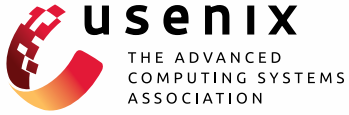



# NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems

**Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai Leong Yong,**
***Data Storage Institute, A-STAR;* Bingsheng He, *Nanyang Technological University***

https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

**This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).**

**February 16–19, 2015 • Santa Clara, CA, USA**

ISBN 978-1-931971-201

**Open access to the Proceedings of the 13th USENIX Conference on File and Storage Technologies is sponsored by USENIX**

# NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems

Jun Yang[1], Qingsong Wei[§ 1], Cheng Chen[1], Chundong Wang[1], Khai Leong Yong[1] and Bingsheng He[2]

[1]Data Storage Institute, A-STAR, Singapore
[2]Nanyang Technological University

## Abstract

The non-volatile memory (NVM) has DRAM-like performance and disk-like persistency which make it possible to replace both disk and DRAM to build single level systems. To keep data consistency in such systems is non-trivial because memory writes may be reordered by CPU and memory controller. In this paper, we study the consistency cost for an important and common data structure, B$^+$Tree. Although the memory fence and CPU cacheline flush instructions can order memory writes to achieve data consistency, they introduce a significant overhead (more than 10X slower in performance). Based on our quantitative analysis of consistency cost, we propose NV-Tree, a consistent and cache-optimized B$^+$Tree variant with reduced CPU cacheline flush. 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 4.8X under YCSB workloads compared to Redis.

## 1 Introduction

For the past few decades, DRAM has been de facto building block for the main memory of computer systems. However, it is becoming insufficient with an increasing need of large main memory due to its density limitation [40, 43]. To address this issue, several Non-Volatile Memory (NVM) technologies have been under active development, such as phase-change memory (PCM) [49], and spin-transfer torque memory (STT-RAM) [29]. These new types of memory have the potential to provide comparable performance and much higher capacity than DRAM. More important, they are persistent which makes failure recovery faster [31, 33].

Considering the projected cost [21] 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 [21, 53, 45]. Such systems can (i) 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 [16, 30, 56, 7, 6]. However, with data stored only in NVM, data structures and algorithms must be carefully designed to avoid any inconsistency caused by system failure. In particular, if the system crashes when an update is being made to a data structure in NVM, the data structure may be left in a corrupted state as the update is only half-done. In that case, we need certain mechanism to recover the data structure to its last consistent state. To achieve data consistency in NVM, ordered memory writes is fundamental. However, existing CPU and memory controller may reorder memory writes which makes it non-trivial to develop consistent NVM-based systems and data structures, as demonstrated in previous works [44, 53, 58, 55, 14, 12, 18, 35, 22, 46, 10, 32, 17]. To maintain memory writes to NVM in certain order, we must (1) prevent them from being reordered by CPU and (2) manually control CPU cacheline flush to make them persistent on NVM. Most studies use CPU instructions such as memory fence and cacheline flush. However, these operations introduce significant overhead [14, 53, 44]. We observe a huge amplification of CPU cacheline flush when using existing approaches to keep B$^+$Tree [13] consistent, which makes the consistency cost very high.

In this paper, we propose **NV-Tree**, a consistent and cache-optimized $B^+$Tree variant which reduces CPU cacheline 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 cacheline flush

§ Corresponding author: WEI_Qingsong@dsi.a-star.edu.sg

by 82% to 96% for keeping LN consistent. Last but not least, to overcome the drawback of slowing down searches and deletions due to the write-optimized design in LN, NV-Tree adopts a pointer-less layout for INs to further increase the CPU cache efficiency.

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 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 our **NV-Tree**, which (1) decouples LNs and INs, only enforces consistency on LNs; (2) keeps entries in LN unsorted, updates LN consistently without logging or versioning; (3) organizes INs in a cache-optimized format to further increase CPU cache efficiency.
3. To evaluate NV-Tree in system level, we have also implemented a key-value store, called **NV-Store**, using NV-Tree as the core data structure.
4. Both NV-Tree and NV-Store are implemented and evaluated on a real NVDIMM [1] platform. The experimental results show that NV-Tree outperforms CDDS-Tree [53], the state-of-art consistent tree structure, by up to **12X** under write-intensive workloads. The speedup drops but still reaches 2X under read-intensive workloads. NV-Store increases the throughput by up to **4.8X** under YCSB workloads compared to Redis [50].

The rest of this paper is organized as follows. Section 2 discusses the background, related work and motivation. Section 3 presents the detailed design and implementation of NV-Tree. The experimental evaluation of NV-Tree and NV-Store is shown in Section 4. Finally, Section 5 concludes this paper.

# 2 Related Work and Motivation

## 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 [21, 55]. Early work [36, 41, 35, 57, 47, 51, 11, 52, 59, 60, 2, 27, 39] focuses on flash memory. As shown in Table 1, flash is faster than HDD but is still unsuitable to replace DRAM due to much higher latency and limited endurance [24]. Recent work has focused on the next generation NVM [28], such as PCM [49, 42, 4, 8, 23] and STT-RAM [29], which (i) is byte addressable, (ii) has DRAM-like performance, and (iii) 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 [48, 61]. 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 [20]. In this paper, **NVM** is referred to the next generation of non-volatile memory excluding flash memory.

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 | $\infty$ |
| 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 | $10^5$ |
| HDD | 3,000,000 | 3,000,000 | $\infty$ |

Due to the price and prematurity of NVM, mass production with large capacity is still impractical today. As an alternative, NVDIMM [44], which is commercially available [1], 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 power failure, 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 [21, 14, 53, 58, 44] 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) [25] which is addressed in previous work [55, 14, 44], 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. To implement these approaches, we must make sure memory writes are in a certain order, e.g., the memory writes for making the copy of data must be completed before updating the data itself. Similar write ordering requirement also exists in pointer-based data structures, e.g., in $B^+$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 wrong write order will make the parent node contain an invalid pointer if the system crash right after the pointer being added.

Unfortunately, memory writes may be reordered by either CPU or memory controller. Alternatively, without modifying existing hardware, we can use the sequence of {`MFENCE`, `CLFLUSH`, `MFENCE`} instruction (referred to `flush` in the rest of this paper) to form ordered memory writes [53]. 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 cachelines so that they can be flushed to NVM by CPU which makes the memory write persistent eventually. However, `CLFLUSH` can only flush a dirty cacheline by explicitly invalidating it which makes CPU cache very inefficient. Although such invalidations can be avoided if we can modify the hardware itself to implement *epoch* [14], CPU cacheline flush cannot be avoided. 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

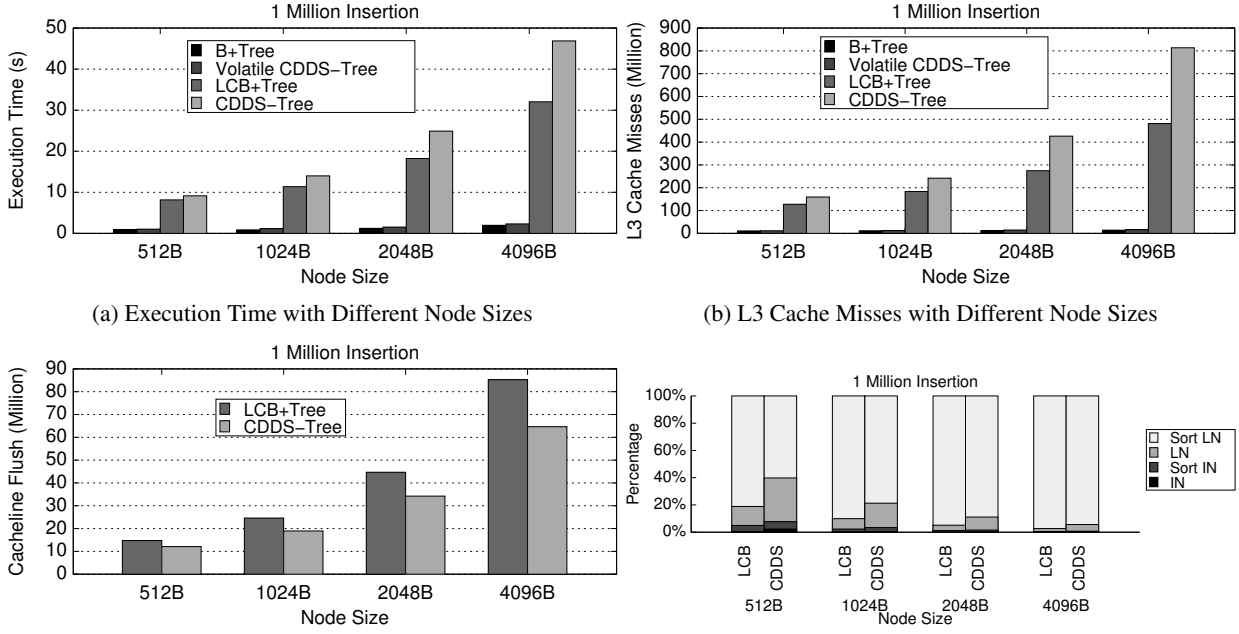
Recent work proposed mechanisms to provide data consistency in NVM-based systems by either modifying existing hardware or using CPU primitive instructions such as `MFENCE` and `CLFLUSH`. BPFS [14] proposed a new file system which is resided in NVM. It adopts a copy-on-write approach called short-circuit shadow paging using *epoch* which can flush dirty CPU cachelines without invalidating them to order memory writes for keeping data consistency. However, it still suffers from the overhead of cacheline flush. It must be implemented by modifying existing hardware which is not practical in most cases. Volos et al. [55] proposed Mnemosyne, a new program interface for memory allocations in NVM. To manage memory consistency, it presents persist memory region, persist primitives and durable memory transaction which consist of `MFENCE` and `CLFLUSH` eventually. NV-Heaps [12] 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. [44] proposed a way to keep the whole system status when power failure happens. Realizing the significant overhead of flushing CPU cacheline 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 cacheline is necessary to order memory writes and used in almost all the existing NVM-based systems [34, 37, 54, 19].

The most related work to our NV-Tree is CDDS-Tree [53] 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.

## 2.4 Motivation

To quantify the consistency cost, we compare the execution of performing one million insertion in (a) a standard B$^+$Tree [13] without consistency guarantee, (b) a log-based consistent B$^+$Tree (**LCB$^+$Tree**), (c) a CDDS-Tree [53] using versioning, and (d) a volatile CDDS-Tree with `flush` disabled. In LCB$^+$Tree, before modifying a node, its original copy is logged and `flush`ed. The modified part of it is then `flush`ed 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 [53]. 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 the increased number of cache misses and additional cacheline flush.

Remembering that `CLFLUSH` flushes a dirty cacheline by explicitly invalidating it, which causes a cache miss




(a) Execution Time with Different Node Sizes

(b) L3 Cache Misses with Different Node Sizes

(c) Total Number of CPU Cacheline Flush

(d) Percentage Breakdown of CPU Cacheline Flush

Figure 1: Consistency Cost Analysis of B$^+$Tree and CDDS-Tree

when reading the same memory address later. 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 cacheline flushes in CDDS-Tree and LCB$^+$Tree for one million insertion. With 0.5KB/1KB/2KB/4KB nodes, the total amount of cacheline 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. This indicates that **keeping consistency causes a huge amplification of the CPU cacheline invalidation and flush, which increases the cache misses significantly**, as shown in Figure 1b.

The numbers of both the cache misses and cacheline 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 `flush`ed to make the insertion persistent. As a result, the amount of data to be `flush`ed is related to the node size for both trees.

We further categorize the CPU cacheline flush into four types, as shown in Figure 1d, *Sort LN*/*Sort IN* stands for the cacheline 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 `flush`ing new nodes and updated pointers after split, etc. The result shows that **the consistency cost due to `flush` mostly comes from `flush`ing shifted entries in order to keep LN sorted**, about 60%-94% in CDDS-Tree, and 81%-97% in LCB$^+$Tree.

Note that CDDS-Tree is slower than LCB$^+$Tree by 11-32% even though it produces less cacheline flush. The reasons are that (1) the size of each `flush` 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 [53].

Last but not least, we observe that given a data structure, **not all the data needs to be consistent to keep the entire data structure consistent.** As long as some parts of it (denoted as *critical data*) is consistent, the rest (denoted as *reconstructable data*) can be reconstructed without losing consistency for the whole data structure. For instance, in $B^+$Tree, where all the data is stored in LNs, they can be considered as *critical data* while INs are *reconstructable data* because they can always be reconstructed from LNs at a reasonably low cost. That suggests we may only need to enforce consistency on *critical data*, and reconstruct the entire data structure from the consistent *critical data* during the recovery.

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

## 3 NV-Tree Design and Implementation

In this section, we present NV-Tree, a consistent and cache-optimized B$^+$Tree variant with reduced consistency cost.

### 3.1 Design Decisions

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

$\mathcal{D}1$. ***Selectively Enforce Data Consistency.*** NV-Tree decouples LNs and INs by treating them as *critical data* and *reconstructable data*, respectively. Different from the traditional design where all nodes are updated with consistency guaranteed, NV-Tree only enforces consistency on LNs (*critical data*) but processes INs (*reconstructable data*) with no consistency guaranteed to reduce the consistency cost. Upon system failure, INs are reconstructed from the consistent LNs so that the whole NV-Tree is always consistent.

$\mathcal{D}2$. ***Keep Entries in LN Unsorted.*** NV-Tree uses 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 [9], we are the first one that quantify its impact on the consistency cost and propose to 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 designed to be visible only 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 LN is also increased because searching in LN is no longer blocked by the concurrent on-going update.

$\mathcal{D}3$. ***Organizing IN in Cache-optimized Format.*** The CPU cache efficiency is a key factor to the performance. In NV-Tree, all INs are stored in a consecutive memory space and located by offset instead of pointers, and all nodes are aligned to CPU cacheline. As a result, NV-Tree achieves higher space utilization and cache hit rate.

### 3.2 NV-Tree

In this subsection, we present the details of tree node layout design and all the tree operations of NV-Tree.

#### 3.2.1 Overview

In NV-Tree, as shown in Figure 2, all the data is stored in LNs which are linked together with right-sibling pointers. Each LN can also be accessed by the LN pointer stored in the last level of IN, denoted as PLN (parent of leaf node). All the IN/PLNs are stored in a pre-allocated consecutive 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*.

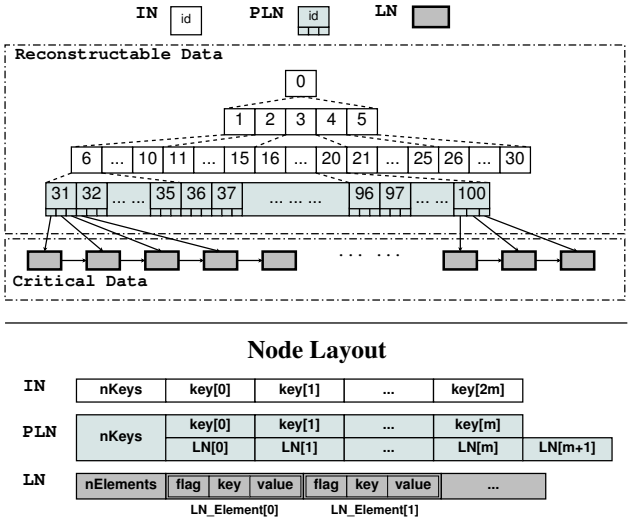



Figure 2: NV-Tree Overview and Node Layout

Keeping each LN and the LN list consistent in NV-Tree without using logging or versioning is non-trivial. Different from a normal B$^+$Tree, both update and deletion are implemented as insertion using an append-only strategy discussed in Section 3.2.3. Any insertion/update/deletion operations may lead to a full LN which triggers either *split*/*replace*/*merge* discussed in Section 3.2.4. We carefully design the write order for insertion (update/deletion) and split/replace/merge using `flush` to guarantee the changes made by these operations 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/PLN to accommodate more LNs, discussed in Section 3.2.5.

#### 3.2.2 Locating Target LN

We first present how to find the target LN in NV-Tree. Due to the hybrid design, the procedure of locating target LN with a given key in NV-Tree is different from that in 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. On each level, which child to go in the next level is determined by a binary search based on the given search key. For instance, with

**Algorithm 1:** NV-Tree LN Lookup

```
1  Function find_leaf(k, r)
       Input: k: key, r: root
       Output: LNpointer: the pointer of target leaf
               node
       /* Start from root (id=0).      */
2      id ← 0;
3      while id ∉ PLNIDs do /* Find PLN.    */
4          IN ← memory address of node id;
5          pos ← BinarySearch(key, IN);
6          id ← id * (2m + 1) + 1 + pos;
           /* m is the maximum number of
              keys in a PLN.             */
7      PLN ← memory address of node id;
8      pos ← BinarySearch(key, PLN);
9      return PLN.LNpointers[pos]
```

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

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 increases the scalability of NV-Tree.

### 3.2.3 Insertion, Update, Deletion and Search

**Insertion** starts with finding target LN. 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. Otherwise, the target LN will split before insertion (discussed in Section 3.2.4). The pseudo-code of insertion is shown in Algorithm 2. 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 shows an example of deleting the `{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 is found. The space of both the NEGATIVE and normal *LN_element*s are recycled by later split.

**Algorithm 2:** NV-Tree Insertion

```
   Input: k: key, v: value, r: root
   Output: SUCCESS/FAILURE
1  begin
2      if r = NULL then /* Create new tree
       with the given KV-pair.          */
3          r ← create_new_tree(k, v);
4          return SUCCESS
5      leaf ← find_leaf(k, r);
6      if LN has space for new KV-pair then
7          newElement ← CreateElement(k, v);
8          flush(newElement);
9          AtomicInc(leaf.number);
10         flush(leaf.number);
11     else
12         leaf_split_and_insert(leaf, k, v)
13     return SUCCESS
```

**Update** is implemented by inserting two *LN_element*s, a NEGATIVE 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.

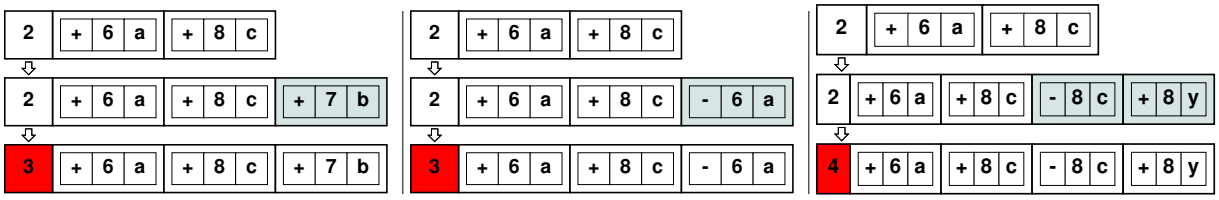
Note that the order of appending *LN_element* before updating *nElement* in LN is guaranteed by `flush`. The appended *LN_element* is only visible after the *nElement* is increased by a successful atomic write to make sure LN cannot be corrupted by system failure.

**Search** a key 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 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 4.4.

All the modification made to LNs/PLNs is protected by light-weight latches. Meanwhile, given the nature of the append-only strategy, searching in LNs/PLNs can be executed without being blocked by any ongoing modification as long as the *nElement* is used as the boundary of the search range in LNs/PLNs.

### 3.2.4 LN Split/Replace/Merge

When an LN is full, the first step is to scan the LN to identify the number of valid *LN_element*s. Those NEGATIVE ones and the corresponding normal ones are




(a) Insert (7,b) (b) Delete (6,a) (c) Update (8,c)→(8,y)

Figure 3: Example of NV-Tree Insertion/Deletion/Update

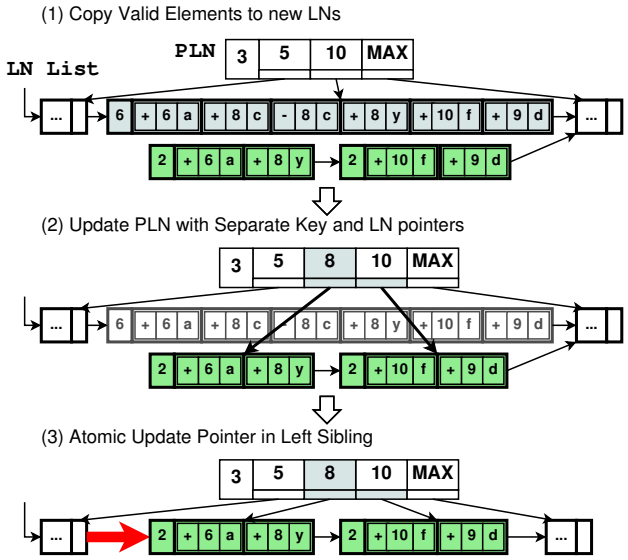



Figure 4: Example of LN Split

both considered invalid. The second step is to determine whether the LN needs a split.

If the percentage of valid elements is above the minimal fill factor (e.g., 50% in standard B$^+$Tree), we perform **split**. 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. Figure 4 shows an example of an LN split.

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, and the new LN replaces the old LN in the LN list using an atomic write. For **merge**, those valid *LN_element*s from both the old LN and its right sibling are copied to a new LN, and the new LN replaces both of them in the LN list using an atomic write. 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* are omitted here.

### 3.2.5 Rebuilding

As the memory address of each IN/PLN is fixed upon creation, IN/PLNs are not allowed to split. Therefore, when one PLN is full, all IN/PLNs have to be reconstructed to make space in PLNs to hold more LN pointers. The first step 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 under a workload with uniformly distributed access pattern, each PLN stores exactly one LN pointer after rebuilding. Optimizing rebuilding for workloads with different access patterns is one of our future work.

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 large NV-Tree. However, given the frequency of rebuilding, such overhead is neglectable in a long-running application (less than 1% in most cases, details can be found in Section 4.7).

If the memory space is enough to hold the new IN/PLNs without deleting the old ones, search can still proceed during rebuilding because it can always access the tree from the old IN/PLNs. In that case, the memory requirement of rebuilding is the total size of both old and new IN/PLNs. For instance, when inserting 100 million entries with random keys to a NV-Tree with 4KB nodes, rebuilding is executed only for two times. The memory requirement to enable parallel rebuilding for the first/second rebuilding is only about 1MB/129MB which is totally acceptable.

#### 3.2.6 Recovery

Since the LN list (*critical data*) is consistent, *rebuild-from-LN* is sufficient to recover a NV-Tree from either normal shutdown or system failure.

To further optimize the recovery after normal shutdown, our current implementation is able to achieve **instant recovery** by storing IN/PLNs persistently 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.

## 4 Evaluation

In this section, we evaluate our NV-Tree by comparing it with LCB$^+$Tree and CDDS-Tree in terms of insertion performance, overall performance under mixed workloads and throughput of all types of tree operations. We also study the overhead of rebuilding by quantifying its impact on the overall performance. We use YCSB [15], a benchmark for KV-stores, to perform an end-to-end comparison between our NV-Store and Redis [50], a well-known in-memory KV-store. Finally, we discuss the performance of NV-Tree on different types of NVM and estimated performance with *epoch*.

### 4.1 Implementation Effort

We implement our NV-Tree from scratch, an LCB$^+$Tree by applying `flush` and logging to a standard B$^+$Tree [5], and a CDDS-Tree [53]. 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 continuous (virtual) memory space. The node "pointer" stored in NV-Tree is actually 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 a reserved location, our NV-Tree is 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 is able to support lock-free concurrent access using timestamp-based multi-version concurrency control (MVCC) [38]. Based on that, we implement NV-Store to support Snapshot Isolation [3] transactions. Finally, we implement a database interface layer to extend YCSB to support NV-Store to facilitate our performance evaluation.

### 4.2 Experimental Setup

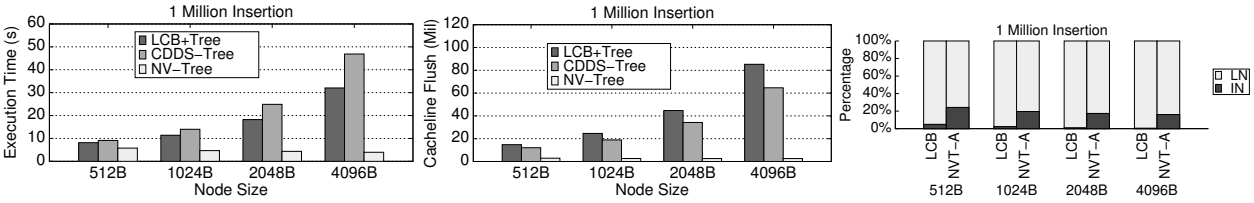
All of our experiments are conducted on a Linux server (Kernel version 3.13.0-24) with an Intel Xeon E5-2650 2.4GHz CPU (512KB/2MB/20MB L1/L2/L3 cache), 8GB DRAM and 8GB NVDIMM [1] which has practically the same read/write latency as DRAM. In the end-to-end comparison, we use YCSB (0.1.4) to compare 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.

### 4.3 Insertion Performance

We first compare the insertion performance of LCB$^+$Tree, CDDS-Tree and NV-Tree with different node sizes. Figure 5a 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 8X and 16X 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 cacheline flush for shifting entries. The size of those cacheline 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 cacheline flush thanks to both the unsorted LN and decoupling strategy of enforcing consistency selectively. Specifically, as shown in Figure 5b, NV-Tree reduces the total CPU cacheline flush by 80%-97% compared to LCB$^+$Tree and 76%-96% compared to CDDS-Tree.

Although the consistency cost of INs is almost neglectable for LCB$^+$Tree and CDDS-Tree as shown in Figure 1d, such cost becomes relatively expensive in NV-Tree. This is because the consistency cost for LN is significantly reduced after our optimization for LN, such as keeping entries unsorted and modifying LN with a log-free append-only approach. To quantify the consistency cost of INs after such optimization, we implement a mod-




(a) Execution Time with Varied Node Sizes (b) Number of CPU Cacheline Flush (c) Cacheline Flush Breakdown for IN/LN

Figure 5: Insertion Performance and Cacheline Flush Comparison

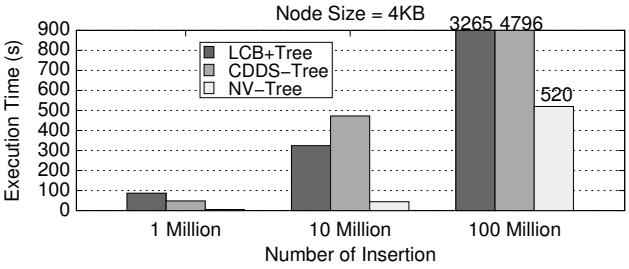



Figure 6: Execution Time of 1/10/100 Million Insertion

ified 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 5c shows the breakdown of CPU cacheline flush for IN and LN in LCB$^+$Tree and NVT-A. The percentage of CPU cacheline flush for IN increase 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 6 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 15.2X/6.3X/5.3X over LCB$^+$Tree and 8X/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.

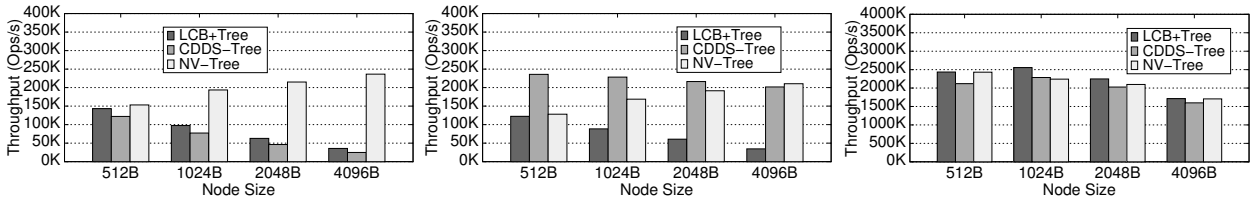
## 4.4 Update/Deletion/Search Throughput

This subsection compares the throughput of update/deletion/search operations in LCB$^+$Tree, CDDS-Tree and NV-Tree. In this experiment, 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 7. As shown in Figure 7a, 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 `flush`es 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 7b. 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 7a and 7b 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. In NV-Tree, by contrast, the throughput of update and deletion 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




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

Figure 7: Update/Deletion/Search Throughput Comparison

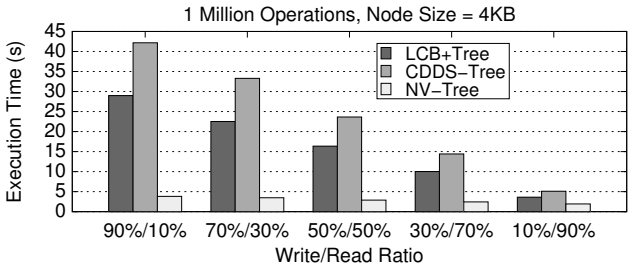



Figure 8: Execution Time of Mixed Workloads

number of split as well as rebuilding.

Although NV-Tree uses unsorted LN, thanks to the cache-optimized IN layout, the search throughput of NV-Tree is comparable to that of LCB$^+$Tree and CDDS-Tree as shown in Figure 7c, which is consistent to the published result [9].

## 4.5 Mixed Workloads

Figure 8 shows the execution time of performing one million insertion/search operations with varied 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.

Firstly, all three trees have better performance under workloads with less insertion. 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 NV-Tree drops but is still better than both competitors under the most read-intensive workload (10% insertion/90% search). This is because NV-Tree has much better insertion performance and comparable search throughput as well.

## 4.6 CPU Cache Efficiency

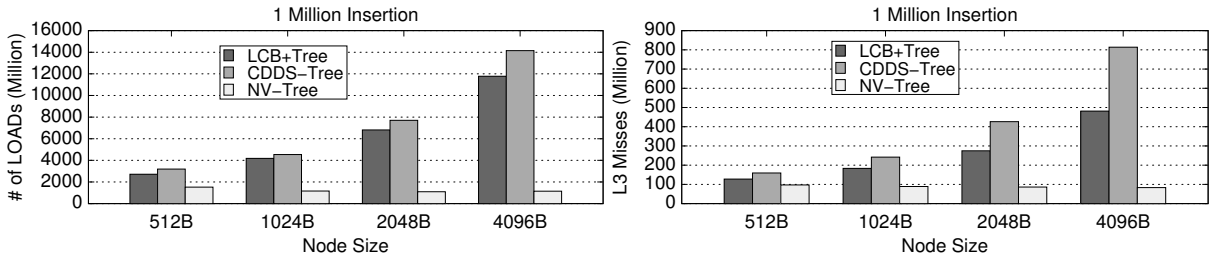
This subsection shows the underlying CPU cache efficiency of LCB$^+$Tree, CDDS-Tree and NV-Tree by using vTune Amplifier. Figure 9a shows the total number of LOAD instructions executed for inserting one million KV-pairs in each tree. NV-Tree reduces the number of LOAD instruction by about 44%-90% and 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 9b 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 cacheline invalidation and flush.

## 4.7 Rebuilding and Failure Recovery

To quantify the impact of rebuilding on the overall performance of NV-Tree, we measure the total number and time of rebuilding with different node sizes under different number of insertion. Compared to the total execution time, as shown in Table 2, the percentage of rebuilding time in the total execution time is below 1% for all types of workloads, which is totally neglectable. Moreover, we can tune the rebuilding frequency by increasing the size of tree nodes because the total number of splits decreases with larger nodes as shown in Figure 10a. With less splits, the frequency of rebuilding also becomes less, e.g., for 100 million insertion, with node size equals to 512B/1KB/2KB/4KB, the number of rebuilding is 7/4/3/2.

We also compare the performance of *rebuild-from-PLN* and *rebuild-from-LN*. Note that *rebuild-from-LN* is only used upon system failure. Figure 10b shows the total rebuilding time of both strategies for inserting 100 million KV-pairs to NV-Tree with different node sizes.

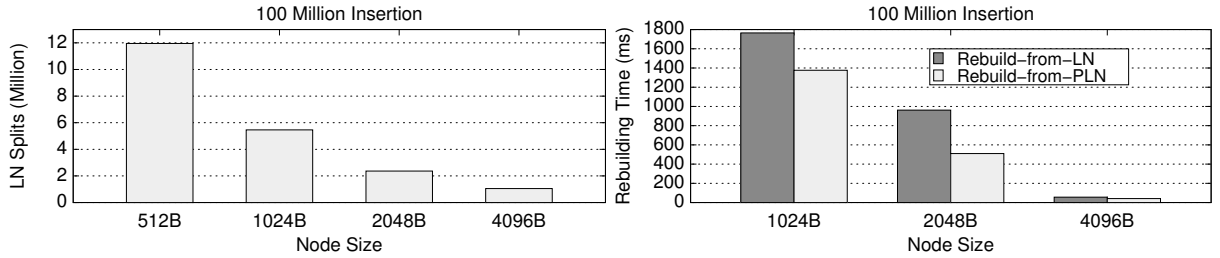



(a) Number of LOAD Instruction Executed (b) Number of L3 Cache Miss

Figure 9: Cache Efficiency Comparison

Table 2: Rebuilding Time (***ms***) for 1/10/100 Million Insertion with 512B/1KB/2KB/4KB Nodes

| Rebuild | 1M | | | | 10M | | | | 100M | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Node Size | | | | Node Size | | | | Node Size | | | |
| # | 0.5KB | 1KB | 2KB | 4KB | 0.5KB | 1KB | 2KB | 4KB | 0.5KB | 1KB | 2KB | 4KB |
| 1 | 0.104 | 0.119 | 0.215 | 0.599 | 0.058 | 0.091 | 0.213 | 0.603 | 0.066 | 0.091 | 0.206 | 0.572 |
| 2 | 0.779 | 2.592 | 8.761 | - | 0.503 | 2.525 | 8.526 | 41.104 | 0.520 | 2.118 | 8.594 | 41.077 |
| 3 | 7.433 | 50.021 | - | - | 4.782 | 54.510 | - | - | 4.706 | 47.219 | 814.989 | - |
| 4 | 31.702 | - | - | - | 39.546 | - | - | - | 37.481 | 1310.004 | - | - |
| 5 | - | - | - | - | 312.139 | - | - | - | 322.606 | - | - | - |
| 6 | - | - | - | - | - | - | - | - | 2567.219 | - | - | - |
| 7 | - | - | - | - | - | - | - | - | 16231.647 | - | - | - |
| Rebuilding Time | 40.018 | 52.559 | 8.976 | 0.599 | 357.016 | 57.126 | 8.739 | 41.707 | 19164.135 | 1359.432 | 823.789 | 41.649 |
| Execution Time | 6107.971 | 4672.032 | 4349.421 | 3955.227 | 62649.634 | 55998.473 | 46874.810 | 44091.494 | 692341.866 | 604111.327 | 570825.594 | 518323.920 |
| **Percentage** | **0.66%** | **1.13%** | **0.21%** | **0.02%** | **0.57%** | **0.10%** | **0.02%** | **0.09%** | **2.77%** | **0.23%** | **0.14%** | **0.01%** |




(a) Number of LN Splits with Different Node Sizes (b) Rebuilding Time for Different Rebuilding Strategy

Figure 10: Rebuilding Overhead Analysis

*Rebuild-from-PLN* is faster than *rebuild-from-LN* by 22-47%. This is because *rebuild-from-PLN* only scans the PLNs but *rebuild-from-LN* has to scan the entire LN list.

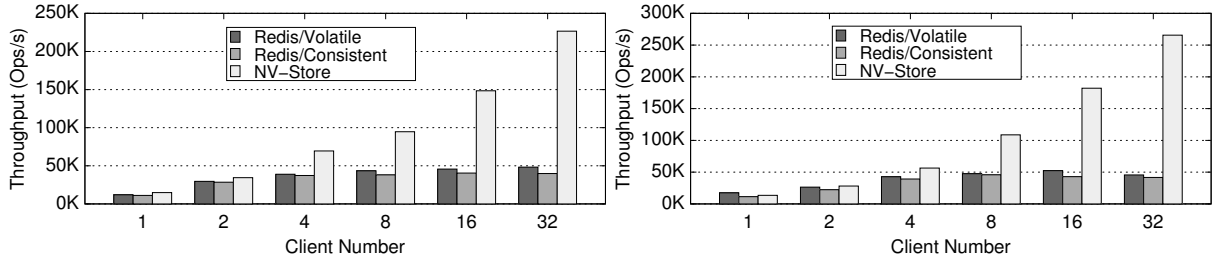
As the failure recovery of NV-Tree simply performs a *rebuild-from-LN*. The recovery time depends on the total number of LNs, but is bounded by the time of *rebuild-from-LN* as shown in Figure 10b.

To validate the consistency, we manually trigger the failure recovery by (1) killing NV-Tree process and (2) cutting the power supply when running both 100M insertion workload and YCSB workloads. Then we check whether NV-Tree has any data inconsistency or memory leak. We repeat these tests a few thousand times for NV-Tree and find it pass the check in all cases.

## 4.8 End-to-End Performance

In this subsection, we present the performance of our NV-Store under two YCSB workloads, StatusUpdate (read-latest) and SessionStore (update-heavy), compared to Redis. NV-Store is practically durable and consistent because it stores data in the NVM space directly allocated from NVDIMM using our modified system call. 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 (fsync).

Figure 11a shows the throughput of NV-Store and Redis under StatusUpdate workload which 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. The result shows that NV-Store improve the throughput by up to 3.2X over both volatile and consistent Redis. This indicates the optimization of reducing cacheline flush for insertion can significantly improve the performance even with as low as 5% insertion percentage. Moreover, both volatile and




(a) Throughput of YCSB Workload: StatusUpdate (b) Throughput of YCSB Workload: SessionStore

Figure 11: Throughput Comparison of NV-Store and Redis

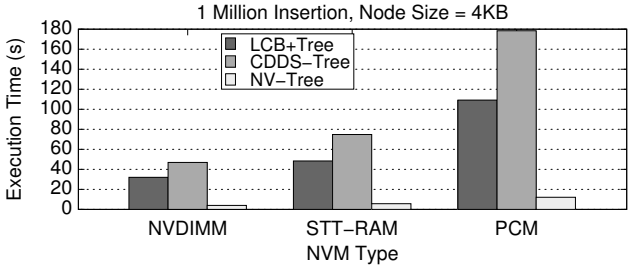



Figure 12: Execution Time on Different Types of NVM

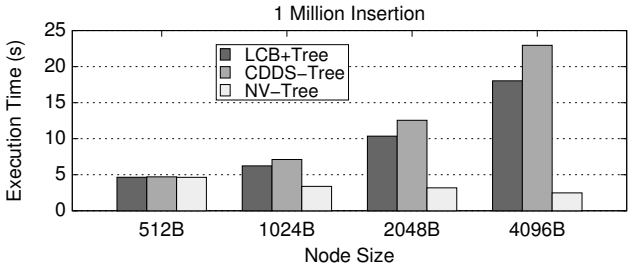



Figure 13: Estimated Execution Time with Epoch

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 (1) allowing concurrent search in LN while it is being updated, (2) searching in IN/PLNs without locks. Figure 11b shows the throughput under SessionStore workload which has 50%/50% search/update ratio on keys chosen from a Zipf distribution to represent applications in which people record recent actions. NV-Store can improve the throughput by up to 4.8X over Redis because the workload is more write-intensive.

## 4.9 Discussion

### 4.9.1 NV-Tree on 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 12 shows the execution time of one million insertion 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 from less performance drop than LCB$^+$Tree and CDDS-Tree on slower NVM because of the reduction of CPU cacheline flush.

### 4.9.2 NV-Tree on Future Hardware: *Epoch* and *CLWB/CLFLUSHOPT/PCOMMIT*

Comparing to `MFENCE` and `CLFLUSH`, *epoch* and a couple of new instructions for non-volatile storage (`CLWB/CLFLUSHOPT/PCOMMIT`) added by Intel recently [26] are able to flush CPU cachelines without explicit invalidations which means it does not trigger any additional cache misses. As these approaches are still unavailable in existing hardware, we estimate LCB$^+$Tree, CDDS-Tree and our NV-Tree performance by removing the cost of L3 cache misses due to cacheline flushes the execution time (Figure 5a). For B$^+$Tree and volatile CDDS-Tree, such cost can be derived by deducting the number of L3 cache misses without cacheline flushes (Figure 1b) from that with cacheline flushes (Figure 9b). As shown in Figure 13, with the cache miss penalty removed, the performance improvement of NV-Tree over LCB$^+$Tree/CDDS-Tree is 7X/9X with 4KB nodes. This indicates our optimization of reducing cacheline flush is still valuable when flushing a cacheline without the invalidation becomes possible.

## 5 Conclusion and Future Work

In this paper, we quantify the consistency cost of applying existing approaches such as logging and versioning on B$^+$Tree. Based on our observations, we propose our NV-Tree which require the data consistency in NVM connected through a memory bus, e.g., NVM-based single level systems. By selectively enforcing consistency, adopting unsorted LN and organizing IN cache-optimized, NV-Tree can reduce the number of cacheline flushes under write-intensive workloads by more than 90% compared to CDDS-Tree. Using NV-Tree as the core data structure, we 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 8X and 12X under write-intensive workloads, respectively. Our NV-Store increases the throughput by up to 4.8X under YCSB workloads compared to Redis. In our future work, we will continue to reduce the overhead of the rebuilding in larger datasets, validate and improve the performance of NV-Tree under skewed and TPC-C workloads, and explore NV-Tree in the distributed environment.

## Acknowledgment

We would like to thank our teammates, Mingdi Xue and Renuga Kanagavelu, the anonymous reviewers and our shepherd, Nisha Talagala, for their helpful comments. This work was supported by Agency for Science, Technology and Research (A*STAR), Singapore under Grant No. 112-172-0010. Bingsheng's work was partly supported by a MoE AcRF Tier 2 grant (MOE2012-T2-1-126) in Singapore.

## References


[1] AGIGATECH. Arxcis-nv (tm) non-volatile dimm. *http://www.vikingtechnology.com/arxcis-nv* (2014).

[2] BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., , KRANER, R., KUSZMAUL, B. C., MEDJEDOVIC, D., MONTES, P., SHETTY, P., SPILLANE, R. P., AND ZADOK, E. Don't thrash: How to cache your hash on flash. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB '12)* (Istanbul, Turkey, August 2012), Morgan Kaufmann.

[3] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ansi sql isolation levels. In *ACM SIGMOD Record* (1995), vol. 24, ACM, pp. 1–10.

[4] BHASKARAN, M. S., XU, J., AND SWANSON, S. Bankshot: caching slow storage in fast non-volatile memory. *Operating Systems Review 48*, 1 (2014), 73–81.

[5] BINGMANN, T. Stx b+ tree c++ template classes, 2008.

[6] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA* (2010), pp. 385–395.

[7] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing safe, user space access to fast, solid state disks. *ACM SIGPLAN Notices 47*, 4 (2012), 387–400.

[8] CAULFIELD, A. M., AND SWANSON, S. Quicksan: a storage area network for fast, distributed, solid state disks. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013* (2013), pp. 464–474.

[9] CHEN, S., GIBBONS, P. B., AND NATH, S. Rethinking database algorithms for phase change memory. In *CIDR* (2011), pp. 21–31.

[10] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* (Farmington, PA, November 2013).

[11] CHO, S., PARK, C., OH, H., KIM, S., YI, Y., AND GANGER, G. R. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, ACM, pp. 91–102.

[12] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 105–118.

[13] COMER, D. Ubiquitous b-tree. *ACM Computing Surveys (CSUR) 11*, 2 (1979), 121–137.

[14] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.

[15] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[16] CULLY, B., WIRES, J., MEYER, D., JAMIESON, K., FRASER, K., DEEGAN, T., STODDEN, D., LEFEBVRE, G., FERSTAY, D., AND WARFIELD, A. Strata: High-performance scalable storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 17–31.

[17] DEBRABANT, J., ARULRAJ, J., PAVLO, A., STONEBRAKER, M., ZDONIK, S., AND DULLOOR, S. R. A prolegomenon on oltp database systems for non-volatile memory. *Proceedings of the VLDB Endowment 7*, 14 (2014).

[18] DHIMAN, G., AYOUB, R., AND ROSING, T. Pdram: a hybrid pram and dram main memory system. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE* (2009), IEEE, pp. 664–669.

[19] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 15.

[20] EVERSPIN. Second generation mram: Spin torque technology. *http://www.everspin.com/products/second-generation-st-mram.html* (2004).

[21] FREITAS, R. F., AND WILCKE, W. W. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development 52*, 4.5 (2008), 439–447.

[22] FRYER, D., SUN, K., MAHMOOD, R., CHENG, T., BENJAMIN, S., GOEL, A., AND BROWN, A. D. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)* (San Jose, CA, February 2012), pp. 73–86.

[23] GAO, S., XU, J., HE, B., CHOI, B., AND HU, H. Pcmlogging: Reducing transaction logging overhead with pcm. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (New York, NY, USA, 2011), CIKM '11, ACM, pp. 2401–2404.

[24] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012* (2012), p. 2.

[25] INTEL. Intel 64 and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part 1* (2014).

[26] INTEL. Intel architecture instruction set extensions programming reference. *https://software.intel.com* (2014).

[27] JUNG, M., CHOI, W., SHALF, J., AND KANDEMIR, M. T. Triple-a: a non-ssd based autonomic all-flash array for high performance storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014), ACM, pp. 441–454.

[28] JUNG, M., WILSON III, E. H., CHOI, W., SHALF, J., AKTULGA, H. M., YANG, C., SAULE, E., CATALYUREK, U. V., AND KANDEMIR, M. Exploring the future of out-of-core computing with compute-local non-volatile memory. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis* (2013), ACM, p. 75.

[29] KAWAHARA, T. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Design & Test of Computers 28*, 1 (2011), 0052–63.

[30] KIM, H., SESHADRI, S., DICKEY, C. L., AND CHIU, L. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 33–45.

[31] KIM, M., SHIN, J., AND WON, Y. Selective segment initialization: Exploiting nvram to reduce device startup latency. In *Embedded Systems Letters* (2014), pp. 33–36.

[32] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 273–285.

[33] LEE, D., AND WON, Y. Bootless boot: Reducing device boot latency with byte addressable NVRAM. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013* (2013), pp. 2014–2021.

[34] LEE, E., BAHN, H., AND NOH, S. H. Unioning of the buffer cache and journaling layers with non-volatile memory. In *FAST* (2013), pp. 73–80.

[35] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: a capacity-optimized ssd cache for primary storage. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference* (2014), USENIX Association, pp. 501–512.

[36] LI, Y., HE, B., YANG, R. J., LUO, Q., AND YI, K. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1195–1206.

[37] LIU, R.-S., SHEN, D.-Y., YANG, C.-L., YU, S.-C., AND WANG, C.-Y. M. Nvm duet: unified working memory and persistent store architecture. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014), ACM, pp. 455–470.

[38] LOMET, D., AND SALZBERG, B. *Access methods for multiversion data*, vol. 18. ACM, 1989.

[39] LV, Y., CUI, B., HE, B., AND CHEN, X. Operation-aware buffer management in flash-based systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), ACM, pp. 13–24.

[40] MANDELMAN, J. A., DENNARD, R. H., BRONNER, G. B., DEBROSSE, J. K., DIVAKARUNI, R., LI, Y., AND RADENS, C. J. Challenges and future directions for the scaling of dynamic random-access memory (dram). *IBM Journal of Research and Development 46*, 2.3 (2002), 187–212.

[41] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. Nvmkv: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (2014), USENIX Association.

[42] MORARU, I., ANDERSEN, D. G., KAMINSKY, M., TOLIA, N., RANGANATHAN, P., AND BINKERT, N. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the 2013 Conference on Timely Results in Operating Systems* (2013).

[43] MUELLER, W., AICHMAYR, G., BERGNER, W., ERBEN, E., HECHT, T., KAPTEYN, C., KERSCH, A., KUDELKA, S., LAU, F., LUETZEN, J., ET AL. Challenges for the dram cell scaling to 40nm. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International* (2005), IEEE, pp. 4–pp.

[44] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 401–410.

[45] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on* (2014), IEEE, pp. 265–276.

[46] PILLAI, T. S., CHIDAMBARAM, V., ALAGAPPAN, R., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)* (Broomfield, CO, October 2014).

[47] QIN, D., BROWN, A. D., AND GOEL, A. Reliable writeback for client-side flash caches. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 451–462.

[48] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News 37*, 3 (2009), 24–33.

[49] RAOUX, S., BURR, G. W., BREITWISCH, M. J., RETTNER, C. T., CHEN, Y.-C., SHELBY, R. M., SALINGA, M., KREBS, D., CHEN, S.-H., LUNG, H.-L., ET AL. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development 52*, 4.5 (2008), 465–479.

[50] SANFILIPPO, S., AND NOORDHUIS, P. Redis. *http://redis.io* (2009).

[51] SESHADRI, S., GAHAGAN, M., BHASKARAN, S., BUNKER, T., DE, A., JIN, Y., LIU, Y., AND SWANSON, S. Willow: A user-programmable ssd. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 67–80.

[52] SRIRAM SUBRAMANIAN, SWAMI SUNDARARAMAN, NISHA TALAGALA, ANDREA C. ARPACI-DUSSEAU, REMZI H. ARPACI-DUSSEAU. Snapshots in a Flash with ioSnap. In *EuroSys '14* (Amsterdam, Netherlands, April 2014).

[53] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., CAMPBELL, R. H., ET AL. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST* (2011), pp. 61–75.

[54] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 14.

[55] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 91–104.

[56] VUČINIĆ, D., WANG, Q., GUYOT, C., MATEESCU, R., BLAGOJEVIĆ, F., FRANCA-NETO, L., MOAL, D. L., BUNKER, T., XU, J., SWANSON, S., AND BANDIĆ, Z. Dc express: Shortest latency protocol for reading phase change memory over pci express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 309–315.

[57] WANG, C., VAZHKUDAI, S. S., MA, X., MENG, F., KIM, Y., AND ENGELMANN, C. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International* (2012), IEEE, pp. 957–968.

[58] WU, X., AND REDDY, A. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 39.

[59] YIYING ZHANG, ANDREA C. ARPACI-DUSSEAU, REMZI H. ARPACI-DUSSEAU. Warped Mirrors for Flash. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST '13)* (Long Beach, California, May 2013).

[60] YIYING ZHANG, LEO ARULRAJ, ANDREA C. ARPACI-DUSSEAU, REMZI H. ARPACI-DUSSEAU. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST '12)* (San Jose, California, February 2012).

[61] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 14–23.