Software Systems for Advanced Memory Technologies

Yihe Huang

Harvard University

yihehuang@g.harvard.edu

Abstract

Emerging trends in memory technologies could change how software storage systems are built. Traditionally, software storage systems rely on slow and block-based persistent storage devices for capacity and durability, and fast but scarce volatile memory is used as a caching layer for performance. With storage devices not catching up with the performance improvements in other aspects of computer systems over the years, smart use of main memory becomes increasingly crucial for performance. Much complexity has arisen from the need to use memory more efficiently. But that trend is bound to change with advances in memory technologies. Emerging non-volatile random access memory (NVRAM) devices provide high-performance byte-addressable persistent storage right from the memory bus, with both capacity and endurance surpassing today's flash drives. Memory is going to become cheap, abundant, and even persistent, and that fundamentally changes the way computer systems use memory.

Today we present two systems that adapt and extend existing software systems to make use of advanced memory technologies. First, we look at adapting a modern state-of-the-art concurrent in-memory data structure to be completely NVRAM-resident. We took Masstree, a highly concurrent in-memory key-value store designed for volatile DRAM, and ported it to work in NVRAM with crash consistency. We used novel shadow paging techniques at internal nodes to avoid write-ahead logging, transforming Masstree into an NVRAM-ready data structure with very little space and performance overhead. Our NVRAM Masstree allows an in-memory storage system to achieve high performance and durability without slow accesses to flash or disk. Second, we use the power of abundant or even persistent memory to provide a powerful programming construct: in-memory transactional snapshots of concurrent data structures. We extended a type-based software transactional memory library STO to support snapshots for STO data types. We also implemented snapshot using the new extension for one STO data type – the linked list. STO-Snapshot supports near-instant snapshot-taking with negligible impact on read/write performance.

1. Introduction

Emerging trends in memory technologies have the potential to change the way storage systems are built. Today's storage systems increasingly rely on efficient use of main memory to overcome the bottleneck of slow access to flash or disk, and much complexity has arisen in these systems as a result. Despite shifting more data and organizing them in a more complex manner in main memory, the way computer systems treat main memory hasn't changed for decades. Main memory has always been used as a scarce scratch space to store temporary objects that are frequently accessed, and it is assumed to be volatile in nature. Emergence of NVRAM is bound to change all that: memory is going to be abundant, and it can even be non-volatile.

Adapting modern systems to make use of NVRAM technology raises several challenges. The "volatility" of main memory may seem like an undesired side-effect, but it's actually the basis of most computer programs. Volatility grants a program the right to "start fresh" every time it launches, and not to worry about keeping a consistent layout of its in-memory data before exiting. As a result, directly porting computer programs that deal with volatile data to work in NVRAM does not give us a durable system – they do not treat memory as persistent and they won't recover from failures.

Computer programs that explicitly deal with persistence, e.g. a file system, are not ready to take advantage of NVRAM technology either. These systems assume a block-based underlying storage device, and that it can only be reached via separate I/O buses. Some of them also assume the storage device intrinsically supports atomic block-level accesses. NVRAM provides none of these, so a block emulation layer would be necessary for a traditional storage system to work out-of-the-box on top of NVRAM. Studies in [3] show that such an unnecessary emulation layer comes with substantial cost: block-based systems, such as NTFS, perform as much as 5x worse in RAM than a system that's designed to work with byte-addressable memory.

The opportunities NVRAM presents, however, are too huge to ignore. NVRAM, like volatile DRAM, operates right from the memory bus, saving storage systems expensive accesses to slow I/O storage devices via a separate bus. For the first time, persistence can be achieved by using the CPU's load and store instructions, fundamentally changing the way programs interact with storage. NVRAM is also abundant in storage capacity, potentially making the scarcity of memory resources a thing of the past. The abundant and persistent memory enables new programming abstractions that present memory as an advanced service, like a database, instead of a scratch space.

We present systems that adapt and extend existing software systems to make use of these advanced memory technologies. We first adapt a highly-optimized concurrent key-value store data structure, Masstree [11], to work in NVRAM, such that it becomes resilient to power failures during operations. We then extend STO [5], a fast type-based software transactional memory library, to support in-memory transactional snapshots of concurrent data structures. The main contributions of this work are:

- A prototype of Persistent Masstree, suitable to work in NVRAM.
- An in-place shadow paging technique for adapting concurrent data structures to be crash-consistent in NVRAM, without write-ahead logging or slow accesses to flash or disk.
- An extension of the core STO interface to support transactional snapshots on arbitrary data types.
- An extended STO linked list that supports transactional snapshots.

2. Background

2.1 NVRAM and Persistence Across the Memory Bus

Emerging NVRAM technologies like memristor [17] turn main memory into a persistent storage device. Advances in material sciences [10] demonstrate a persistent storage device with 10-nanosecond switching latencies and 10^{12}-cycle write endurance. Such devices can handle workloads in the working-memory space just like volatile DRAM, which, in comparison, is believed to have a similar access latency and 10^{15}-cycle write endurance [8].

Technologies like this sound like a dream-come-true, but direct adaptations of on-disk or in-memory data structures are still challenging. Data structures and software systems designed for traditional block-based storage devices such as disks don't work out-of-the-box on NVRAM, because their correct operation generally relies on atomic block-wise access (both read and write). NVRAM doesn't have the notion of a "block" since it is byte-addressable just as regular memory, breaking the invariants that block-based storage systems rely on. Volatile in-memory data structures also don't work out-of-the-box in NVRAM. Since NVRAM retains data even if power is removed, these data structures may become corrupt and unusable if unexpected power loss occurs during operation.

Further more, adapting these data structures to work with NVRAM requires careful consideration of the underlying architectural artifacts such as caches. NVRAM devices provide persistent storage themselves, but accesses to data stored in NVRAM go through the memory subsystem and the CPU's cache hierarchy. All caches and buffers involved are volatile in nature, so special flush and fence instructions will be required to ensure persistence of certain critical operations before a program can proceed. These instructions may impede the compiler and the run time system's ability to optimize code, and therefore negatively impact performance even without considering the slower access speed of NVRAM when compared to DRAM.

2.2 NVRAM and Abundance of Storage

Apart from providing byte-addressable persistence right from the memory bus, NVRAM also promises much higher storage capacity when compared to today's DRAM. HP is on track to release the world's first 100TB memristor device by 2018 [13], which, if achieved, could fundamentally change the way computer programs interact with memory. At such high capacity, NVRAM could eliminate the need of separate I/O-bus based storage devices and let main memory become a unified hub that houses all data in the system. Programs, on the other hand, no longer have to use memory as a scratch space to temporarily cache data objects in use, but to treat memory more like a traditional database service – it should be easy for programs to retrieve information, update persistent records, and perform ACID transactions all directly within main memory.

2.3 Masstree

Masstree [11] is an in-memory key-value store optimized for scalability. The basic internal data structures that comprise Masstree are B-link trees. Masstree nodes have fan-out of 15, and are cache-line-aligned (256 bytes, or exactly 4 cache lines on most x86 implementations) and pre-fetched to maximize performance.

Concurrency control in Masstree is heavily optimized. Concurrent writers use fine-grained locking to arbitrate access to tree structures, and readers are completely lock-free by using version validations. Readers and writers rely on an 8-byte version value in each node for synchronization.

Four methods make up the core of Masstree: `insert()`, `update()`, `remove()`, and `split()`. Many of these operations take effect atomically by performing Read-Copy Update (RCU) [12]. It is done by modifying part of the data structure in an invisible buffer space, and atomically swap in a pointer that points the reader to the updated copy. RCU is used in Masstree for better concurrency control between reads and writes, it also simplifies crash recovery for these operations. Since RCU takes effect in an atomic step, a system crash can only happen before or after the final atomic step, and in either case the data structure remains consistent. `insert()`, `remove()`, and `update()` all use RCUs and thus work well in NVRAM, requiring only minimum modifications to ensure correct memory ordering.

One operation needs special attention, and that is `split()`. `split()` is actually a special case of `insert()`. A Masstree node "splits" when a key is inserted to a node that is already full and has no room for the new key. After a node splits into two, both of them become children of the previous parent node, thus resulting in another insertion at the parent level. The insertion at the parent level can end up with a split as well, and the process continues until no more splits are required. Hence, `split()` is a complex iterative operation that generates many intermediate states in the data structures. Masstree also overwrites existing data with the intermediate states in-place, without using RCU, and relies on locking and other run time concurrency control to protect the intermediate states from concurrent readers. To make this process recoverable in NVRAM, however, run time concurrency control clearly won't help. We may have to recover a data structure left in an intermediate state back to a consistent one. The current `split()` process makes it extremely difficult, if not impossible, to do so.

2.4 STO Software Transactional Memory

STO [5] is a software transactional memory (STM) [16] library that uses the power of abstraction to provide fast in-memory transactional semantics with very little overhead. STO outperforms traditional word-based STM systems by more than 10x under certain benchmarks while preserving the same benefits of a composable and declarative programming interface for end users. STO even outperforms Silo [18], a state-of-the-art in-memory database, despite being a more general and powerful system.

STO's performance comes from abstractions. While traditional word-based STM systems view memory as untyped words or even bytes, STO tracks the true intentions of transactions by taking into account the actual data types involved. The results are smaller read/write set sizes and many fewer false conflicts. For example, when a binary search tree data structure re-balances in a word-based STM, the internal pointer flipping may invalidate and abort many transactions that touched these pointers. In STO, however, since the re-balancing operation doesn't change the abstract state of the binary search tree, it generates zero read/write transaction set entries, and other concurrent STO transactions can proceed without aborting due to re-balancing.

In order to benefit from STO's type-level reasoning of transactional intentions, programs must use data types from a library of data structures that are designed to work with STO. The STO core library interacts with these data structures via callbacks and a read/write-set manipulation interface known as the "`TItem`". Each data structure internally divides itself into logical segments that maps to the interface exposed to the user, and each segment can be registered via STO's `TItem` interface to receive callback at certain stages of a transaction. For example, in a binary search tree used as a map, a natural choice of a logical segment would be a node containing a key-value pair, since it maps nicely to the key-value put/get interface the user sees. The data structure implementation will ensure that these logical segments are registered with STO properly via the `TItem` interface to indicate the intention of an on-going transaction. Once STO determines that the transaction is safe to commit, it will issue callbacks for every `TItem` in the transaction's write-set so that proper actions can be taken on the corresponding logical segments to finalize the commit stage. STO uses an optimistic 2-phase locking [9] commit protocol to ensure serializability.

2.5 Simple STO Walkthrough

Details about the internals of STO are available in the STO paper. We provide a simple walkthrough example involving the linked list to introduce the background that's important to understand STO-Snapshot, our snapshot extension.

STO's core interface includes 4 callback methods that must be properly implemented by all conforming data structures: `lock()`, `unlock()`, `check()`, and `install()`. Each of them also takes a `TItem` as an argument to identify the logical segment that receives the callback.

In the `find(K key)` method of a linked list, the data type implementation first walks the list to find the node with the specified key, atomically observes the associated value and a version number within the node, and registers the version information with STO by constructing a new `TItem` associated with this node and passing it to STO along with the observed version. If no node with the specified key is found, the data type observes a list-wise version value, again passing it to STO with a corresponding `TItem`. Different `TItems` contain keys that differentiate between different logical segments within the data type. STO will consolidate these `TItems` by keys and store them in the read-set of the transaction.

The `insert(K key, V value)` method updates the linked list. The data type implementation again first walks the list to find the node with the specified key, and depending on the result to either perform an update or insert a new node. In either case, a `TItem` associated with the involved node (the updated one or the newly inserted one) is created and registered with STO along with its "write value". STO data types normally perform lazy updates so updates are not applied (or installed) until commit time. Such `TItems` with associated write values comprise the write set of a transaction.

When a transaction tries to commit, STO will first call `lock()` for all `TItems` in the write set to initiate the 2-phase commit protocol. In the linked list, locking a `TItem` maps naturally to locking the corresponding node. Once all locks are acquired, STO calls `check()` for each `TItem` in the read set of the transaction, and the implementation of the `check()` method should ensure that validation only succeeds if the observed version in the `TItem` is not locked and has not changed since. After all checks succeed, the transaction is guaranteed to commit, and STO will call `install()` methods to apply all buffered changes to the data structure, and finally call `unlock()` for all locked `TItems`.

Note that STO doesn't keep track of the internal details about the data structures, such as the head/tail and all `next` pointers between nodes. The data type implements proper concurrency control independent of STO to maintain its internal structural invariants.

STO assigns each committed transaction a "transaction ID" (TID) from a global TID space. Every time a transaction commits, STO advances a global TID value, and assigns the old value to the transaction as its "commit TID". Most STO data types, when installing updates, will stamp this commit TID on the corresponding logical segments along with the updates. The value is used by STO to efficiently support opacity [4]. Our snapshot extension also uses this value and the TID space to identify snapshots.

3. Related Work and Motivation

3.1 NVRAM for Fast Persistence

Developing software systems for NVRAM has been an active area of research. Most of them focus on using NVRAM as a fast persistent storage device, and adapting disk-based systems to better utilize NVRAM.

BPFS [3] provides insight about the performance benefits that can be gained from a byte-addressable interface versus a clumsy block-oriented interface. It adapts the Microsoft® NTFS file system to make all its data structures NVRAM-resident. It proposed a technique known as "short-circuit" shadow paging, which leverages persistent memory's fine-grained (8-byte) atomic update capability.

Shadow paging is a technique used by block-based storage systems to maintain crash-consistency of their on-disk data structures. It creates "shadow copies" of parts of a data structure that require updates, until a point when a complex multi-step update can be made visible by "activating" the shadow structures in one atomic step (usually a block update representing a pointer swing). Shadow paging makes disk-resident data structures resilient against crashes because of the atomicity of the last step. It guarantees that the update either takes effect in whole, or takes no effect at all.

In a typical block-oriented file system, shadow paging performs copy-on-writes at a block-level. Creating a shadow copy of a block changes the address of the affected block and thus requires updates to upstream pointers. If one of these upstream pointers does not reside in the same block, another copy-on-write is needed to continue shadow paging. This process continues until everything can be contained in one block update, and the sheer block size (often 512 bytes) results in copying an excessive amount of unmodified data. BPFS realized that by using byte-addressable NVRAM, the amount of data copied can be drastically reduced. For example, updating an 8-byte pointer value requires only an atomic 8-byte write in NVRAM, but it requires moving at least 512 bytes of data on traditional disks. While still built on top of a block-oriented file system, short-circuit shadow paging in BPFS greatly reduced the amount of data transferred during the shadow paging process.

BPFS and Persistent Masstree share the same goal of making data structures completely NVRAM-resident and resilient to crashes. BPFS works with file system data structures that were originally designed with persistence in-mind, albeit for disks, while Persistent Masstree works with a complex in-memory data structure originally designed for concurrency and performance, but not persistence. Persistent Masstree also borrows the idea of shadow paging to keep intermediate states invisible. However, regular shadow paging techniques using copy-on-writes do not work well in Masstree, where backward links are common. Backward links are useful for efficient in-memory operations, but as a result changing the identity (address) of a node would invalidate pointers to this node *both* up and down the tree. Doing copy-on-write for every node needs updating easily leads to a cascade of copy-on-writes covering pretty much the entire data structure. Persistent Masstree uses an "in-place" shadow paging technique that doesn't change the identity of the node being updated.

REWIND [2] aims at adapting the techniques used to support durable database transactions in NVRAM. It features write-ahead logging as a systematic approach to support crash-consistent NVRAM-resident data structures. We will show that Persistent Masstree's shadow paging technique actually shares very similar ideas to write-ahead logging, but is completely localized. REWIND, however, requires serializing writes to a global write-ahead log, which translates to lower concurrent performance for updates. Persistent Masstree achieves a level of concurrent read/write performance close to the original Masstree.

BPFS and REWIND see the same problem as we do: traditional software systems are not ready to take advantage of a byte-addressable persistent storage device. However, they both fall short of treating NVRAM as RAM by only attempting to adapt data structures or techniques that have a disk origin. Persistent Masstree chooses a different angle. By allowing a concurrent in-memory data structure to operate completely in NVRAM all by itself, we perceive a way to achieve the "no-compromise" result: having both the performance of a DRAM-resident data structure and the durability of a disk-based storage system.

3.2 NVRAM as Backing Storage for DRAM

Recent works on NVRAM software systems also have focused on using NVRAM as backing store for DRAM, leveraging NVRAM's capacity to provide abundant main memory resources. pVM [7], as an example, extends the operating system's virtual memory system to utilize NVRAM. The extended operating system is able to automatically scale main memory capacity on-demand, and shows 2.5x speedup for memory intensive applications. pVM also provides a simple object store service, and it out-performs state-of-the-

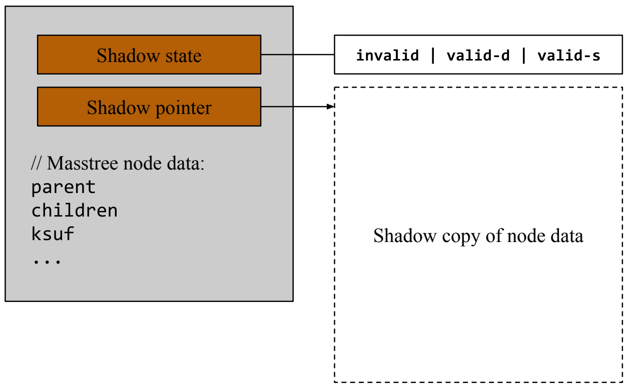

Figure 1. Layout of an augmented Masstree node to support operations in NVRAM. Shadow state can be in one of the three states shown on the right, and when it's in the `invalid` state the shadow pointer does not point to a valid shadow copy.

art block-based solutions by 2x, with 4x less overhead at the operating system level.

We share the same vision as pVM-like systems. As memory becomes abundant, computer systems are more likely to treat memory as a service rather than a simple scratch space. pVM's simple object store service shows what NVRAM-backed memory systems can do, but we want to take it even further. STO-Snapshot builds two more services on top of a NVRAM-backed memory system: serializable transactions and transactional snapshots.

We choose to add snapshot support to a software transactional memory system because it provides a means to offload long-running read-only transactions, without having them limiting the throughput of the system. The goal of the extended software transactional memory system is to provide a declarative and easy-to-use programming interface as a new service, just as those provided by database systems. We also want to provide this service all directly from main memory, and with the same (or even greater) level of performance one would expect from today's in-memory systems that do not have these advanced features.

4. Persistent Masstree

4.1 Overview

Persistent Masstree is a version of Masstree that's designed to work well in NVRAM. It guarantees the recoverability of Masstree's `split()` operation. A light-weight shadow paging technique is used for nodes that undergo splits, and it does not rely on a global write-ahead log. The technique itself, however, does share many similarities with write-ahead logging, except being completely localized and operating at a finer granularity.

We modified Masstree both in data layout and in execution logic to support recoverability in NVRAM. We augment every original Masstree node by 2 words, or 16 bytes: one word for a special "shadow state" field for each node, and the other word for a pointer that optionally points to a shadow copy of the node.

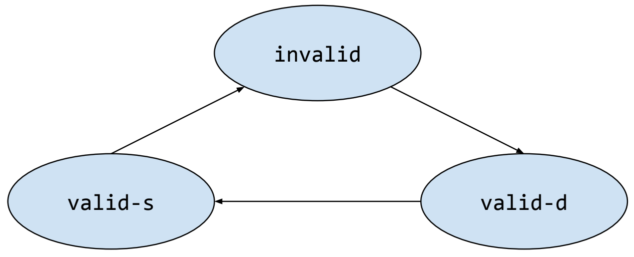

Figure 2. Allowed transitions among shadow states.

State name	Description
`invalid` (`I`)	The shadow copy is currently invalid or contains inconsistent data. Any one other than the active writer shall not try to dereference the shadow pointer.
`valid-d` (`VD`)	The shadow copy is valid and contains consistent data. All information about the on-going split can be found in the shadow copy. Shadow copy is not visible.
`valid-s` (`VS`)	The shadow copy is valid and contains consistent data. The shadow copy can be made visible at any moment (structurally valid).

Table 1. Shadow States

Figure 1 shows the layout of a node in Persistent Masstree. Currently we actually only use two bits in the shadow state field, so we can reduce the space overhead further by combing it with the pointer field. For clarity and simplicity we will treat them as separate words for the rest of the paper.

4.2 Hand-over-hand State Transition

In this section we describe how we make splits recoverable for any unclean shutdowns. As in many other systems, Persistent Masstree uses a shadow copy to store and manipulate intermediate states that are not easily recoverable. Once the data in a shadow copy becomes "consistent" (safe to access even if power fails **now**), the shadow state associated with the shadow copy will be updated. Each shadow state can only be in one of the three allowed states. Table 1 describes the meaning of each state, and Figure 2 is the state transition diagram among the three states.

State transitions are governed in such a way that for a given parent-child node pair, only one node changes its state at a time. The transitions happen at the parent-child pair in a hand-over-hand fashion: for a split at the child, let *pState* and *cState* denote the shadow states at the parent and the child, respectively, then the state pair $[pState, cState]$ transitions in the following manner:

$$[I,I] \rightarrow [I,VD] \rightarrow [VD,VD] \rightarrow [VD,VS] \rightarrow [VD,I]$$

Now we describe in detail when these transitions are allowed to happen. A node starts with the I state, and when it needs to split, a shadow copy of the node will be created, and the shadow pointer in the node is updated to point to the shadow copy. The split operation takes place in the shadow copy, and therefore not visible to concurrent readers. Concurrent writers are locked out of the node because the `split()` operation locks the splitting node. After the split is complete in the shadow copy, the splitting node's state changes from I to VD. At this point, the split becomes an insert to the parent node, which locks the parent. After the lock at the parent is acquired, again a shadow copy is created and the insert (or an additional split, if necessary) takes place in the shadow copy just like before. Once the operation is complete at the shadow copy in the parent, the parent's state changes from I to VD. Since the parent may split, it may be necessary to update the parent pointers in the split child (which are two child nodes, because of the split). The parent updates these parent pointers in the child nodes, changes the child's shadow state from VD to VS. At this point the all information in the child is up-to-date and visible. Note that the valid information is not in the child node itself, but in its shadow copy instead. The VS shadow state indicates that the information in the node itself is obsolete, and anyone looking at the node should go to the shadow copy instead. In the mean time the child node copies information from the shadow copy to the main node, bringing the node itself up-to-date, switching the shadow state back to I, and unlocks.

Note that the shadow copy is only "temporary" and gets deallocated once the split is complete. This keeps the identity of the splitting node unchanged and alleviates pointer updates up or down the tree. We call this technique "in-place" shadow paging due to the fact that it does not change the identity of a node, just like an in-place update.

This process gives us a transition path described above. Note that after the last step, if the parent splits and needs an additional insert to the "grandparent", then the parent essentially becomes the new child, and the state pair becomes equivalent to $[I,VD]$ if we look one level up the tree. This fits seamlessly with the iterative `split()` process implemented by Masstree.

The iterative process comes to an end when there is enough room for an insert and no more splits are required. In this case, no more locks need to be acquired up the tree, and the node undergoing an insert (which should be in state VD) simply switches itself to state VS, brings itself up-to-date by copying from the shadow copy, and finally changes its shadow state back to I before unlocking. Figure 3 describes the complete process in pseudo-code.

```
split(node, new_key):
  node.lock()
  // create the shadow copy
  node.shadow_ptr = new NodeData(node)

  // split the node using the shadow copy
  // returned key is the one to insert to parent
  p_key = do_split(node.shadow_ptr, new_key)

  node.shadow_state = VD

  while true {
    parent = node.parent
    parent.lock()
    parent.shadow_ptr = new NodeData(parent)
    if parent.has_room() {
      do_insert(parent.shadow_ptr, p_key)
      parent.state = VD
      // shadow shate at parent can be elevated to VS
      // since no more splits are needed
      parent.state = VS
      node.state = VS
      node.copy_from_shadow()
      node.state = I
      node.unlock()

      parent.copy_from_shadow()
      parent.unlock()
      return
    }
    // parent also needs splitting
    p_key = do_split(parent.shadow_ptr, p_key)
    parent.state = VD
    node.state = VS
    node.copy_from_shadow()
    node.state = I
    node.unlock()

    // go to the next iteration
    node = parent
  }
```

Figure 3. Pseudo-code of the recoverable `split()` in Persistent Masstree.

4.3 Recovery

The recovery process needs two pieces of information to recover a system stopped in an inconsistent state: 1) identify operations that were incomplete, and 2) all necessary information and data to redo or undo these operations. In a system with write-ahead logging, all these information would be readily available in the log. Persistent Masstree achieves the same by clever use of the shadow states and shadow copies of nodes where updates did not finalize. Persistent Masstree **re-executes** incomplete operations during recovery.

First, the recovery process needs to identify any split operations that were left incomplete when the system stopped.

The shadow state I is the stable state for nodes not undergoing splits, and a shadow state of VD or VS indicates in-progress split operations. Shadow states serve as clear signals for the recovery process to identify incomplete operations.

Second, to re-execute an identified in-complete split, the recovery process needs to know which key to insert to the upper level, if any. One invariant in our system is that for any incomplete split, there will be at least one node left in shadow state VD or VS. Both states mean that the associated shadow copy contains complete information (information that would have made it to the main node if the split completes). By looking at the shadow copy and following B-tree invariants, the recovery process has enough information to infer if an insertion is needed at the next level, and if so which key to insert. In other words, if we borrow terms from Figure 3, shadow copies provide enough information for the recovery process to recover `p_key` for the iteration where the split is stopped. The recovery process can pick right up from where the program was stopped and finish the rest of it by following exactly the same process depicted in Figure 3.

4.4 Memory Fences and Cache Flushes

Although NVRAM devices provide data persistence by themselves, accesses to these devices still go through the CPU's cache hierarchy and buffers in the memory subsystem, which are still volatile. Additional care needs to be taken to ensure these architectural artifacts do not interfere with our durability and consistency goals [1]. Additionally, the compiler, run time systems, and the hardware may optimize code by reordering them, which can be problematic when we work with persistent data. As an example, before a Persietent Masstree insert operation returns, it should make sure that the inserted data safely made it to persistent storage. We don't want caching to weaken this guarantee, and we certainly don't want anything at this critical stage to get reordered.

We use specialized hardware instructions to deal with these issues. Intel's latest Instruction Set Architecture (ISA) extension [6] provides instructions for working with non-volatile memory. The newly introduced `pcommit` instruction can be used to flush cached/buffered updates in the NVRAM domain to persistent storage, and it does so without invalidating cache lines. A regular `mfence` instruction is all we need to prevent reordering. In Persistent Masstree operations that take effect by doing an atomic write (insert/remove), we need an `mfence` and a `pcommit` instruction before the atomic write. For `split()`, we need to do this every time before a shadow state transition.

5. STO-Snapshot

5.1 Overview

STO-Snapshot is an extension to the STO software transactional memory library to support efficient and transactionally consistent snapshots for STO data types. It heavily relies on the STO concept of "logical segments" in data types and implements snapshots using copy-on-write.

STO-Snapshot provides a framework to help data type implementers maintain snapshots for each logical segment of the data type. It also provides public interfaces for an application to take a snapshot of all active transactional data types and to query them at a specified snapshot.

Snapshots taken by STO-Snapshot are system-wide, meaning that they are transactionally consistent for all STO data types that are active in the system. STO-Snapshot identifies snapshots by "snapshot IDs" (or SIDs), which are 64-bit numbers taken from the TID space.

5.2 Interface

The STO-Snapshot interface consists of the user-facing public run time interfaces and a data-type-implementer-facing framework for managing snapshot for the logical segments of a particular data type. Figure 4 describes the public interfaces as an extension to the STO core interface. A user who wants to save the current state of the system as a snapshot simply calls `Sto::take_snapshot()` from outside of a transaction. The method will return a SID that can be used to reference the snapshot being taken at the moment. To access any given snapshot, the user simply starts a new read-only transaction and calls `Sto::set_active_snapshot()` with the proper SID before issuing reads.

SIDs and TIDs share the same space. STO uses monotonically increasing TIDs, which reflect the serialization order of committed transactions. When a user takes a snapshot and gets a SID s, the snapshot reflects the collective effect of all committed transactions with TIDs $(s-1)$ or smaller.

STO-Snapshot also provides a framework to help data type implementations organize snapshot copies. The framework defines a "`Base`" object for each type of logical segment that needs copy-on-write snapshots. It contains a most current version of this segment as well as a history of snapshot copies of the segment. To better support in-memory data structures that heavily use pointers, STO-Snapshot also provides the option for data types to manage pointers separately from semantic data. Snapshots are read-only, so all updates to a logical segment go to the most current version in the `Base` object. To support STO transactions, a compatible version number is also included in the `Base` object. Snapshot

```
namespace Sto:
  // take a snapshot; only to be used outside of a transaction
  sid_type take_snapshot();
  // specify the snapshot to be used this transaction
  // transaction must be read-only to use this method
  void set_active_sid(sid_type sid);
  // return the snapshot being used by the current transaction
  sid_type active_sid();
```

Figure 4. The STO-Snapshot public interface.

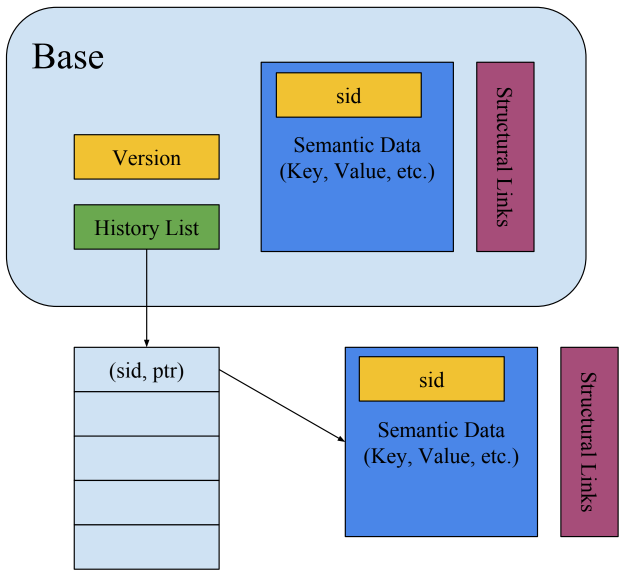

Figure 5. Sto-Snapshot "`Base`" object layout. A `Base` object exists for each data structure logical segment (node, page, element, etc.) that requires copy-on-write snapshots.

reads of a data type may lead to searches in the history list of the logical segments to return the proper data. The layout of a STO-Snapshot `Base` object is shown in Figure 5. To follow this layout and also for explanatory reasons, we will refer to the most current version of a segment in the `Base` object as the "top-level" version in the following sections.

The data type implementation is completely free to use this framework however it wants, but the framework works most naturally when the data type is implemented as a union of all logical segments ever created, snapshots or current. We will elaborate more on this in § 5.5.

5.3 STO-Snapshot in Action

This section gives a high-level overview of how transactional reads and writes interact with the STO-Snapshot framework.

There are two types of transactional reads, snapshot reads and non-snapshot reads. Non-snapshot reads observe the most current state of the data type. They simply ignore the history lists and always access the top-level version of each logical segment. Snapshot reads are more complicated since they may need to traverse the history list. For each STO-Snapshot `Base` object, a snapshot read will look at both the top-level and all versions stored in the history list to find the valid copy. If a copy is found, it will follow the corresponding data in that version to go to the next segment, otherwise it simply skips the segment.

Transactional writes are not allowed to modify an existing snapshot and also only operate at top-level versions. At execution time, transactional writes simply ignore the history list of a `Base` object and apply the same conflict-detection logic as the original STO design.

At commit time, instead of performing in-place updates to the top-level version of the segment, the implementation must check whether this version can be referenced in a valid snapshot. If yes, a copy-on-write operation is required: saving the old version to the history list, creating a new version with the updates applied, and making it the new top-level version.

Deleting or unlinking a logical segment from a data structure should also be treated as an update, as future read-only transactions may still refer to snapshot versions in a recently unlinked segment. When a write tries to delete a segment that's part of a snapshot, it moves the old top-level version into the history list and creates a new version, marks it as "deleted", and makes it the new top-level. Non-snapshot reads will skip segments whose top-level versions are marked as "deleted".

5.4 The Global Snapshot Clock

In the previous section, we briefly mentioned that while an overwriting transaction installs its changes, it should somehow detect if a top-level version of a data structure segment belongs to an existing snapshot. We implement this efficiently by using a Global Snapshot Clock, or GSC, which is updated to the global TID value every time a snapshot is taken. Each data structure segment also keeps track of the last transaction that committed changes to it, by stamping the transaction's commit TID along with the updates. In our design, a version of a data structure segment with a stamped TID t could be referenced by a valid snapshot if and only if $t <$ GSC. We call this check "the GSC test". An overwrite operation must perform a copy-on-write before overwriting a data structure segment that passes the GSC test. It is important that these GSC tests are only performed *after* the overwriting transaction obtains its commit TID. This guarantees that no snapshots are inadvertently overwritten. We demonstrate its correctness in § 5.6.

Snapshot reads identifies snapshot copies of data structures also by looking at the stamped commit TID in each segment. A data structure segment with stamped TID t is the snapshot copy for a given SID s, if and only if t is the largest value that satisfies $t < s$, among all copies both at the top level and in the history list for that segment. Correctness of this design will also be discussed in § 5.6.

The GSC also simplifies the process of taking a snapshot. Taking a snapshot now only involves 1) reading from the global TID value and assigning it to GSC, 2) incrementing the global TID value, and 3) returning the updated GSC, all atomically. This guarantees that no transactions can have commit TIDs that are ever returned as SIDs. No data structure data are copied when we take a snapshot; a few global clock value updates are all we need to capture a transactionally consistent state of the entire system.

Note that updating the GSC requires an atomic read-modify-write operation across two different words (the TID word and the GSC word). This can be achieved using hard-

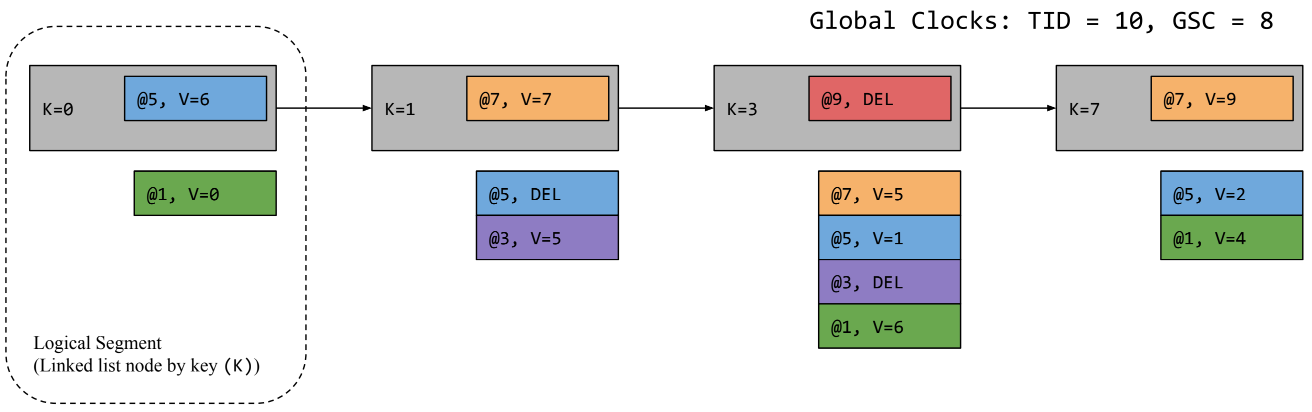

Figure 6. Illustration of a key-value linked list in STO-Snapshot. Grey nodes are `Base` objects, and logical segments are defined by the "keys". In each copy (or version) of a logical segment, and the number following the "`@`" sign is the SID that identifies the transaction that installed an update via copy-on-write. We can see that the `Base` object with key 3 contains a top-level record marked as "deleted", prompting non-snapshot reads and writes to ignore this `Base` object.

ware transactional memory (HTM), such as the Intel® Transactional Synchronization Extensions [14], available in modern processors. Right now, we collocate TID and GSC within a 16-byte struct, and use a 16-byte compare-and-swap loop instead for better compatibility. To make sure that a snapshot seen by the user refers to a set of transactions that are not only committed but also *installed*, `Sto::take_snapshot()` waits for all active transactions in the system to finish before returning.

5.5 Linkage between Base Objects

Most data structures implement linkage between different logical segments via pointers. Correctly managing these pointers with copy-on-write snapshots is a major challenge. A naive design, which treats pointers simply as regular data and copies them also as part of a snapshot, is much harder to work with for complex data structures that contain backwards links, such as a self-balancing binary search tree.

With STO-Snapshot, data type implementations can elegantly address this problem by organizing the internal data structures of the data type as a union of all logical segments that are 1) ever created, and 2) still referenced in snapshots. The logical segments are stored "folded" according to data type semantics, with each semantic logical segment only represented once in the form of a `Base` object, and different versions of it contained in the corresponding history list. `Base` objects link to each other the same way logical segments do in a normal data structure. The resulting design is similar to multi-version concurrency control [15].

We use a linked list example to demonstrate this idea. Figure 6 illustrates what a key-value linked list looks like with snapshots enabled. In this particular example we take a snapshot every time after a transaction commits, except for the most recent one with commit TID 9, which deletes key 3. Copy-on-write installation ensures that all snapshot versions of a logical segment are safely copied to the history list before the updates are applied. For example, when a snapshot read tries to access the value associated with key 3 at a specified SID 4, it will walk the list, from one `Base` object to another, until it reaches the one that represents key 3. It then looks for the valid snapshot copy of that logical segment by comparing the stamped commit TID of each copy with SID 4, and concludes that "`@3`" (the purple one) is the match. The data within that copy, however, suggest that the record was deleted at the time, so the snapshot read shall return a negative (key absent) result.

The linked list illustration in Figure 6 shows that linkages are only maintained by `Base` objects. This can be a useful invariant and keeps programing with STO-Snapshot simple. Nevertheless, the system does give data type implementations the freedom to have direct links between history list entries (directly between colored nodes in the figure). This may not always be possible, however, if the "next" node of a given node depends on the SID specified for a snapshot read. For example, in Figure 6, the green copy (`@1, V=0`) of the logical segment with `K=0` is followed by the green copy (`@1, V=6`) of `K=3` if we query the data structure with SID $= 2$, but followed by the purple copy (`@3, V=5`) of `K=1` instead for SID $= 5$.

5.6 Correctness

STO-Snapshot operates independently of STO's concurrency control and conflict detection logic for non-snapshot reads and transactional writes, so its correctness is based on the following two properties: 1) writes never throw away useful snapshots; 2) snapshot reads return the correct historical version of a data structure segment requested.

Writer correctness Let's say a transactional write is in its commit stage and has obtained commit TID t. It is guaranteed to commit, and is bound to overwrite a data structure segment previously stamped with TID c. The transactional write will directly overwrite it without creating any copies if c fails the GSC test, or GSC $\leqslant c$. We won't have any problems if GSC stays less than or equal to c for the duration of our commit, but what if someone else takes a snapshot in the mean time, and changes the GSC value after we performed the GSC test? Will that be a problem? No. Note that the write performs the GSC test after obtaining TID c, so anyone who takes a snapshot after the GSC test is guaranteed to get a SID that's greater than or equal to $t+1$. Such a SID will refer to the value written by the current write, which stamps TID t to the segment, as the valid snapshot value, so directly overwriting the copy with TID c doesn't throw away useful snapshots. □

Reader correctness Recall that a SID s is defined as referring to the snapshot that's the collective effect of all committed transactions with TIDs $s-1$ or lower. We demonstrate the reader correctness by showing that our reader-side algorithm yields the defined snapshot behavior. We mention in § 5.4 that reader looks for the data structure segment copy with a stamped TID t that's the largest among all available ones while still satisfying $t < s$. The $t < s$ condition ensures that the snapshot read won't return data that too up-to-date, but we also need to make sure it doesn't return stale data. Note that by the time we obtain SID s, all committed transactions with TIDs lower than s must have already finished execution. Given that the writers are correct in the sense that they never throw away useful snapshots, every available snapshot version should be accessible to us. t being the highest qualifying value indicates that, among all transactions with TIDs $s-1$ or lower, t is the last one in the serailization order that modified this logical segment. Hence the updates installed by t reflects the collective effect of all transactions up to $s-1$. □

Other considerations There are other aspects of the implementation that are crucial for correctness. For example, `Sto::take_snapshot()` must not return until all active transactions (at the instant it accesses the TID value) finish executing, otherwise a snapshot read using the prematurely returned SID may miss committed but not-yet-installed transactions. Also, it is the data type's responsibility to make sure that accesses to its internal data structures are synchronized and linearizable, and that deleted segments are not physically unlinked if valid snapshots still reference them. The reader/writer-side correctness provided by the STO-Snapshot framework makes it easier for data type implementers to keep track of such invariants.

6. Evaluation

With functioning designs and implementations, we perform experiments on both Persistent Masstree and STO-Snapshot

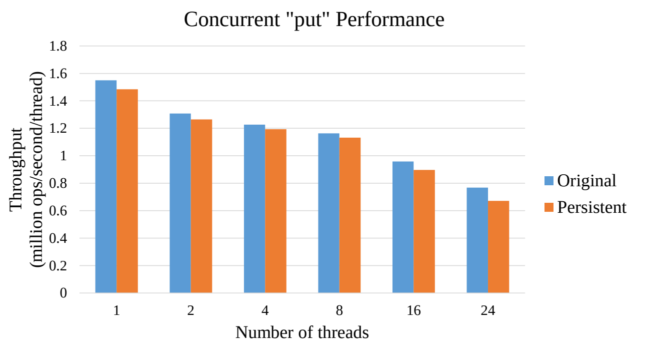

Figure 7. Per-thread throughput comparison for puts.

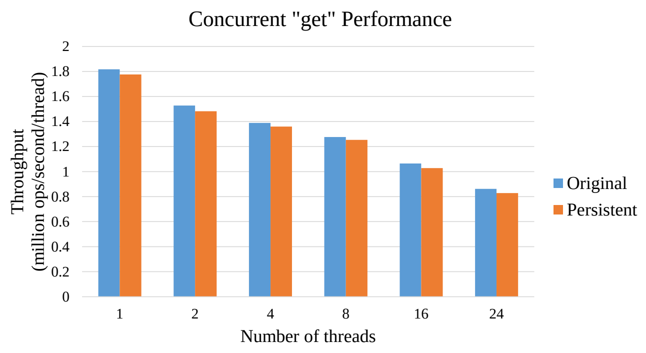

Figure 8. Per-thread throughput comparison for gets.

to evaluate whether they achieve their respective goals. We compile all source code using gcc-5.3, and run all experiment on a server powered by two Intel® Xeon® X5690 processors (12 cores/24 hardware threads in total). Reported results for each experiment are the average of five consecutive runs.

6.1 Persistent Masstree

We measure the overhead added by Persistent Masstree's new `split()` operation as well as its stricter restrictions on memory ordering, by comparing it with the original Masstree. All experiments are conducted in DRAM without simulating the lower access speed of NVRAM. Since NVRAM devices are not physically available yet, the purpose of the experiments is to find out the overhead compared to the original Masstree, rather than the absolute performance of Persistent Masstree in NVRAM.

We test the systems throughput for both reads and updates. We let each system run in full-speed at different concurrent settings for 20 seconds, where the first 10 seconds are "puts" (or inserts) and the second 10 seconds "gets". We measure the per-thread put/get throughput, and results are in Figure 7 and Figure 8.

From the graphs we can tell that both systems demonstrate good scalability. Persistent Masstree's recoverable split operation and memory ordering restrictions does introduce a little overhead, but the overhead seems to be con-

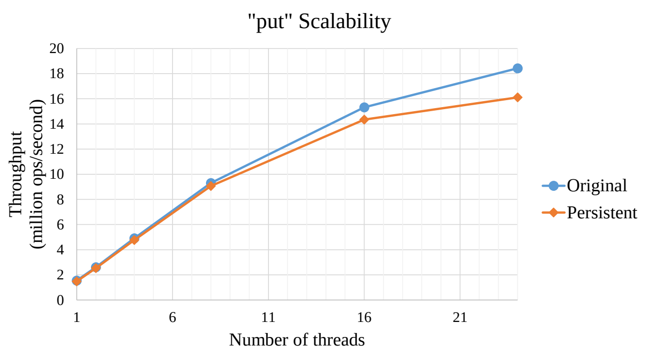

Figure 9. Scalability of the original Masstree and Persistent Masstree under all-put workloads.

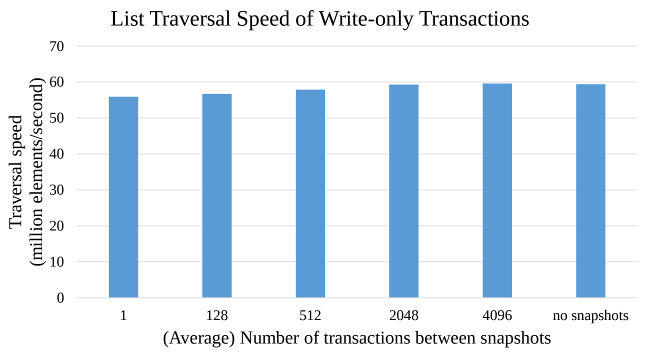

Figure 10. The impact of snapshots on writes, in terms of overall effective list traversal speed. The overhead is mainly introduced by copy-on-write.

stant at all concurrency levels and does not affect overall scalability. Persistent Masstree's `get()` operation involves the additional step of checking shadow states of nodes visited. The augmented nodes are also slightly misaligned with cache lines. These modifications lead to a slight overhead at around 4% for read-only operations, also independent of concurrency levels. The scalability of the two systems with all-put workloads are shown in Figure 9.

6.2 STO-Snapshot Microbenchmarks

In this section, we measure the inherent overhead associated with STO-Snapshot, and evaluate how well it achieves the goal of providing fast in memory transactional snapshots.

We evaluate the overhead of STO-Snapshot by inserting random key-value pairs to a linked list, taking snapshots, and performing random snapshot look-ups. The keys are randomly generated integers from 1 to 2048. We perform transactions repeatedly such that every key is, on average, overwritten approximately 120 times over the course of the benchmark. Before measurements are performed, the linked list is also populated to the full length. The benchmarks are single-threaded to factor out contention.

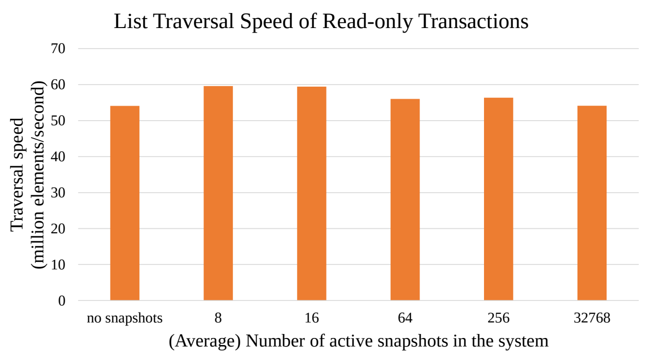

Figure 11. The overhead of snapshot reads. For the run with no snapshots in the system (the left-most bar in the graph), reads are executed as regular transactional reads.

6.2.1 Writer Overhead

We ask several questions for the benchmarks. First, does snapshot support comes with a cost for writers? If so, how much?

We measured the system's performance, in terms of list nodes visited per second, using the linked list. We vary the frequency by which snapshots are taken, and Figure 10 shows the results. When snapshots are taken very often, most transactional installs have to perform copy-on-writes to preserve the old snapshot. This leads to a 5.9% slowdown with respect to the zero-snapshot baseline in the worst case, where a snapshot is taken after every transaction. As the interval between snapshots increases, performance improves quickly, and the copy-on-write overhead soon becomes negligible. In the case where a snapshot is taken (on average) for every 4096 transactions, $\sim$10 snapshots are taken during the 5 seconds the benchmark is executed, and the snapshot-taking run somehow slightly outperforms the zero-snapshot run. We are still trying to investigate this effect, but it shows that copy-on-write at run time incurs a near-zero overhead when data structures are well populated and snapshots are reasonably spaced between each other.

6.2.2 Reader Overhead

The second question we want to answer is how snapshots affect reads. We expect snapshots to have a negative impact on snapshot reads since they involve the additional step of searching the history list. But is the impact really going to be noticeable?

For this set of experiments, we again pre-populate the list to its full length before measurements, but we also vary the number of valid snapshots in those pre-populated lists. The snapshot reads then access "middle-aged" snapshots on the lists to simulate the average case. Figure 11 shows the results.

Compared to writes, regular non-snapshot reads (the left-most bar in Figure 11) are slightly more expensive, since they involve the additional `check()` step. For snapshot reads, however, the elimination of the commit protocol actually makes them faster. Searching the history list does come with a slight cost when we compare the two extreme cases in the graph: performance degrades for 9.17% as the number of active snapshots goes from 8 all the way up to 32768. Most of this performance drop is attributed to our relatively simple history list design, which only does linear searches albeit maintaining data in sorted order. This results in poor performance when the history list grows large. The system shows little slowdown when the number of active snapshots in the system remains small. Snapshot reads also out-performs regular read-only transactions in all measured cases.

6.2.3 Discussions

STO-Snapshot's performance remains relatively stable in the microbenchmarks, but its perceived performance can depend a lot on the actual workload. In our tests we did not stress the linked list with deletes, but too many deletes in conjunction with frequent snapshots will result in many "placeholder nodes" in the linked list just to make the snapshots reachable. This may not affect our microbenchmark results since these placeholder nodes are also considered meaningful "elements", but it does affect the perceived end-to-end performance of a list look up.

It's also worth noting that the current design of STO-Snapshot faces a garbage collection problem – we keep creating snapshots via copy-on-write but never free them. An epoch-based garbage collection design is currently under development. The garbage collection scheme should operate mostly independently of STO-Snapshot with minimal performance impact on the system, but the actual effect of a garbage collector is yet to be measured.

We do not yet have any end-to-end throughput results for STO-Snapshot because the linked list is the only data structure that supports it right now, and the throughput of an $O(n)$ data structure depends on too many variables. We will measure STO-Snapshot's overall performance using a synthetic benchmark (e.g. TPCC) once we have more data structures implemented.

7. Future Directions

Persistent Masstree and STO-Snapshot both show great promise as first steps, and we have future plans for both of them.

For Persistent Masstree, the very immediate next step is to evaluate more on recoverability. We already have a recovery program implemented but we haven't done any fault injection testing yet. We would also like to measure the performance of the system with NVRAM emulation or even physical devices as they become available. With Persistent Masstree as a powerful building block of novel NVRAM-optimized systems, we also plan to design and implement a complete database system with Persistent Masstree itself or its core ideas.

On the STO-Snapshot side, we are currently in-progress in terms of implementing and testing a garbage collection scheme that allows the system to reclaim memory usage, once the user tells the system he/she no longer needs a snapshot. We would also like to apply it to more data structures so that we can more thoroughly test the system with standard benchmarks. We also feel very excited about more powerful applications of STO-Snapshot, such as using it in conjunction with NVRAM's non-volatility to support durable STM transactions.

8. Conclusion

Software system designs always follow the intrinsic properties and limitations of the underlying technology on which the system operates. With advanced memory technologies like NVRAM arriving in the near future, software systems must also adapt to catch up with and take advantage of these technologies. NVRAM provides fast and byte-addressable persistence storage right from the memory bus, rendering traditional block-based storage systems obsolete. NVRAM is also abundant in capacity, making scarce main memory resource a thing of the past. With cheap and abundant memory in our systems, programs can use memory as a service, rather than just a simple scratch space. We adapt and extend two existing systems to answer the calls for change.

In Persistent Masstree, we take a concurrent data structure designed for volatile DRAM and make it an NVRAM-resident crash-resilient data structure. We used novel techniques that borrow ideas from both shadow paging and write-ahead logging, but are optimized for byte-addressable NVRAM. Persistent Masstree achieves durability and crash consistency with only $< 10\%$ performance overhead compared to the original Masstree, at very high concurrency settings.

We also perceive STO-Snapshot, an extension to the STO software transactional memory library to support transactional snapshots of in-memory data structures. It makes use of the abundant storage capacity of NVRAM devices, and makes memory function more like a database service. Based on a fast transactional memory system, STO-Snapshot further simplifies and improves the performance of concurrent programming by providing the means to offload long-running read-only transactions with snapshots, and it does so with very little performance impact on regular reads and writes.

With both systems showing promise as first steps, we will continue to improve and extend the their functionalities and to explore more exciting applications.

Acknowledgements

Many thanks to Professors Eddie Kohler and Margo Seltzer for their input, support, suggestions, and critiques on this work. We would also like to thank Harvard undergrads Nathaniel Herman, Lilian Tsai, and our collaborators from

MIT and other institutions for their excellent work on STO that makes the second half of this work possible.

References

[1] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Implications of CPU caching on byte-addressable non-volatile memory programming, 2012.

[2] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[4] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[5] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the 11th ACM European Conference on Computer Systems*, page 31, 2016.

[6] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, chapter 10. Intel Corporation, feb 2016.

[7] S. Kannan, A. Gavrilovska, and K. Schwan. pvm: persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 13. ACM, 2016.

[8] K. Kim and D. J. Jung. *Future memory technology and ferroelectric memory as an ultimate memory solution*.

[9] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[10] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, S. Seo, et al. A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta_2O_{5-x}/TaO_{2-x} bilayer structures. *Nature materials*, 10(8):625–630, 2011.

[11] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 183–196. ACM, 2012.

[12] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[13] C. Mellor. HP 100 TB memristor drives by 2018 – if you're lucky, admits tech titan, Nov 2013. URL http://www.theregister.co.uk/2013/11/01/hp_memristor_2018/.

[14] R. Rajwar and M. Dixon. Intel transactional synchronization extensions. In *Intel Developer Forum San Francisco*, volume 2012, 2012.

[15] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross. Reducing database locking contention through multi-version concurrency. *Proceedings of the VLDB Endowment*, 7 (13):1331–1342, 2014.

[16] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

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

[18] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.