NVTM: A Transactional Interface for Next-Generation Non-volatile Memories

UCSD CSE Tech Report XXXXXXXXXX

Joel Coburn Adrian Caulfield Laura Grupp Ameen Akel Steven Swanson
The Department of Computer Science and Engineering
University of California, San Diego
{jdcoburn,acaulfie,lgrupp,aakel,swanson}@cs.ucsd.edu

Abstract

Advanced non-volatile, solid-state storage technologies such as phase change memory promise enormous gains in performance relative to both conventional disks and flash memory-based storage devices. However, existing abstractions for accessing non-volatile data (i.e., file systems and the associated system calls) cannot exploit the performance and flexibility these memories offer. We describe a new, transaction-based interface that allows programmers to implement fast, scalable, durable data structures that are robust in the face of unexpected system failures. The interface maps the non-volatile storage directly into the application's address space and allows volatile and non-volatile data structures to interact seamlessly in programs. Since the interface removes the operating system from the critical path of basic read and write operations, it can realize up to 2350$\times$ speedups relative to transactional storage systems that target the conventional file-based interface. We describe the system's internal operation, describe the support it requires from hardware, and quantify its performance.

1 Introduction

The interface that programmers use to access, manipulate, and manage non-volatile data has remained unchanged for a generation. The interface relies on a trusted file system to mediate access to non-volatile storage (traditionally spinning disks) and to provide access to named, untyped arrays of bytes (i.e., files). Since spinning disks are extremely slow, the efficiency of the software that mediates access to them was a secondary concern.

In addition to being slow, the file system interface is also much less flexible than the rich language constructs (classes, types, inheritance, etc.) that systems provide for dealing with volatile data. To bridge the gap between volatile and non-volatile storage, we rely on tools such as parsers, object serialization facilities, and databases.

Until recently, this gap between the semantics of volatile and non-volatile data was necessary, because the technology underlying the two types of data had radically different performance characteristics: Spinning disks are block addressable and have access times measured in 10s of milliseconds, while memory is byte addressable with access times measured in nanoseconds. Consequently, the fast, volatile representations that are most convenient for programmers would perform extremely poorly if mapped onto a hard disk.

This is about to change. In the near future, extremely fast non-volatile, solid-state memories such as phase-change and spin-torque transfer memories will begin to appear in mainstream computing systems. These technologies represent a decrease in latency relative to disk of between two and five orders of magnitude along with dramatic increases in bandwidth. Properly harnessed, these changes will blur the performance distinction between volatile and non-volatile memories.

Fully exploiting these memories will require changes in both the interface that the OS provides to non-volatile data and in how applications access it. The current OS overheads required to access non-volatile storage (i.e., microseconds for IO system calls) will dominate the access time to these memories, squandering the potential performance gains. To harness that performance, we must remove the operating system from common-case read and write operations without sacrificing safety or reliability.

For applications, these new technologies provide the promise of a unified interface to volatile and non-volatile data. With the advent of fast, byte-addressable, non-volatile storage, it should be possible for applications to use familiar tools to build specialized, durable, scalable, high-performance data structures. The challenge is in ensuring that these structures remain reliable even in the face of system crashes, power failures, and unexpected application termination.

```
NVHeap * nv = Open("foo.nvheap");
NVList<int>::Ptr a = nv->GetRoot();
TXBegin {
  TXOpenRead(a);
  while(a->next != NULL) {
    TXOpenRead(a->next);
    if (a->next->value == 42) {
      TXOpenWrite(a);
      a->next = a->next->next;
    }
     a = a->next;
  }
} TXEnd;
nv->Close();
```

Figure 1: **NVTM Example.** A simple NVTM function that atomically removes all links with value 42 from a non-volatile linked list.

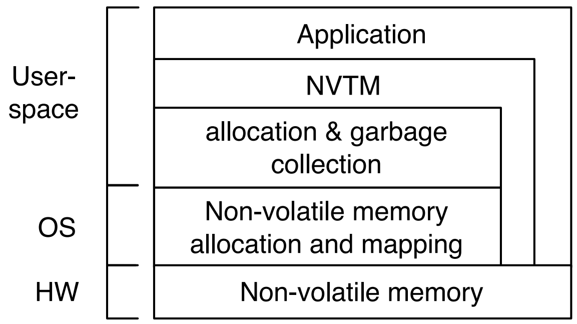
User-space
OS
HW
Application
NVTM
allocation & garbage collection
Non-volatile memory allocation and mapping
Non-volatile memory

Figure 2: **A system stack to provide durable transactional memory.** This organization allows common read and write operations to bypass the operating system entirely, while still allowing programmers to create robust data structures.

This paper describes a new interface, *non-volatile transactional memory* (NVTM), that integrates non-volatile data smoothly with existing language features, gets the operating system out of the way for accesses to non-volatile data, and provides robustness and reliability. With NVTM, programmers can build complex, persistent data structures that can interact seamlessly with volatile data structures. We have engineered NVTM to meet the following criteria:

1. **Memory-like interface** Programmers should be able to implement rich, non-volatile data structures in much the same way they implement volatile data structures today. This means accessing and modifying data directly via load and store instructions and using pointers to link components together.

2. **Atomic, durable semantics** Multiple threads should be able to make transactional modifications to those data structures that are durable and atomic, thereby making them robust against application and system failure.

3. **High performance** Access to data must be extremely fast. In the common case, accessing non-volatile storage should be nearly as fast as accessing volatile memory. In particular, accesses to non-volatile data should benefit from caching in the memory hierarchy and performance should scale with thread count.

4. **Scalability** The interface should scale to very large (many gigabytes to terabytes) data structures. This means that operation costs and volatile storage requirements should not grow with the size of the non-volatile storage.

To evaluate NVTM's performance, we compare it to existing non-durable TM systems as well as a transactional storage system optimized for spinning disks. We also evaluate NVTM's performance scalability, and evaluate the impact of the underlying non-volatile storage technology on its overall performance. In particular, we focus on phase-change (PCM) and spin-transfer torque memories (STTM), both of which are projected to provide near-DRAM levels of performance.

Our results show that by side-stepping the operating system for common-case accesses, NVTM offers enormous performance improvements without sacrificing safety or reliability. NVTM outperforms a transactional storage system designed for disks by up to 870$\times$ on average, scales to large data sets, and offers good performance scaling with additional threads.

The remainder of this paper is organized as follows. We present an overview of NVTM in Section 2. Section 3 describes the emerging non-volatile memory technologies we use and the hardware support NVTM requires. Section 4 describes NVTM in detail. Sections 5 and 6 describe our methodology and results. We discuss related work in Section 7. Finally, Section 8 concludes.

2 NVTM overview

The goal of NVTM is to expose the performance advantages of the non-volatile storage technologies we mentioned in the introduction while making it easy for programmers to safely construct large (i.e., multi-terabyte), robust, high-performance data structures.

Technology	Latency	
	Read	Write
Phase Change Memory [5, 25]	67ns	215ns
Spin-Torque Transfer [40, 25]	29ns	95ns
DRAM	25ns	35ns
Seagate 7200.10 3.5 in HD [27]	9ms	6ms
Intel 2.5 in X-25E SSD [1]	190μs	125μs

Table 1: **Memory technology summary.** NVTM targets solid-state, non-volatile memories such as phase change and spin-torque transfer memories. We assume that phase change and spin-torque transfer memories occur in DIMMs on the processor's memory bus. The DRAM, hard disk, and SSD values are measurements from the hardware we use in Section 6.

The basic building block for these structures is the *non-volatile heap* (NV-heap). NV-heaps are similar to conventional heaps, but in addition to providing mechanisms for allocating and deallocating memory, they also provide automatic storage reclamation and durable transactional operations.

From the user's perspective, the interface to an NV-heap comes in two parts. The first is a mechanism for creating, managing, and accessing NV-heaps via a set of library calls. These calls correspond roughly to conventional file creation and open operations. The second part of the interface is an object-oriented, non-volatile transactional memory system. NVTM defines a base class from which all non-volatile objects must be derived and provides automatic storage reclamation.

Figure 1 contains the sample NVTM code that atomically removes all instances of the value 42 from a linked list. The example is similar to code for a volatile transactional memory system with a few exceptions: First, once per application, we must explicitly open the NV-heap before we use it. Second, the call to `GetRoot()` returns a pointer to an object at the beginning of the NV-heap – in this case, a non-volatile linked list of integers. Third, the code uses a special pointer type, `NVList<int>::Ptr` to access the list. The final and most important difference is in the guarantees that the code makes: If the applications crashes, and we re-open the NV-heap, it will appear as though the filter operation either ran to completion or never started.

Figure 2 shows NVTM's hardware and software components and their relationships. The key components, in addition to the NV-heap abstraction, are the non-volatile storage array, the operating system facilities for managing NV-heaps and mapping them into the application's address space, the user-space memory allocation and garbage collection system, and a non-volatile transactional memory system.

An NV-heap comprises two sections of storage. The first is a statically-sized section that holds NVTM meta-data. The second is the heap itself, which is a large contiguous region of allocatable, non-volatile memory.

In terms of scalability, NVTM aims to support very large NV-heaps and to do so regardless (within reason) of the amount of volatile memory available. The running times of all operations are functions only of the amount of data modified by the operation, not the total size of storage. In particular, it is never necessary to scan the memory or traverse the object graph in our system, even on recovery from a crash. In addition, the NV-heap libraries maintain some volatile state (allocated in DRAM), but the size of that state is small (less than 1 MB) and independent of the NV-heap size. Transactions require a small, fixed amount of volatile storage as well.

On the performance side, NVTM removes the operating system from all common-case operations. All accesses to data use load and store instructions. Our system only makes system calls to setup and tear down NV-heaps and to perform infrequent heap enlargement operations.

In the next section, we describe the non-volatile technologies that motivate our design and the support NVTM requires from the storage array. Then, we describe NVTM in more detail in Section 4.

3 Non-volatile memory technologies

NVTM does not rely on the characteristics of any particular solid-state memory technology. Its only requirement is that the technology can present a DRAM-like interface and achieve performance within a small factor of DRAM in both latency and bandwidth. Table 1 describes the specific technologies we use to evaluate our system. We consider two advanced non-volatile memories: phase change memory (PCM) and spin-torque transfer memory (STTM).

Phase-change memory stores data as the crystalline state of a chalcogenide layer [8], and recent work [25, 30, 42] has demonstrated that PCM has the potential to become a viable main memory technology as DRAM's scaling begins to falter. PCM may also eventually surpass flash memory in density according to the ITRS [2]. Current PCM devices can withstand a limited number of set/reset cycles (between 1e4 and 1e9, depending on the device). NVTM is compatible with PCM wear management techniques such as write coelescing [25] and "segment swapping" [42], but we do not address endurance in this work. The analysis in [25] provides a good characterization of PCM's performance and power consumption characteristics.

Spin-torque transfer memories store bits as a magnetic orientation of one layer of a magnetic tunnel junction (MTJ). Depending on the orientation, the junction's resistance is either low (the "parallel" state) or high (the "anti-parallel" state) [13]. Currently several companies including Grandis, Sony [20], Hitatchi [22] and Renesas [41] have developed STTM prototypes. Eventually STTM's density, latency, and power consumption will approach those of DRAM. In this work we assume 22nm STTM technology and base our estimates for performance on published papers [40, 23] and discussions with industry.

3.1 Storage system architecture

For this work we assume a DRAM-like DDR interface for non-volatile memories. This organization allows DRAM and non-volatile memory to share a single memory bus and controller. We assume that bus negotiation and transfer times are similar to existing DRAM interfaces, and the only additional overhead for non-volatile accesses arises from the differences in memory technology. Other system-level architectures are possible. For instance, the non-volatile memory could be on PCIe expansion card. Exploration of this and other alternatives and their impact on performance is the subject of ongoing work.

If NVTM is going to provide well-defined semantics, the underlying hardware must provide well-defined semantics as well. The key capability that the hardware must provide is the ability to be certain that a store has succeeded in writing data to non-volatile storage. Synchronous `write()`s and `sync()` provide similar capabilities for disk-based systems. In our descriptions below, we assume a non-volatile write barrier instruction that does not complete until all previous stores to non-volatile memory have been committed to non-volatile storage. This is similar to the atomic 8-byte writes and epoch barriers which provide atomicity and consistency in BPFS [11], a byte-addressable, persistent file system.

4 Non-volatile transactional memory

This section describes NVTM in detail. In addition to the hardware interface described in the previous section, the system has three primary components. The first is an interface for allocating and mapping NV-heaps. The second is an atomic, durable memory allocator that manages non-volatile memory for the user and provides the foundations of durability for NVTM. Finally, there is NVTM itself.

4.1 System-level storage allocation and mapping

The operating system is in charge of allocating and deallocating all non-volatile memory that applications use. To allocate space, the application requests a new NV-heap and the space for it from the library. The library maps the NV-heap into a contiguous region of the application's address space and returns the base address to the application. Later, if the application requires additional space, the library will request it from the operating system.

The operating system must also provide permissions, protection, and naming for NV-heaps. We do not address these issues directly in this work, but our implementation stores NV-heaps as normal files in a conventional file system and uses the associated naming and protection mechanisms. In a system with non-volatile memory, the `mmap()` system call would map the physical pages of non-volatile storage directly into the application's address space and transfer the file's read and write permissions into the virtual memory protection bits for the mapped region.

4.2 Durable, atomic memory allocation

NVTM uses its memory allocator to provide the foundation for its durability support. The allocator performs allocation and automatic garbage collection via reference counting. It is thread safe and all the operations it provides are atomic and durable. It also provides a set of special pointer types that NVTM and user code use to access data. NVTM uses these capabilities to bootstrap full-blown durable transactions.

Application interface When an application creates a NV-heap, a user space library initializes it by setting up the data structures for the memory allocation system. This system resembles conventional memory allocators, except that allocation and deallocation must be atomic and durable. This is required to prevent the orphaning or multi-allocating of memory as well as any other corruption of the NV-heap's internal data structures.

A reference counting scheme provides automatic deallocation for application code and NVTM. Automatic reclamation is necessary, since memory leaks are especially pernicious in a non-volatile storage system: Once a piece of memory leaks away (i.e., remains allocated but unreachable), the system can never reclaim it. Each transactional object has a reference count that the run-time atomically updates whenever a pointer to the object is copied or created. We selected reference counting instead of garbage collection, since it avoids the need to scan the entire NV-heap to reclaim unused space.

Combining volatile and non-volatile data in the same address space creates four different kinds of pointers: There are pointers stored in volatile storage that point to non-volatile data (*V-to-NV* pointers), non-volatile pointers to non-volatile data (*NV-to-NV*), and volatile pointers that point to volatile data (*V-to-V* or conventional pointers). We disallow the final category, non-volatile pointers to volatile data (*NV-to-V*), since they would be meaningless after a system failure.

Algorithm 1 Pseudo-code for assignment to a reference counted pointer. This code uses a two-phase commit protocol to atomically and durably assigns a new value to a reference counting pointer and updates the reference count on the object.

```
RCAssign(RefCountPtr & dstPtr, RefCountPtr object)
  object→Lock();
  /* Set up the operation descriptor */
  opDescriptor→dstPtr = &dstPtr;
  opDescriptor→newCount =
          object→refCount + 1;
  NonvolatileWriteBarrier();
  opDescriptor→valid = true;
  NonvolatileWriteBarrier();
  /* Apply the operation descriptor */
  *(opDescriptor→dstPtr) = opDescriptor→object;
  opDescriptor→object→refCount =
          opDescriptor→newCount;
  NonvolatileWriteBarrier();
  opDescriptor→valid = false;
  NonvolatileWriteBarrier();
  object→Unlock();
```

For simplicity, the reference counts only track references from NV-to-NV pointers. In practice, this means that the programmer must be careful not to drop the last NV-to-NV reference while modifying a data structure. This can lead to subtle bugs, but they only arise in code that does not use NVTM, since NVTM eliminates this possibility by keeping reference-counted pointers to objects involved in transactions.

Atomicity and Durability Algorithm 1 contains the pseudo-code for atomically assigning the address of a reference counted object to a reference counting pointer that is initially NULL. The allocator uses a simple two-phase protocol to provide atomicity and durability. It uses a non-volatile *operation descriptor* to record the changes required to perform the assignment. In this case, the descriptor contains the address of the pointer receiving the new value, the address of the reference counted object, and the new value of the object's reference count.

To gather this data, the code starts by recording the address of the object. It then acquires a lock stored in the object that will prevent other threads from concurrently modifying its reference count and/or destroying it. Once it has gathered the rest of the data, the code uses the non-volatile write barrier instruction to guarantee that the data is recorded in non-volatile memory. Then, it sets a valid bit and issues another write barrier. At this point, the assignment has logically completed. It then "plays" the operation descriptor by performing the necessary assignments. Once the assignments are complete, it issues a third write barrier and then marks the descriptor invalid. In the case of one or more system failures, it can replay the assignments as many times as needed. The memory allocator and reference counting system use a similar scheme to make all operations durable and atomic.

To provide support for concurrent accesses to objects, each reference counted object contains a lock that protects its reference count and the ability to deallocate its storage. Storing locks in non-volatile storage could potentially violate our scalability requirement, since, in the case of a system failure, all the locks need to be unlocked. This could necessitate a scan of the entire storage array. We avoid this problem by using *epoch-based locks*: We associate a current *epoch* with each NV-heap, and the system increments the epoch number when the heap is reloaded. A non-volatile lock is an integer, and if the integer is equal to the current epoch number, it is held by a thread, otherwise, it is available. Therefore, incrementing the NV-heap's epoch instantly releases all the locks in the NV-heap.

Both NVTM and the application code use `NVNew()` to allocate reference counted objects. During allocation, storage must atomically move from control of the allocator to control of the code that requested the allocation to prevent orphaned data. `NVNew()` accomplishes this by allocating storage, calling the object's constructor, and performing the assignment to the requester's pointer all as part of one atomic operation.

To support concurrent allocations and deallocations each thread has a private free list and set of operation descriptors for each of the atomic operations the allocator may need to perform (allocation, deallocation, reference count update, recursive object destruction, etc.). We also provide a global free list and periodically move free space from the per-thread free lists to the global list.

On allocation, if the thread's free list does not contain a suitable memory block, the thread acquires a lock on the global

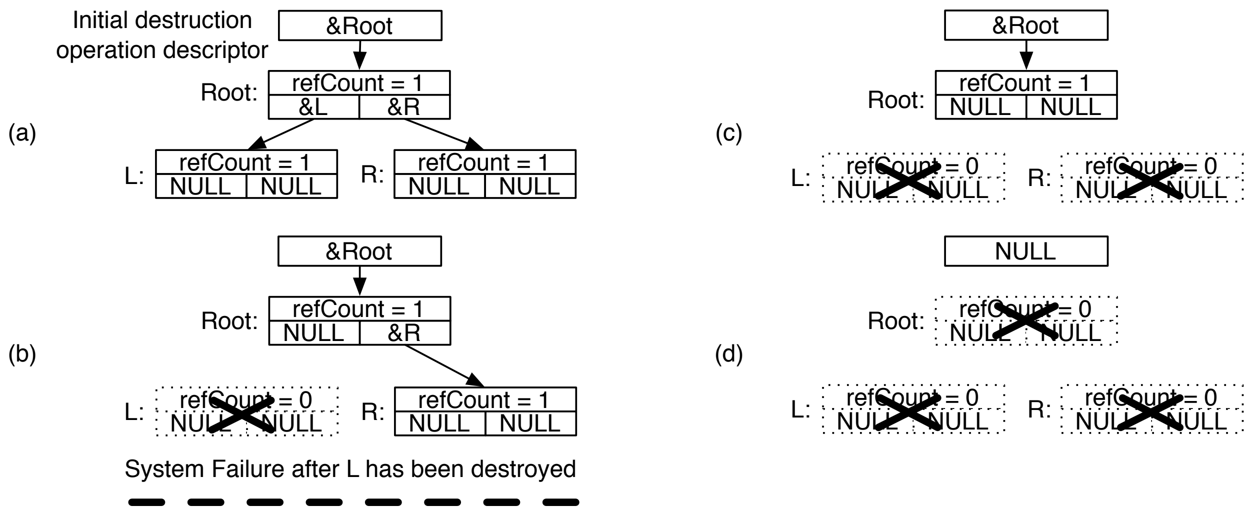

Figure 3: **Restartable object destruction.** Durable, atomic reference counting requires being able to restart the recursive destruction of a potentially large numbers of objects. In this example the root node of a tree is destroyed and triggers the deletion of the entire tree.

free list and uses its operation descriptor to complete the allocation. If the global list cannot satisfy the allocation, a system call (similar to `sbrk()` in a conventional allocator) will enlarge the NV-heap.

When the reference counting system discovers that an object is dead, it uses `NVDelete()` to perform the reverse operation. It atomically calls the destructor, deallocates the memory, and sets the requester's pointer to NULL. This is a potentially complex process since the destructor may cause the reference counts on other objects to go to zero, necessitating their destruction as well. Furthermore, NVTM must be able to restart this recursive destruction process in the case of a system failure.

Our solution is to record the top level object that must be destroyed in an operation descriptor and then call its destructor. When the destructor encounters an object with a pointer to another object, it checks that object's reference count. If the reference count is 1 (i.e., this is the last pointer to the object), it calls the destructor, and when the destructor completes, it atomically deallocates the memory and sets the pointer to NULL. If the reference count is greater than 1, it atomically decrements it and sets the pointer to NULL.

This algorithm provides the useful guarantee that, if a destructor finds a non-null pointer, it must point to a live object, albeit one that that may have been partially destroyed. However, even if it is has been partially destroyed, calling the destructor is still safe: The non-null pointers in the object point to live objects and the NULL pointers have already been properly disposed of.

This means that if a system failure occurs, it is safe to restart the top-level call to the destructor. As it proceeds a second (or even nth) time, it will try to perform the same set of recursive destructions. It will encounter NULL pointers up until the point of the system failure and then resume deletion where it was interrupted.

Figure 3 shows the algorithm in action. The system has just begun to delete a tree rooted at `Root` since the last pointer to it has just been set to NULL by the application. In (a), the operation descriptor for the starting point for the deletion holds the address of `Root`. The destruction proceeds by destroying the left child, `L`, and setting the left child pointer in `Root` to NULL.

At this point, the system fails. During recovery, it finds a valid operation descriptor and restarts the delete. Since the original destruction of `L` was atomic, the pointer structure of the tree is still valid and the destruction operation can destroy `R` (c) and `Root` before invalidating the operation descriptor (d).

The only caveat is that an object's destructor may be called more than once. In most cases, this not a problem, but some idioms that require the destructor to modify global state (e.g., tracking the number of live instances of an object) will be more difficult to implement.

Reloading NV-heaps It must be possible for one program to load an NV-heap created by another. This presents two challenges. The first is that the NV-heaps may end up mapped into any part of the application's address space. To support this, our system implements NV-to-NV pointers as relative pointers: Instead of holding the actual address (which would change from application to application or execution to execution), the relative pointer holds an offset from the pointer's address to data it points to. The conversion between relative pointer and virtual addresses requires just a single add instruction.

The second challenge is the virtual table pointers stored in objects with virtual methods. There is no guarantee that the virtual tables or virtual functions will be at the same virtual address across executions. There are (at least) two solutions: The first is to use epoch-numbers to determine if the virtual table pointer has been update during the current epoch (i.e., during the current program's execution), and, if it has not, overwrite it with the correct value (obtained using the dynamic loader). In the common case, this adds an extra integer comparison and branch (to check the epoch) to each virtual method call. To call a virtual method, a simple non-virtual stub function can implement the epoch-based scheme and call the virtual method. A second approach is to modify the compiler to emit relocatable virtual pointer tables for non-volatile classes and then store the necessary code in the NV-heap. This approach is preferable, but requires changing C++'s application binary interface and type system. We have implemented the first option.

Recovery To recover from a system failure, the memory allocator performs a sequence of four operations. First, it replays any valid operation descriptors for allocation or deallocation operations and marks them invalid. Then, it replays any reference count updates and/or reference counting pointer assignments, which may include the recursive destruction process described above. After these tasks are complete, all the reference counts are valid and up-to-date and the recovery process passes to NVTM.

4.3 Non-volatile transactional memory

Our transactional memory system builds on previous work developing TM systems for volatile memories. Transactional memories have received a great deal of attention in recent years [19, 37, 15, 7, 32, 18, 16]. We briefly describe the application interface and then describe NVTM's internals, focusing on the mechanisms it provides for durability and support for non-volatile storage.

Application interface All data stored in a NV-heap are in the form of objects derived from a transactional base class, `TMObject`, that provides reference counting and transaction bookkeeping meta-data. Objects derived from `TMObject` can only exist in non-volatile storage. This means they cannot be declared as local variables, static global variables, or as part of another class. The only way to create a transactional object is through a call to `NVNew()`. This is similar to the Java-based scheme in [18]).

An application indicates the beginning of transaction with `TXBegin`. Once inside a transaction, it calls `TXOpenWrite()` or `TXOpenRead()` to gain exclusive write access or shared read access to a transactional object. When it is through, the application uses `TXEnd` to denote the end of the transaction. Nested transactions are flattened into a single transaction.

Our implementation of NVTM provides nearly ACID (atomic, consistent, isolated, and durable) semantics for its transactions. Its only limitation is with respect to isolation. Since NVTM is a software only scheme, it is not possible to efficiently provide complete isolation, since there is no run-time protection against an application circumventing NVTM and issuing loads or stores to non-volatile storage. This is a well-known limitation of software TM systems. A second caveat is that our implementation only protects non-volatile storage, so transactions can communicate through shared, volatile variables. However, this is not a fundamental limitation, and extending the system to include volatile data as is the subject of ongoing work. When this work is complete, well-behaved NVTM programs will provide full ACID semantics.

NVTM internals NVTM is a software-only TM system: The only support NVTM requires from the hardware is the primitives described in Section 3.1. However, it relies heavily on the transactional memory allocator described above and uses reference counted objects for all of its internal data structures. It is implemented in C++.

NVTM is a blocking TM implementation that relies on writer locks and read versioning. We perform eager conflict detection of writes by requiring a transaction to become the owner of an object before modifying it. Read conflicts are detected lazily by validating object version numbers at commit time. NVTM allows direct update of objects in memory by making a copy of each object in the log as it is opened for writing. We maintain durability by storing the logs of any outstanding transactions in non-volatile memory and rolling back uncommitted changes at restart. We use a simple contention management scheme [34] that aborts transactions as soon as a conflict is detected.

NVTM consists of three key data structures: (i) an NVTM base class from which all transactional objects inherit, (ii) a set of logs (one per thread) implemented as linked list of log entries, and (iii) an ownership record table. The object base class provides per-object data including the reference count, object size information, and an object ID.

The ownership record table is the only volatile structure NVTM uses. Ownership records enforce exclusive write access and to detect read-write conflicts during transactions. Each ownership record is protected by a lock, and it stores a pointer to the transaction that currently owns the data, if there is one, and a version number. We use a table of ownership records stored in volatile memory and indexed by the low-order bits of the object ID that the system assigns at object creation time. Using a table of ownership records leads to some unnecessary transaction aborts, but it has two key advantages: First, volatile locks are faster than non-volatile locks. Second, using an array indexed by an ID avoids the need for a more complicated map lookup on lock acquisition. In practice, we find that the number of unnecessary aborts to be very low.

The structure of the log and how it is processed are the main differences between volatile and non-volatile TM systems.

NVTM must ensure that if the system fails in the midst of a transaction, the startup recovery code can restore memory to the state it was in before the failed transaction began. This means that the log must be non-volatile. It also means that, in the case of multiple system failures, it must be safe to re-start the restore procedure at any point. Similarly, in the case of a commit, it must be safe to restart the log deletion at any point.

NTVM maintains one log for each thread accessing the NV-heap. Each log is a pair of linked lists (one for reads and one for writes) of log entries. Algorithm 2 contains the pseudo-code for `TXOpenWrite()`. To open an object for writing, the running transaction must take ownership of the object. If the object is owned by another transaction already, then a write conflict has occurred and the transaction aborts. Once the transaction becomes the owner, NVTM copies the object into the log along with a pointer to the original object. To open an object for reading, NVTM stores a pointer to the original object and its current version number in the log.

Algorithm 2 Pseudo-code for opening an NVTM object for modification. Before modifications are allowed, NVTM copies the object to the log and marks the entry valid.

```
TXOpenWrite(RefCountPtr p)
  TransactionPtr trans = getCurrentTransaction();
  int index = hash(p→objID);
  orecTable[index].acquireLock();
  if orecTable[index].owner == NULL then
    orecTable[index].owner = trans;
  else if orecTable[index].owner != trans then
    orecTable[index].releaseLock();
    TXAbort();
    return
  end if
  orecTable[index].releaseLock();
  NVNew(log→tail→next);
  newEntry = log→tail→next;
  log→tail = newEntry;
  NVNew(newEntry→copy);
  *(newEntry→copy) = *p;
  NonvolatileWriteBarrier();
  newEntry→valid = true;
  NonvolatileWriteBarrier();
```

The pointer in the log is a reference-counting pointer, just like all other pointers to a transactional object, so creating the copy increments the original object's reference count. Likewise, when we create the copy, the reference counting system also increments the reference counts on any objects referred to by pointers in the object. This is necessary to postpone the deletion of objects until the transaction commits.

When the copy is complete, the application can safely modify the original object. If a system failure occurs or the transaction aborts, NVTM rolls back the log entries by restoring the original data from the log, marks the log entry as invalid, and then drops the pointer to the log entry so that the reference counting system deletes it. Note that is safe to perform the restore multiple times (as might happen if the system crashed during recovery) since, during the re-executions, previously completed assignments will have no effect.

It is also possible that the system will fail during the initial copy. In this case, the log entry will be invalid on restart, and dropping the pointer to the entry will trigger its destruction. Here, the danger is in calling a destructor on an object whose constructor has not completed and which might contain garbage pointer values. The memory allocator prevents this by ensuring that the memory it returns is all zeros, guaranteeing that all pointers are initially NULL.

Transaction abort and crash recovery The process for aborting a transaction and recovering from a system failure are very similar: NVTM rolls back the transaction by restoring data from the write log into the application's objects, marking log entries invalid, and deleting them as it goes.

The only additional concern with crash recovery is that recovery must be restartable in the case of multiple failures. NVTM ensures this by only rolling back valid log entries and by using a non-volatile memory barrier to ensure that recovery of an entry is completely recorded in non-volatile storage before marking it invalid.

Cores	GHz	L2 Cache	System DRAM	NV storage type	NV storage size
4	2.4	2x 4MB	4	Hard disk	256GB
4	2.4	2x 4MB	4	Intel SSD	32GB
4	2.4	2x 4MB	4	Advanced NVM	4GB

Table 2: **Machine configurations.** We gather data for three different machine configurations in this study.

5 Methodology

This section describes the methodology we use to evaluate our system. First, we describe our system for emulating large amounts of PCM and STTM. Then we describe the applications we use in our evaluation.

5.1 Model fast non-volatile storage

NVTM aims to support systems with several terabytes of high-performance non-volatile memory, but since the memory technologies in Section 3 are still maturing, these systems are not available.

To emulate these systems, we have built a set of tools that allows us run full applications for many hundreds of billions of instructions while simulating the performance impact of using an advance non-volatile memory for either a portion of application memory or as a disk-like block device. This section describes these tools and the system configurations we model with them.

5.1.1 Modeling byte-addressable, non-volatile storage

The first scheme models the effect of adding advanced non-volatile memories to a conventional memory hierarchy as DRAM-like, byte-addressable memory. The system identifies a region of an application's address space as non-volatile and samples its memory behavior to generate a signature that separates accesses along three axes: loads vs stores; last-level cache hits vs misses; and volatile vs non-volatile. The tool uses the signature to estimate the performance impact of non-volatile storage on application performance.

To accomplish this with minimal overhead, the tool divides program execution into intervals of 10 billion instructions (across all executing threads). The tool is based on Pin [28]. At the beginning of each interval, we turn on instrumentation to perform a detailed cache simulation of the first 100 million instructions to collect the signature.

After the sampling period, we turn off the cache simulator and use hardware performance counters to track the number of L2 cache misses and L1 cache accesses. We use the signature to estimate how many of each type of access are to non-volatile storage and then use a technology model to estimate the additional latency those operations would incur. We perform all these measurements and calculations on a per-thread basis and then use the new running time of the longest-running thread to estimate the total latency. To generate accurate results we ensure that the simulated cache configuration matches the hardware that machine is running on (See below).

There are several potential problems with this system. First, program behavior may change during the interval, invalidating the signature. To mediate this problem, we annotate the applications with a special function call between phases that triggers the start of a new interval. Our applications have predictable, consistent behavior, so identifying phase boundaries is easy. For more complex workloads, a phase-based sampling methodology such as [38] could be used. Second, it does not capture fine-grain parallelism among accesses to non-volatile memory. This is a conservative assumption, since it assumes that all non-volatile memory accesses within a thread occur sequentially.

To calibrate our system we used a simple program that empirically determines the L2 miss latency (a common exercise in architecture courses). We ran the program on top of the simulated PCM and STTM arrays and its estimates matched our target latencies to within 10%.

The overhead due to Pin's instrumentation varies unpredictably with thread count and application (although it is consistent for a particular application and thread count combination). To account for this overhead, we run the instrumented version several times and compare it to the run-time without Pin. We then subtract this value from the Pin instrumented run-times we report. This methodology delivers results accurate to within 5%.

5.1.2 Modeling a SSD based on advance non-volatile memory

The transactional storage system we compare our system to in Section 6 targets a block device (i.e., a disk) instead of a non-volatile main memory. To model an SSD built out of the non-volatile technologies in Section 3, we implemented a RAM disk in that allows us to insert extra delay on accesses to match the latency of non-volatile memories. Measurements using a simple disk access latency benchmark show that we can control the effective latency to within about 2% for values we are interested in for this work.

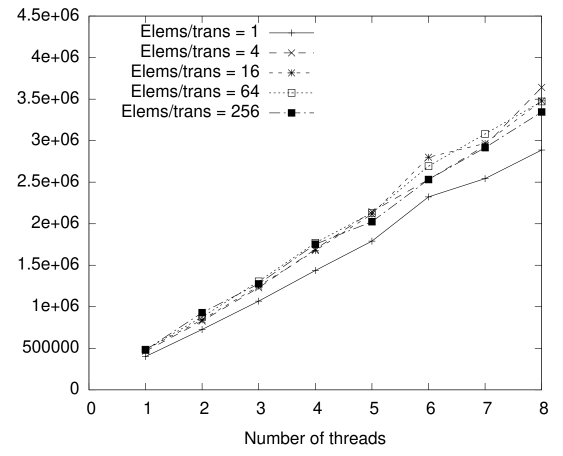

Figure 4: **Throughput vs. transaction size.** For the GUPS benchmark, we scaled the number of elements a transaction modifies from one to 256 as we scaled the number of threads in order to show the effect of transaction size on throughput.

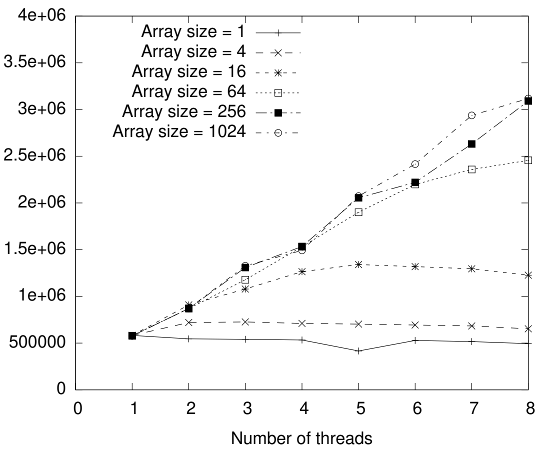

Figure 5: **Contention among transactions for shared objects.** When multiple threads contend for write access to the same object, their execution is serialized. NVTM performance scales well as contention decreases.

5.1.3 System configurations

We present results collected on two socket, Core 2 Duo (a total of 4 cores) machines running at 2.4Ghz with 8GB of physical DRAM and two 4MB L2 caches. These machines are equipped with both a conventional 250GB hard drive and an 32GB Intel Extreme flash-based SSD.

We use these physical machines to model three different target machines (Table 2). In each configuration we configure a portion of physical DRAM as either fast solid-state disk or as addressable non-volatile memory, regardless of whether the experiment requires it. This places all the systems on an equal footing with respect to usable system memory.

For the experiments that use disks and/or the SSD, we use the small machines and the timing measurements we present are "wall clock" times for program execution.

5.2 Workloads

Because existing interfaces to non-volatile storage make it difficult to build complex data structures in non-volatile memory, there are no "off the shelf" workloads with which to evaluate our system. Instead, we have written a set of benchmarks from scratch and ported an additional one to use NVTM. All of them include large data footprints and a large amount of concurrency.

To evaluate NVTM we use four workloads, each implemented in two different styles. The first is an NVTM-based implementation, and the second uses a transactional storage system called Stasis [35] that targets conventional disk-based storage systems.

GUPS and GSPS Giga-updates per second (GUPS) is a common benchmark for measuring random access performance. GUPS typically consists of a random read-modify-write operations to a large array of scalar values. We implement this algorithm as well as a Giga-swaps per second (GSPS) version that randomly selects two elements and transactionally swaps their values. We use GUPS to quantify the random IO performance of NVTM.

SSCA SSCA version 2.2 [4] is one of several workloads provided by government agencies to model workloads they are interested in. The application generates a very large, scale-free graph in memory. It then performs three analyses on the graph. The first analysis traverses the given graph structure in search of its highest weighted edges. The second analysis maps all possible paths of a given length that begin with the highest-weighted edges found in the first analysis. The third analysis computes the number of shortest paths that pass through a given vertex.

Our implementation is based on an OpenMP implementation that we ported to pthreads and then transactionalized. It stores the graph as well several auxiliary structures of meta data in non-volatile storage, modeling an application in which the graph is stored permanently on in non-volatile memory and analyzed repeatedly.

System	Nop	Read	Write	Read-Write
NVTM-nodur	0.058μs	0.50μs	0.72μs	1.2μs
NVTM-DRAM	0.063μs	0.90μs	1.8μs	2.6μs
NVTM-PCM	0.063μs	7.3μs	18μs	25μs
NVTM-STTM	0.063μs	3.0μs	7.2μs	10μs

Table 3: **Latency for minimal transactions.** We measure the latency in micro-seconds for each operation on the small-NVM hardware configuration for several versions of NVTM (see text).

6-degrees of separation This workload solves the 6-degrees of separation problem on a large, scale-free graph that models a social network or the link structure of the world wide web (y = 2.72) [9]. Each thread alternates between two states. The first chooses a random pair of vertices and determines whether a path of at most 6 links connects them. The second type of thread performs mutations on the graph, adding and removing edges at random while preserving its scale-free properties.
B-Tree This workload builds a large B-Tree and then randomly performs read and update operations. The B-Tree implementation is based on the description in [12]. Each thread selects a number at random and searches for it in the B-Tree. It if it present, the thread removes it. Otherwise, the thread inserts it.

6 Results

To evaluate our system we start by quantifying the performance impact of both adding durability to the transactional memory system and the underlying non-volatile memory technology. Then we examine its performance and scalability on our benchmark suite, and compare it to an existing disk-based transactional storage system called Stasis [35]. Finally, we examine the impact of non-volatile memory latencies and the importance of hardware support.

6.1 Basic operation performance

Our system differs from conventional TM systems in two respects. First, it targets memory technologies that are somewhat slower than DRAM. Second, it provides durability, which adds complexity to our transactions and memory allocation operations.

To understand the impact of these two changes, we characterize NVTM's performance on basic operations for several different configurations: Our TM system without support for durability (NVTM-nodur), NVTM with durability but running on normal DRAM (NVTM-DRAM), and NVTM running on PCM and STTM at the technology points described in Section 3 (NVTM-PCM and NVTM-STTM).

We wrote a simple micro-benchmark to measure the execution time of an empty transaction, a transaction with a single read or write, and a transaction with both a read and a write. We ran a single thread operating on a single transactional object to eliminate contention effects, and we measured the execution time for a tight loop that executed 100 million operations. Table 3 contains the latency results.

The difference in latencies between NVTM-nodur and NVTM-DRAM indicate that the cost of durability is roughly a $2X$ increase in latency. This is due to the frequent non-volatile write barriers that flush transactional state or memory allocation state to storage. With durability added to the system, nop transactions exhibit only a slight increase in latency because they do not perform any logging. Also, the cost of a read is approximately half the cost of a write because write logging requires copying the actual object data in addition to a pointer to the object. The effect that storage technology has on transaction latency is consistent with the latencies we presented in Table 1.

Figure 4 illustrates how performance scales with the size of a transaction. We measure GUPS write throughput over an array of one million elements as we varied the number of updates performed in a single transaction from one to 256 elements. The overhead of beginning and completing a transaction is most pronounced for transactions that access only a single element. Increasing the number of elements to four amortizes most of this cost, and beyond that, the improvement is marginal.

Figure 5 highlights the effect of contention for write access to shared objects in the GUPS workload. We measure GUPS write throughput for various array sizes with each transaction accessing only a single element. Under high contention (array sizes of one, four, and 16), we see that throughput is nearly independent of the total number of threads. For larger arrays, contention is minimized and there are fewer aborts which results in an overall throughput that scales with the number of threads, delivering over 6$\times$ speedup with 8 threads.

6.2 Benchmark performance

Figure 6 shows the performance of the workloads in Section 5.2, running on each of the configurations listed in Table 3. We ran each configuration with 4GB memory footprints and one, two, and four threads. These workloads suffer a smaller

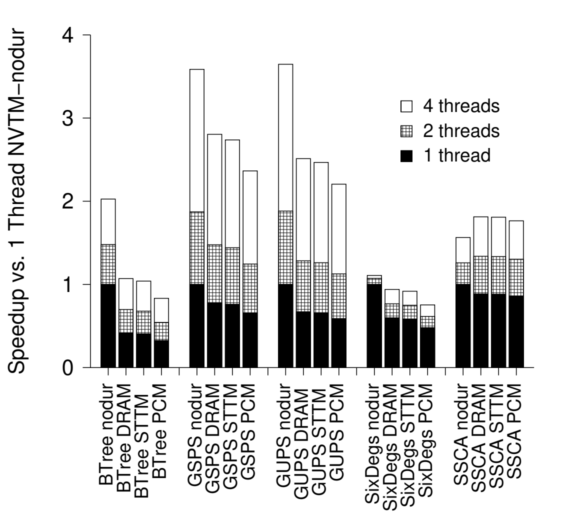

Figure 6: **NVTM vs. TM performance.** Both adding durability and running on non-volatile memory affect NVTM's performance.

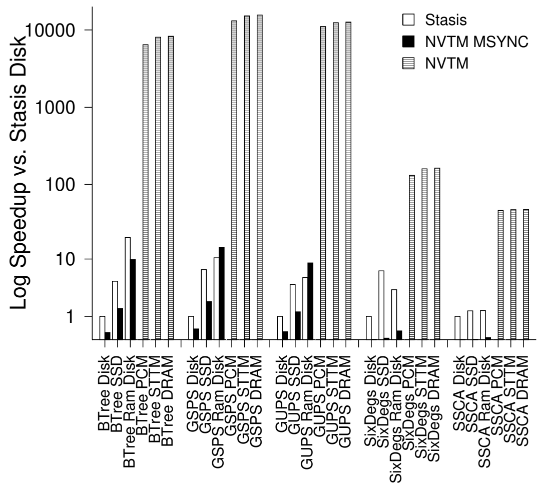

Figure 7: **The benefits of avoiding the operating system.** NVTM outperforms Stasis by almost three orders of magnitude in large part because it does not require system calls to provide transactional guarantees.

penalty for adding durability (between 15 and 58% for four threads) than our micro-benchmarks, since they include normal computation in addition to transactions. Likewise, the difference between the DRAM, STTM, and PCM configurations is smaller as well.

The simplest workloads, GUPS and GSPS, show very good scalability with thread count achieving 3.7× and 3.6× speedups with four thread, respectively. BTree shows good speedup as well with 2.6×. Six-degrees scales less well, but this is due in large part to contention for nodes in the graph: Reducing the search depth to one eliminates much of the contention and increases scalability by 60%. A more sophisticated contention manager would probably improve performance as well.

6.3 Comparison to conventional IO-based systems

In the introduction, we argued that existing interfaces to non-volatile storage were unsuitable for storage systems based on new non-volatile memories. To test that hypothesis we compare NVTM's performance to that of Stasis [35], a transactional storage system tuned for disk-based systems.

For these experiments we created a modified version of NVTM, NVTM-msync, that uses `msync()` in place of the non-volatile write barrier instruction. The `msync()` system call forces dirty pages of an mapped file to disk.

Figure 7 compares NVTM to NVTM-msync and Stasis on the three machine configurations with four threads, and note that performance appears on a log scale. A fourth configuration, RAM-disk, creates a block device out of DRAM which models an SSD composed of advanced non-volatile memories. The first thing to notice is the impact that solid-state memories can have on performance even without NVTM. For both Stasis and NVTM-msync, moving from the disk to the SSD provides a 4.8× increase on average and the RAM disk provides an additional 4×.

The data also show the value of minimizing writes in "disk"-based systems. Stasis carefully manages buffers to minimize writes, and as a result, it out-performs NVTM-msync by between 3× and 200×.

The largest jump in performance, though, comes from avoiding the operating system altogether. Baseline NVTM out-performs Stasis on the RAM disk by between 46× and 2,350×. The largest speedups come for GUPS and GSPS. For these, Stasis pays a double penalty since it both uses systems calls and writes whole pages when the transactions touch just one or two integers. More complex benchmarks see smaller but still very large performance gains (between 34× and 415×).

This makes a strong argument for moving away from the current set of IO abstractions for fast non-volatile memories: The cost they extract is enormous compared to memory latency. A system call to write a 4KB page to the RAM disk takes at least 6μs, according to our measurements, and a single transaction requires several such writes. NVTM's non-volatile write barriers are much faster and can proceed in parallel with other instructions. The data make it clear that continuing to rely on the operating system for access to non-volatile data will squander a vast amount of potential performance.

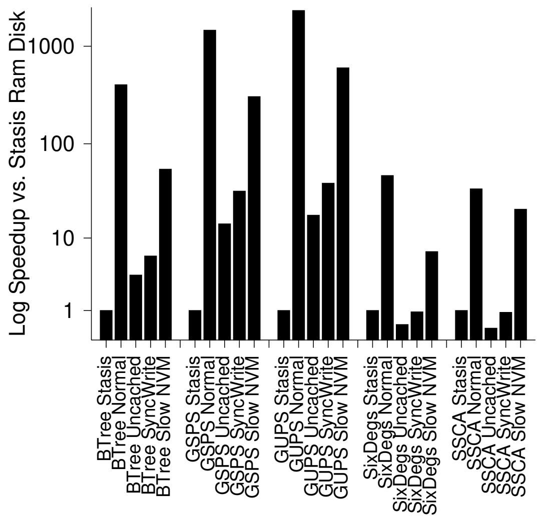

Figure 8: **Sensitivity analysis.** Even with less efficient mechanisms for flushing writes to non-volatile memory or slower non-volatile storage, NVTM still delivers good performance.

6.4 Sensitivity to technology

This section discusses the sensitivity of NVTM's performance to technology factors: The absence of a non-volatile write barrier instruction in existing architectures and the impact of non-volatile memory performance on NVTM.

In Section 3.1 we described the non-volatile write barrier instruction that forces data into non-volatile storage. This instruction, or something equivalent to it, is essential for implementing NVTM. However, current processors do not have such an instruction. Figure 8 compares two alternative implementations for our workloads running with four threads on the small machine configuration. The first, is to mark non-volatile memory pages as uncached, an option available on modern machines. Without caching, NVTM performance drops significantly (by 121× on average). At that level of performance, NVTM still outperforms Stasis for three of out the five benchmarks and by a factor of 197× on average. The second option is to make all stores synchronous and all caches write-through for non-volatile data, still allowing loads to access data from the caches. This alternative reduces performance by a smaller margin – 55× on average and by no more than 10% for any one application.

The final bar in the figure plots NVTM's performance with the non-volatile memory barriers for a hypothetical, non-volatile memory technology with read and write latency of 1μs. Performance drops by about 82% on average, demonstrating that even if non-volatile memories do not reach the levels of performance projected for PCM and STTM, NVTM still offers a large margin of performance over the status quo.

7 Related Work

Transactions are used extensively in database management systems (DBMSs) to provide the properties of atomicity, consistency, isolation, and durability (ACID) of data stored on disk. The relational model that most databases use makes building pointer-based data structures difficult. The database community has addressed this shortcoming with object-oriented databases [24], streaming databases [10], and relational extensions that support user-defined abstract data types [39]. These systems still require a fixed physical data model and that applications be formulated as queries into a database. NVTM does not provide or require a query-based interface.

Berkeley DB is a lightweight alternative to conventional databases that provides transactional semantics for primitive data structures such as B-trees and hash tables [36]. Stasis, a transactional storage library optimized for disk, provides a more flexible set of transactional storage primitives without using a database [35]. As discussed in Section 6, Stasis targets block devices rather than byte-addressable media, and in that domain, it outperforms NVTM. Several other systems, including QuickSilver [17], RVM [33], and Argus [26], provide transactional storage for disk-based storage.

BPFS [11] is a file system designed specifically for byte-addressable, advanced non-volatile memories. It supports ACID semantics at the file level, but it is still a conventional file system in that applications must use system calls to access data. We feel that BPFS and NVTM are complimentary, since both take important steps towards modernizing storage abstractions for these new memories.

NVTM owes much of its heritage to the volatile transactional memory [19] systems that provide an alternative to locks for managing concurrent access to volatile memory. Recent work has focused on utilizing conventional coherence and consistency protocols and overcoming data size restrictions [15, 3, 31, 7]. Given the expense of hardware support, Shavit and Touitou proposed a software transactional memory system which provided weak isolation and explicit statically-sized transactions [37]. The operation descriptors that NVTM's allocator and reference-counting system use are similar. Dynamic software transactional memory (DSTM) provided strong isolation and overcame the static limitation to allow dynamic data structures such as lists and trees [18]. NVTM's object-based TM model is similar to that of DSTM and others [29, 14]. Parts of NVTM's internal algorithms for read versioning, write locking, and undo logging were inspired by McRT-STM, a C++ library-based multi-core runtime system that supports software transactional memory [32].

Prior work in memory management for multi-core systems played a part in the design of NVTM's atomic, durable allocator. Memory allocators such as Hoard [6] use a global heap and per-thread heaps to efficiently support parallel applications. McRT-Malloc is a memory allocator designed specifically for a software transactional memory system in a multi-core environment [21].

8 Conclusion

We have described a non-volatile transactional memory system that provides durable, transactional semantics for rich, pointer-based data structures stored in advanced non-volatile memories. NVTM removes the operating systems from the common-case path for accessing non-volatile storage while still providing reliability in the face of application and system failure. Our performance measurements show that while adding durability adds some overhead compared to existing volatile transactional memory systems, NVTM out-performs existing file-based transactional interfaces by a wide margin.

References

[1] Intel x25-e sata solid state drive product manual.

[2] International technology roadmap for semiconductors: Emerging research devices, 2007.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[4] D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *HiPC 2005: Proc. 12th International Conference on High Performance Computing*, pages 465–476, December 2005.

[5] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, M. Tosi, R. Bez, R. Gastaldi, and G. Casagrande. An 8mb demonstrator for high-density 1.8v phase-change memories. *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 442–445, June 2004.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 117–128, New York, NY, USA, 2000. ACM.

[7] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 24–34, New York, NY, USA, 2007. ACM.

[8] M. J. Breitwisch. Phase change memory. *Interconnect Technology Conference, 2008. IITC 2008. International*, pages 219–221, June 2008.

[9] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309 – 320, 2000.

[10] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 203–214. VLDB Endowment, 2002.

[11] J. Condit, E. B. Nightingale, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *SOSP '09: Proceedings of the twenty-second ACM Symposium on Operating systems principles*. To appear.

[12] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[13] B. Dieny, R. Sousa, G. Prenat, and U. Ebels. Spin-dependent phenomena and their implementation in spintronic devices. *VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on*, pages 70–71, April 2008.

[14] J. E. Gottschlich and D. A. Connors. Dracostm: a practical c++ approach to software transactional memory. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 52–66, New York, NY, USA, 2007. ACM.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[16] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[17] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in quicksilver. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 107–108, New York, NY, USA, 1987. ACM.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[20] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, pages 459–462, Dec. 2005.

[21] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 74–83, New York, NY, USA, 2006. ACM.

[22] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, I. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno. 2mb spin-transfer torque ram (spram) with bit-by-bit bidirectional current write and parallelizing-direction current read. *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 480–617, Feb. 2007.

[23] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. M. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno. 2 mb spram (spin-transfer torque ram) with bit-by-bit bi-directional current write and parallelizing-direction current read. *Solid-State Circuits, IEEE Journal of*, 43(1):109–120, Jan. 2008.

[24] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Commun. ACM*, 34(10):50–63, 1991.

[25] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[26] B. Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.

[27] S. T. LLC. Barracude 7200.10 datasheet. http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_7200_10.pdf.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[29] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006. Held in conjunction with PLDI 2006. Expanded version available as TR 893, Department of Computer Science, University of Rochester, March 2006.

[30] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *International Symposium on Computer Architecture*, June 2009.

[31] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[32] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[33] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, 1994.

[34] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[35] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 29–44, Berkeley, CA, USA, 2006. USENIX Association.

[36] M. Seltzer and M. Olson. Libtp: Portable, modular transactions for unix. In *Proceedings of the 1992 Winter Usenix*, pages 9–25, 1992.

[37] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[38] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[39] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, 1991.

[40] R. Takemura, T. Kawahara, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno. 2mb spram design: Bi-directional current write and parallelizing-direction current read schemes based on spin-transfer torque switching. *Integrated Circuit Design and Technology, 2007. ICICDT '07. IEEE International Conference on*, pages 1–4, 30 2007-June 1 2007.

[41] H. Tanizaki, T. Tsuji, J. Otani, Y. Yamaguchi, Y. Murai, H. Furuta, S. Ueno, T. Oishi, M. Hayashikoshi, and H. Hidaka. A high-density and high-speed 1t-4mtj mram with voltage offset self-reference sensing scheme. pages 303–306, Nov. 2006.

[42] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 14–23, New York, NY, USA, 2009. ACM.