Consistent and Durable Data Structures for
Non-Volatile Byte-Addressable Memory

Shivaram Venkataraman[†], Niraj Tolia[‡], Parthasarathy Ranganathan[†], and Roy H. Campbell[*]*
[†]HP Labs, Palo Alto, [‡]Maginatics, and [*]University of Illinois, Urbana-Champaign

Abstract

The predicted shift to non-volatile, byte-addressable memory (e.g., Phase Change Memory and Memristor), the growth of "big data", and the subsequent emergence of frameworks such as memcached and NoSQL systems require us to rethink the design of data stores. To derive the maximum performance from these new memory technologies, this paper proposes the use of single-level data stores. For these systems, where no distinction is made between a volatile and a persistent copy of data, we present Consistent and Durable Data Structures (CDDSs) that, on current hardware, allows programmers to safely exploit the low-latency and non-volatile aspects of new memory technologies. CDDSs use versioning to allow atomic updates without requiring logging. The same versioning scheme also enables rollback for failure recovery. When compared to a memory-backed Berkeley DB B-Tree, our prototype-based results show that a CDDS B-Tree can increase put and get throughput by 74% and 138%. When compared to Cassandra, a two-level data store, Tembo, a CDDS B-Tree enabled distributed Key-Value system, increases throughput by up to 250%–286%.

1 Introduction

Recent architecture trends and our conversations with memory vendors show that DRAM density scaling is facing significant challenges and will hit a scalability wall beyond 40nm [26, 33, 34]. Additionally, power constraints will also limit the amount of DRAM installed in future systems [5, 19]. To support next generation systems, including large memory-backed data stores such as memcached [18] and RAMCloud [38], technologies such as Phase Change Memory [40] and Memristor [48] hold promise as DRAM replacements. Described in Section 2, these memories offer latencies that are comparable to DRAM and are orders of magnitude faster than ei-

ther disk or flash. Not only are they byte-addressable and low-latency like DRAM but, they are also non-volatile.

Projected cost [19] and power-efficiency characteristics of Non-Volatile Byte-addressable Memory (NVBM) lead us to believe that it can replace both disk and memory in data stores (e.g., memcached, database systems, NoSQL systems, etc.) but not through legacy interfaces (e.g., block interfaces or file systems). First, the overhead of PCI accesses or system calls will dominate NVBM's sub-microsecond access latencies. More importantly, these interfaces impose a two-level logical separation of data, differentiating between in-memory and on-disk copies. Traditional data stores have to both update the in-memory data and, for durability, sync the data to disk with the help of a write-ahead log. Not only does this data movement use extra power [5] and reduce performance for low-latency NVBM, the logical separation also reduces the usable capacity of an NVBM system.

Instead, we propose a single-level NVBM hierarchy where no distinction is made between a volatile and a persistent copy of data. In particular, we propose the use of Consistent and Durable Data Structures (CDDSs) to store data, a design that allows for the creation of logless systems on non-volatile memory without processor modifications. Described in Section 3, these data structures allow mutations to be safely performed directly (using loads and stores) on the single copy of the data and metadata. We have architected CDDSs to use versioning. Independent of the update size, versioning allows the CDDS to atomically move from one consistent state to the next, without the extra writes required by logging or shadow paging. Failure recovery simply restores the data structure to the most recent consistent version. Further, while complex processor changes to support NVBM have been proposed [14], we show how primitives to provide durability and consistency can be created using existing processors.

We have implemented a CDDS B-Tree because of its non-trivial implementation complexity and widespread

use in storage systems. Our evaluation, presented in Section 4, shows that a CDDS B-Tree can increase put and get throughput by 74% and 138% when compared to a memory-backed Berkeley DB B-Tree. Tembo[1], our Key-Value (KV) store described in Section 3.5, was created by integrating this CDDS B-Tree into a widely-used open-source KV system. Using the Yahoo Cloud Serving Benchmark [15], we observed that Tembo increases throughput by up to 250%–286% when compared to memory-backed Cassandra, a two-level data store.

2 Background and Related Work

2.1 Hardware Non-Volatile Memory

Significant changes are expected in the memory industry. Non-volatile flash memories have seen widespread adoption in consumer electronics and are starting to gain adoption in the enterprise market [20]. Recently, new NVBM memory technologies (e.g., PCM, Memristor, and STTRAM) have been demonstrated that significantly improve latency and energy efficiency compared to flash.

As an illustration, we discuss Phase Change Memory (PCM) [40], a promising NVBM technology. PCM is a non-volatile memory built out of Chalcogenide-based materials (e.g., alloys of germanium, antimony, or tellurium). Unlike DRAM and flash that record data through charge storage, PCM uses distinct phase change material states (corresponding to resistances) to store values. Specifically, when heated to a high temperature for an extended period of time, the materials crystallize and reduce their resistance. To reset the resistance, a current large enough to melt the phase change material is applied for a short period and then abruptly cut-off to quench the material into the amorphous phase. The two resistance states correspond to a '0' and '1', but, by varying the pulse width of the reset current, one can partially crystallize the phase change material and modify the resistance to an intermediate value between the '0' and '1' resistances. This range of resistances enables multiple bits per cell, and the projected availability of these MLC designs is 2012 [25].

Table 1 summarizes key attributes of potential storage alternatives in the next decade, with projected data from recent publications, technology trends, and direct industry communication. These trends suggest that future non-volatile memories such as PCM or Memristors can be viable DRAM replacements, achieving competitive speeds with much lower power consumption, and with non-volatility properties similar to disk but without the power overhead. Additionally, a number of recent studies have identified a slowing of DRAM

growth [25, 26, 30, 33, 34, 39, 55] due to scaling challenges for charge-based memories. In conjunction with DRAM's power inefficiencies [5, 19], these trends can potentially accelerate the adoption of NVBM memories.

NVBM technologies have traditionally been limited by density and endurance, but recent trends suggest that these limitations can be addressed. Increased density can be achieved within a single-die through multi-level designs, and, potentially, multiple-layers per die. At a single chip level, 3D die stacking using through-silicon vias (TSVs) for inter-die communication can further increase density. PCM and Memristor also offer higher endurance than flash (10^8 writes/cell compared to 10^5 writes/cell for flash). Optimizations at the technology, circuit, and systems levels have been shown to further address endurance issues, and more improvements are likely as the technologies mature and gain widespread adoption.

These trends, combined with the attributes summarized in Table 1, suggest that technologies like PCM and Memristors can be used to provide a single "unified datastore" layer - an assumption underpinning the system architecture in our paper. Specifically, we assume a storage system layer that provides disk-like functionality but with memory-like performance characteristics and improved energy efficiency. This layer is persistent and byte-addressable. Additionally, to best take advantage of the low-latency features of these emerging technologies, non-volatile memory is assumed to be accessed off the memory bus. Like other systems [12, 14], we also assume that the hardware can perform atomic 8 byte writes.

While our assumed architecture is future-looking, it must be pointed out that many of these assumptions are being validated individually. For example, PCM samples are already available (e.g., from Numonyx) and an HP/Hynix collaboration [22] has been announced to bring Memristor to market. In addition, aggressive capacity roadmaps with multi-level cells and stacking have been discussed by major memory vendors. Finally, previously announced products have also allowed non-volatile memory, albeit flash, to be accessed through the memory bus [46].

2.2 File Systems

Traditional disk-based file systems are also faced with the problem of performing atomic updates to data structures. File systems like WAFL [23] and ZFS [49] use shadowing to perform atomic updates. Failure recovery in these systems is implemented by restoring the file system to a consistent snapshot that is taken periodically. These snapshots are created by shadowing, where every change to a block creates a new copy of the block. Recently, Rodeh [42] presented a B-Tree construction that can provide efficient support for shadowing and this tech-

---

[1] Swahili for elephant, an animal anecdotally known for its memory.

Technology	Density um^2/bit	Read/Write Latency ns		Read/Write Energy pJ/bit		Endurance writes/bit
HDD	0.00006	3,000,000	3,000,000	2,500	2,500	∞
Flash SSD (SLC)	0.00210	25,000	200,000	250	250	10^5
DRAM (DIMM)	0.00380	55	55	24	24	10^{18}
PCM	0.00580	48	150	2	20	10^8
Memristor	0.00580	100	100	2	2	10^8

Table 1: Non-Volatile Memory Characteristics: 2015 Projections

nique has been used in the design of BTRFS [37]. Failure recovery in a CDDS uses a similar notion of restoring the data structure to the most recent consistent version. However the versioning scheme used in a CDDS results in fewer data-copies when compared to shadowing.

2.3 Non-Volatile Memory-based Systems

The use of non-volatile memory to improve performance is not new. eNVy [54] designed a non-volatile main memory storage system using flash. eNVy, however, accessed memory on a page-granularity basis and could not distinguish between temporary and permanent data. The Rio File Cache [11, 32] used battery-backed DRAM to emulate NVBM but it did not account for persistent data residing in volatile CPU caches. Recently there have been many efforts [21] to optimize data structures for flash memory based systems. FD-Tree [31] and Buffer-Hash [2] are examples of write-optimized data structures designed to overcome high-latency of random writes, while FAWN [3] presents an energy efficient system design for clusters using flash memory. However, design choices that have been influenced by flash limitations (e.g., block addressing and high-latency random writes) render these systems suboptimal for NVBM.

Qureshi et al. [39] have also investigated combining PCM and DRAM into a hybrid main-memory system but do not use the non-volatile features of PCM. While our work assumes that NVBM wear-leveling happens at a lower layer [55], it is worth noting that versioning can help wear-leveling as frequently written locations are aged out and replaced by new versions. Most closely related is the work on NVTM [12] and BPFS [14]. NVTM, a more general system than CDDS, adds STM-based [44] durability to non-volatile memory. However, it requires adoption of an STM-based programming model. Further, because NVTM only uses a metadata log, it cannot guarantee failure atomicity. BPFS, a PCM-based file system, also proposes a single-level store. However, unlike CDDS's exclusive use of existing processor primitives, BPFS depends on extensive hardware modifications to provide correctness and durability. Further, unlike the data structure interface proposed in this work, BPFS implements a file system interface. While this is transparent to legacy applications, the system-call overheads reduce NVBM's low-latency benefits.

2.4 Data Store Trends

The growth of "big data" [1] and the corresponding need for scalable analytics has driven the creation of a number of different data stores today. Best exemplified by NoSQL systems [9], the throughput and latency requirements of large web services, social networks, and social media-based applications have been driving the design of next-generation data stores. In terms of storage, high-performance systems have started shifting from magnetic disks to flash over the last decade. Even more recently, this shift has accelerated to the use of large memory-backed data stores. Examples of the latter include memcached [18] clusters over 200 TB in size [28], memory-backed systems such as RAMCloud [38], in-memory databases [47, 52], and NoSQL systems such as Redis [41]. As DRAM is volatile, these systems provide data durability using backend databases (e.g., memcached/MySQL), on-disk logs (e.g., RAMCloud), or, for systems with relaxed durability semantics, via periodic checkpoints. We expect that these systems will easily transition from being DRAM-based with separate persistent storage to being NVBM-based.

3 Design and Implementation

As mentioned previously, we expect NVBM to be exposed across a memory bus and not via a legacy disk interface. Using the PCI interface (256 ns latency [24]) or even a kernel-based syscall API (89.2 and 76.4 ns for POSIX `read/write`) would add significant overhead to NVBM's access latencies (50–150 ns). Further, given the performance and energy cost of moving data, we believe that all data should reside in a single-level store where no distinction is made between volatile and persistent storage and all updates are performed in-place. We therefore propose that data access should use userspace libraries and APIs that map data into the process's address space.

However, the same properties that allow systems to take full advantage of NVBM's performance properties also introduce challenges. In particular, one of the biggest obstacles is that current processors do not provide primitives to order memory writes. Combined with the fact that the memory controller can reorder writes (at a cache line granularity), current mechanisms for updat-

ing data structures are likely to cause corruption in the face of power or software failures. For example, assume that a hash table insert requires the write of a new hash table object and is followed by a pointer write linking the new object to the hash table. A reordered write could propagate the pointer to main memory before the object and a failure at this stage would cause the pointer to link to an undefined memory region. Processor modifications for ordering can be complex [14], do not show up on vendor roadmaps, and will likely be preceded by NVBM availability.

To address these issues, our design and implementation focuses on three different layers. First, in Section 3.1, we describe how we implement ordering and flushing of data on existing processors. However, this low-level primitive is not sufficient for atomic updates larger than 8 bytes. In addition, we therefore also require versioning CDDSs, whose design principles are described in Section 3.2. After discussing our CDDS B-Tree implementation in Section 3.3 and some of the open opportunities and challenges with CDDS data structures in Section 3.4, Section 3.5 describes Tembo, the system resulting from the integration of our CDDS B-Tree into a distributed Key-Value system.

3.1 Flushing Data on Current Processors

As mentioned earlier, today's processors have no mechanism for preventing memory writes from reaching memory and doing so for arbitrarily large updates would be infeasible. Similarly, there is no guarantee that writes will not be reordered by either the processor or by the memory controller. While processors support a `mfence` instruction, it only provides write visibility and does not guarantee that all memory writes are propagated to memory (NVBM in this case) or that the ordering of writes is maintained. While cache contents can be flushed using the `wbinvd` instruction, it is a high-overhead operation (multiple ms per invocation) and flushes the instruction cache and other unrelated cached data. While it is possible to mark specific memory regions as write-through, this impacts write throughput as all stores have to wait for the data to reach main memory.

To address this problem, we use a combination of tracking recently written data and use of the `mfence` and `clflush` instructions. `clflush` is an instruction that invalidates the cache line containing a given memory address from all levels of the cache hierarchy, across multiple processors. If the cache line is dirty (i.e., it has uncommitted data), it is written to memory before invalidation. The `clflush` instruction is also ordered by the `mfence` instruction. Therefore, to commit a series of memory writes, we first execute an `mfence` as a barrier to them, execute a `clflush` on every cacheline of all

modified memory regions that need to be committed to persistent memory, and then execute another `mfence`. In this paper, we refer to this instruction sequence as a *flush*. As microbenchmarks in Section 4.2 show, using `flush` will be acceptable for most workloads.

While this description and tracking dirty memory might seem complex, this was easy to implement in practice and can be abstracted away by macros or helper functions. In particular, for data structures, all updates occur behind an API and therefore the process of `flushing` data to non-volatile memory is hidden from the programmer. Using the simplified hash table example described above, the implementation would first write the object and `flush` it. Only after this would it write the pointer value and then `flush` again. This two-step process is transparent to the user as it occurs inside the insert method.

Finally, one should note that while `flush` is necessary for durability and consistency, it is not sufficient by itself. If any metadata update (e.g., rebalancing a tree) requires an atomic update greater than the 8 byte atomic write provided by the hardware, a failure could leave it in an inconsistent state. We therefore need the versioning approach described below in Sections 3.2 and 3.3.

3.2 CDDS Overview

Given the challenges highlighted at the beginning of Section 3, an ideal data store for non-volatile memory must have the following properties:

- **Durable**: The data store should be durable. A fail-stop failure should not lose committed data.

- **Consistent**: The data store should remain consistent after every update operation. If a failure occurs during an update, the data store must be restored to a consistent state before further updates are applied.

- **Scalable**: The data store should scale to arbitrarily-large sizes. When compared to traditional data stores, any space, performance, or complexity overhead should be minimal.

- **Easy-to-Program**: Using the data store should not introduce undue complexity for programmers or unreasonable limitations to its use.

We believe it is possible to meet the above properties by storing data in Consistent and Durable Data Structures (CDDSs), i.e., hardened versions of conventional data structures currently used with volatile memory. The ideas used in constructing a CDDS are applicable to a wide variety of linked data structures and, in this paper, we implement a CDDS B-Tree because of its non-trivial implementation complexity and widespread use in storage systems. We would like to note that the design and implementation of a CDDS only addresses *physical* consistency, i.e., ensuring that the data structure is readable
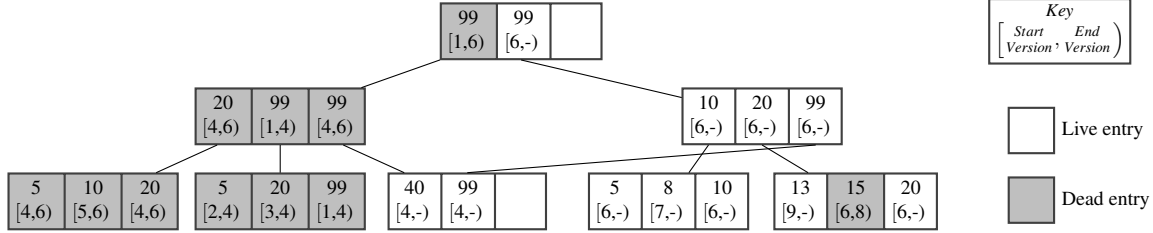
Key:
$$\begin{bmatrix} Start & End \\ Version & Version \end{bmatrix}$$

| 99 [1,6) | 99 [6,-) | |

| 20 [4,6) | 99 [1,4) | 99 [4,6) | | 10 [6,-) | 20 [6,-) | 99 [6,-) |

| 5 [4,6) | 10 [5,6) | 20 [4,6) | | 5 [2,4) | 20 [3,4) | 99 [1,4) | | 40 [4,-) | 99 [4,-) | | | 5 [6,-) | 8 [7,-) | 10 [6,-) | | 13 [9,-) | 15 [6,8) | 20 [6,-) |

□ Live entry
▧ Dead entry

Figure 1: Example of a CDDS B-Tree

and never left in a corrupt state. Higher-level layers control *logical* consistency, i.e., ensuring that the data stored in the data structure is valid and matches external integrity constraints. Similarly, while our current system implements a simple concurrency control scheme, we do not mandate concurrency control to provide isolation as it might be more efficient to do it at a higher layer.

A CDDS is built by maintaining a limited number of versions of the data structure with the constraint that an update should not weaken the structural integrity of an older version and that updates are atomic. This versioning scheme allows a CDDS to provide consistency without the additional overhead of logging or shadowing. A CDDS thus provides a guarantee that a failure between operations will never leave the data in an inconsistent state. As a CDDS never acknowledges completion of an update without safely committing it to non-volatile memory, it also ensures that there is no silent data loss.

3.2.1 Versioning for Durability

Internally, a CDDS maintains the following properties:

- There exists a version number for the most recent consistent version. This is used by any thread which wishes to read from the data structure.
- Every update to the data structure results in the creation of a new version.
- During the update operation, modifications ensure that existing data representing older versions are never overwritten. Such modifications are performed by either using atomic operations or copy-on-write style changes.
- After all the modifications for an update have been made persistent, the most recent consistent version number is updated atomically.

3.2.2 Garbage Collection

Along with support for multiple versions, a CDDS also tracks versions of the data structure that are being accessed. Knowing the oldest version which has a non-zero reference count has two benefits. First, we can garbage collect older versions of the data structure. Garbage collection (GC) is run in the background and helps limit the space utilization by eliminating data that will not be referenced in the future. Second, knowing the oldest active version can also improve performance by enabling intelligent space reuse in a CDDS. When creating a new entry, the CDDS can proactively reclaim the space used by older inactive versions.

3.2.3 Failure Recovery

Insert or delete operations may be interrupted due to operating system crashes or power failures. By definition, the most recent consistent version of the data structure should be accessible on recovery. However, an in-progress update needs to be removed as it belongs to an uncommitted version. We handle failures in a CDDS by using a 'forward garbage collection' procedure during recovery. This process involves discarding all update operations which were executed after the most recent consistent version. New entries created can be discarded while older entries with in-progress update operations are reverted.

3.3 CDDS B-Trees

As an example of a CDDS, we selected the B-Tree [13] data structure because of its widespread use in databases, file systems, and storage systems. This section discusses the design and implementation of a consistent and durable version of a B-Tree. Our B-Tree modifications[2] have been heavily inspired by previous work on multi-version data structures [4, 50]. However, our focus on durability required changes to the design and impacted our implementation. We also do not retain all previous versions of the data structure and can therefore optimize updates.

In a CDDS B-Tree node, shown in Figure 1, the key and value stored in a B-Tree entry is augmented with a start and end version number, represented by unsigned 64-bit integers. A B-Tree node is considered 'live' if it has at least one live entry. In turn, an entry is considered 'live' if it does not have an end version (displayed as a '−' in the figure). To bound space utilization, in addition to ensuring that a minimum number of entries in a B-Tree node are used, we also bound the minimum number of

---

[2]In reality, our B-Tree is a B+ Tree with values only stored in leaves.

Algorithm 1: CDDS B-Tree Lookup

Input: k: key, r: root
Output: val: value
1 **begin** lookup(*k, r*)
2    $v \leftarrow current\_version$
3    $n \leftarrow r$
4   **while** is_inner_node(*n*) **do**
5       $entry\_num \leftarrow$ find(*k, n, v*)
6       $n \leftarrow n[entry\_num].child$
7    $entry\_num \leftarrow$ find(*k, n, v*)
8   **return** $n[entry\_num].value$
9 **end**

10 **begin** find(*k, n, v*)
11    $l \leftarrow 0$
12    $h \leftarrow$ get_num_entries(*n*)
13   **while** $l < h$ **do**   // Binary Serch
14       $m \leftarrow (l+h)/2$
15   **if** $k \le n[m].key$ **then**
16          $h \leftarrow m - 1$
17   **else** $l \leftarrow m + 1$
18   **while** $h <$ get_num_entries(*n*) **do**
19   **if** $n[h].start \le v$ **then**
20   **if** $n[h].end > v$ || $n[h].end = 0$ **then**
21   *break*
22       $h \leftarrow h + 1$
23   **return** h
24 **end**

Algorithm 2: CDDS B-Tree Insertion

Input: k: key, r: root
1 **begin** insert_key(*k, r*)
2    $v \leftarrow current\_version$
3    $v' \leftarrow v + 1$
      // Recurse to leaf node (n)
4    $y \leftarrow$ get_num_entries(*n*)
5   **if** $y = node\_size$ **then**   // Node Full
6   **if** $entry\_num =$ can_reuse_version(*n,y*) **then**
7          $n[entry\_num].key \leftarrow k$
8          $n[entry\_num].start \leftarrow v'$
9          $n[entry\_num].end \leftarrow 0$
10   flush(*n[entry_num]*)
11   **else**
12   split_insert(*n, k, v'*)
         // Update inner nodes
13   **else**
14       $n[y].key \leftarrow k$
15       $n[y].start \leftarrow v'$
16       $n[y].end \leftarrow 0$
17   flush(*n[y]*)
18    $current\_version \leftarrow v'$
19   flush(*current_version*)
20 **end**

21 **begin** split_insert(*n, k, v*)
22    $l \leftarrow$ num_live_entries(*n*)
23    $m_l \leftarrow$ min_live_entries
24   **if** $l > 4 * m_l$ **then**
25       $nn_1 \leftarrow$ new_node
26       $nn_2 \leftarrow$ new_node
27   **for** $i = 1$ *to* $l/2$ **do**
28   insert(*nn_1, n[i].key, v*)
29   **for** $i = l/2 + 1$ *to* l **do**
30   insert(*nn_2, n[i].key, v*)
31   **if** $k < n[l/2].key$ **then**
32   insert(*nn_1, k, v*)
33   **else** insert(*nn_2, k, v*)
34   flush(*nn_1, nn_2*)
35   **else**
36       $nn \leftarrow$ new_node
37   **for** $i = 1$ *to* l **do**
38   insert(*nn, n[i].key, v*)
39   insert(*nn, k, v*)
40   flush(*nn*)
41   **for** $i = 1$ *to* l **do**
42       $n[i].end \leftarrow v$
43   flush(*n*)
44 **end**

---

live entries in each node. Thus, while the CDDS B-Tree API is identical to normal B-Trees, the implementation differs significantly. In the rest of this section, we use the *lookup*, *insert*, and *delete* operations to illustrate how the CDDS B-Tree design guarantees consistency and durability[3].

3.3.1 Lookup

We first briefly describe the lookup algorithm, shown in Algorithm 1. For ease of explanation and brevity, the pseudocode in this and following algorithms does not include all of the design details. The algorithm uses the find function to recurse down the tree (lines 4–6) until it finds the leaf node with the correct key and value.

Consider a lookup for the key 10 in the CDDS B-Tree shown in Figure 1. After determining the most current version (version 9, line 2), we start from the root node and pick the rightmost entry with key 99 as it is the next largest valid key. Similarly in the next level, we follow the link from the leftmost entry and finally retrieve the value for 10 from the leaf node.

Our implementation currently optimizes lookup performance by ordering node entries by key first and then by the start version number. This involves extra writes during inserts to shift entries but improves read performance by enabling a binary search within nodes

(lines 13–17 in find). While we have an alternate implementation that optimizes writes by not ordering keys at the cost of higher lookup latencies, we do not use it as our target workloads are read-intensive. Finally, once we detect the right index in the node, we ensure that we are returning a version that was valid for v, the requested version number (lines 18–22).

---

[3]A longer technical report [51] presents more details on all CDDS B-Tree operations and their corresponding implementations.

3.3.2 Insertion

The algorithm for inserting a key into a CDDS B-Tree is shown in Algorithm 2. Our implementation of the algorithm uses the `flush` operation (described in Section 3.1) to perform atomic operations on a cacheline. Consider the case where a key, 12, is inserted into the B-Tree shown in Figure 1. First, an algorithm similar to lookup is used to find the leaf node that contains the key range that 12 belongs to. In this case, the right-most leaf node is selected. As shown in lines 2–3, the current consistent version is read and a new version number is generated. As the leaf node is full, we first use the `can_reuse_version` function to check if an existing dead entry can be reused. In this case, the entry with key 15 died at version 8 and is reused. To reuse a slot we first remove the key from the node and shift the entries to maintain them in sorted order. Now we insert the new key and again shift entries as required. For each key shift, we ensure that the data is first `flushed` to another slot before it is overwritten. This ensures that the safety properties specified in Section 3.2.1 are not violated. While not described in the algorithm, if an empty entry was detected in the node, it would be used and the order of the keys, as specified in Section 3.3.1, would be maintained.

If no free or dead entry was found, a `split_insert`, similar to a traditional B-Tree split, would be performed. `split_insert` is a copy-on-write style operation in which existing entries are copied before making a modification. As an example, consider the node shown in Figure 2, where the key 40 is being inserted. We only need to preserve the 'live' entries for further updates and `split_insert` creates one or two new nodes based on the number of live entries present. Note that setting the end version (lines 41–42) is the only change made to the existing leaf node. This ensures that older data versions are not affected by failures. In this case, two new nodes are created at the end of the split.

The inner nodes are now updated with links to the newly created leaf nodes and the parent entries of the now-dead nodes are also marked as dead. A similar procedure is followed for inserting entries into the inner nodes. When the root node of a tree overflows, we split the node using the `split_insert` function and create one or two new nodes. We then create a new root node with links to the old root and to the newly created split-nodes. The pointer to the root node is updated atomically to ensure safety.

Once all the changes have been `flushed` to persistent storage, the current consistent version is update atomically (lines 18–19). At this point, the update has been successfully committed to the NVBM and failures will not result in the update being lost.
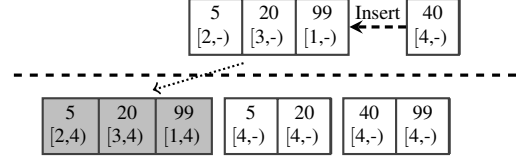

Figure 2: CDDS node split during insertion

Algorithm 3: CDDS B-Tree Deletion

Input: k: key, r: root

```
 1  begin delete(k, r)
 2      v ← current_version
 3      v' ← v + 1
        // Recurse to leaf node (n)
 4      y ← find_entry(n, k)
 5      n[y].end ← v'
 6      l ← num_live_entries(n)
 7      if l = m_l then                        // Underflow
 8          s ← pick_sibling(n)
 9          l_s ← num_live_entries(s)
10          if l_s > 3 × m_l then
11              copy_from_sibling(n, s, v')
12          else merge_with_sibling(n, s, v')
            // Update inner nodes
13      else flush(n[y])
14      current_version ← v'
15      flush(current_version)
16  end
17  begin merge_with_sibling(n, s, v)
18      y ← get_num_entries(s)
19      if y < 4 × m_l then
20          for i = 1 to m_l do
21              insert(s, n[i].key, v)
22              n[i].end ← v
23      else
24          nn ← new_node
25          l_s ← num_live_entries(s)
26          for i = 1 to l_s do
27              insert(nn, s[i].key, v)
28              s[i].end ← v
29          for i = 1 to m_l do
30              insert(nn, n[i].key, v)
31              n[i].end ← v
32          flush(nn)
33      flush(n, s)
34  end
35  begin copy_from_sibling(n, s, v)
        // Omitted for brevity
36  end
```

3.3.3 Deletion

Deleting an entry is conceptually simple as it simply involves setting the end version number for the given key. It does not require deleting any data as that is handled by GC. However, in order to bound the number of live blocks in the B-Tree and improve space utilization, we shift live entries if the number of live entries per node reaches m_l, a threshold defined in Section 3.3.6. The only exception is the root node as, due to a lack of siblings, shifting within the same level is not feasible. However,
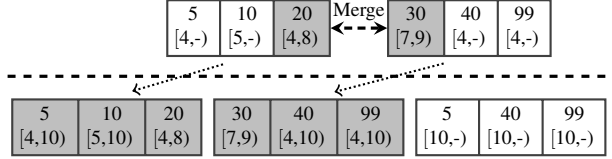
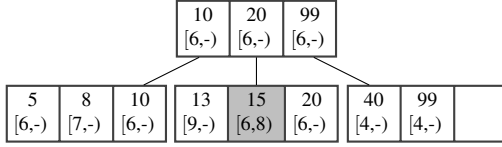Figure 3: CDDS node merge during deletion

Figure 4: CDDS B-Tree after Garbage Collection

as described in Section 3.3.4, if the root only contains one live entry, the child will be promoted.

As shown in Algorithm 3, we first check if the sibling has at least $3 \times m_l$ live entries and, if so, we copy m_l live entries from the sibling to form a new node. As the leaf has m_l live entries, the new node will have $2 \times m_l$ live entries. If that is not the case, we check if the sibling has enough space to copy the live entries. Otherwise, as shown in Figure 3, we merge the two nodes to create a new node containing the live entries from the leaf and sibling nodes. The number of live entries in the new node will be $\geq 2 \times m_l$. The inner nodes are updated with pointers to the newly created nodes and, after the changes have been flushed to persistent memory, the current consistent version is incremented.

3.3.4 Garbage Collection

As shown in Section 3.3.3, the size of the B-Tree does not decrease when keys are deleted and can increase due to the creation of new nodes. To reduce the space overhead, we therefore use a periodic GC procedure, currently implemented using a mark-and-sweep garbage collector [8]. The GC procedure first selects the latest version number that can be safely garbage collected. It then starts from the root of the B-Tree and deletes nodes which contain dead and unreferenced entries by invalidating the parent pointer to the deleted node. If the root node contains only one live entry after garbage collection, the child pointed to by the entry is promoted. This helps reduce the height of the B-Tree. As seen in the transformation of Figure 1 to the reduced-height tree shown in Figure 4, only live nodes are present after GC.

3.3.5 Failure Recovery

The recovery procedure for the B-Tree is similar to garbage collection. In this case, nodes newer than the more recent consistent version are removed and older nodes are recursively analyzed for partial updates. The recovery function performs a physical 'undo' of these

updates and ensures that the tree is physically and logically identical to the most recent consistent version. While our current recovery implementation scans the entire data structure, the recovery process is fast as it operates at memory bandwidth speeds and only needs to verify CDDS metadata.

3.3.6 Space Analysis

In the CDDS B-Tree, space utilization can be characterized by the number of live blocks required to store N key-value pairs. Since the values are only stored in the leaf nodes, we analyze the maximum number of live leaf nodes present in the tree. In the CDDS B-Tree, a new node is created by an insert or delete operation. As described in Sections 3.3.2 and 3.3.3, the minimum number of live entries in new nodes is $2 \times m_l$.

When the number of live entries in a node reaches m_l, it is either merged with a sibling node or its live entries are copied to a new node. Hence, the number of live entries in a node is $> m_l$. Therefore, in a B-Tree with N live keys, the maximum number of live leaf nodes is bound by $O(\frac{N}{m_l})$. Choosing m_l as $\frac{k}{5}$, where k is the size of a B-Tree node, the maximum number of live leaf nodes is $O(\frac{5N}{k})$.

For each live leaf node, there is a corresponding entry in the parent node. Since the number of live entries in an inner node is also $> m_l$, the number of parent nodes required is $O\left(\frac{\frac{5N}{k}}{m_l}\right) = O(\frac{N}{(\frac{k}{5})^2})$. Extending this, we can see that the height of the CDDS B-Tree is bound by $O(log_{\frac{k}{5}} N)$. This also bounds the time for all B-Tree operations.

3.4 CDDS Discussion

Apart from the CDDS B-Tree operations described above, the implementation also supports additional features including iterators and range scans. We believe that CDDS versioning also lends itself to other powerful features such as instant snapshots, rollback for programmer recovery, and integrated NVBM wear-leveling. We hope to explore these issues in our future work.

We also do not anticipate the design of a CDDS preventing the implementation of different concurrency schemes. Our current CDDS B-Tree implementation uses a multiple-reader, single-writer model. However, the use of versioning lends itself to more complex concurrency control schemes including multi-version concurrency control (MVCC) [6]. While beyond the scope of this paper, exploring different concurrency control schemes for CDDSs is a part of our future work.

CDDS-based systems currently depend on virtual memory mechanisms to provide fault-isolation and like

other services, it depends on the OS for safety. Therefore, while unlikely, placing NVBM on the memory bus can expose it to accidental writes from rogue DMAs. In contrast, the narrow traditional block device interface makes it harder to accidentally corrupt data. We believe that hardware memory protection, similar to IOMMUs, will be required to address this problem. Given that we map data into an application's address space, stray writes from a buggy application could also destroy data. While this is no different from current applications that `mmap` their data, we are developing lightweight persistent heaps that use virtual memory protection with a RVM-style API [43] to provide improved data safety.

Finally, apart from multi-version data structures [4, 50], CDDSs have also been influenced by Persistent Data Structures (PDSs) [17]. The "Persistent" in PDS does not actually denote durability on persistent storage but, instead, represents immutable data structures where an update always yields a new data structure copy and never modifies previous versions. The CDDS B-Tree presented above is a weakened form of semi-persistent data structures. We modify previous versions of the data structure for efficiency but are guaranteed to recover from failure and rollback to a consistent state. However, the PDS concepts are applicable, in theory, to all linked data structures. Using PDS-style techniques, we have implemented a proof-of-concept CDDS hash table and, as evidenced by previous work for functional programming languages [35], we are confident that CDDS versioning techniques can be extended to a wide range of data structures.

3.5 Tembo: A CDDS Key-Value Store

We created Tembo, a CDDS Key-Value (KV) store, to evaluate the effectiveness of a CDDS-based data store. The system involves the integration of the CDDS-based B-Tree described in Section 3.3 into Redis [41], a widely used event-driven KV store. As our contribution is not based around the design of this KV system, we only briefly describe Tembo in this section. As shown in Section 4.4, the integration effort was minor and leads us to believe that retrofitting CDDS into existing applications will be straightforward.

The base architecture of Redis is well suited for a CDDS as it retains the entire data set in RAM. This also allows an unmodified Redis to serve as an appropriate performance baseline. While persistence in the original system was provided by a write-ahead append-only log, this is eliminated in Tembo because of the CDDS B-Tree integration. For fault-tolerance, Tembo provides master-slave replication with support for hierarchical replication trees where a slave can act as the master for other repli-
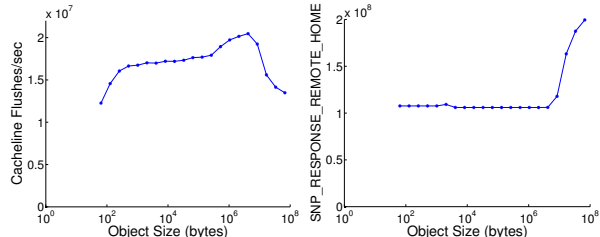

Figure 5: Flushes/second   Figure 6: Cache Snooping

cas. Consistent hashing [27] is used by client libraries to distribute data in a Tembo cluster.
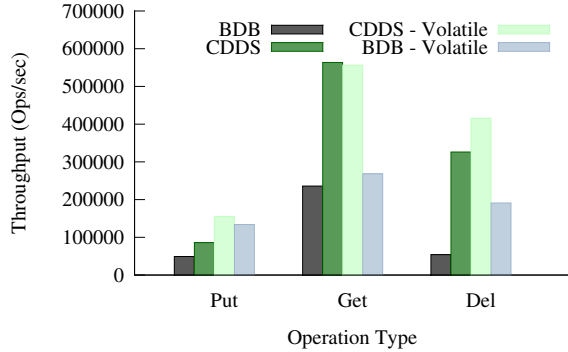
4 Evaluation

In this section, we evaluate our design choices in building Consistent and Durable Data Structures. First, we measure the overhead associated with techniques used to achieve durability on existing processors. We then compare the CDDS B-tree to Berkeley DB and against log-based schemes. After briefly discussing CDDS implementation and integration complexity, we present results from a multi-node distributed experiment where we use the Yahoo Cloud Serving Benchmark (YCSB) [15].

4.1 Evaluation Setup

As NVBM is not commercially available yet, we used DRAM-based servers. While others [14] have shown that DRAM-based results are a good predictor of NVBM performance, as a part of our ongoing work, we aim to run micro-architectural simulations to confirm this within the context of our work. Our testbed consisted of 15 servers with two Intel Xeon Quad-Core 2.67 GHz (X5550) processors and 48 GB RAM each. The machines were connected via a full-bisection Gigabit Ethernet network. Each processor has 128 KB L1, 256 KB L2, and 8 MB L3 caches. While each server contained 8 300 GB 10K SAS drives, unless specified, all experiments were run directly on RAM or on a ramdisk. We used the Ubuntu 10.04 Linux distribution and the 2.6.32-24 64-bit kernel.

4.2 Flush Performance

To accurately capture the performance of the `flush` operation defined in Section 3.1, we used the "Mult-CallFlushLRU" methodology [53]. The experiment allocates 64 MB of memory and subdivides it into equally-sized cache-aligned objects. Object sizes ranged from 64 bytes to 64 MB. We write to every cache line in an object, `flush` the entire object, and then repeat the process with the next object. For improved timing accuracy, we stride over the memory region multiple times.

Mean of 5 trials. Max. standard deviation: 2.2% of the mean.

Figure 7: Berkeley DB Comparison

	Lines of Code
Original STX B-Tree	2,110
CDDS Modifications	1,902
Redis (v2.0.0-rc4)	18,539
Tembo Modifications	321

Table 2: Lines of Code Modified

Remembering that each `flush` is a number of `clflush`es bracketed by `mfence`s on both sides, Figure 5 shows the number of `clflush`es executed per second. Flushing small objects sees the worst performance (∼12M cacheline flushes/sec for 64 byte objects). For larger objects (256 bytes–8 MB), the performance ranges from ∼16M–20M cacheline flushes/sec.

We also observed an unexpected drop in performance for large objects (>8 MB). Our analysis showed that this was due to the cache coherency protocol. Large objects are likely to be evicted from the L3 cache before they are explicitly flushed. A subsequent `clflush` would miss in the local cache and cause a high-latency "snoop" request that checks the second off-socket processor for the given cache line. As measured by the UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE performance counter, seen in Figure 6, the second socket shows a corresponding spike in requests for cache lines that it does not contain. To verify this, we physically removed a processor and observed that the anomaly disappeared[4]. Further, as we could not replicate this slowdown on AMD platforms, we believe that cache-coherency protocol modifications can address this anomaly.

Overall, the results show that we can `flush` 0.72–1.19 GB/s on current processors. For applications without networking, Section 4.3 shows that future hardware support can help but applications using `flush` can still outperform applications that use file system `sync` calls. Distributed applications are more likely to encounter network bottlenecks before `flush` becomes an overhead.

4.3   API Microbenchmarks

This section compares the CDDS B-Tree performance for puts, gets, and deletes to Berkeley DB's (BDB) B-Tree implementation [36]. For this experiment, we insert, fetch, and then delete 1 million key-value tuples

into each system. After each operation, we flush the CPU cache to eliminate any variance due to cache contents. Keys and values are 25 and 2048 bytes large. The single-threaded benchmark driver runs in the same address space as BDB and CDDS. BDB's cache size was set to 8 GB and could hold the entire data set in memory. Further, we configure BDB to maintain its log files on an in-memory partition.
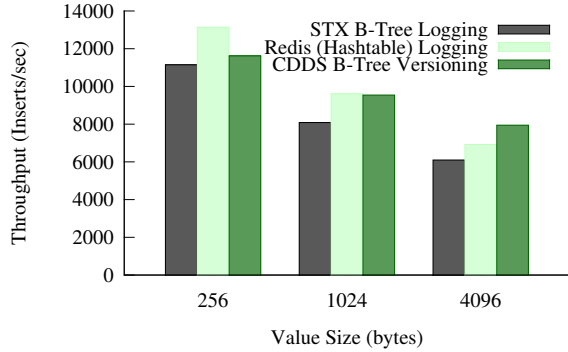
We run both CDDS and BDB (v4.8) in durable and volatile modes. For BDB volatile mode, we turn transactions and logging off. For CDDS volatile mode, we turn `flush`ing off. Both systems in volatile mode can lose or corrupt data and would not be used where durability is required. We only present the volatile results to highlight predicted performance if hardware support was available and to discuss CDDS design tradeoffs.

The results, displayed in Figure 7, show that, for memory-backed BDB in durable mode, the CDDS B-Tree improves throughout by 74%, 138%, and 503% for puts, gets, and deletes respectively. These gains come from not using a log (extra writes) or the file system interface (system call overhead). CDDS delete improvement is larger than puts and gets because we do not delete data immediately but simply mark it as dead and use GC to free unreferenced memory. In results not presented here, reducing the value size, and therefore the log size, improves BDB performance but CDDS always performs better.

If zero-overhead epoch-based hardware support [14] was available, the CDDS volatile numbers show that performance of puts and deletes would increase by 80% and 27% as `flush`es would never be on the critical path. We do not observe any significant change for gets as the only difference between the volatile and durable CDDS is that the `flush` operations are converted into a noop.

We also notice that while volatile BDB throughput is lower than durable CDDS for gets and dels by 52% and 41%, it is higher by 56% for puts. Puts are slower for the CDDS B-Tree because of the work required to maintain key ordering (described in Section 3.3.1), GC overhead, and a slightly higher height due to nodes with a mixture of live and dead entries. Volatile BDB throughput is also higher than durable BDB but lower than volatile CDDS for all operations.

Finally, to measure versioning overhead, we compared the volatile CDDS B-Tree to a normal B-Tree [7]. While not presented in Figure 7, volatile CDDS's performance

---

[4]We did not have physical access to the experimental testbed and ran the processor removal experiment on a different dual-socket Intel Xeon (X5570) machine.

Mean of 5 trials. Max. standard deviation: 6.7% of the mean.

Figure 8: Versioning vs. Logging

Mean of 5 trials. Max. standard deviation: 7.8% of the mean.

Figure 9: YCSB: SessionStore

was lower than the in-memory B-Tree by 24%, 13%, and 39% for puts, gets, and dels. This difference is similar to other performance-optimized versioned B-trees [45].
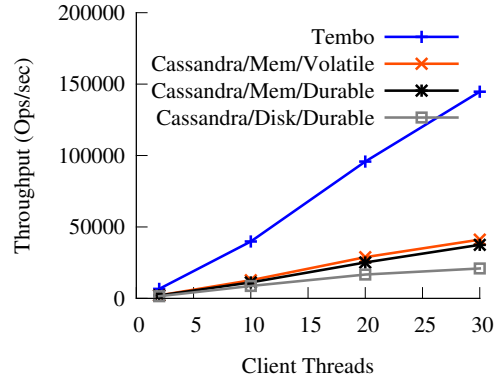
4.4 Implementation Effort

The CDDS B-Tree started with the STX C++ B-Tree [7] implementation but, as measured by `sloccount` and shown in Table 2, the addition of versioning and NVBM durability replaced 90% of the code. While the API remained the same, the internal implementation differs substantially. The integration with Redis to create Tembo was simpler and only changed 1.7% of code and took less than a day to integrate. Since the CDDS B-Tree implements an interface similar to an STL Sorted Container, we believe that integration with other systems should also be simple. Overall, our experiences show that while the initial implementation complexity is moderately high, this only needs to be done once for a given data structure. The subsequent integration into legacy or new systems is straightforward.

4.5 Tembo Versioning vs. Redis Logging

Apart from the B-Tree specific logging performed by BDB in Section 4.3, we also wanted to compare CDDS versioning when integrated into Tembo to the write-ahead log used by Redis in fully-durable mode. Redis uses a hashtable and, as it is hard to compare hashtables and tree-based data structures, we also replaced the hashtable with the STX B-Tree. In this single-node experiment, we used 6 Tembo or Redis data stores and 2 clients[5]. The write-ahead log for the Redis server was stored on an in-memory partition mounted as `tmpfs` and did not use the hard disk. Each client performed 1M inserts over the loopback interface.

The results, presented in Figure 8, show that as the value size is increased, Tembo performs up to 30% better
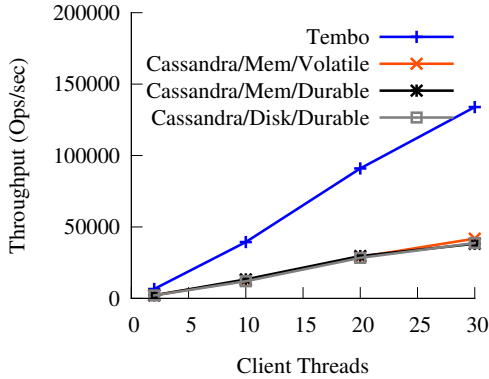
than Redis integrated with the STX B-Tree. While Redis updates the in-memory data copy and also writes to the append-only log, Tembo only updates a single copy. While hashtable-based Redis is faster than Tembo for 256 byte values because of faster lookups, even with the disadvantage of a tree-based structure, Tembo's performance is almost equivalent for 1 KB values and is 15% faster for 4 KB values.

The results presented in this section are lower than the improvements in Section 4.3 because of network latency overhead. The `fsync` implementation in `tmpfs` also does not explicitly flush modified cache lines to memory and is therefore biased against Tembo. We are working on modifications to the file system that will enable a fairer comparison. Finally, some of the overhead is due to maintaining ordering properties in the CDDS-based B-Tree to support range scans - a feature not used in the current implementation of Tembo.

4.6 End-to-End Comparison

For an end-to-end test, we used YCSB, a framework for evaluating the performance of Key-Value, NoSQL, and cloud storage systems [15]. In this experiment, we used 13 servers for the cluster and 2 servers as the clients. We extended YCSB to support Tembo, and present results from two of YCSB's workloads. Workload-A, referred to as SessionStore in this section, contains a 50:50 read:update mix and is representative of tracking recent actions in an online user's session. Workload-D, referred to as StatusUpdates, has a 95:5 read:insert mix. It represents people updating their online status (e.g., Twitter tweets or Facebook wall updates) and other users reading them. Both workloads execute 2M operations on values consisting of 10 columns with 100 byte fields.

We compare Tembo to Cassandra (v0.6.1) [29], a distributed data store that borrows concepts from BigTable [10] and Dynamo [16]. We used three different Cassandra configurations in this experiment. The

---

[5]Being event-driven, both Redis and Tembo are single-threaded. Therefore one data store (or client) is run per core in this experiment.

Mean of 5 trials. Max. standard deviation: 8.1% of the mean.

Figure 10: YCSB: StatusUpdates

first two used a ramdisk for storage but the first (Cassandra/Mem/Durable) flushed its commit log before every update while the second (Cassandra/Mem/Volatile) only flushed the log every 10 seconds. For completeness, we also configured Cassandra to use a disk as the backing store (Cassandra/Disk/Durable).

Figure 9 presents the aggregate throughput for the SessionStore benchmark. With 30 client threads, Tembo's throughput was 286% higher than memory-backed durable Cassandra. Given Tembo and Cassandra's different design and implementation choices, the experiment shows the overheads of Cassandra's in-memory "memtables," on-disk "SSTables," and a write-ahead log, vs. Tembo's single-level store. Disk-backed Cassandra's throughput was only 22–44% lower than the memory-backed durable configuration. The large number of disks in our experimental setup and a 512 MB battery-backed disk controller cache were responsible for this better-than-expected disk performance. On a different machine with fewer disks and a smaller controller cache, disk-backed Cassandra bottlenecked with 10 client threads.

Figure 10 shows that, for the StatusUpdates workload, Tembo's throughput is up to 250% higher than memory-backed durable Cassandra. Tembo's improvement is slightly lower than the SessionStore benchmark because StatusUpdates insert operations update all 10 columns for each value, while the SessionStore only selects one random column to update. Finally, as the entire data set can be cached in memory and inserts represent only 5% of this workload, the different Cassandra configurations have similar performance.

5 Conclusion and Future Work

Given the impending shift to non-volatile byte-addressable memory, this work has presented Consistent and Durable Data Structures (CDDSs), an architecture that, without processor modifications, allows for the cre-

ation of log-less storage systems on NVBM. Our results show that redesigning systems to support single-level data stores will be critical in meeting the high-throughput requirements of emerging applications.

We are currently also working on extending this work in a number of directions. First, we plan on leveraging the inbuilt CDDS versioning to support multi-version concurrency control. We also aim to explore the use of relaxed consistency to further optimize performance as well as integration with virtual memory to provide better safety against stray application writes. Finally, we are investigating the integration of CDDS versioning and wear-leveling for better performance.

References

[1] The data deluge. *The Economist*, 394(8671):11, Feb. 2010.

[2] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large cams for high performance data-intensive networked systems. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI '10)*, pages 433–448, San Jose, CA, Apr. 2010.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, pages 1–14, Big Sky, MT, Oct. 2009.

[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multi-version b-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[5] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally,

M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008. DARPA IPTO, ExaScale Computing Study, `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECS_reports.htm`.

[6] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[7] T. Bingmann. STX B+ Tree, Sept. 2008. `http://idlebox.net/2007/stx-btree/`.

[8] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practices and Experience*, 18(9):807–820, 1988.

[9] R. Cattell. High performance data stores. `http://www.cattell.net/datastores/Datastores.pdf`, Apr. 2010.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[11] P. M. Chen, W. T. Ng, S. Chandra, C. M. Aycock, G. Rajamani, and D. E. Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 74–83, Cambridge, MA, Oct. 1996.

[12] J. Coburn, A. Caulfield, L. Grupp, A. Akel, and S. Swanson. NVTM: A transactional interface for next-generation non-volatile memories. Technical Report CS2009-0948, University of California, San Diego, Sept. 2009.

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

[14] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, MT, Oct. 2009.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154, Indianapolis, IN, June 2010.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pages 205–220, Stevenson, WA, 2007.

[17] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[18] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[19] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4):439–447, 2008.

[20] FusionIO, Sept. 2010. `http://www.fusionio.com/`.

[21] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37:138–163, June 2005.

[22] Hewlett-Packard Development Company. HP Collaborates with Hynix to Bring the Memristor to Market in Next-generation Memory, Aug. 2010. `http://www.hp.com/hpinfo/newsroom/press/2010/100831c.html`.

[23] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, San Francisco, California, 1994.

[24] B. Holden. Latency comparison between hypertransport and pci-express in communications systems. Whitepaper, Nov. 2006.

[25] International Technology Roadmap for Semiconductors, 2009. `http://www.itrs.net/Links/2009ITRS/Home2009.htm`.

[26] International Technology Roadmap for Semiconductors: Process integration, Devices, and Structures, 2007. `http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_PIDS.pdf`.

[27] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on*

Theory of Computing (STOC '97), pages 654–663, El Paso, TX, 1997.

[28] M. Kwiatkowski. memcache@facebook, Apr. 2010. QCon Beijing 2010 Enterprise Software Development Conference. http://www.qconbeijing.com/download/marc-facebook.pdf.

[29] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[30] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pages 2–13, Austin, TX, 2009.

[31] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*, pages 1303–1306, Washington, DC, USA, Apr. 2009.

[32] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 92–101, St. Malo, France, Oct. 1997.

[33] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM Journal of Research and Development*, 46(2-3):187–212, 2002.

[34] W. Mueller, G. Aichmayr, W. Bergner, E. Erben, T. Hecht, C. Kapteyn, A. Kersch, S. Kudelka, F. Lau, J. Luetzen, A. Orth, J. Nuetzel, T. Schloesser, A. Scholz, U. Schroeder, A. Sieck, A. Spitzer, M. Strasser, P.-F. Wang, S. Wege, and R. Weis. Challenges for the DRAM cell scaling to 40nm. In *IEEE International Electron Devices Meeting*, pages 339–342, May 2005.

[35] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, July 1999. ISBN 0521663504.

[36] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–191, Monterey, CA, June 1999.

[37] Oracle Corporation. BTRFS, June 2009. http://btrfs.wiki.kernel.org.

[38] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43:92–105, January 2010.

[39] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, pages 24–33, Austin, TX, June 2009.

[40] S. Raoux, G. W. Burr., M. J. Breitwisch., C. T. Rettner., Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. Lam. Phase-change random access memory: a scalable technology. *IBM Journal of Research and Development*, 52(4):465–479, 2008.

[41] Redis, Sept. 2010. http://code.google.com/p/redis/.

[42] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3:2:1–2:27, February 2008.

[43] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994.

[44] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, Ottawa, Canada, Aug. 1995.

[45] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, San Francisco, CA, Mar. 2003.

[46] Spansion, Inc. Using spansion ecoram to improve tco and power consumption in internet data centers, 2008. http://www.spansion.com/jp/About/Documents/Spansion_EcoRAM_Architecture_J.pdf.

[47] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (its time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 1150–1160, Vienna, Austria, Sept. 2007.

[48] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, May 2008.

[49] Sun Microsystems. ZFS, Nov. 2005. `http://www.opensolaris.org/os/community/zfs/`.

[50] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[51] S. Venkataraman and N. Tolia. Consistent and durable data structures for non-volatile byte-addressable memory. Technical Report HPL-2010-110, HP Labs, Palo Alto, CA, Sept. 2010.

[52] VoltDB, Sept. 2010. `http://www.voltdb.com/`.

[53] R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software – Practice and Experience*, 38(15): 1621–1642, 2008.

[54] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 86–97, San Jose, CA, Oct. 1994.

[55] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 14–23, Austin, TX, June 2009.