized keys. An 8-byte bitmap can support up to 63 index entries, and 1-byte sized slots can support up to 255 index entries. For example, if an index entry is 16-byte large (with 8-byte keys and 8-byte pointers on an 64-bit machine), then a slot+bitmap node can be as large as 1KB (i.e. 16 cache lines), which is often sufficient for main memory B$^+$-Trees [28, 6, 12]. Support for variable sized keys will be described in Section 4.6.

4.3 Insertion

Figure 9 shows the algorithm for inserting a new entry to a slot-only node using atomic writes. The algorithm starts by finding the insertion position using the sorted slot array (line 3). Then it examines the valid slots to locate an unused entry (line 5) and writes to the new entry (line 6). It ensures that the new entry is stable in NVMM with a `clflush` and a `mfence` (line 7). Then it generates an up-to-date slot array using an 8-byte space from the call stack (line 8–16). Finally, it performs an atomic write to update the slot array (line 17–19). Upon failure recovery, if this atomic write succeeds, the insertion has successfully completed. If the atomic write has not occurred, the original data in the node is kept intact because the insertion uses an unused entry. Both states are consistent. As shown in Table 3, $N_w = 3$ for writing the new entry and the slot array, $N_{clf} = 2$, $N_{mf} = 2$. For bitmap-only leaf nodes, we can apply a similar algorithm with the same cost, which writes the new entry to an unused location and atomically updates the bitmap.

Figure 10 shows the algorithm for inserting into a slot+bitmap node using atomic writes. It clears the slot array valid bit in the bitmap (line 6–8). Then it writes and flushes the new entry to an unused location (line 9–12), and modifies and flushes the slot array (line 13–24). Next, it issues a `mfence` to ensure the new entry and the slot array are stable in NVMM (line 25). Finally, the algorithm updates the bitmap atomically to enable the slot valid bit and the new entry (line 26–28). During failure recovery, a slot+bitmap node may be in one of three consistent states: (i) the original state before insertion, (ii) the original state with invalid slot array, or (iii)

```
1: procedure INSERT2SLOTBMP_ATOMIC(leaf, newEntry)
2:     if (leaf.bitmap & 1 == 0) /* Slot array is invalid? */ then
3:         Recover by using the bitmap to find the valid entries,
           building the slot array, and setting the slot valid bit;
4:     end if
5:     pos= leaf.GetInsertPosWithBinarySearch(newEntry);
6:     /* Disable the slot array */
7:     leaf.bitmap = leaf.bitmap - 1;
8:     clflush(&leaf.bitmap); mfence();
9:     /* Write and flush newEntry */
10:    u= leaf.GetUnusedEntryWithBitmap();
11:    leaf.entry[u]= newEntry;
12:    clflush(&leaf.entry[u]);
13:    /* Modify and flush the slot array */
14:    for (j=leaf.slot[0]; j≥pos; j - -) do
15:        leaf.slot[j+1]= leaf.slot[j];
16:    end for
17:    leaf.slot[pos]=u;
18:    for (j=pos-1; j≥1; j - -) do
19:        leaf.slot[j]= leaf.slot[j];
20:    end for
21:    leaf.slot[0]=leaf.slot[0]+1;
22:    for (j=0; j≤leaf.slot[0]; j += 8) do
23:        clflush(&leaf.slot[j]);
24:    end for
25:    mfence(); /* Ensure new entry and slot array are stable */
26:    /* Enable slot array, new entry and flush bitmap */
27:    leaf.bitmap = leaf.bitmap + 1 + (1<<u);
28:    clflush(&leaf.bitmap); mfence();
29: end procedure
```

Figure 10: Insertion to a slot+bitmap node with atomic writes.

successful insertion with valid bitmap and valid slot array. If the failure occurs before the first atomic bitmap write, the node will be in state (i). If the failure occurs between the first and the second atomic writes, the node will be in state (ii). The slot array is disabled. The insertion is to a previously unused entry. Therefore, any potentially partial writes of the new entry and the slot array have not modified the original valid entries in the node. The algorithm checks and recovers for this case (line 2–4). If the second actomic write to the bitmap also completes, the node will be in state (iii).

`Insert2SlotBmp_Atomic` incurs an NVM write, a `clflush`, and a `mfence` for either atomic bitmap write. It performs 2 NVM writes and a clflush for the new entry. For the slot array, it writes and flushes $(m + 2)$ bytes, leading to $(m + 2)/8$ NVM writes and $(m + 2)/64$ `clflush`s. Then it issues a `mfence` at line 25. Putting it all together, $N_w = 0.125m + 4.25$, $N_{clf} = \frac{1}{64}m + 3\frac{1}{32}$, $N_{mf} = 3$, as shown in Table 3.

When an insertion leads to the split of a leaf node J, we allocate a new leaf node $\tilde{J}$ and balance the index entries between J and $\tilde{J}$. Note that the balance operation simply copies index entries from J to $\tilde{J}$. As entries are unsorted, there is no need to move any entries in J. Then we write the bitmap/slot fields and the sibling pointer field of the new node. The new node can be updated in place. On the other hand, we also need to update the bitmap/slot fields and the sibling pointer field of J. While we cannot perform a single atomic write in this case, it is easy to employ redo-logging. Then we insert the new leaf node to the parent node using the algorithms in Figure 9 or Figure 10, and commit the redo writes.

4.4 Deletion

The deletion algorithm is similar to the insertion algorithm. As the index entries are unsorted, we do not need to move any entries

Table 3: Comparison of persistent B$^+$-Tree solutions.

Solution	Insertion without node splits	Insertion with l node splits	Deletion without node merges
B$^+$-Trees undo-redo logging	$N_w = 4m + 12$, $N_{clf} = N_{mf} = m + 3$	$N_w = l(4n + 15) + 4m + 19$, $N_{clf} = l(0.375n + 3.25) + m + 4.125$, $N_{mf} = l(0.25n + 2) + m + 5$	$N_w = 4m$, $N_{clf} = N_{mf} = m$
Unsorted leaf undo-redo logging	$N_w = 10$, $N_{clf} = 2$, $N_{mf} = 2$	$N_w = l(4n+15)+n+4m+19$, $N_{clf} = l(0.375n+3.25)+0.25n+m+4.125$, $N_{mf} = l(0.25n+2)+0.25n+m+5$	$N_w = 12$, $N_{clf} = 3$, $N_{mf} = 3$
Unsorted leaf w/ bitmap undo-redo logging	$N_w = 10$, $N_{clf} = 2$, $N_{mf} = 2$	$N_w = l(4n+15)-n+4m+19$, $N_{clf} = l(0.375n+3.25)-0.25n+m+4.125$, $N_{mf} = l(0.25n+2)-0.25n+m+5$	$N_w = 4$, $N_{clf} = 1$, $N_{mf} = 1$
B$^+$-Trees shadowing	$N_w = 2m + 11$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2.5$	$N_w = l(2n + 5) + 2m + 12$, $N_{clf} = l(0.25n + 1.5) + 0.25m + 2.625$, $N_{mf} = 2$	$N_w = 2m + 7$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2$
Unsorted leaf shadowing	$N_w = 2m + 11$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2.5$	$N_w = l(2n + 5) + 2m + 12$, $N_{clf} = l(0.25n + 1.5) + 0.25m + 2.625$, $N_{mf} = 2$	$N_w = 2m + 7$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2$
Unsorted leaf w/ bitmap shadowing	$N_w = 2m + 11$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2.5$	$N_w = l(2n + 5) + 2m + 12$, $N_{clf} = l(0.25n + 1.5) + 0.25m + 2.625$, $N_{mf} = 2$	$N_w = 2m + 7$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2$
wB$^+$-Tree	$N_w = 0.125m+4.25$, $N_{clf} = \frac{1}{64}m + 3\frac{1}{32}$, $N_{mf} = 3$	$N_w = l(1.25n' + 9.75) + 0.125m + 8.25$, $N_{clf} = l(\frac{19}{128}n' + 1\frac{105}{128}) + \frac{1}{64}m + 3\frac{13}{32}$, $N_{mf} = 3$	$N_w = 0.125m + 2$, $N_{clf} = \frac{1}{64}m + 2$, $N_{mf} = 3$
wB$^+$-Tree w/ bitmap-only leaf	$N_w = 3$, $N_{clf} = 2$, $N_{mf} = 2$	$N_w = l(1.25n' + 9.75) - 0.25n' + 0.125m + 7.5$, $N_{clf} = l(\frac{19}{128}n' + 1\frac{105}{128}) - \frac{3}{128}n' + \frac{1}{64}m + 3\frac{43}{128}$, $N_{mf} = 3$	$N_w = 1$, $N_{clf} = 1$, $N_{mf} = 1$
wB$^+$-Tree w/ slot-only nodes	$N_w = 3$, $N_{clf} = 2$, $N_{mf} = 2$	$N_w = l(n + 9) + 7$, $N_{clf} = l(0.125n + 1.75) + 2.375$, $N_{mf} = 2$	$N_w = 1$, $N_{clf} = 1$, $N_{mf} = 1$

Note: The estimated N_{clf}s are lower bounds because they do not cover the case where a log record spans the cache line boundary, and requires two flushes. For 512-byte sized nodes, $n = 31$, $n' = 29$, m is about 21 if a node is 70% full.

for deletion in a leaf node. The algorithm simply updates the slot array and/or the bitmap to reflect the deletion. Either atomic writes or redo-only logging can be employed in a fashion similar to the insertion algorithms. The main difference is that there is no need to write any index entries for deletion without node merges.

4.5 Search

One benefit provided by the slot array is that it maintains the sorted order of the index entries in a node. This is especially useful for non-leaf nodes. Let us consider the search operation in an unsorted non-leaf node without the slot array. For a search key, we will have to examine every entry in the node to find the largest entry that is smaller than the search key. Note that this can be more expensive than searching a leaf node, which requires only equality comparison. If the search key exists in a leaf node, it is expected to examine only half of the entries in the leaf node, while all the entries in an unsorted non-leaf node must be examined.

With the slot array, we apply binary search with logarithm comparisons. In our implementation, we find that the slot array dereference incurs certain non-trivial overhead due to memory access to retrieve the slot contents. We optimize the search procedure to stop the binary search when the range narrows down to less than eight slots. Then we retrieve all the remaining slots into an 8-byte integer variable. From then on, we use shift and logic operation on this integer to obtain slot contents avoiding further slot dereferences.

4.6 wB$^+$-Trees for Variable Sized Keys

While primary key indices and foreign key indices usually contain fixed-sized keys, variable sized keys are also widely used. For example, in the Memcached key-value store [19], a key can be a string with up to 250 characters. Memcached maintains a slab-based memory allocation pool, and all the key-value items are stored in this pool. Memcached constructs a hash index for associative lookups. The hash index stores pointers to the key-value items.

We would like to extend wB$^+$-Trees to support variable sized keys. When the key size is similar to or larger than a cache line, it is less beneficial to store all the keys of a node contiguously in order to reduce CPU cache misses. Therefore, we instead store 8-byte pointers to the variable sized keys rather than the actual keys in the trees. That is, the tree structure contains 8-byte keys, which are pointers to the actual variable sized keys. We call these 8-byte keys as key pointers. In this way, the above wB$^+$-Tree structures can be easily adapted to support variable sized keys. Essentially, a slot+bitmap node has two indirection layers. The first indirection layer is the key pointers, while the second indirection layer is the slot array. With this simple extension, we are able to support string keys and achieve similar benefits as fixed sized keys.

Compared to wB$^+$-Trees with fixed sized keys, wB$^+$-Trees with variable sized keys incur larger key comparison costs because of (i) the key pointer dereference to access the actual key, and (ii) the usually larger size of the key. As a result, the performance benefit of maintaining the sorted order in a slot array will be higher because it effectively reduces the number of unnecessary key comparisons.

4.7 Comparison with Previous Solutions

Table 3 compares the cost of all the persistent B$^+$-Trees that we have discussed, assuming 8-byte keys. We apply undo-redo logging and shadowing to main-memory B$^+$-Trees and the two variants of PCM-friendly B$^+$-Trees. Then we consider three wB$^+$-Tree structures. From the table, we see that wB$^+$-Tree schemes reduce the number of NVM writes, and / or the number of CPU cache flushes and memory fences compared to the previous solutions based on shadowing and undo-redo logging.

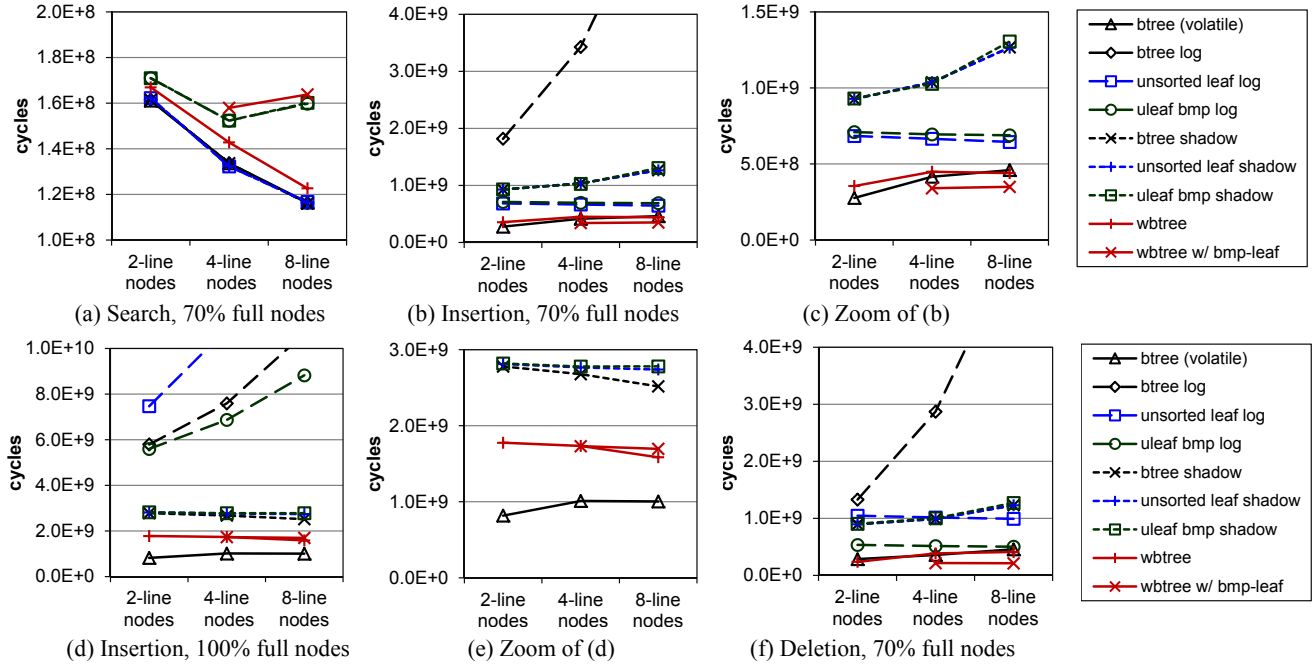
5. EXPERIMENTAL RESULTS

We conduct an extensive performance study for persistent B$^+$-Tree solutions with different key types and different NVMMs.

5.1 Experimental Setup

Real machine modeling DRAM-like fast NVMM. We model fast NVMM using DRAM on a real machine. The machine configuration is shown in Table 4. It is an Intel Xeon x86-64 machine running Ubuntu 12.04. For each real-machine experiment, we perform 10 runs and report the average performance across the 10 runs.

Simulation modeling PCM-based NVMM. We model PCM using a cycle-accurate out-of-order x86-64 simulator, PTLsim [25], PTLsim has been used in our previous work [7]. However, we find that the simulator does not implement the actual functionality of `clflush`. It is treated as a nop and ignored. We extended the simulator with the following modifications to support `clflush`: (i) the reorder buffer in the processor checks the dependence between a `clflush` and prior memory accesses; (ii) when a `clflush` commits in the reorder buffer, it issues a memory store operation for the specified cache line if it is dirty; (iii) a `mfence` will check all prior stores initiated by `clflush` instructions, and wait for them to complete. We tune the implementation so that the impact of `clflush`



(a) Search, 70% full nodes (b) Insertion, 70% full nodes (c) Zoom of (b)

(d) Insertion, 100% full nodes (e) Zoom of (d) (f) Deletion, 70% full nodes

Figure 11: Index performance on a cycle-accurate simulator modeling PCM-based NVMM. (We bulkload a tree with 20M entries, then perform 100K random back-to-back lookups, insertions, or deletions. Keys are 8-byte integers.)

Table 4: Experimental Setup.

Real Machine Description	
Processor	2 Intel Xeon E5-2620, 6 cores/12 threads, 2.00GHz
CPU cache	32KB L1I/core, 32KB L1D/core, 256KB L2/core 15MB shared L3, all caches with 64B lines
OS Compiler	Ubuntu 12.04, Linux 3.5.0-37-generic kernel gcc 4.6.3, compiled with -O3
Simulator Description	
Processor	Out-of-order X86-64 core, 3GHz
CPU cache	Private L1D (32KB, 8-way, 4-cycle latency), private L2 (256KB, 8-way, 11-cycle latency), shared L3 (8MB, 16-way, 39-cycle latency), all caches with 64B lines, 64-entry DTLB, 32-entry write back queue
PCM	4 ranks, read latency for a cache line: 230 cycles, write latency per 8B modified word: 450 cycles, E_{rb} = 2 pJ, E_{wb} = 16 pJ

and `mfence` on writes are similar to that shown in Figure 4 on the real machine. We use the same simulator configurations as in previous work [7]. Table 4 lists the detailed configuration.

B$^+$-Trees Implementations. We compare a total nine B$^+$-Tree solutions for fixed-sized keys: (1) *btree (volatile)*: cache optimized prefetching B$^+$-Tree; (2) *btree log*: employ undo-redo logging for *btree*; (3) *unsorted leaf log*: employ undo-redo logging for B$^+$-Tree with unsorted leaf nodes; (4) *uleaf bmp log*: employ undo-redo logging for B$^+$-Tree with bitmap-only unsorted leaf nodes; (5) *btree shadow*: employ shadowing for *btree*; (6) *unsorted leaf shadow*: employ shadowing for B$^+$-Tree with unsorted leaf nodes; (7) *uleaf bmp shadow*: employ shadowing for B$^+$-Tree with bitmap-only unsorted leaf nodes; (8) *wbtree*: wB$^+$-Tree; (9) *wbtree w/ bmp-leaf*: wB$^+$-Tree with bitmap-only leaf nodes. If the node size $\leq$ two cache lines, we use wB$^+$-Tree with slot-only nodes to replace both (8) and (9), and report results as *wbtree*. We employ the principle of cache optimized prefetching B$^+$-Trees [6] for all the B$^+$-Tree solutions. All but (1) are persistent B$^+$-Tree solutions.

We compare a total seven B$^+$-Tree solutions for variable sized keys. All the above variants of B$^+$-Trees are included except for the two variants involving unsorted leaf nodes, as we have seen similar performance between unsorted leaf nodes and bitmap-only unsorted leaf nodes.

Memcached Implementation. We have replaced the hash index in Memcached 1.4.17 to use our tree implementations with variable sized keys. We modified about 100 lines of code in Memcached to use tree-based indices.

B$^+$-Tree Workload. Unless otherwise noted, we bulkload a tree with B entries that are randomly uniformly generated, and perform back-to-back random search, insertion, or deletion operations. We use 8-byte integer keys for fixed-sized keys and 20-byte strings for variable sized keys . We make B large enough so that the tree size is much larger than the last-level cache in the system. For fixed-sized keys, we perform real-machine experiments and simulation experiments. For simulation, B = 20 million. For real machine experiments, B = 50 million. Thus, the total size of valid leaf entries is 320MB in simulation and 800MB on the real machine, respectively. For variable sized keys, we perform only real-machine experiments because experiments on fixed sized keys have already contrasted the two types of NVMM models. $B = 50$ million. There will be an additional 1GB memory space for storing the actual strings on the real machine.

5.2 Simulation Modeling PCM

Figure 11 reports simulation results for 100 thousand search, insertion, and deletion operations on B$^+$-Trees bulkloaded with 20 million entries, while varying the tree node size from 2 cache lines to 8 cache lines. From Figure 11, we see the following:

- The wB$^+$-Tree achieves similar search performance compared to the baseline main-memory non-persistent B$^+$-Trees. The indirection through the slot array incurs 2%–4% slowdowns. Solutions using bitmap-only leaf nodes see up to 16% slowdowns because of the sequential search overhead in leaf nodes.
- Applying undo-redo logging incurs drastic 6.6–13.7x slowdowns for B$^+$-Trees and 2.7–12.6x slowdowns for PCM-friendly B$^+$-Trees. This is because undo-redo logging requires a significant

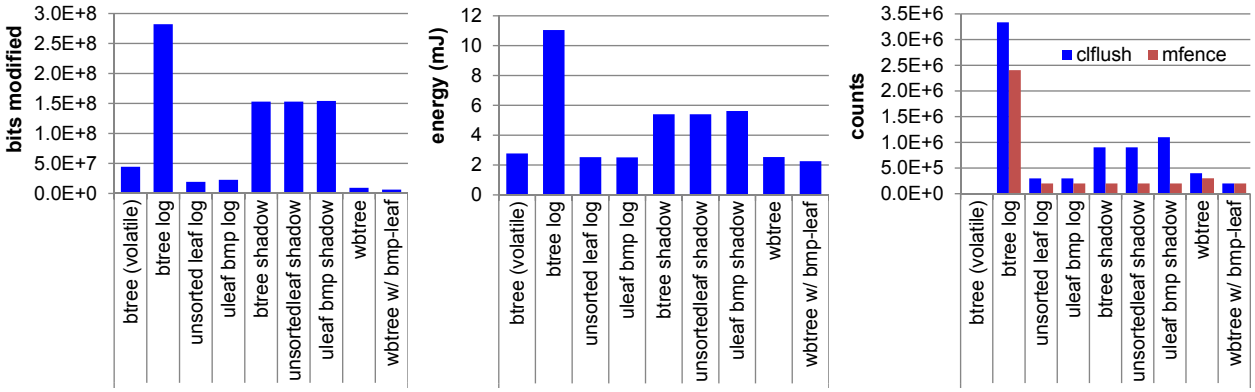
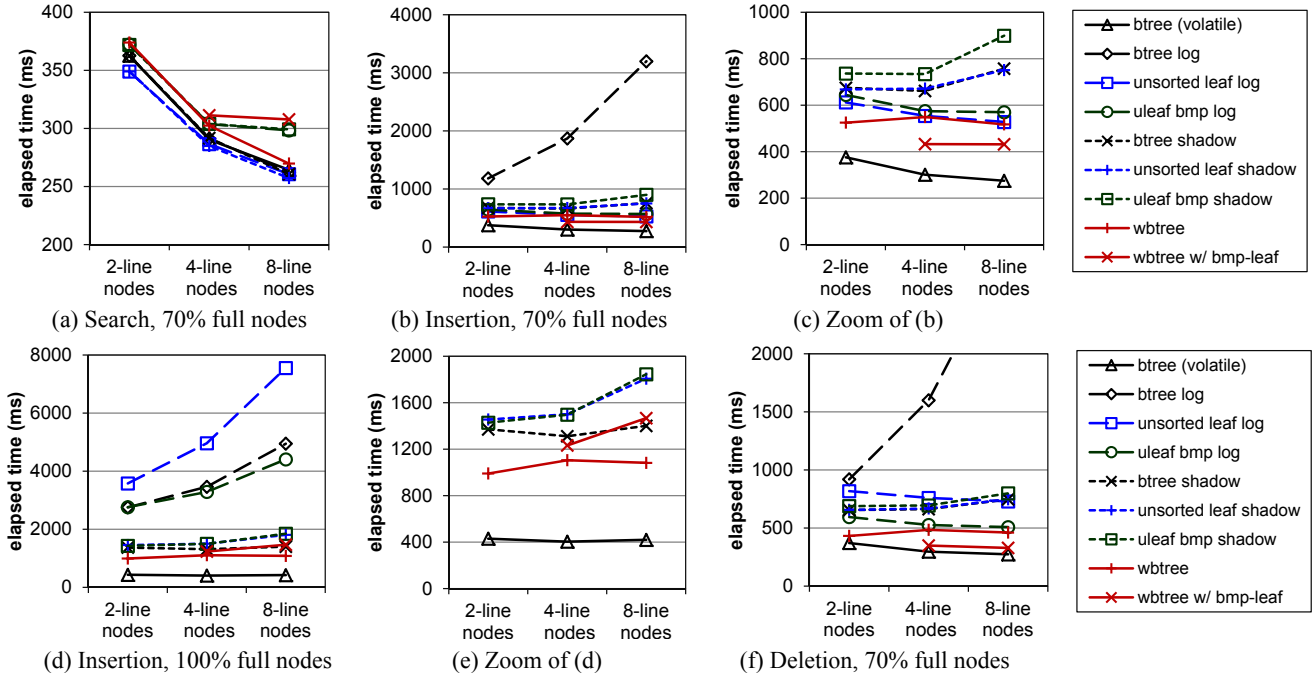


Figure 12: Wear, energy, and clflush/mfence counts of index operations for Figure 11(b) with 8-line nodes.



(a) Search, 70% full nodes (b) Insertion, 70% full nodes (c) Zoom of (b)

(d) Insertion, 100% full nodes (e) Zoom of (d) (f) Deletion, 70% full nodes

Figure 13: Index performance on a real machine modeling DRAM-like fast NVMM. (We bulkload a tree with 50M entries, then perform 500K random back-to-back lookups, insertions, or deletions. Keys are 8-byte integers.)

number of PCM writes and cache line flushes. The worst slowdowns happen in Figure 11(d), where there are many node splits.

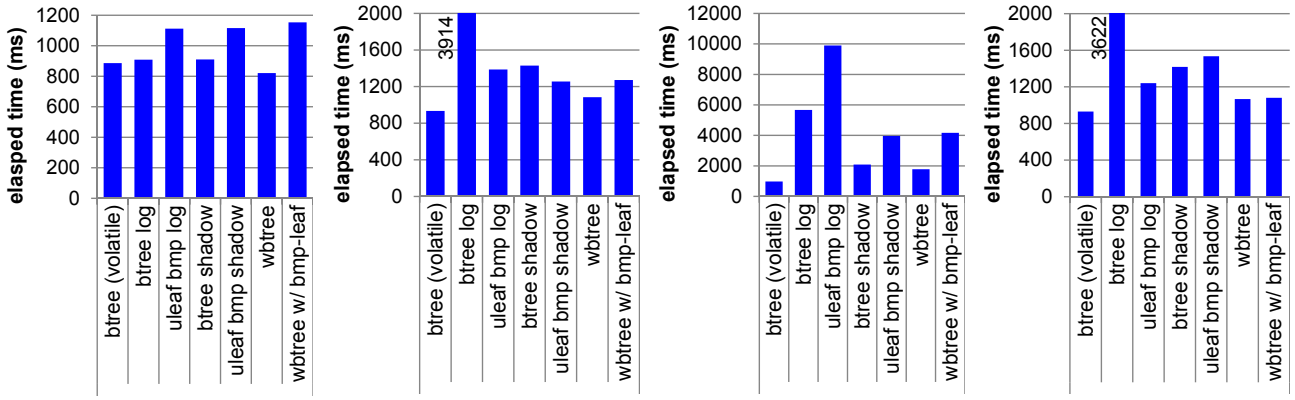
- Shadowing incurs 2.1–7.8x slowdowns because it performs extensive data copying to create a new node for every insertion or deletion. The benefits of unsorted leaf nodes are lost; *unsorted leaf shadow* and *uleaf bmp shadow* are as slow as *btree shadow*.
- For each update experiment, we compare the performance of wB$^+$-Trees with the slowest and the fastest previous persistent solutions. Our wB$^+$-Trees achieve a factor of 4.2–27.1x improvement over the slowest previous persistent solution. The best wB$^+$-Tree result is 1.5–2.4x better than the fastest previous persistent solution in each insertion or deletion experiment.
- *wbtree w/ bmp-leaf* achieves slightly better insertion and deletion performance than *wbtree*, but sees worse search performance. *wbtree w/ bmp-leaf* saves the cost of updating the slot array for insertions and deletions, but pays the cost of sequential search in leaf nodes. Note that for 2-line nodes, the figure shows only wB$^+$-Tree with slot-only nodes as *wbtree*.

Figure 12 shows the wear (bits modified), energy, and counts of `clflush` and `mfence` for Figure 11(b). Comparing the counts with the cost analysis in Table 3, we see that the `mfence` counts are estimated accurately, while the `clflush` estimations are smaller than the real measurements. This is because our `clflush` estimation has not considered the case where a log record spans the cache line boundary causing two flushes.

5.3 Real Machine Experiments Modeling Fast DRAM-Like NVM

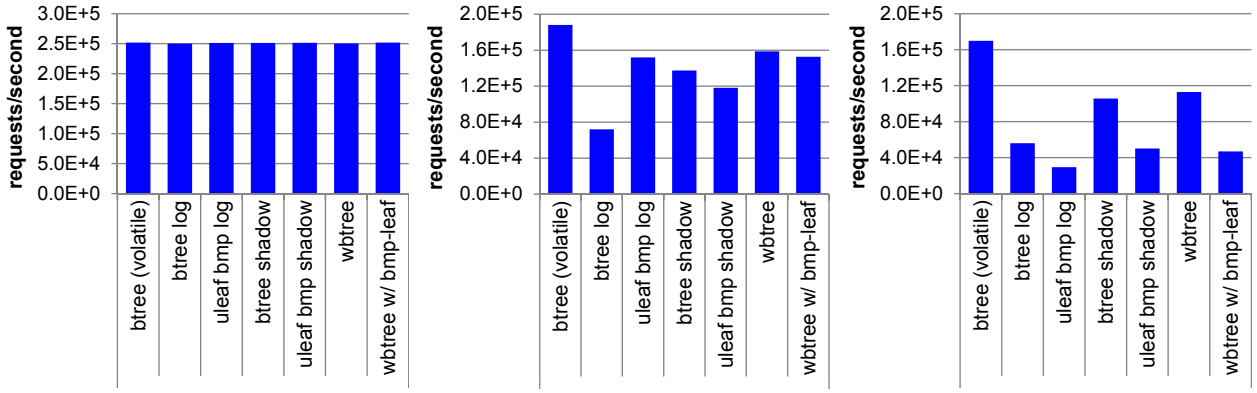
Figure 13 reports the elapsed time for 500 thousand random search, insertion, and deletion operations on B$^+$-Trees bulkloaded with 50 million keys on the real machine. The layout of Figure 13 is the same as Figure 11. Similar to the simulation results, we see that (i) the wB$^+$-Tree achieves similar search performance compared to the baseline main-memory non-persistent B$^+$-Trees; (ii) Applying undo-redo logging incurs 1.6–11.8x slowdowns; (iii) Shadowing incurs 1.7–3.3x slowdowns; and (iv) Our wB$^+$-Trees achieve 2.1–8.8x improvement over the slowest previous persistent solution, and the best wB$^+$-Tree result is 1.2–1.6x better than the best previous persistent solution in each insertion or deletion experiment.

There are interesting differences between real machine and simulation results. We redraw Figure 13 and Figure 11 as bar charts,



(a) Search, 70% full nodes, 20-byte string keys (b) Insertion, 70% full nodes, 20-byte string keys (c) Insertion, 100% full nodes, 20-byte string keys (d) Deletion, 70% full nodes, 20-byte string keys

Figure 14: Index performance with string keys on a real machine. (We bulkload a tree with 50M entries, then perform 500K random back-to-back lookups, insertions, or deletions).



(a) Search, 70% full nodes (b) Insertion, 70% full nodes (c) Insertion, 100% full nodes

Figure 15: Memcached throughput on a real machine. (We replace the hash index in Memcached with various types of trees. We bulkload a tree with 50M entries, and use mc-benchmark to insert and search 500K random keys. Keys are 20-byte random strings.)

and generate the wear, energy, and `clflush`/`mfence` counts figures for all experiments. These charts are omitted from the paper because of space limitation. Comparing the figures, we find that the bar charts of the simulation results have similar shape to the bits modified charts, while the bar charts of the real machine results have similar shape to the `clflush` charts. This means that PCM writes play a major role in determining the elapsed times on PCM based NVMM, while on fast DRAM-like NVMM, cache line flushes are the major factor affecting the elapsed times.

5.4 Real Machine Experiments for Trees with String Keys

Figure 14 shows the real machine experimental results for index operations with 20-byte string keys. The trees use the structure for the variable sized keys as described in Section 4.6. The actual keys are stored in a separate memory area outside the trees, and the trees contain 8-byte sized key pointers that point to the actual string keys.

Compared to 8-byte integer keys in Figure 13, the key comparison operation becomes much more expensive because of the pointer dereference and the significantly larger key size. A search is about 3 times as expensive as that in the trees with 8-byte integer keys, as can be seen by comparing Figure 14(a) and Figure 13(a).

From Figure 14(a)–(d), we see that *wbtree* is the best persistent tree solution. It achieves 1.2–5.6x speedups compared to previous persistent tree solutions based on undo-redo logging or shadowing. On the other hand, *wbtree w/ bmp-leaf* has significantly poorer performance because it has to perform very costly sequential search and compare many more keys.

5.5 Real-Machine Memcached Performance

Figure 15 reports the throughput of Memcached with various tree implementations as its internal index. (Note that unlike the elapsed time, throughput is the higher the better.) We run the mc-benchmark[5], which performs a set of insertions followed by a set of search operations over network. The experiments use two machines, one running the Memcached server, the other running the mc-benchmark tool, with a 1Gbps Ethernet switch connecting the two machines. By default, the mc-benchmark uses 50 parallel connections to maximize the throughput. Memcached employs a locking based scheme to protect shared data structures. We see that the performance difference across solutions is smaller than Figure 14 because of the communication overhead and the parallel execution. This effect is more pronounced for search because the shorter search time is outweighed more by the communication overhead, and read-only operations can be executed fully in parallel. We see that *wbtree* achieves the highest throughput for insertions among persistent tree structures. It achieves 1.04–3.8X improvements over previous persistent tree structures with undo-redo logging or shadowing. Similar to Section 5.4, we see non-trivial overhead for using

[5]https://github.com/antirez/mc-benchmark

bitmap-only leaf nodes. In essence, for variable sized keys, it is important to employ the slot array in leaf nodes.

6. CONCLUSION

This paper studies persistent B$^+$-Trees that take advantage of the non-volatility provided by NVMM for instantaneous failure recovery. We propose and evaluate write atomic B$^+$-Trees (wB$^+$-Trees), a new type of main-memory B$^+$-Trees. Based on our analysis and experiments, we draw the following conclusions. (i) Traditional solutions, such as undo-redo logging and shadowing, can incur drastic overhead because of extensive NVM writes and cache line flushes. (ii) The factors affecting performance have different weights for different NVM technologies. The number of NVM writes plays a major role in determining the elapsed times on PCM based NVMM, while cache line flush is the major factor for fast DRAM-like NVMM. (iii) Compared to previous persistent B$^+$-Tree solutions, our proposed write atomic B$^+$-Trees (wB$^+$-Trees) significantly improve the insertion and deletion performance, while achieving good search performance similar to that of non-persistent cache-optimized B$^+$-Trees.

Acknowledgment. We thank the anonymous reviewers for their valuable comments. This work is partially supported by the CAS Hundred Talents program and by NSFC Innovation Research Group No. 61221062. The second author is supported by the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China (No. 14XNLQ01), and Beijing Natural Science Foundation (No. 4142029).

7. REFERENCES

[1] R. Agrawal and H. V. Jagadish. Recovery algorithms for database machines with nonvolatile main memory. In *IWDM*, pages 269–285, 1989.

[2] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *JETC*, 9(2):13, 2013.

[3] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, S. Szabo, R. Sidle, and K. Stolze. Blink: Not your father's database! In *BIRTE*, pages 1–22, 2011.

[4] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy. Phase change memory technology. *J. Vacuum Science*, 28(2), 2010.

[5] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM J. Res. Dev.*, 52(4):449–464, July 2008.

[6] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *SIGMOD*, 2001.

[7] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[10] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Conference*, 2013.

[11] E. Doller. Phase change memory and its impacts on memory hierarchy. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf, 2009.

[12] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B$^+$-trees. In *SIGMETRICS*, 2003.

[13] Intel Corp. Intel 64 and ia-32 architectures software developers manual. Order Number: 325462-047US, June 2013.

[14] ITRS. International technology roadmap for semiconductors (2011 edition executive summary). http://www.itrs.net/Links/2011ITRS/2011Chapters/2011ExecSum.pdf.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[16] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. *Operating Systems Review*, 31, 1997.

[17] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[18] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. S. III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT*, 2006.

[19] Memcached. http://memcached.org/.

[20] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS*, 2012.

[21] W. T. Ng and P. M. Chen. Integrating reliable memory in databases. In *VLDB*, 1997.

[22] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, 2011.

[23] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *PVLDB*, 7(2):121–132, 2013.

[24] H. Plattner. The impact of columnar in-memory databases on enterprise systems (keynote). In *VLDB*, 2014.

[25] PTLsim. http://www.ptlsim.org/.

[26] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA*, 2009.

[27] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[28] J. Rao and K. A. Ross. Making B$^+$-trees cache conscious in main memory. In *SIGMOD*, 2000.

[29] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, 2011.

[30] S. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5):413–424, 2014.

[31] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS*, 2011.

[32] M. Wu and W. Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *ASPLOS*, 1994.

[33] X. Wu and A. L. N. Reddy. Scmfs: a file system for storage class memory. In *SC*, 2011.

[34] J. J. Yang and R. S. Williams. Memristive devices in computing system: Promises and challenges. *JETC*, 9(2):11, 2013.

[35] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[36] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA*, 2009.