NV-Tree: A Consistent and Workload-adaptive Tree Structure for Non-volatile Memory

Jun Yang, Qingsong Wei, *Member, IEEE,* Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He

Abstract—The non-volatile memory (NVM) which can provide DRAM-like performance and disk-like persistency has the potential to build single-level systems by replacing both DRAM and disk. Keeping data consistency in such systems is non-trivial because memory writes may be reordered by CPU. Although ordered memory writes for achieving data consistency can be implemented using the memory fence and the CPU cache line flush instructions, they introduce a significant overhead (more than 10X slower in performance). In this paper, we focus on an important and common data structure, B$^+$Tree. Based on our quantitative analysis for consistent tree structures, we propose NV-Tree, a consistent, cache-optimized and workload-adaptive B$^+$Tree variant with significantly reduced consistency cost (up to 96% reduction in CPU cache line flush). To further optimize NV-Tree under various workloads, we propose a workload-adaptive scheme in which the sizes of individual nodes can be dynamically adjusted to improve the performance over time. We implement and evaluate NV-Tree and NV-Store, a key-value store based on NV-Tree, on an NVDIMM server. NV-Tree outperforms the state-of-art consistent tree structures by up to 12X under write-intensive workloads. NV-Store increases the throughput by up to 7.3X under YCSB workloads compared to Redis.

Index Terms—Non-volatile Memory, Data Consistency, Tree, Workload-adaptive

1 Introduction

Recently, the next generation of Non-Volatile Memory (NVM) technologies has attracted more and more attentions from both industrial and academic researchers. New types of memory such as phase-change memory (PCM) [1], spin-transfer torque memory (STT-RAM) [2] and Memristor [3] are under active development and have the potential to provide comparable performance and much higher capacity than DRAM. More important, they are persistent. The possibility of NVM working as both the main memory and the storage device at the same time brings a revolution to the traditional multi-layered memory and storage architecture.

Given the projected cost [4] and power efficiency of NVM, there have been a number of proposals that replace both disk and DRAM with NVM to build a single-level system [4], [5], [6]. Such systems can (1) eliminate the data movement between disk and memory, (2) fully utilize the low-latency byte-addressable NVM by connecting it through memory bus instead of legacy block interface [7], [8], [9], [10], [11]. However, with data stored only in NVM, in-memory data structures and algorithms must be carefully designed to not only support high-throughput data processing, but also efficiently avoid any inconsistency caused by system failure. In particular, if the system crashes when an update is being made to an in-NVM data structure, it may be left in a corrupted (inconsistent) state as the update is only half-done. To achieve data consistency in NVM, implementing persistent and ordered memory writes is fundamental. However, a modern CPU which is designed on the basis that main memory is always volatile DRAM may cache and reorder memory writes to NVM for the sake of performance. Therefore, it is non-trivial to develop consistent NVM-based systems and data structures, as demonstrated in previous works [12], [13], [14], [15], [16], [17]. Specifically, to guarantee the order of memory writes to NVM, we must (1) prevent them from being reordered by CPU and (2) manually flush CPU cache lines to make data persistent in NVM. Due to the read performance penalty [18], instead of using non-temporal writes, most studies use CPU instructions such as issuing memory fences and flushing cache lines to order memory writes when reads are involved. However, these operations introduce significant overhead [5], [19], [20]. Our experiment shows that the performance drop caused by these operations is more than one order of magnitude on a traditional B$^+$Tree [21], one of the most commonly used data structures in storage systems [22], [23]. Therefore, such consistency cost on NVM must be properly addressed in order to fully utilize the superb performance of NVM.

In this paper, we propose **NV-Tree**, a consistent, cache-optimized and workload-adaptive B$^+$Tree variant which minimizes CPU cache line flush for keeping data consistency in NVM. Specifically, NV-Tree decouples tree nodes into two parts, leaf nodes (LNs) as *critical data* and internal nodes (INs) as *reconstructable data*. By enforcing consistency only on LNs and reconstructing INs from LNs during failure recovery, the consistency cost for INs is eliminated but the data consistency of the entire NV-Tree is still guaranteed. Moreover, NV-Tree keeps entries in each LN *unsorted* which can reduce CPU cache line flush by 82% to 96% for keeping LN consistent. On the other hand, to further increase the CPU cache efficiency, NV-Tree adopts a sorted but pointer-less layout for INs to facilitate the search at the expense of

- *J. Yang, Q. Wei (corresponding author), C. Wang, C. Chen, K. L. Yong are with Data Storage Institute, A-STAR, Singapore.*
E-mail: {yangju, WEI_Qingsong, wangc, CHEN_Cheng, YONG_Khai_Leong}@dsi.a-star.edu.sg
- *B. He is with Nanyang Technological University, Singapore.*
E-mail: BSHE@ntu.edu.sg

periodically performing a special operation called rebuilding. The excessive rebuilding may affect the overall performance but the frequency of rebuilding highly depends on the undergoing workload which is extremely diversified in the real world. To optimize NV-Tree for its unfavorable workloads such as the skewed write-intensive ones, we propose a workload-adaptive scheme which dynamically adjusts the node sizes to reduce the number of rebuilding under different and changing workloads.

Our contributions can be summarized as follows:

1) We quantify the consistency cost for B$^+$Tree using existing approaches, and present two insightful observations: (1) keeping entries in LN sorted introduces a large amount of CPU cacheline flush which dominates the overall consistency cost (over 90%); (2) enforcing consistency only on LN is sufficient to keep the entire tree consistent because INs can always be reconstructed even after system failure.
2) Based on the observations, we present **NV-Tree**, which (1) decouples LNs and INs, and only enforces consistency on LNs; (2) keeps entries in LN unsorted, and updates LN consistently without logging or versioning; (3) organizes INs in a cache-optimized format to increase CPU cache efficiency.
3) We propose a workload-adaptive scheme for NV-Tree to further optimize for various workloads in terms of read/write ratios and localities. In particular, with the rebuilding time reduced significantly, the performance of NV-Tree under skewed write-intensive workloads can be improved by 3X.
4) We implement and evaluate NV-Tree and a key-value store based on it, named **NV-Store**, on a real NVDIMM [24] platform. The experimental results show that NV-Tree outperforms CDDS-Tree [5], the state-of-art consistent tree structure, by up to 12X under write-intensive workloads. Under read-intensive workloads, the speedup can still be as high as 2X. NV-Store increases the throughput by up to 7.3X under YCSB workloads compared to Redis [25].

Organization. The rest of this paper is organized as follows. Section 2 discusses the background of NVM and reviews the related work. Section 3 presents the motivation and design decisions of a consistent tree structure for NVM, followed by detailed design, implementation and optimization of NV-Tree in Section 4 and 5. The evaluation is shown in Section 6. Finally, Section 7 concludes this paper.

2 Background and Related Work

2.1 Non-Volatile Memory (NVM)

Computer memory has been evolving rapidly in recent years. A new category of memory, NVM, has attracted more and more attention in both academia and industry [4], [26], [27]. Early work [28], [29], [30] focuses on flash to overcome its physical limitation such as out-of-place updates, asymmetric I/O performance and unequal unit size for I/O and erasure. However, flash is unsuitable to replace DRAM due to much longer latency and worse endurance.

Recent work has focused on the next generation NVM [31], such as PCM [32] and STT-RAM [2], which (a) is byte-addressable, (b) has DRAM-like performance, (c) supports in-place updates and (d) provides better endurance than flash. PCM is several times slower than DRAM and its write endurance is limited to as few as 10^8 times. However, PCM has larger density than DRAM and shows a promising potential for increasing the capacity of main memory. Although wear-leveling is necessary for PCM, it can be done by memory controller [33], [34]. STT-RAM has the advantages of lower power consumption over DRAM, unlimited write cycles over PCM, and lower read/write latency than PCM. Recently, Everspin announced its commercial 64Mb STT-RAM chip with DDR3 interface [35]. In this paper, our target **NVM** is referred to the next generation of non-volatile memory which is byte-addressable and in-place-update supported (no garbage collection needed).

TABLE 1
Characteristics of Different Types of Memory

Category	Read Latency (*ns*)	Write Latency (*ns*)	Endurance (*# of writes per bit*)
SRAM	2-3	2-3	∞
DRAM	15	15	10^{18}
STT-RAM	5-30	10-100	10^{15}
PCM	50-70	150-220	10^8-10^{12}
Flash	25,000	200,000-500,000	10^5

As an alternative to NVM, NVDIMM [20], which is commercially available [24], provides persistency and DRAM-like performance. NVDIMM is a combination of DRAM and NAND flash. During normal operations, NVDIMM is working as DRAM while flash is invisible to the host. However, upon scheduled or unscheduled power offs, NVDIMM saves all the data from DRAM to flash by using supercapacitor to make the data persistent. Since this process is transparent to other parts of the system, NVDIMM can be treated as NVM. In this paper, our NV-Tree and NV-Store are implemented and evaluated on a NVDIMM platform.

2.2 Data Consistency in NVM

NVM-based single level systems [4], [5], [19], [20], [36] have been proposed and evaluated using the simulated NVM in terms of cost, power efficiency and performance. As one of the most crucial features of storage systems, data consistency guarantees that stored data can survive system failure. Based on the fact that data is recognizable only if it is organized in a certain format, updating data consistently means preventing data from being lost or partially updated after a system failure. However, the atomicity of memory writes can only be supported with a very small granularity or no more than the memory bus width (8 bytes for 64-bit CPUs) [37] which is addressed in previous work [19], [20], [26], so updating data larger than 8 bytes requires certain mechanisms to make sure data can be recovered even if system failure happens before it is completely updated. Particularly, the approaches such as logging and copy-on-write make data recoverable by writing a copy elsewhere before updating the data itself. For append-only data structures, using an item counter for visibility control [38] can also prevent the exposure of incomplete or half-updated data. To implement these approaches, memory writes must be executed in a certain order, e.g., the memory writes for the copy of data must be completed before updating the data itself. Such ordered writes are also required in pointer-based data structures such as B$^+$Tree. If one tree node is split, the new node must be written completely before its pointer being added to the parent node, otherwise, the parent node may contain an invalid pointer if the system crashes.

Unfortunately, memory writes may be reordered by either CPU or memory controller. Alternatively, without modifying existing hardware, the sequence of {MFENCE, CLFLUSH, MFENCE} instruction (referred to FLUSH in the rest of this paper) can be executed to form ordered memory writes [5]. Specifically, MFENCE issues a memory barrier which guarantees the memory operations after the barrier cannot proceed until those before the barrier complete, but it does not guarantee the order of write-back to the memory from CPU cache. On the other hand, CLFLUSH can explicitly invalidate the corresponding dirty CPU cache lines so that they can be flushed to NVM by CPU which makes the memory write persistent eventually. However, CLFLUSH can only flush a dirty cache line by explicitly invalidating it which makes CPU cache very inefficient. Although such invalidations can be avoided if the hardware itself can be modified to implement *epoch* [19], CPU cache line flush must be executed. Reducing it is still necessary to not only improve performance but also extend the life cycle of NVM with reduced memory write.

2.3 Related Work

Recent work proposed mechanisms to provide data consistency in NVM-based systems by either modifying existing hardware or using CPU primitive instructions such as non-temporal stores with write-combining (e.g. MOVNTDQ), or MFENCE with CLFLUSH. As non-temporal stores with write-combining cause severe read performance degradation due to the absence of CPU cache [18], the use of these instructions is limited to write-only components such as logging [39]). BPFS [19] proposed a new file system which is resided in NVM. It adopts a copy-on-write approach called short-circuit shadow paging using *epoch* which can flush dirty CPU cache lines without invalidating them to order memory writes for keeping data consistency. However, it still suffers from the overhead of cache line flush. It must be implemented by modifying existing hardware which is not practical in most cases. Volos et al. [26] proposed Mnemosyne, a new program interface for memory allocations in NVM. To manage memory consistency, it presents persistent memory region, persistency primitives and durable memory transaction which consist of MFENCE and CLFLUSH eventually. NV-Heaps [40] is another way to consistently manage NVM directly by programmers based on *epoch*. It uses *mmap* to access spaces in NVM and gives a way to allocate, use and deallocate objects and their pointers in NVM. Narayanan et al. [20] proposed a way to keep the whole system status when power failure happens. Realizing the significant overhead of flushing CPU cache line to NVM, they propose to flush-on-fail instead of flush-on-demand. However, they cannot protect the system from any software failure. In general, flushing CPU cache line is necessary to order memory writes and used in almost all the existing NVM-based systems [41], [42], [43], [44], [45].

The most related work to NV-Tree is CDDS-Tree [5] which uses FLUSH to enforce consistency on all the tree nodes. In order to keep entries sorted, when an entry is inserted to a node, all the entries on the right side of the insertion position need to be shifted. CDDS-Tree performs FLUSH for each entry shift, which makes the consistency cost very high. Moreover, it uses the entry-level versioning approach to keep consistency for all tree operations. Therefore, a background garbage collector and a relatively complicated recovery process are both needed. The consistency cost of tree structures on NVM has been proven to be crucial as similar work [45] has been published concurrently with ours [46] to address this issue. In comparison, the wB$^+$-Tree in [45] does redo-logging during node split and is tested through simulation while our NV-Tree is log-free and evaluated on a real NVDIMM platform with practical failure-recoverability and consistency guarantee.

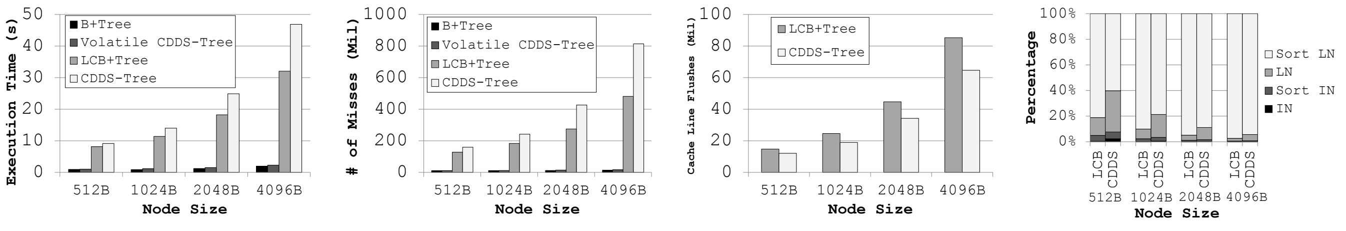
A preliminary version of NV-Tree has been presented in a previous paper [46]. This paper goes beyond the preliminary version with the enhancement of workload-adaptive feature as following: (1) We develop a workload profiling algorithm for NV-Tree to learn the undergoing workload in terms of read/write ratios and localities (Section 5.2); (2) With different workload-adaptive requirement, we present three different decision-making strategies for adjusting the node size (Section 5.3); (3) We propose an analytical model to help identify the boundary of node sizes to avoid the performance drop because of too large/small nodes (Section 5.4); (4) We extend the evaluation to study the workload-adaptive NV-Tree under more workloads, and demonstrate the self-tuning feature of NV-Tree under skewed and dynamic workloads. Note that our preliminary version [46] only evaluates the previous version of NV-Tree without workload-adaptivity under low-locality workloads; (5) Performance-wise, compared with the previous version, the throughput of NV-Tree under skewed write-intensive workloads can be improved by up to 3X. The throughput of NV-Store based on workload-adaptive NV-Tree can be improved by up to 30% under YCSB workloads.

3 Problem Formulation

3.1 Motivations

To quantify the consistency cost, we compare the execution of performing one million insertion in (a) a standard B$^+$Tree [21] without consistency guarantee, (b) a log-based consistent B$^+$Tree (LCB$^+$Tree), (c) a CDDS-Tree [5] using versioning, and (d) a volatile CDDS-Tree with FLUSH disabled. In LCB$^+$Tree, before modifying a node, its original copy is logged and FLUSHed. The modified part of it is then FLUSHed to make the changes persistent. Note that we only use LCB$^+$Tree as the baseline to illustrate one way to use logging to guarantee the consistency. We understand optimizations (such as combining several modification to one node into one flush) can be made to improve the performance of LCB$^+$Tree but it is beyond the scope of this paper. Since CDDS-Tree is not open-source, we implement it ourselves and achieve similar performance to that in the original paper [5]. We are motivated by three major observations of the experimental results.

OB1. ***Keeping consistency causes a huge amplification of the CPU cache line invalidation and flush, which increases the cache misses significantly.*** As shown in Figure 1a, for one million insertion with 4KB nodes, the LCB$^+$Tree and CDDS-Tree are up to 16X and 20X slower than their volatile version, respectively. Such performance drop is caused by



(a) Execution Time (b) L3 Cache Misses (c) Number of Cache Line Flushes (d) Percentage Breakdown

Fig. 1. Consistency Cost Analysis of B$^+$Tree and CDDS-Tree For One Million Insertion With Different Node Sizes

the increased number of cache misses and additional cache line flush. We use Intel vTune Amplifier[1], a CPU profiling tool, to count the L3 cache misses during the one million insertion. As shown in Figure 1b, while the volatile CDDS-Tree or B$^+$Tree produces about 10 million L3 cache misses, their consistent version causes about 120-800 million cache misses which explains the performance drop. Figure 1c shows the total number of cache line flushes in CDDS-Tree and LCB$^+$Tree for one million insertion. With 0.5KB/1KB/2KB/4KB nodes, the total amount of cache line flushes is 14.8/24.6/44.7/85.26 million for LCB$^+$Tree, and 12.1/19.0/34.2/64.7 million for CDDS-Tree.

OB2. ***The consistency cost of* FLUSH *mostly comes from* FLUSH*ing shifted entries in order to keep LN sorted.*** The numbers of both the cache misses and cache line flushes in LCB$^+$Tree and CDDS-Tree are proportional to the node size due to the FLUSH for keeping the entries sorted. Specifically, for LCB$^+$Tree and CDDS-Tree, all the shifted entries caused by inserting an entry inside a node need to be FLUSHed to make the insertion persistent. As a result, the amount of data to be FLUSHed is related to the node size for both trees. We further categorize the CPU cache line flush into four types, as shown in Figure 1d, *Sort LN*/*Sort IN* stands for the cache line flush of shifted entries. It also includes the FLUSH of logs in LCB$^+$Tree. *LN*/*IN* stands for the FLUSH of other purpose such as FLUSHing new nodes and updated pointers after split, etc. The result shows that most of the cache line flushes (60%-94% in CDDS-Tree, 81%-97% in LCB$^+$Tree) are for shifting entries in LN. Note that CDDS-Tree is slower than LCB$^+$Tree by 11%-32% even though it produces less FLUSHes. The reasons are that (1) the FLUSH unit in CDDS-Tree is the entry size, which is much smaller than that in LCB$^+$Tree, and (2) the performance of FLUSH for small objects is over 25% slower than that for large objects [5].

OB3. ***Not all the data need to be consistent to keep the entire data structure consistent.*** Given a data structure, as long as some parts of it (denoted as *critical data*) are consistent, the rest (denoted as *reconstructable data*) can be reconstructed without losing consistency for the whole data structure. For instance, in B$^+$Tree, LNs are *critical* while INs are *reconstructable* as they can always be reconstructed from LNs. That suggests we may reduce the consistency cost by only enforcing consistency on *critical data*.

3.2 Design Decisions

Based on our observations above, we make three major design decisions as the following.

1. https://software.intel.com/en-us/intel-vtune-amplifier-xe

D1. ***Selectively Enforce Data Consistency.*** We distinguish LNs and INs in a B$^+$Tree by decoupling them as *critical* and *reconstructable* data, respectively. Different from the traditional approach where all nodes are updated with consistency guaranteed, our design only enforces consistency on LNs but processes INs with no consistency guaranteed to reduce the consistency cost. Upon system failure, INs are reconstructed from the consistent LNs so that the whole tree is always consistent.

D2. ***Keep Entries in LN Unsorted.*** We adopt unsorted LNs so that the FLUSH operation used in LCB$^+$Tree and CDDS-Tree for shifting entries upon insertion can be avoided. Meanwhile, entries of INs are still sorted to optimize search performance. Although the unsorted LN strategy is not new [38], we are the first to quantify its impact on the consistency cost and use it to reduce the consistency cost in NVM. Moreover, based on our unsorted scheme for LNs, both the content (entry insertion/update/deletion) and structural (split) changes in LNs are only visible after a CPU primitive atomic write. Therefore, LNs can be protected from being corrupted by any half-done updates due to system failure without using logging or versioning. Thanks to the invisibility of on-going updates, the parallelism of accessing an LN is also increased because searching is no longer blocked by the concurrent on-going update.

D3. ***Organizing IN in Cache-optimized Format.*** The CPU cache efficiency is a key factor to the performance of memory-only systems. Inspired by the cache-optimized data format [47], all INs are designed to be stored in a contiguous memory space and located by offset instead of pointers, and all nodes are aligned to CPU cache line. As a result, higher space utilization and cache hit rate can be achieved.

4 Design and Implementation

4.1 Overview of NV-Tree

We propose NV-Tree, a consistent and workload-adaptive B$^+$Tree variant for NVM with minimized consistency overhead. As shown in Figure 2, all the data is stored in LNs in a single-linked list. Each LN can also be accessed by the LN pointer stored in its parent, denoted as PLN (parent of leaf node). All the IN/PLNs are stored in a contiguous memory space which means the position of each IN/PLN is fixed upon creation. The *node id* of each IN/PLN is assigned sequentially from 0 (root). Therefore it can be used to calculate the offset of each IN/PLN to the root. Given the memory address of the root, all the IN/PLNs can be located without using any pointers. Each key/value pair (KV-pair) stored in LNs is encapsulated in an *LN_element*.

Keeping each LN and the LN list consistent in NV-Tree without using logging or versioning is non-trivial. Different

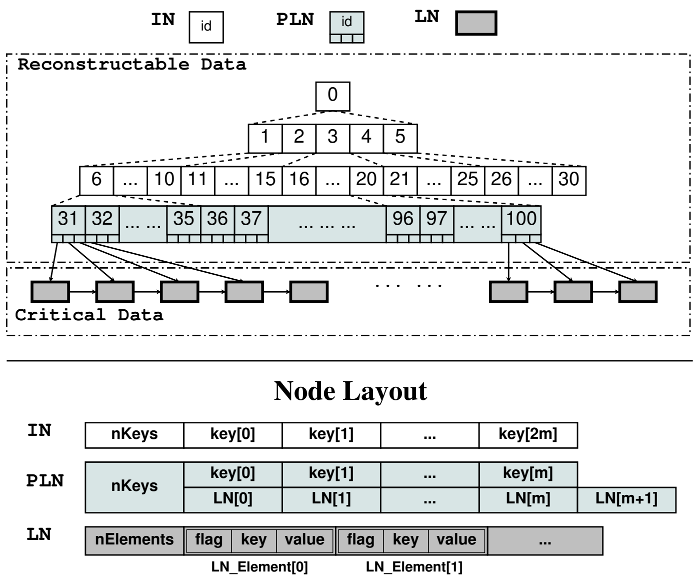


Fig. 2. NV-Tree Overview and Node Layout

Algorithm 1 NV-Tree LN Lookup

```
 1: procedure FINDLEAF($k, r$)
Input: $k : key, r : root$
Output: $LNpointer : the\ pointer\ of\ target\ leaf\ node$
 2:     $id \leftarrow 0$
 3:     while $id \notin PLNIDs$ do
 4:         $IN \leftarrow$ memory address of node id
 5:         $pos \leftarrow$ BINARYSEARCH($key, IN$)
 6:         $id \leftarrow id * (2m+1) + 1 + pos$
 7:     end while
 8:     $PLN \leftarrow$ memory address of node id
 9:     $pos \leftarrow$ BINARYSEARCH($key, PLN$)
10:     return $PLN.LNpointers[pos]$
11: end procedure
```

from an ordinary B$^+$Tree, both update and deletion are implemented as insertion using an append-only strategy discussed in Section 4.3. Any insertion/update/deletion operations may lead to a full LN which triggers either *split*/*replace*/*merge* discussed in Section 4.4. We carefully design the write order for insertion/update/deletion and split/replace/merge using FLUSH to guarantee the modified data cannot be seen until a successful atomic write. When one PLN is full, a procedure called *rebuilding* is executed to reconstruct a new set of IN/PLNs to accommodate more LNs, discussed in Section 4.5. To address the performance issue caused by frequent rebuilding under skewed write-intensive workloads, instead of using a fixed and equal size for all the nodes, we design a workload-adaptive scheme to tune the node sizes based on a workload profiling algorithm and an analytical model, discussed in Section 5.

4.2 Locating Target LN

The procedure of locating a target LN with a given key in NV-Tree is different from that in a standard B$^+$Tree. As shown in Algorithm 1, given the search key and the memory address of root, INs are searched level by level, starting from root with node id 0 (line 2). On each level, which child to go in the next level is determined by a binary search based on the given search key (line 3-7). For instance, with keys and pointers having the same length, if a PLN can hold m keys and $m+1$ LN pointers, an IN can hold $2m$ keys. If the node id of current IN is i and the binary search finds the smallest key which is no smaller than the search key is at position

Algorithm 2 NV-Tree Insertion

```
Input: k: key, v: value, r: root
Output: SUCCESS/FAILURE
 1: if $r = NULL$ then
 2:     $r \leftarrow$ CREATENEWTREE($k, v$)
 3:     return SUCCESS
 4: end if
 5: $leaf \leftarrow$ FINDLEAF($k, r$)
 6: if LN has space for new KV-pair then
 7:     $newElement \leftarrow$ CREATEELEMENT($k, v$)
 8:     FLUSH($newElement$)
 9:     ATOMICINC($leaf.number$)
10:     FLUSH($leaf.number$)
11: else
12:     LEAFSPLITANDINSERT($leaf, k, v$)
13: end if
14: return SUCCESS
```

k in current IN, then the next node to visit should have the node id $(i \times (2m+1) + 1 + k)$ (line 6). On reaching a PLN, the address of the target LN can be retrieved from the leaf node pointer array (line 8-10).

As every IN/PLN has a fixed location once rebuilt, PLNs are not allowed to split. Therefore, the content of INs (PLNs excluded) remains unchanged during normal execution. Therefore, NV-Tree does not need to use locks in INs for concurrent tree operations which in turn increases the concurrency of NV-Tree.

4.3 Insertion, Deletion, Update and Search

Insertion is one of the most commonly used operations. As shown in Algorithm 2, if the target tree is empty, a new tree contains the provided KV-pair will be created (line 1-4), otherwise, the first step is to find the target LN (line 5). After target LN is located, a new *LN_element* will be generated using the new KV-pair. If the target LN has enough space to hold the *LN_element*, the insertion completes after the *LN_element* is appended, and the *nElement* is increased by one successfully (line 6-10). Otherwise, the target LN will split before insertion (line 12, discussed in Section 4.4). Figure 3a shows an example of inserting a KV-pair {7,b} into an LN with existing two KV-pairs {6,a} and {8,c}.

Deletion is implemented just the same as insertion except a special NEGATIVE flag. Figure 3b illustrates the deletion of {6,a} in the original LN. A NEGATIVE *LN_element* {6,a} (marked as '-') is inserted. Note that the NEGATIVE one cannot be inserted unless a normal one (with a POSITIVE flag) is found. The space of both the NEGATIVE and normal *LN_element*s are recycled by later splits.

Update is implemented by inserting two *LN_element*s, a NEGATIVE one with the same *value* and a normal one with updated *value*. For instance, as shown in Figure 3c, to update the original {8,c} with {8,y}, the NEGATIVE *LN_element* for {8,c} and the normal one for {8,y} are appended accordingly.

Note that the order of appending *LN_element* before updating *nElement* in LN is guaranteed by FLUSH (line 8-10). Inspired by the visibility control mechanism used in an appended-only data structure [38], the appended *LN_element* is only visible after the *nElement* is increased by a successful atomic write to make sure LN cannot be corrupted by system failure.

Search starts with locating the target LN with the given key. After the target LN is located, since keys are unsorted in LN, a scan is performed to retrieve the *LN_element* with

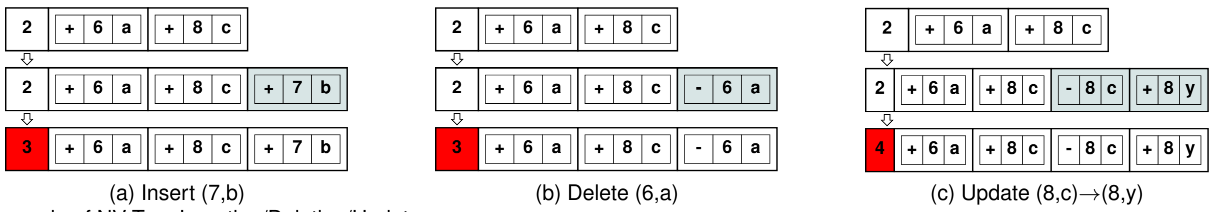


Fig. 3. Example of NV-Tree Insertion/Deletion/Update

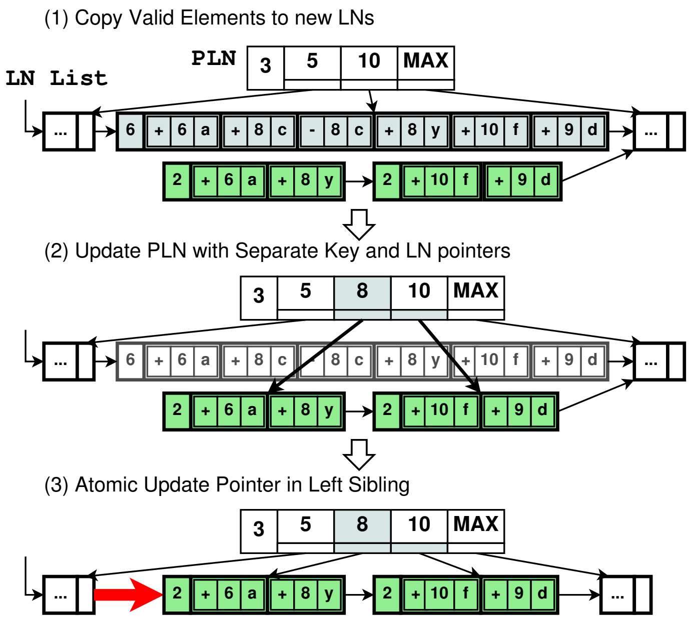

(1) Copy Valid Elements to new LNs
PLN
LN List
(2) Update PLN with Separate Key and LN pointers
(3) Atomic Update Pointer in Left Sibling

Fig. 4. Example of LN Split

the given *key*. Note that if two *LN_element*s have the target key and same *value* but one of them has a NEGATIVE flag, both of them are ignored because that indicates the corresponding KV-pair is deleted. Although the unsorted leaf increases the searching time inside LN, the entries in IN/PLNs are still sorted so that the search performance is still acceptable as shown in Section 6.3.

All the modification of LNs/PLNs is protected by light-weight locks. Meanwhile, searching is never blocked by any ongoing modification as long as the *nElement* is used as the boundary of the search range in LNs/PLNs.

4.4 LN Split/Replace/Merge

When an LN is full, it is scanned to identify the number of valid *LN_element*s. Those NEGATIVE ones and the corresponding normal ones are both considered invalid. The next step is determined by the number of valid elements.

If the percentage of valid elements is above the minimal fill factor (e.g., 50% in standard B$^+$Tree), we perform **split** as illustrated in Figure 4. Two new LNs (left and right) are created and valid elements are copied to either of them according to the selected separate key. Then the new KV-pair is inserted accordingly. The split completes after the pointer in the left sibling of the old LN is updated to point to new left LN using an atomic write. Before that, all the changes made during split are not visible to the tree.

If the percentage is below the minimal fill factor, we check the number of *LN_element*s in the right sibling of the old LN. If it is above the minimal fill factor, we perform **replace**, otherwise, we perform **merge**. For **replace**, those valid *LN_element*s in the old LN are copied to a new LN which replaces the old one in the LN list using an atomic write. For **merge**, the old LN and its right sibling will be replaced by a new LN which contains their valid *LN_element*s. Note that we use the *nElement* instead of the number of valid elements in the right sibling to decide which operation to perform because finding the latter needs to perform a scan which is relatively more expensive. Due to space limitation, examples of replace and merge operations are omitted here.

4.5 Rebuilding

As the memory address of each IN/PLN is fixed upon creation, cache-optimized IN/PLNs are not allowed to split. Therefore, the disadvantage of the cache-optimized design is that when one PLN is full, all IN/PLNs have to be reconstructed to make space in PLNs to hold more LN pointers. The rebuilding procedure introduces both temporal and spatial overhead but our experimental results in Section 6.5 show that such overhead is negligible. The first step of rebuilding is to determine the new number of PLNs based on the current number of LNs. In our current implementation, to delay the next rebuilding as much as possible, each PLN stores exactly one LN pointer after rebuilding.

During normal execution, we can use *rebuild-from-PLN* strategy by redistributing all the keys and LN pointers in existing PLNs into the new set of PLNs. However, upon system failure, we use *rebuild-from-LN* strategy. Because entries are unsorted in each LN, *rebuild-from-LN* needs to scan each LN to find its maximum key to construct the corresponding key and LN pointer in PLN. *Rebuild-from-LN* is more expensive than *rebuild-from-PLN* but is only executed upon system failure. Compared to a single tree operation (e.g., insertion or search), one rebuilding may be very time-consuming in a large NV-Tree. Although such overhead is neglectable (less than 1%) in a long-running application under low-locality workloads (e.g., key selection is random and uniformly distributed), the performance of NV-Tree can be significantly affected by the excessive rebuilding under skewed (high-locality, in which the key selection is limited in a small region) write-intensive workloads. To optimize NV-Tree for such workloads, we propose a workload-adaptive scheme which is discussed in detail in Section 5.

During rebuilding, insertion/update is not allowed to proceed but search can still be executed. It is achieved by storing the new IN/PLNs generated by the on-going rebuilding without deleting the existing ones. Since all the existing nodes remain unchanged during rebuilding, they can still be used by search operations without any locks. Given the fact that IN/PLNs occupy much less space than LNs do in NV-Tree, the spatial and temporal overhead of rebuilding is totally acceptable. For instance, our experiment results in Section 6.5 show that when inserting 100 million entries with random keys to an NV-Tree with 4KB nodes

(>4GB), rebuilding is executed only for two times. The rebuilding time is 41.6*ms*, which occupies 0.01% of the total execution time. The size of the INs generated by the first/second rebuilding is only about 1MB/129MB, which occupies 0.02%/3% of the total tree size.

4.6 Recovery

Since all LNs are consistent, *rebuild-from-LN* described in Section 4.5 is sufficient to recover an NV-Tree from either normal shutdown or system failure.

To further optimize the recovery after normal shutdown, our current implementation can achieve **instant recovery** by storing IN/PLNs in NVM. More specifically, during normal shutdown, we (1) FLUSH all IN/PLNs to NVM, (2) save the root pointer to a reserved position in NVM, (3) and use an atomic write to mark a special flag along with the root pointer to indicate a successful shutdown. Then, the recovery can (1) start with checking the special flag, (2) if it is marked, reset it and use the root pointer stored in NVM as the current root to complete the recovery. Otherwise, it means a system failure occurred, and a *rebuild-from-LN* procedure is executed to recover the NV-Tree.

5 Optimization for Skewed Workloads

5.1 Motivation

One of the major difference between NV-Tree and other tree structures is the additional rebuilding operation. Its negative impact on performance highly depends on the undergoing workload because different workloads may have different rebuilding frequency. As the rebuilding is only triggered by a full PLN, frequent rebuilding is essentially caused by excessive splits in a small region of the entire tree, e.g., under skewed write-intensive workloads. In order to solve this problem, we propose a workload-adaptive scheme for NV-Tree to (1) identify the workload locality and (2) adjust the individual node sizes accordingly to reduce the total number of splits and rebuilding.

5.2 Workload Profiling

The purpose of understanding a workload is to determine the proper size of the individual tree nodes to reduce the number of splits and rebuilding. We profile a workload from two aspects: (1) the read/write ratio (**r/w ratio**) of each node to identify a node is read-/write-intensive, and (2) read/write locality (**r/w locality**) of the entire workload to distinguish the hot/cold nodes for both read and write.

In general, we maintain two global counters, $nWrites$ and $nReads$, to track the numbers of write (insertion/deletion/update) and read (search) operations, respectively. Since only LNs are allowed to split in NV-Tree, we also maintain two counters in each LN to keep track of the number of write and read operations, $nWrites_{LN[i]}$ and $nReads_{LN[i]}$. Thus, we can tell whether a node is read or write intensive by comparing the two counters of it. To be more precise, as the amount of data written by different write operations is not equal, $nWrites$ or $nWrites_{LN[i]}$ represents the actual number of *LN_element*s appended instead of the write operations. The description of the statistic information used by NV-Tree is summarized in Table 2.

Table 3 shows how to maintain the statistic information for different tree operations in NV-Tree. Note that each update operation appends two *LN_element*s instead of one in an insertion/deletion operation. And if one LN (i) is full and split into two new LNs (j and k), the $nWrites_{LN[j/k]}$ and $nReads_{LN[j/k]}$ will be half of $nWrites_{LN[i]}$ and $nReads_{LN[i]}$, respectively.

TABLE 2
Statistic Information for Workload Profiling

Counter	Description
$nWrites$	Total number of *LN_element*s appended
$nReads$	Total number of search operations
$nWrites_{LN[0..N]}$	Number of *LN_element*s appended in each LN
$nReads_{LN[0..N]}$	Number of search operations performed in each LN

TABLE 3
Summary of Maintaining Statistic Information for Different Operations

{Operation}[LN_id]	Change of Statistic Information
{Insertion}[i] /{Deletion}[i]	$nWrites \leftarrow nWrites + 1$ $nWrites_{LN[i]} \leftarrow nWrites_{LN[i]} + 1$
{Update}[i]	$nWrites \leftarrow nWrites + 2$ $nWrites_{LN[i]} \leftarrow nWrites_{LN[i]} + 2$
{Search}[i]	$nReads \leftarrow nReads + 1$ $nReads_{LN[i]} \leftarrow nReads_{LN[i]} + 1$
{Split}[i]⇒[j,k]	$nWrites_{LN[j]} \leftarrow nWrites_{LN[k]} \leftarrow Writes_{LN[i]}/2$ $nReads_{LN[j]} \leftarrow nReads_{LN[k]} \leftarrow nReads_{LN[i]}/2$

Given the above statistic information, the **r/w ratio** of a given LN(i) can be calculated as $R_{r/w}[i] = \frac{nReads_{LN[i]}}{nWrites_{LN[i]}}$. The LNs with $R_{r/w} > 1$ are **read-intensive** while those with $R_{r/w} < 1$ are **write-intensive**. On the other hand, the **r/w locality** of a workload can be profiled by comparing the individual $nWrites_{LN[i]}$ and $nReads_{LN[i]}$ to the average $\frac{nWrites}{N_{LN}}$ and $\frac{nReads}{N_{LN}}$. Those nodes with $nWrites_{LN[i]} > \frac{nWrites}{N_{LN}}$ or $nReads_{LN[i]} > \frac{nReads}{N_{LN}}$ can be considered as **hot-write** nodes or **hot-read** nodes, respectively. Similarly, there are also **cold-write/-read** nodes.

5.3 Selecting Proper Node Sizes

Based on the statistic information collected over time, the workload-adaptive algorithm determines the new node size of the two new nodes for splitting a full node. The sizes of write-intensive or read-intensive nodes are preferred to be increased or decreased, respectively. However, in some circumstances, that may not be the best choice. For instance, if a write-intensive node is also hot-read, increasing its size may slow down the in-node search performance. Similarly, if a read-intensive node is also hot-write, decreasing its size may trigger more splits.

In addition to making decision of increasing or decreasing the node size, how much exactly to increase or decrease is also an important issue because it affects the effectiveness and sensitivity of the workload-adaptive approach. Because of the CPU cache efficiency, the node size should be the integral multiple of the CPU cache line size. Therefore, the minimum size to increase or decrease is the CPU cache line size. However, such small changes in node sizes make the algorithm inefficient under highly-skewed workloads because it takes a long time to get the optimal node size

Algorithm 3 Determine the new LN size

```
1: procedure GETNEWSIZE($id, strategy$)
2:     $UPorDOWN \leftarrow 0$
3:     if $\frac{nReads_{LN[id]}}{nWrites_{LN[id]}} < 1$ then
4:         if $((s = conservative)$
5:             $\&(nWrites_{LN[id]} > \frac{nWrites}{N_{LN}})$
6:             $\&(nReads_{LN[id]} < \frac{nReads}{N_{LN}}))$
7:             $\|(s = aggressive)\|(s = balanced)$ then
8:             $UPorDOWN \leftarrow 1$
9:         end if
10:    else
11:        if $((s = conservative)$
12:            $\&(nWrites_{LN[id]} < \frac{nWrites}{N_{LN}})$
13:            $\&(nReads_{LN[id]} > \frac{nReads}{N_{LN}}))$
14:            $\|((s = balanced)$
15:            $\&(nReads_{LN[id]} > \frac{nReads}{N_{LN}}))$ then
16:            $UPorDOWN \leftarrow -1$
17:        end if
18:    end if
19:    if $UPorDOWN = 0$ then
20:        return $Size_{LN[id]}$
21:    end if
22:    if $UPorDOWN = -1$ then
23:        return $Size_{LN[id]} - Size_{cacheline}$
24:    else
25:        if $\big((s = balanced)\ \&\ (nWrites_{LN[id]} > \frac{nWrites}{N_{LN}})\big)$
26:            $\|(s = aggressive)$ then
27:            return $2 \cdot Size_{LN[id]}$
28:        else
29:            return $Size_{LN[id]} + Size_{cacheline}$
30:        end if
31:    end if
32: end procedure
```

for hot-write nodes. On the other hand, too much increase or decrease in node sizes can lead to the performance drop under low locality workloads.

Based on the criteria for whether to increase or decrease the node size and how much change each time in node size, we propose three basic strategy in this section.

Conservative Strategy. In this strategy, only the size of write-intensive, hot-write, cold-read nodes are increased, and only the size of read-intensive, cold-write, hot-read nodes are decreased. For the other types of nodes, the node size remains unchanged. Moreover, the size only changes one cache line size each time. Therefore, the overall performance can always be improved steady but slowly. However, it takes long time to achieve the peak performance under highly-skewed workloads. Moreover, if the workload is dynamically changing over time, the node size will not be changed in time to achieve better performance.

Aggressive Strategy. In this strategy, the size of all write-intensive nodes are increased but the size of the other types of nodes remains unchanged. Moreover, the new size doubles the original node size. Therefore, the overall performance can be improved much faster than that in the conservative strategy under highly-skewed write-intensive workloads. However, if the locality of the workload is low, such rapid size change may lead to worse performance, especially for those read-intensive workloads. Note that as the node size is always increased, the space utilization is much worse in aggressive strategy than that in the conservative strategy.

Balanced Strategy. Unlike the above two strategies, this strategy looks for the balance among the reduction of splits, the search performance and the space utilization. The size of write-intensive, hot-write nodes is doubled, the size of write-intensive, cold-write nodes is increased by the CPU cache line size. On the other hand, the size of read-intensive, hot-read nodes is decrease by the CPU cache line size. The size of the rest remains unchanged.

Algorithm 3 shows the pseudo-code of selecting proper size of the two new LNs for splitting a full LN in NV-Tree with the above three basic strategies. Specifically, for a write-intensive LN, the size is increased if the aggressive/balanced strategy is in use or the LN is also a hot-write cold-read one under the conservative strategy (line 3-9). For non-write-intensive LN, the size is decreased if the LN is a hot-read cold-write one under the conservative strategy or a hot-read one under the balanced strategy (line 10-18). After that, the new node size is returned according to the strategy in use (line 19-31). Note that all these three strategies change the node size one step at a time without knowing the optimal, but the size of a write-intensive or read-intensive node cannot be increased/decreased without limitation because the trade-off between the cost of split and in-node search. Enlarging or shrinking any LNs too much can lead to performance drop. To address issue, we propose an analytical model to help identify the maximum/minimum node size that prevents the performance drop under on-going workload.

5.4 Analytical Model

The optimal node size should be the one that maximizing the difference between the positive performance impact (i.e., the saved split cost when enlarging LNs or the saved in-node search cost when shrinking LNs) and the negative impact (i.e., the extra in-node search cost when enlarging LNs or the early split cost when shrinking LNs). As the future workload is unknown, we can only change the node size gradually towards the local optimum instead of the global one. However, we can still derive the boundary of the node size when increasing or decreasing more can cause performance drop in the near future.

We propose an analytical model to quantify the positive and negative impact of increasing/decreasing the size of the two new nodes created upon split. In this model, we also take the performance of the memory in use into account to improve the accuracy. Specifically, the positive impact of increasing the node size is the saved split cost, denoted as in Equation 1 where $T_{r(oldSize)}$ is the execution time of reading the node with $oldSize$, $T_{w(\frac{oldSize}{2})}$ is the execution time of `FLUSH`ing the two new nodes.

$$Positive_{\Uparrow} = T_{r(oldSize)} + 2 \cdot T_{w(\frac{oldSize}{2})} \tag{1}$$

The negative impact of increasing the node size is the extra cost of in-node scan due to the larger size. We set the time window from the current split to the next split of the two new nodes with the new node size. Paying extra cost of in-node scan starts when the content of the new node grows over the old size, every in-node scan after this point has to scan more (equal to the old size) until the new node splits. Therefore, the negative impact can be denoted as in Equation 2 where $Size_{element}$ is the size of one LN_element.

$$Negative_{\Uparrow} = \left(\frac{newSize - oldSize}{Size_{element}}\right) \cdot \frac{nReads_{LN[i]}}{nWrites_{LN[i]}} \cdot T_{r(oldSize)} \cdot 2 \tag{2}$$

Similarly, the positive and negative impact of shrinking a LN can be denoted as in Equation 3 and 4.

$$Positive_{\Downarrow} = \left(\frac{oldSize-newSize}{Size_{element}}\right) \cdot \frac{nReads_{LN[i]}}{nWrites_{LN[i]}} \cdot T_{r(newSize)} \cdot 2 \quad (3)$$

$$Negative_{\Downarrow} = T_{r(newSize)} + 2 \cdot T_{w(\frac{newSize}{2})} \quad (4)$$

When a LN is decided to be enlarged, the $newSize$ will be checked to make sure $Positive_{\Uparrow} > Negative_{\Uparrow}$, otherwise, the node size will remain unchanged. Shrinking a LN must make sure the $newSize$ satisfies $Positive_{\Downarrow} > Negative_{\Downarrow}$. As long as the read/write performance $T_{r(size)}$ and $T_{w(size)}$ of the memory in use is provided, the size limit of LNs can be determined by the analytical model using the statistic information $nReads_{LN[i]}$ and $nWrites_{LN[i]}$ to prevent the potential performance drop under a specific workload. Note that $T_{r(size)}$ and $T_{w(size)}$ must be tested thoroughly for different types of NVM because they are not linearly proportional to the $size$ due to the effect of cache line hit and the use of memory fences and cache line flushes.

5.5 Rebuilding Optimization

Rebuilding is executed when any PLNs are full. Using the workload-adaptive approach to reduce the number of splits can definitely help reduce this overhead. To further optimize NV-Tree in terms of rebuilding, we identify and increase the size of the hot-write PLNs during each rebuilding to delay the next one. Specifically, A PLN is considered **hot-write** if $nWrites_{PLN[i]} = \sum_{j\in(IDs\ of\ LNs\ in\ PLN[i])} nWrites_{LN[j]} > \frac{nWrites}{N_{PLN}}$. During rebuilding, we increase the size of all the **hot-write** PLNs. To minimize the space usage, we use the default size of INs as the unit size for enlarging PLNs.

6 Evaluation

In this section, we firstly show the advantage of NV-Tree over other tree structures in terms of insertion/update/deletion/search performance and overall performance under mixed workloads. To evaluate the workload-adaptive scheme, we measure the throughput of NV-Tree under skewed write-intensive workloads with three strategies (conservative, aggressive and balanced), and compare it with the throughput of NV-Tree without workload-adaptivity (denoted as NV-Tree'). Then we quantify the overhead of rebuilding by calculating the percentage of the rebuilding time of NV-Tree in the total execution time under insertion-only workloads. To evaluate NV-Tree from a system perspective, we use YCSB [48], a benchmark for KV-stores, to compare our NV-Tree-based KV-Store (NV-Store) with Redis [25], a well-known in-memory KV-store. Finally, we discuss the performance of NV-Tree on different hardware including different types of NVM and CPUs with new instructions available.

6.1 Methodology

6.1.1 Implementation Effort

We implement our NV-Tree from scratch, an LCB$^+$Tree by applying FLUSH and logging to a standard B$^+$Tree [49], and a CDDS-Tree [5]. To further optimize NV-Tree under various workloads, we implement the workload-adaptive feature into NV-Tree. All the results showed in this paper is measured using the NV-Tree with workload-adaptivity.

To make use of NVDIMM as a persistent storage device, we modify the memory management of Linux kernel to add new functions (e.g., `malloc_NVDIMM`) to directly allocate memory space from NVDIMM. The NVDIMM space used by NV-Tree is guaranteed to be mapped to a contiguous (virtual) memory space. The node "pointer" stored in NV-Tree is the memory offset to the start address of the mapped memory space. Therefore, even if the mapping is changed after reboot, each node can always be located using the offset. With the position information of the first LN stored in the reserved location, our NV-Tree can be practically recoverable after power down.

We build our NV-Store based on NV-Tree by allowing different sizes of key and value. Moreover, by adding a timestamp in each *LN_Element*, NV-Store can support lock-free concurrent accesses using timestamp-based multi-version concurrency control (MVCC) [50]. Based on that, we implement NV-Store to support Snapshot Isolation [51] transactions. Finally, we extended YCSB to support NV-Store to facilitate our performance evaluation.

6.1.2 Experimental Setup

All of our experiments are conducted on a Linux server (Kernel version 3.13.0-24) with an Intel Xeon E5-2650 2.4GHz CPU (512KB/2MB/20MB L1/L2/L3 cache), 8GB DRAM and 8GB NVDIMM [24] which has practically the same read/write latency as DRAM. In the end-to-end comparison, we use YCSB (0.1.4) to compare NV-Store with Redis (2.8.13). Note that all results shown in this section are produced by running application on NVDIMM server instead of simulation. The execution time measured for NV-Tree and NV-Store includes the rebuilding overhead.

6.2 Insertion Performance

We first compare the random-key insertion performance of LCB$^+$Tree, CDDS-Tree and NV-Tree with different node sizes. Figure 5 shows the execution time of inserting one million KV-pairs (8B/8B) with randomly selected keys to each tree with different sizes of tree nodes from 512B to 4KB. The result shows that NV-Tree outperforms LCB$^+$Tree and CDDS-Tree up to 7X and 12X with 4KB nodes, respectively. Moreover, different from LCB$^+$Tree and CDDS-Tree that the insertion performance drops when the node size increases, NV-Tree shows the best performance with larger nodes. This is because (1) NV-Tree adopts unsorted LN to avoid CPU cache line flush for shifting entries. The size of those cache line flush is proportional to the node size in LCB$^+$Tree and CDDS-Tree; (2) larger nodes lead to less LN split resulting in less rebuilding and reduced height of NV-Tree.

The performance improvement of NV-Tree over the competitors is mainly because of the reduced number of cache line flush thanks to both the unsorted LN and decoupling strategy of enforcing consistency selectively. Specifically, as shown in Figure 6, NV-Tree reduces the total CPU cache line flush by 80%-97% compared to LCB$^+$Tree and 76%-96% compared to CDDS-Tree.

Although the consistency cost of INs is neglectable for LCB$^+$Tree and CDDS-Tree as shown in Figure 1d, such cost becomes relatively expensive in NV-Tree because the

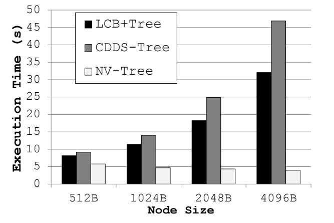


Fig. 5. Execution Time for 1 Million Insertions

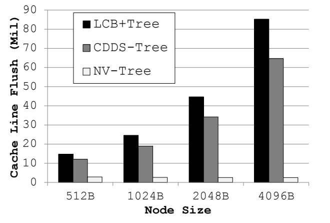


Fig. 6. Number of Cache Line Flushes for 1 Million Insertions

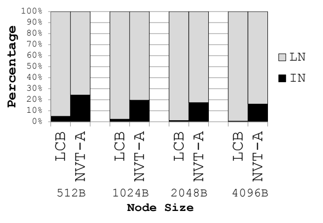


Fig. 7. Cache Line Flush Breakdown for IN/LN

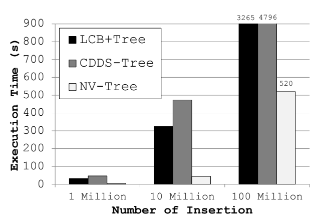


Fig. 8. Execution Time for 1/10/100 Million Insertions

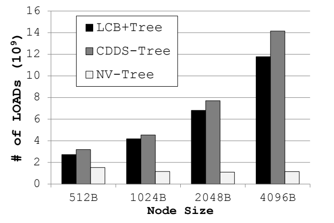


Fig. 9. Number of LOADs Executed for 1 Million Insertions

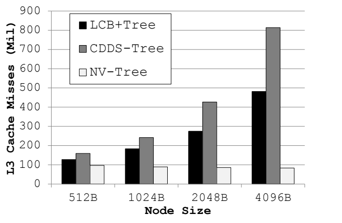


Fig. 10. Number of L3 Cache Misses for 1 Million Insertions

consistency cost for unsorted LNs is significantly reduced. To quantify the consistency cost of INs after such optimization, we implement a modified NV-Tree, denoted as NVT-A, which does the same optimization for LN as NV-Tree, but manages INs in the same way as LCB$^+$Tree and enforces consistency for all INs. Figure 7 shows the breakdown of CPU cache line flush for IN and LN in LCB$^+$Tree and NVT-A. The percentage of CPU cache line flush for IN increases from around 7% in LCB$^+$Tree to more than 20% in NVT-A. This result proves that decoupling IN/LN and enforcing consistency selectively are necessary and beneficial.

Figure 8 shows the execution time of inserting different number of KV-pairs with 4KB node size. The result shows that for inserting 1/10/100 million KV-pairs, the speedup of NV-Tree can be 7.1X/6.3X/5.3X over LCB$^+$Tree and 12X/9.7X/8.2X over CDDS-Tree. This suggests that although inserting more KV-pairs increases the number and duration of rebuilding, NV-Tree can still outperform the competitors thanks to the write-optimized design.

We further study the underlying CPU cache efficiency of the three trees by using vTune Amplifier. Figure 9 shows the total number of LOAD instructions executed for inserting one million KV-pairs in each tree. NV-Tree reduces the number of LOAD instruction by about 44%-90% and 52%-92% compared to LCB$^+$Tree and CDDS-Tree, respectively. We also notice the number of LOAD instructions is not sensitive to the node size in NV-Tree while it is proportional to the node size in LCB$^+$Tree and CDDS-Tree. This is because NV-Tree (1) eliminates entry shifting during insertion in unsorted LN, (2) adopts cache-optimized layout for IN/PLNs.

Most important, NV-Tree produces much less cache misses. Since memory read is only needed upon L3 cache miss, we use the number of L3 cache misses to quantify the read penalty of FLUSH. Figure 10 shows the total number of L3 cache misses when inserting one million KV-pairs. NV-Tree can reduce the number of L3 cache misses by 24%-83% and 39%-90% compared to LCB$^+$Tree and CDDS-Tree, respectively. This is because NV-Tree reduces the number of CPU cache line invalidation and flush.

6.3 Update/Deletion/Search Throughput

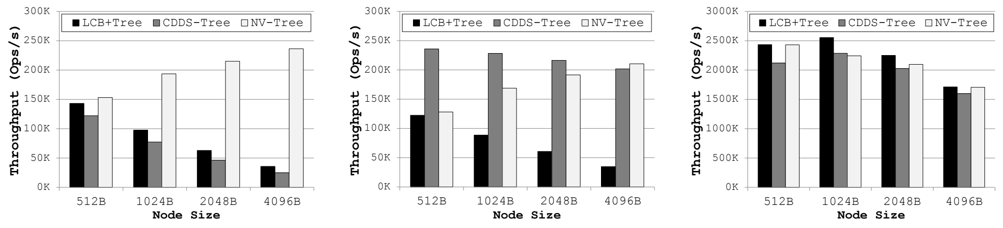
To compare the throughput of update/deletion/search operations in the three trees, we first insert one million KV-pairs, then update each of them with new value (same size), then search with every key and finally delete all of them. For each type of operation, each key is randomly and uniquely selected. After each type of operation, we flush the CPU cache to remove the cache influence between different types of operation.

The update/deletion/search performance with node size varied from 512B to 4KB is shown in Figure 11. As shown in Figure 11a, NV-Tree improves the throughput of update by up to 5.6X and 8.5X over LCB$^+$Tree and CDDS-Tree. In CDDS-Tree, although update does not trigger the split if any reusable slots are available, entry shifting is still needed to keep the entries sorted. LCB$^+$Tree does not need to shift entries for update, but in addition to the updated part of the node, it FLUSHes the log which contains the original copy of the node. In contrast, NV-Tree uses log-free append-only approach to modify LNs so that only two *LN_element*s need to be FLUSHed.

Upon deletion, NV-Tree is better than LCB$^+$Tree but not as good as CDDS-Tree as shown in Figure 11b. This is because CDDS-Tree simply does an in-place update to update the end version of a corresponding key. However, with the node size increased, NV-Tree is able to achieve comparable throughput to CDDS-Tree because of the reduction of split.

Note that the throughput of update and deletion in Figure 11a and 11b in LCB$^+$Tree decreases when the node size increases. This is because both the log size and the amount of data to FLUSH for shifting entries are proportional to the node size. The same trend is observed in CDDS-Tree. By contrast, the throughput of update and deletion in NV-Tree always increases when the node size increases because (1) the amount of data to FLUSH is irrelevant to the node size, (2) a larger node reduces the number of split and rebuilding.

Figure 11c shows the search throughput of the three trees. Although the search performance of NV-Tree is affected by the unsorted LN design, the results show that it can still outperform its competitors in some cases. This is because for the same number of KV-pairs, (1) with the same node size, NV-Tree is always shorter than CDDS-Tree and no taller than LCB$^+$Tree thanks to the removal of pointers in INs, (2) the total size of INs in NV-Tree is much smaller than that in its competitors so the INs visited during search in NV-Tree are more likely to be found in CPU caches. Overall, NV-Tree with the unsorted LN design (but keys in INs are still sorted) shows comparable search performance to its competitors with sorted LNs, which is consistent to the published result in [38].



(a) Update (b) Deletion (c) Search

Fig. 11. Update/Deletion/Search Throughput Comparison

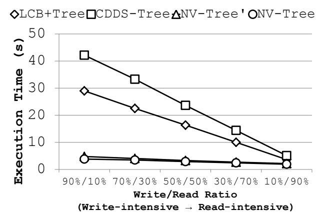


Fig. 12. Execution Time of Different R/W Ratios (Node Size = 4KB)

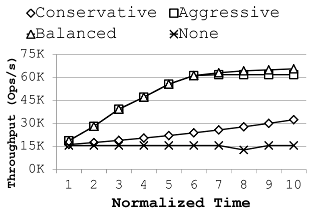


Fig. 13. Update Performance Over Time under Skewed Workloads

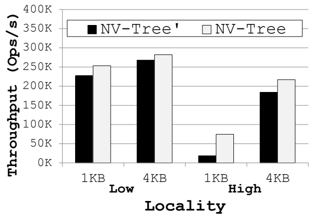


Fig. 14. Insertion Performance with And without Workload-adaptivity

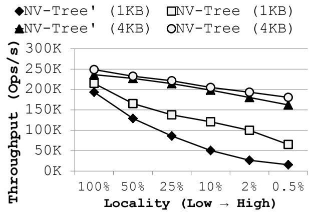


Fig. 15. Update Performance with And without Workload-adaptivity

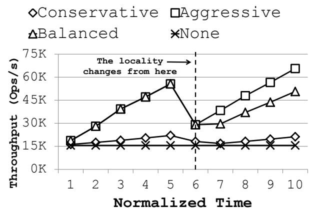


Fig. 16. Update Performance Over Time under Dynamic Workloads

6.4 Workload-adaptive Optimization

6.4.1 *Different Read/Write Ratios*

Firstly, we investigate the performance of NV-Tree under low-locality workloads with different r/w ratios. Figure 12 shows the execution time of performing one million random-key update/search operations with varied r/w ratios on an existing tree with one million KV-pairs. NV-Tree has the best performance under mixed workloads compared to LCB$^+$Tree and CDDS-Tree. All three trees have better performance under workloads with fewer update. This is because memory writes must be performed to write LN changes to NVM persistently through FLUSH while searches can be much faster if they hit the CPU cache. Moreover, NV-Tree shows the highest speedup, 6.6X over LCB$^+$Tree and 10X over CDDS-Tree, under the most write-intensive workload (90% insertion/10% search). As the write/read ratio decreases, the speedup of each tree drops but NV-Tree is still better than its counterparts under the most read-intensive workload (10% insertion/90% search). This indicates NV-Tree has much better insertion performance and comparable search throughput as well. Note that NV-Tree′ (NV-Tree without workload-adaptivity) has similar performance to NV-Tree because the rebuilding only happens once under such low-locality workloads.

6.4.2 *Skewed (High-locality) Write-intensive Workloads*

In addition to the low-locality workloads above, we also study the tree performance under skewed workloads. The workload we choose is the update-only workload where keys are selected within 5% of the tree with 1KB node size. Figure 13 shows the update throughput of NV-Tree with different workload-adaptive strategies under such a workload over time. Due to the excessive number of splits and rebuilding, without the workload-adaptive algorithm, the update throughput under the skewed workload drops 92% compared to that under low-locality workloads shown in Figure 11a, and the drop is consistent over time. Note that the slightly worse performance at time 8 is caused by a rebuilding. However, with the workload-adaptivity enabled, the performance can be improved over time under different strategy.

As shown in Figure 13, every strategy can help improve the performance over time but the balanced and aggressive one can do that much faster than the conservative one. This is because at the early stage when doubling the node size does not reach the size limitation set by the analytical model, the performance can be improved much faster than slowly increasing the node size by one cache line size. Moreover, after a certain point of time, the performance remains unchanged for the aggressive strategy while it still gets better but slowly for the balanced strategy. This is because in our current implementation, if doubling the size of a node is not allowed due to the limitation calculated by the analytical model, the node size remains unchanged in the aggressive strategy. However in the balanced strategy, the node size will be increased by only one cache line size as long as it does not exceed the limitation which can improve the performance just like the conservative strategy does.

To quantify the effectiveness of the workload-adaptive scheme, we compare the throughput of NV-Tree (workload-adaptive with balanced strategy) and NV-Tree′ (NV-Tree without the workload-adaptive scheme) under insertion and update workloads with different localities. As shown in Figure 14, under high-locality insertion workloads, the throughput can be improved up to 305% with workload-adaptive algorithm for 1KB node. Figure 15 shows the update throughput with different localities, which are represented by the percentages of how many keys over the total number of keys are selected for the update operations. 100% represents the key selection is uniformly distributed which is the lowest locality. Under the high-locality workload (0.5%), the update throughput can be improved up to 3.2X with the workload-adaptive scheme. Moreover, although the update throughput drops when the locality increases due to the more frequent rebuilding, the throughput of NV-Tree only drops 70%/27% while that of NV-Tree′ drops 92%/31% with 1KB/4KB node sizes when the locality changes from 100% to 0.5%. This indicates the workload-adaptive algorithm can effectively reduce the performance drop when the locality increases.

6.4.3 Dynamic Workloads

The workload-adaptive strategy can also detect the change of the on-going workload. To validate this ability, we use the settings similar to those in the previous subsection except that we change the locality of the workload during the execution. There are numerous ways to change the workload in terms of the locality and read/write ratio, but for illustration, we only show how the performance changes when the hot-zone shifts under the update-only workload. Specifically, the workload starts with updating the entries with the large keys (right-most 0.5%), then shifts to updating the entries with the small keys (left-most 0.5%).

As shown in Figure 16, the update throughput of NV-Tree without workload-adaptive strategy remains unchanged over time. In contrast, when the workload shifts at the time between 5 and 6, the performance stops being improved and drops immediately for all three strategies. As the conservative strategy only enlarges the hot-write nodes, the LNs in the new hot zone may not be identified as hot-write immediately after the workload shifts, the performance keeps drop at time 7 and starts to be improved from time 8 eventually. However, such a phenomenon that the performance keeps drop is not observed in the aggressive and balanced strategy. Starting from time 7, the performance immediately starts to be improved under all three strategies. Furthermore, as the aggressive strategy increases the size of write-intensive nodes, it is more responsive to the workload shifts than the balanced strategy. Therefore, the performance improvement of the aggressive strategy right after the workload shifts is much better than that of the balanced strategy. Nevertheless, the performance improvement gets better at time 8 for the balanced strategy because the LNs in the new hot zone start to be identified as the hot-write nodes.

6.5 Rebuilding and Failure Recovery

To quantify the impact of rebuilding on the overall performance, we measure the total number and time of rebuilding with different node sizes under different number of insertions. As shown in Table 4, compared to the total execution time, the percentage of rebuilding time in the total execution time is only about 1% for 1/10/100 million random-key insertion workloads, which is totally neglectable.

Due to the excessive rebuilding under the sequential insertion workload, the rebuilding time is increased. To quantify the rebuilding optimization of the workload-adaptive scheme, we compare the rebuilding time of the NV-Tree with and without workload-adaptive algorithm (denoted as NV-Tree') for 1 million sequential insertions with 1KB node size as shown in Figure 17. With the workload-adaptive optimization on rebuilding, the rebuilding time can be significantly reduced (from 44.5s to 1.4s). The slightly longer rebuilding time under the random insertion workload is due to the extra time for identifying hot-write PLNs.

Different from the *rebuild-from-PLN* during normal execution, the failure recovery performs a *rebuild-from-LN*. Our previous study [46] shows that a *rebuild-from-LN* is 22%-47% slower than a *rebuild-from-PLN*. Since we focus on the tree performance where the off-line recovery time is not crucial, we leave the optimization for recovery as future work.

TABLE 4
Rebuilding Time for 1/10/100 Million Insertions with 1KB/4KB Nodes

	1M		10M		100M	
	Node Size		Node Size		Node Size	
	1KB	4KB	1KB	4KB	1KB	4KB
# of Rebuilding	3	1	3	2	4	2
Rebuilding Time (*ms*)	52.6	0.6	57.1	41.7	1359.4	41.6
Execution Time (*ms*)	4672.0	3955.2	55998.5	44091.5	604111.3	518323.9
Percentage	**1.13%**	**0.02%**	**0.10%**	**0.09%**	**0.23%**	**0.01%**

To validate the recoverability and consistency, when NV-Tree is running a 100M insertion workload, we manually trigger the system failure by either (1) killing the process or cut the power at randomly selected timing, or (2) terminating the program at a certain point of time such as the LN modification (between the steps shown in Figure 3), the LN split (between the steps shown in Figure 4) and rebuilding. After NV-Tree is recovered from such injected failure, no data inconsistency such as invalid pointers, pointers to invalid data or data loss is found.

6.6 End-to-End Performance

In this subsection, we compare our NV-Store with Redis under YCSB workloads. NV-Store is built on top of NV-Tree which is practically durable and consistent. NV-Store' represents the KV-Store based on NV-Tree' without workload-adaptive algorithm. Redis can provide persistency by using `fsync` to write logs to an append-only file (AOF mode). With different `fsync` strategy, Redis can be either volatile if `fsync` is performed in a time interval, or consistent if `fsync` is performed right after each log write. We use the NVM space to allocate a RAMDisk for holding the log file so that Redis can be in-memory persistent. Note that it still goes through the POSIX interface by using `fsync`.

We select two typical workloads in YCSB benchmark, StatusUpdate (read-latest) and SessionStore (update-heavy), to evaluate NV-Store and Redis. StatusUpdate workload has 95%/5% search/insert ratio on keys chosen from a temporally weighted distribution to represent applications in which people update the online status while others view the latest status, which means newly inserted keys are preferentially chosen for retrieval. SessionStore workload has 50%/50% search/update ratio on keys chosen from a Zipfian distribution to represent applications in which people record recent actions.

Figure 18 shows the throughput of NV-Store and Redis under StatusUpdate workload. The result shows that NV-Store improve the throughput by up to 4.7X over both volatile and consistent Redis. This indicates the optimization of reducing cache line flush for insertion can significantly improve the performance even with as low as 5% insertion percentage. Moreover, both volatile and consistent Redis are bottlenecked with about 16 clients while NV-Store can still scale up with 32 clients. The high scalability of NV-Store is achieved by the high concurrency of the tree operations. Figure 19 shows the throughput under SessionStore workload. NV-Store can improve the throughput by up to 7.3X over Redis because the workload is more write-intensive. Note that NV-Store outperforms NV-Store' [46] by up to 9% and 30% under these two workloads, respectively. This indicates our workload-adaptive scheme can signifi-

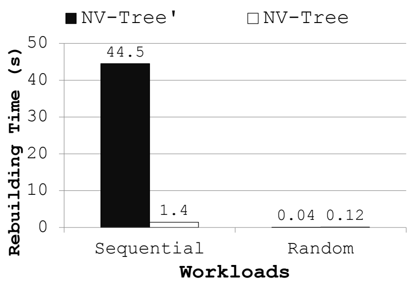


Fig. 17. Rebuilding Time Comparison

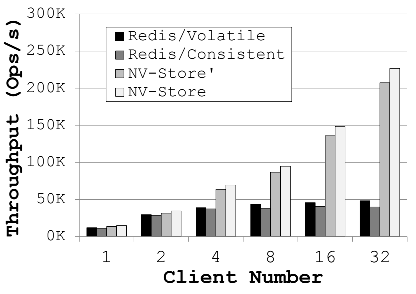


Fig. 18. Throughput of YCSB-StatusUpdate

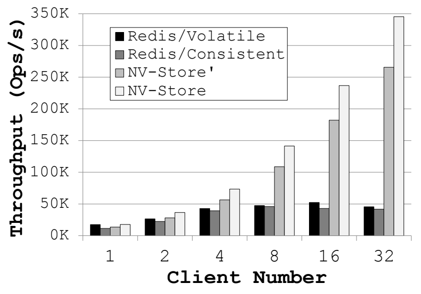
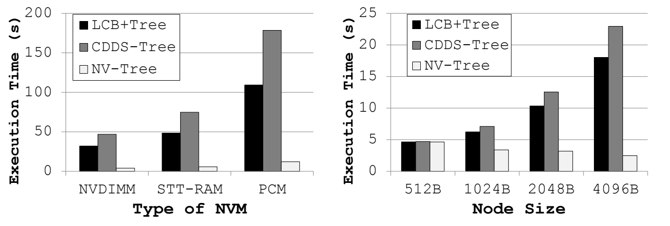


Fig. 19. Throughput of YCSB-SessionStore



(a) Different Types of NVM (b) Hardware with Epoch
Fig. 20. NV-Tree Performance with Different NVM and New Hardware

cantly improve the overall performance under such skewed (Zipfian distribution) write-intensive workloads.

6.7 NV-Tree Performance With Different Hardware

6.7.1 Different Types of NVM

Given the write latency difference of NVDIMM (same as DRAM), PCM (180ns), STT-RAM (50ns) in Table 1, we explicitly add some delay before every memory write in our NV-Tree to investigate its performance on different types of NVM. Figure 20a shows the execution time of one million insertions in NV-Tree with 4KB nodes. Compared to the performance on NVDIMM, NV-Tree is only 5%/206% slower on STT-RAM/PCM, but LCB$^+$Tree is 51%/241% and CDDS-Tree is 87%/281% slower. NV-Tree suffers less performance drop than LCB$^+$Tree and CDDS-Tree on slower NVM because of the reduction of CPU cache line flush.

6.7.2 Future Hardware: New CPU Instructions

Comparing to `MFENCE` and `CLFLUSH`, *epoch* and a couple of new instructions for non-volatile storage (`CLWB/CLFLUSHOPT/PCOMMIT`) added by Intel recently [37] are able to flush CPU cache lines without explicit invalidations which means additional cache misses can be avoided. We estimate LCB$^+$Tree, CDDS-Tree and our NV-Tree performance by removing the cost of L3 cache misses due to cache line flushes the execution time (Figure 5). For B$^+$Tree and volatile CDDS-Tree, such cost can be derived by deducting the number of L3 cache misses without cache line flushes (Figure 1b) from that with cache-line flushes (Figure 10). As shown in Figure 20b, with the cache miss penalty removed, NV-Tree outperforms LCB$^+$Tree/CDDS-Tree by 7X/9X with 4KB nodes. This indicates our optimization of reducing cache line flush is still valuable when flushing a cache line without the invalidation becomes possible.

7 Conclusion

In this paper, we quantify the consistency cost of existing tree structures in NVM. Based on our observations, we propose NV-Tree, a consistent B$^+$Tree variant, which can keep in-NVM data consistent with significantly reduced consistency cost. By selectively enforcing consistency, adopting unsorted LNs and organizing INs in a cache-optimized format, NV-Tree can reduce the number of cache line flushes under write-intensive workloads by more than 90% compared to the state-of-art consistent trees. To further optimize NV-Tree for skewed workloads, we propose a workload-adaptive scheme to improve the performance over time by dynamically adjust the individual node size. We also use NV-Tree as the core data structure to build a key-value store named NV-Store. Both NV-Tree and NV-Store are implemented and evaluated on a real NVDIMM platform instead of simulation. The experimental results show that NV-Tree outperforms LCB$^+$Tree and CDDS-Tree by up to 7X and 12X under write-intensive workloads, respectively. Our NV-Store increases the throughput by up to 7.3X under YCSB workloads compared to Redis. The workload-adaptive enhancement can improve (1) NV-Tree by up to 3X under skewed write-intensive workloads; (2) NV-Store by 30% under YCSB workloads.

Acknowledgments

The authors are grateful to the anonymous reviewers for their constructive comments. This work was supported by Agency for Science, Technology and Research (A*STAR), Singapore under Grant No. 112-172-0010. Bingsheng's work is in part supported by a MoE AcRF Tier 1 (2014-T1-001-145) of Singapore.

References

[1] S. Raoux, G. W. Burr *et al.*, "Phase-change random access memory: A scalable technology," *IBM JRD*, vol. 52, pp. 465–479, 2008.
[2] T. Kawahara, "Scalable spin-transfer torque ram technology for normally-off computing," *DTC*, vol. 28, no. 1, pp. 0052–63, 2011.
[3] L. Chua, "Resistance switching memories are memristors," *Applied Physics A*, vol. 102, no. 4, pp. 765–783, 2011.
[4] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM JRD*, vol. 52, pp. 439–447, 2008.
[5] S. Venkataraman, N. Tolia *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory." in *FAST'11*.
[6] S. Pelley, P. M. Chen *et al.*, "Memory persistency," in *ISCA*, 2014.
[7] B. Cully, J. Wires *et al.*, "Strata: High-performance scalable storage on virtualized non-volatile memory," in *FAST*, 2014.
[8] H. Kim, S. Seshadri *et al.*, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," in *FAST*, 2014.
[9] D. Vučinić, Q. Wang *et al.*, "Dc express: Shortest latency protocol for reading phase change memory over pci express," in *FAST'14*.
[10] A. M. Caulfield, T. I. Mollov *et al.*, "Providing safe, user space access to fast, solid state disks," *SIGPLAN*, vol. 47, 2012.
[11] A. M. Caulfield, A. De *et al.*, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *MICRO*, 2010.

[12] G. Dhiman, R. Ayoub *et al.*, "Pdram: a hybrid pram and dram main memory system," in *DAC*, 2009.
[13] D. Fryer, K. Sun *et al.*, "Recon: Verifying file system consistency at runtime," in *FAST*, 2012.
[14] T. S. Pillai, V. Chidambaram *et al.*, "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications," in *OSDI*, 2014.
[15] V. Chidambaram, T. S. Pillai *et al.*, "Optimistic Crash Consistency," in *SOSP*, 2013.
[16] W.-H. Kim, B. Nam *et al.*, "Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split," in *FAST*, 2014.
[17] J. DeBrabant, J. Arulraj *et al.*, "A prolegomenon on oltp database systems for non-volatile memory," *PVLDB*, vol. 7, no. 14, 2014.
[18] K. Bhandari, D. R. Chakrabarti *et al.*, "Implications of cpu caching on byte-addressable non-volatile memory programming," HP Technical Report HPL-2012-236, Tech. Rep., 2012.
[19] J. Condit, E. B. Nightingale *et al.*, "Better i/o through byte-addressable, persistent memory," in *SIGOPS*, 2009.
[20] D. Narayanan and O. Hodson, "Whole-system persistence," in *ASPLOS*, 2012.
[21] D. Comer, "Ubiquitous b-tree," *CSUR*, vol. 11, pp. 121–137, 1979.
[22] J. Ahn, D. Kang *et al.*, "μ*-tree: An ordered index structure for NAND flash memory with adaptive page layout scheme," *IEEE Trans. Computers*, vol. 62, no. 4, pp. 784–797, 2013.
[23] H. Fang, M. Yeh *et al.*, "An adaptive endurance-aware b$^+$-tree for flash memory storage systems," *IEEE Trans. Computers*, vol. 63, no. 11, pp. 2661–2673, 2014.
[24] VikingTechnology, "Arxcis-nv (tm) non-volatile dimm," *http://www.vikingtechnology.com/arxcis-nv*, 2014.
[25] S. Sanfilippo and P. Noordhuis, "Redis," *http://redis.io*, 2009.
[26] H. Volos, A. J. Tack *et al.*, "Mnemosyne: Lightweight persistent memory," in *SIGARCH*, vol. 39, 2011, pp. 91–104.
[27] C. Chang, C. Yang *et al.*, "Booting time minimization for real-time embedded systems with non-volatile memory," *IEEE Trans. Computers*, vol. 63, no. 4, pp. 847–859, 2014.
[28] C. Zhang, Y. Wang *et al.*, "Deterministic crash recovery for NAND flash based storage systems," in *DAC*, 2014.
[29] Y. Li, B. He *et al.*, "Tree indexing on solid state drives," *PVLDB*, vol. 3, no. 1-2, pp. 1195–1206, 2010.
[30] Y. Lv, B. Cui *et al.*, "Operation-aware buffer management in flash-based systems," in *SIGMOD*, 2011.
[31] M. Jung, E. H. Wilson III *et al.*, "Exploring the future of out-of-core computing with compute-local non-volatile memory," in *SC*, 2013.
[32] Q. Wu, F. Sun *et al.*, "Using multilevel phase change memory to build data storage: A time-aware system design perspective," *IEEE Trans. Computers*, vol. 62, no. 10, pp. 2083–2095, 2013.
[33] M. K. Qureshi, V. Srinivasan *et al.*, "Scalable high performance main memory system using phase-change memory technology," *SIGARCH*, vol. 37, no. 3, pp. 24–33, 2009.
[34] P. Zhou, B. Zhao *et al.*, "A durable and energy efficient main memory using phase change memory technology," in *SIGARCH*, vol. 37, 2009, pp. 14–23.
[35] Everspin, "Second generation mram: Spin torque technology," *http://www.everspin.com/products/second-generation-st-mram.html*.
[36] X. Wu and A. Reddy, "Scmfs: a file system for storage class memory," in *SC*, 2011.
[37] Intel, "Intel 64 and ia-32 architectures software developers manual," *Volume 3A: System Programming Guide, Part*, 2014.
[38] S. Chen, P. B. Gibbons *et al.*, "Rethinking database algorithms for phase change memory." in *CIDR*, 2011.
[39] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *PVLDB*, vol. 7, no. 10, pp. 865–876, 2014.
[40] J. Coburn, A. M. Caulfield *et al.*, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS*, 2011.
[41] E. Lee, H. Bahn *et al.*, "Unioning of the buffer cache and journaling layers with non-volatile memory." in *FAST*, 2013.
[42] R.-S. Liu, D.-Y. Shen *et al.*, "Nvm duet: unified working memory and persistent store architecture," in *ASPLOS*, 2014.
[43] H. Volos, S. Nalli *et al.*, "Aerie: flexible file-system interfaces to storage-class memory," in *EuroSys*, 2014.
[44] S. R. Dulloor, S. Kumar *et al.*, "System software for persistent memory," in *EuroSys*, 2014.
[45] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *PVLDB*, vol. 8, no. 7, pp. 786–797, Feb. 2015.
[46] J. Yang, Q. Wei *et al.*, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *FAST*, 2015.
[47] J. Rao and K. A. Ross, "Making b+-trees cache conscious in main memory," in *ACM SIGMOD Record*, vol. 29, 2000, pp. 475–486.
[48] B. F. Cooper, A. Silberstein *et al.*, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.
[49] T. Bingmann, "Stx b+ tree c++ template classes," 2008.
[50] D. Lomet and B. Salzberg, *Access methods for multiversion data*, 1989, vol. 18.
[51] H. Berenson, P. Bernstein *et al.*, "A critique of ansi sql isolation levels," in *ACM SIGMOD Record*, vol. 24, 1995, pp. 1–10.

Jun Yang received the bachelor degree in computer science from Shanghai Jiaotong University (2003-2007), and the PhD degree in computer science and engineering in Hong Kong University of Science and Technology (2007-2013). Currently he is a Scientist in the Data Storage Institute, A*Star, Singapore. His research interests include database systems, file systems and next-generation non-volatile memory.

Qingsong Wei received his Ph.D. in computer science from University of Electronic Science and Technologies of China in 2004. He was with Tongji University as assistant professor from 2004 to 2005. He is currently with the Data Storage Institute, Agency for Science, Technology and Research (A*STAR), Singapore, as a Research Scientist. His research interests include Non-volatile Memory Technologies, Distributed File and Storage System, Cloud Storage and Operating System. Dr. Wei is a member of IEEE.

Chundong Wang received the bachelor degree in computer science from Xi'an Jiaotong University (2004-2008), and the PhD degree in computer science of National University of Singapore (2008-2013). Currently he is a Scientist in the Data Storage Institute, A*Star, Singapore. His research interests include file system, data storage and next-generation non-volatile memory.

Cheng Chen received the bachelor degree in computer science from Chengdu University of Information Technology (2004-2008), and the Master degree in computer science in University of Electronic Science and Technology of China (2009-2012). He is a research engineer in Data Storage Institute, A*star, Singapore. His research interests are Non-volatile Memory based operating system, high performance computing, distributed and parallel systems, and database systems.

Khai Leong Yong currently works as a Divisional Manager for Data Storage Institute (DSI), a research institute under the Agency for Science, Technology and Research (A*STAR), Singapore. In his role with DSI, Khai Leong leads a team of research scientists and engineers in developing data systems and storage technologies for next generation data centers. Khai Leong has more than 10 years of research and industry experience in network storage systems, networking and software designs with many of these years in leading positions. Khai Leong received his Engineering degree from the National University of Singapore and holds a postgraduate degree in Communication Software and Networks.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an associate professor in the School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.