NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories

Joel Coburn Adrian M. Caulfield Ameen Akel Laura M. Grupp
Rajesh K. Gupta Ranjit Jhala Steven Swanson

Department of Computer Science and Engineering
University of California, San Diego
{jdcoburn, acaulfie, aakel, lgrupp, rgupta, jhala, swanson}@cs.ucsd.edu

Abstract

Persistent, user-defined objects present an attractive abstraction for working with non-volatile program state. However, the slow speed of persistent storage (i.e., disk) has restricted their design and limited their performance. Fast, byte-addressable, non-volatile technologies, such as phase change memory, will remove this constraint and allow programmers to build high-performance, persistent data structures in non-volatile storage that is almost as fast as DRAM. Creating these data structures requires a system that is lightweight enough to expose the performance of the underlying memories but also ensures safety in the presence of application and system failures by avoiding familiar bugs such as dangling pointers, multiple free()s, and locking errors. In addition, the system must prevent new types of hard-to-find pointer safety bugs that only arise with persistent objects. These bugs are especially dangerous since any corruption they cause will be permanent.

We have implemented a lightweight, high-performance persistent object system called NV-heaps that provides transactional semantics while preventing these errors and providing a model for persistence that is easy to use and reason about. We implement search trees, hash tables, sparse graphs, and arrays using NV-heaps, BerkeleyDB, and Stasis. Our results show that NV-heap performance scales with thread count and that data structures implemented using NV-heaps out-perform BerkeleyDB and Stasis implementations by 32× and 244×, respectively, by avoiding the operating system and minimizing other software overheads. We also quantify the cost of enforcing the safety guarantees that NV-heaps provide and measure the costs of NV-heap primitive operations.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Storage hierarchies; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); E.2 [*Data*]: Data Storage Representations

General Terms Design, Performance, Reliability

Keywords Non-volatile Heap, Persistent Objects, Phase-change Memory, Spin-torque Transfer Memory, ACID Transactions, Pointer Safety, Memory Management, Transactional Memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0266-1/11/03…$10.00

1. Introduction

The notion of memory-mapped persistent data structures has long been compelling: Instead of reading bytes serially from a file and building data structures in memory, the data structures would appear, ready to use in the program's address space, allowing quick access to even the largest, most complex persistent data structures. Fast, persistent structures would let programmers leverage decades of work in data structure design to implement fast, purpose-built persistent structures. They would also reduce our reliance on the traditional, un-typed file-based IO operations that do not integrate well with most programming languages.

Many systems (e.g., object-oriented databases) have provided persistent data structures and integrated them tightly into programming languages. These systems faced a common challenge that arose from the performance and interface differences between volatile main memory (i.e., DRAM) and persistent mass storage (i.e., disk): They required complex buffer management and de(serialization) mechanisms to move data to and from DRAM. Despite decades of work optimizing this process, slow disks ultimately limit performance, especially if strong consistency and durability guarantees are necessary.

New non-volatile memory technologies, such as phase change and spin-torque transfer memories, are poised to remove the disk-imposed limit on persistent object performance. These technologies are hundreds of times faster than the NAND flash that makes up existing solid state disks (SSDs). While NAND, like disk, is fundamentally block-oriented, these new technologies offer both a DRAM-like byte-addressable interface and DRAM-like performance. This potent combination will allow them to reside on the processor's memory bus and will nearly eliminate the gap in performance between volatile and non-volatile storage.

Neither existing implementations of persistent objects nor the familiar tools we use to build volatile data structures are a good fit for these new memories. Existing persistent object systems are not suitable, because the gap between memory and storage performance drove many design decisions that shaped them. Recent work [13, 14] has shown that software overheads from the operating system, file systems, and database management systems can squander the performance advantages of these memories. Removing these overheads requires significant reengineering of the way both the kernel and application manage access to storage.

Managing non-volatile memory like conventional memory is not a good solution either. To guarantee consistency and durability, non-volatile structures must meet a host of challenges, many of which do not exist for volatile memories. They must avoid dangling pointers, multiple free()s, memory leaks, and locking errors, but they also must avoid several new types of hard-to-find program-

ming errors. For instance, pointers from non-volatile data structures into volatile memory are inherently unsafe, because they are meaningless after the program ends. The system must also perform some kind of logging if non-volatile structures are to be robust in the face of application or system failure. Trusting the average programmer to "get it right" in meeting these challenges is both unreasonable and dangerous for non-volatile data structures: An error in any of these areas will result in permanent corruption that neither restarting the application nor rebooting the system will resolve.

This paper proposes a new implementation of persistent objects called *Non-volatile Memory Heaps* (NV-heaps). NV-heaps aim to provide flexible, robust abstractions for building persistent objects that expose as much of the underlying memory performance as possible. NV-heaps provide programmers with a familiar set of simple primitives (i.e., objects, pointers, memory allocation, and atomic sections) that make it easy to build fast, robust, and flexible persistent objects. NV-heaps avoid OS overheads on all common case access operations and protect programmers from common mistakes: NV-heaps provide automatic garbage collection, pointer safety, and protection from several novel kinds of bugs that non-volatile objects make possible. NV-heaps are completely self-contained, allowing the system to copy, move, or transmit them just like normal files.

In designing NV-heaps, our goals are to provide safe access to persistent objects, to make persistent objects easy to program, and to achieve high performance. To this end, our system has the following of properties:

1. **Pointer safety.** NV-heaps, to the extent possible, prevent programmers from corrupting the data structures they create by misusing pointers or making memory allocation errors.
2. **Flexible ACID transactions.** Multiple threads can modify NV-heaps concurrently. NV-heaps are robust against application and system failure.
3. **Familiar interface.** The programming interface for NV-heaps is similar to the familiar interface for implementing volatile data structures.
4. **High performance.** Access to the data in an NV-heap is as fast as possible relative to the speed of the underlying non-volatile memory.
5. **Scalability.** NV-heaps are designed to scale to very large (many gigabytes to terabytes) data structures.

The paper describes the NV-heaps model and our Linux-based implementation in detail. It highlights the aspects of NV-heaps that minimize software overheads and evaluates its performance using emulated fast memories. We use a set of benchmarks that includes several complex data structures as well as a non-volatile version of SSCA [5], a sparse graph analysis application. We also convert a persistent version of Memcached [16], the popular memory object caching system, to use NV-heaps. We compare NV-heaps to Stasis [53] and BerkeleyDB [46], two other interfaces to non-volatile data that target conventional block devices (e.g., disks). We also evaluate the performance scalability of NV-heaps, and evaluate the impact of the underlying non-volatile storage technology on their overall performance. In particular, we focus on phase-change memory (PCM) and spin-torque transfer memory (STTM) technologies.

Our results show that NV-heaps out-perform BerkeleyDB and Stasis by 32× and 244×, respectively, because NV-heaps eliminate the operating system from common-case operations and minimize other software overheads. We also compare NV-heaps to a Rio Vista-like [38] version of the system that provides no safety guarantees and find that the cost of safety is a 11× drop in performance. Our experience programming with NV-heaps and the unsafe version of the system leads us to conclude that the benefits in terms of ease-of-use more than make up for the performance cost.

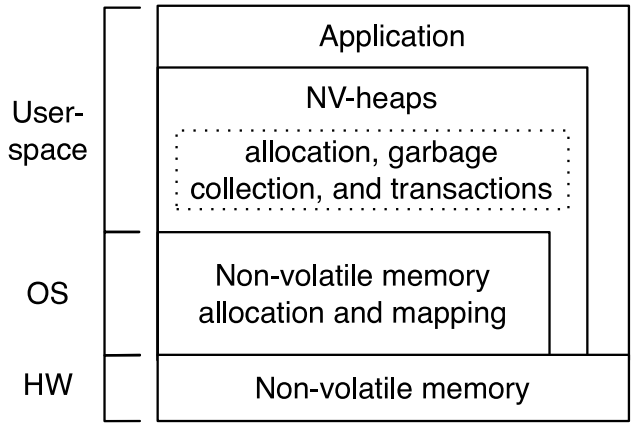

Figure 1. The NV-heap system stack This organization allows read and write operations to bypass the operating system entirely.

The remainder of this paper is organized as follows. We present an overview of NV-heaps in Section 2. Section 3 describes our implementation of NV-heaps. Sections 4 and 5 describe our methodology and results. Section 6 concludes.

2. NV-heaps

The goal of NV-heaps is to make it easy to build and use robust, persistent data structures that can exploit the performance that emerging non-volatile, solid-state memories offer. To achieve this, NV-heaps provides an easy-to-use application-level interface to a persistent object system tailored to emerging non-volatile memories. The designs of previous persistent object systems [3, 10, 12, 34, 57, 61] focus on hiding disk latency rather than minimizing software overhead, making them a poor fit for these new memory technologies.

NV-heaps provide a small set of simple primitives: persistent objects, specialized pointer types, a memory allocator, and atomic sections to provide concurrency and guard against system or application failure. NV-heaps hide the details of locking, logging, and memory management, so building a data structure in an NV-heap is very similar to building one in a conventional, volatile heap.

To use an NV-heap, the application opens it by passing the NV-heap library a file name. The library maps the NV-heap directly into the application's address space without performing a copy, which is possible because the underlying memory is byte-addressable and appears in the processor's physical address space. Once the mapping is complete, the application can access the NV-heap's contents via a *root pointer* from which all data in the NV-heap is accessible.

Figure 1 shows the relationship between NV-heaps, the operating system, and the underlying storage. A key feature of NV-heaps is that they give the application direct access to non-volatile memory, eliminating operating system overhead in most cases and significantly increasing performance.

Below, we make the case for the strong safety guarantees that NV-heaps provide. Then, we describe the transaction, pointer safety, performance, and usability features that NV-heaps include along with an example of NV-heaps in action. We discuss the differences between NV-heaps and existing persistent object systems throughout. Section 3 describes the implementation in detail.

2.1 Preventing programmer errors

Integrating persistent objects into conventional programs presents multiple challenges. Not only must the system (or the programmer) maintain locking and memory allocation invariants, but it must also enforce a new set of invariants on which objects belong to which region of memory. Potential problems arise if one NV-heap contains a pointer into another NV-heap or into the volatile heap. In either case, when a program re-opens the NV-heap, the pointer will be meaningless and potentially dangerous. Furthermore, violating any of these invariants results in errors that are, by definition, persistent. They are akin to inconsistencies in a corrupt filesystem.

We believe that if persistent objects are to be useful to the average programmer, they must be fast *and* make strong safety guarantees. Providing a low-level interface and expecting the programmer to "get it right" has proven to be a recipe for bugs and unreliability in at least two well-known domains: Memory management and locking disciplines. In both of those instances, there is a program-wide invariant (i.e., which code is responsible for `free()`ing an object and which locks protect which data) that the source code does not explicitly describe and that the system does not enforce. A persistent object system must contend with both of these in addition to the constraints on pointer usage in NV-heaps.

To understand how easy it is to create dangerous pointers in NV-heaps, consider a function

```
Insert(Object * a, List<Object> * l) ...
```

that inserts a pointer to a non-volatile object, `a`, into a non-volatile linked list, `l`. The programmer must ensure that `a` and `l` are part of the same non-volatile structure, but there is no mechanism to enforce that constraint since it is not clear whether `a` is volatile or non-volatile or which NV-heap it belongs to. One incorrect call to this function can corrupt `l`: It might, for instance, end up containing a pointer from an NV-heap into volatile memory. In either case, if we move `l` to another system or restart the program, `l` is no longer safe to use: The pointer to object `a` that the list contains has become a "wild" pointer.

There is also real-world evidence that non-volatile data structures are difficult to implement correctly. Microsoft Outlook stores mail and other personal information in a pointer-based Personal Folder File (PFF) file format [43]. The file format is complex enough that implementing it correctly proved difficult. In fact, Microsoft eventually released the "Inbox Repair Tool" [44] which is similar to `fsck`-style tools that check and repair file systems.

Another system, BPFS [17], highlights what it takes to build a robust non-volatile data structure on top of raw non-volatile memory. BPFS implements a transactional file system directly atop the same kind of byte-addressable non-volatile memories that NV-heaps target. BPFS uses carefully designed data structures that exploit the file system's tree structure and limited set of required operations to make transactional semantics easy to implement in most cases. In doing so, however, it enforces stringent invariants on those structures (e.g., each block of data has only a single incoming pointer) and requires careful reasoning about thread safety, atomicity, and memory access ordering. BPFS is an excellent example of what skilled programmers can accomplish with non-volatile memories, but average users will, we expect, be unwilling (or unable) to devise, enforce, and reason about such constraints. NV-heaps remove that burden, and despite additional overheads, they still provide excellent performance (see Section 5).

Existing systems that are similar to NV-heaps, such as Rio Vista [38] and RVM [51], provide direct access to byte addressable, non-volatile memories and let the programmer define arbitrary data structures. But these systems do not offer protection from memory allocation errors, locking errors, or dangerous non-volatile pointers. Concurrent work on a system called Mnemosyne [60] goes further and provides a set of primitives for operating on data in persistent regions. It would be possible to implement the key features of NV-heaps using these primitives. Systems that target disk-based storage have either implemented persistent objects on top of a conventional database (e.g., the Java Persistence API [7]) or in a specialized object-oriented database [3, 10, 12, 34, 57, 61]. In these systems, the underlying storage system enforces the invariants automatically, but they extract a heavy cost in terms of software overhead.

2.2 Transactions

To make memory-mapped structures robust in the face of application and system failures, the system must provide useful, well-defined guarantees about how and when changes to a data structure become permanent. NV-heaps use ACID transactions for this purpose because they provide an elegant method for mediating access to shared data as well as robustness in the face of failures. Programmer-managed locks cannot provide that robustness. Recent work on (volatile) transactional memory [8, 22, 24, 26, 27, 50, 56] demonstrates that transactions may also be easier to use than locks in some cases.

Systems that provide persistence, including persistent object stores [34, 37, 57, 61], some persistence systems for Java [7, 41], Rio Vista [38], RVM [51], Mnemosyne [60], Stasis [53], Argus [36], and QuickSilver [25], provide some kind of transactions, as do relational databases. However, the type of transactions vary considerably. For example, Stasis provides page-based transactions using write-ahead logging, a technique inspired by databases [45]. Mnemosyne also uses write-ahead logging, but operates at the word granularity. RVM and Rio Vista provide transactions without isolation, and RVM provides persistence guarantees only at log flushes. Single-level stores have taken other approaches that include checkpoints [55] and explicitly flushing objects to persistent storage [58].

2.3 Referential integrity

Referential integrity implies that all references (i.e., pointers) in a program point to valid data. Java and other managed languages have demonstrated the benefits of maintaining referential integrity: It avoids memory leaks, "wild" pointers, and the associated bugs.

In NV-heaps, referential integrity is more important and complex than in a conventional system. Integrity problems can arise in three ways, and each requires a different solution.

Memory allocation NV-heaps are subject to the memory leaks and pointer bugs that all programming systems face. Memory leaks, in particular, are more pernicious in a non-volatile setting. Once a region of storage leaks away, reclaiming it is very difficult. Preventing such problems requires some form of automatic garbage collection.

NV-heaps use reference counting, which means that space is reclaimed as soon as it becomes dead and that there is never a need to scan the entire NV-heap. The system avoids memory leaks due to cycles by using a well-known technique: weak pointers that do not affect reference counts. Several other garbage collection schemes [33, 47] for non-volatile storage have been proposed, and integrating a similar system into NV-heaps is a focus of our ongoing work.

Previous systems have taken different approaches to memory management. Non-volatile extensions to Java [4, 41] provide garbage collection, but Rio Vista [38], RVM [51], and Mnemosyne [60] do not. Java's persistence API [7] requires the programmer to specify a memory management policy via flexible (and potentially error-prone) annotations. Object-oriented databases have taken a range of approaches from providing simple

garbage collection [10] to allowing applications to specify complex structural invariants on data structures [3].

Volatile and non-volatile pointers NV-heaps provide several new avenues for creating unsafe pointers because they partition the address space into a volatile memory area (i.e., the stack and volatile heap) and one or more NV-heaps.

This partitioning gives rise to four new types of pointers: Pointers within a single NV-heap (*intra-heap* NV-to-NV pointers), pointers between two NV-heaps (*inter-heap* NV-to-NV pointers), pointers from volatile memory to an NV-heap (V-to-NV pointers), and pointers from an NV-heap to volatile memory (NV-to-V pointers).

Ensuring referential integrity in NV-heaps requires that the system obey two invariants. The first is that there are no NV-to-V pointers, since they become meaningless once the program ends and would be unsafe the next time the program uses the NV-heap.

The second invariant is that there are no inter-heap NV-to-NV pointers. Inter-heap pointers become unsafe if the NV-heap that contains the object is not available. Inter-heap pointers also complicate garbage collection, since it is impossible to tell if a given location in an NV-heap is actually dead if a pointer in another (potentially unavailable) NV-heap may refer to it.

NV-heaps enforce these invariants via a simple dynamic type system. Each pointer and each object carries an identifier of the heap (NV-heap or volatile heap) that it belongs to. Mismatched assignments are a run-time error. As far as we know, NV-heaps are the first system to explicitly identify and prohibit these types of dangerous pointers. Rio Vista [38] and RVM [51] make no attempts to eliminate any of these pointer types. Also, full-fledged persistent object systems such as ObjectStore do not guard against these dangerous pointers, leaving the system and underlying database vulnerable to corruption [23]. However, other systems effectively eliminate dangerous pointers. JavaCard [2], a subset of Java for the very constrained environment of code running on a smart card, makes almost all objects persistent and collects them in a single heap. In Java's persistence API [7] the underlying database determines which NV-to-NV pointers exist and whether they are well-behaved. It prohibits NV-to-V pointers through constraints on objects that can be mapped to rows in the database.

Closing NV-heaps Unmapping an NV-heap can also create unsafe pointers. On closing, any V-to-NV pointers into the NV-heap become invalid (but non-null). Our implementation avoids this possibility by unmapping NV-heaps only at program exit, but other alternatives are possible. For instance, a V-to-NV pointer could use a proxy object to check whether the NV-heap it points into is still open.

2.4 Performance and scalability

NV-heaps provide common case performance that is close to that of the underlying memory for data structures that scale up to the terabyte range. All common case operations (e.g., reading or writing data, starting and completing transactions) occur in user space. NV-heaps make system calls to map themselves into the application's address space and to expand the heap as needed. Entering the OS more frequently would severely impact performance since system call overheads are large (e.g., 6 μs for a 4 KB read on our system) compared to memory access time.

The result is a system that is very lightweight compared to previous persistent object systems that include sophisticated buffer management systems to hide disk latency [11, 30, 34, 57, 61] and/or costly serialization/deserialization mechanisms [7]. Unlike these systems, NV-heaps operate directly on non-volatile data that is accessible through the processor-memory bus, thereby avoiding the operating system. Rio Vista [38] is similar to NV-heaps, since it runs directly on battery-backed DRAM, but it does not provide any of the safety guarantees of NV-heaps. Mnemosyne [60] also provides direct access to fast, non-volatile memories, but providing the level of safety in NV-heaps requires more effort.

We have designed NV-heaps to support multi-terabyte data structures, so we expect that the amount of non-volatile storage available in the system may be much larger than the amount of volatile storage. To allow access to large structures, NV-heaps require a fixed, small amount of volatile storage to access an NV-heap of any size. NV-heaps also ensure that the running time of operations, including recovery, are a function only of the amount of data they access, not the size of the NV-heap.

2.5 Ease of use

To be useful, NV-heaps need to be easy to use and interact cleanly with each other and with existing system components. NV-heaps also need to make it clear which portions of a program's data are non-volatile and which NV-heap they belong to.

NV-heaps exist as ordinary files in a file system in a manner similar to previous persistent object store implementations [57, 61]. Using files for NV-heaps is crucial because the file system provides naming, storage management, and access control. These features are necessary for storage, but they are not needed for systems like [48, 49, 62] that simply use non-volatile memories as a replacement for DRAM.

Like any file, it is possible to copy, rename, and transmit an NV-heap. This portability means that NV-heaps must be completely self-contained. The prohibition against NV-to-V and inter-heap NV-to-NV pointers guarantees this isolation. An additional feature, relative pointers (described in Section 3), provides (nearly) zero-cost pointer "swizzling" and makes NV-heaps completely relocatable within an application's address space.

We chose to make NV-heaps self-contained to make them easier to manage with existing file-based tools. However, this choice means that NV-heaps do not natively support transactions that span multiple NV-heaps. Implementing such transactions would require that NV-heaps share some non-volatile state (e.g., the "committed" bit for the transaction). The shared state would have to be available to both NV-heaps at recovery, something that self-contained NV-heaps cannot guarantee. It is possible to move objects between non-volatile data structures, but those structures need to reside in the same NV-heap.

While NV-heaps make it easy to implement non-volatile data structures, they do not provide the "orthogonal persistence" that persistent object stores [34, 57, 61] and some dialects of Java [4, 18, 41] provide. Orthogonal persistence allows programmers to designate an existing pointer as the "root" and have all objects reachable from that pointer become *implicitly* persistent regardless of their type. This is an elegant abstraction, but using reachability to confer persistence leads to several potential problems. For instance, the programmer may inadvertently make more data persistent than intended. In addition, the abstraction breaks down for objects that cannot or should not be made persistent, such as an object representing a network connection, a file descriptor, or a secret key. Finally, it is possible for a single object to be reachable from two roots, leading to the confusing situation of multiple copies of the same object in two different persistent structures.

NV-heaps provide an alternative model for persistence. Each NV-heap has a designated root pointer, and everything reachable from the root is persistent and part of the same NV-heap. The difference is that the program *explicitly* creates a persistent object in the NV-heap and attaches it to another object in the heap with a pointer. This does not prevent all errors (e.g., it is still possible to inappropriately store a file descriptor in an NV-heap), but it requires that the programmer explicitly add the data to the NV-heap. It also prevents a single object from being part of two NV-heaps.

```
class NVList : public NVObject {
  DECLARE_POINTER_TYPES(NVList);
public:
  DECLARE_MEMBER(int, value);
  DECLARE_PTR_MEMBER(NVList::NVPtr, next);
};

void remove(int k)
{
  NVHeap * nv = NVHOpen("foo.nvheap");
  NVList::VPtr a =
           nv->GetRoot<NVList::NVPtr>();
  AtomicBegin {
    while(a->get_next() != NULL) {
      if (a->get_next()->get_value() == k) {
        a->set_next(a->get_next()->get_next());
      }
      a = a->get_next();
    }
  } AtomicEnd;
}
```

Figure 2. NV-heap example A simple NV-heap function that atomically removes all links with value `k` from a non-volatile linked list.

This model for persistence is similar to what is imposed by the Thor [37] persistent object store, but Thor does so through a type-safe database programming language.

2.6 Example

The code in Figure 2 provides an example of how a programmer can create a non-volatile data structure using NV-heaps. The code removes the value `k` from a linked list. Declaring the linked list class as a subclass of `NV_Object` marks it as non-volatile. The `DECLARE_POINTER_TYPES`, `DECLARE_MEMBER`, and `DECLARE_PTR_MEMBER` macros declare the smart pointer types for NV-to-NV and V-to-NV pointers (`NVList::NV_Ptr` and `NVList::V_Ptr`, respectively) and declare two fields. The declarations generate private fields in the class and public accessor functions (e.g., `get_next()` and `set_next()`) that provide access to data and perform logging and locking.

The program uses `NVHOpen()` to open an NV-heap and then retrieves the root object, in this case a list of integers. It stores the pointer to the linked list as a NVList::V_Ptr. `AtomicBegin` starts a transaction. When the atomic section is complete, the NV-heap attempts to commit the changes. If it fails or if the system crashes, it will roll the operations back.

In the next section we describe the implementation of the NV-heap library in detail.

3. Implementing NV-heaps

Two considerations drove our implementation of NV-heaps: The need for strong safety guarantees and our goal of maximizing performance on fast, non-volatile memories. We implemented NV-heaps as a C++ library under Linux. The system is fully functional running on top of a RAM disk backed by DRAM. Below, we describe the technologies that NV-heaps target and the support they require from the OS and hardware. Then we describe our implementations of memory management, reference safety, and transactions. Finally, we discuss the storage overheads and how we validated our implementation.

3.1 Fast, byte-addressable non-volatile memories

NV-heaps target solid-state memory technologies that present a DRAM-like interface (e.g., via LPDDR [31]) and achieve performance within a small factor of DRAM. To evaluate NV-heaps we consider two advanced non-volatile memories: phase change memory (PCM) and spin-torque transfer memory (STTM).

PCM stores data as the crystalline state of a chalcogenide layer [9] and has the potential to become a viable main memory technology as DRAM's scaling falters [35, 49, 62]. PCM may also eventually surpass flash memory in density according to the ITRS [29]. The analysis in [35] provides a good characterization of PCM's performance and power consumption.

STTM stores bits as a magnetic orientation of one layer of a magnetic tunnel junction [20]. We assume 22nm STTM technology and base our estimates for performance on published papers [32, 59] and discussions with industry.

PCM and, to a lesser extent, STTM, along with most other non-volatile memories require some form of wear management to ensure reasonable device lifetime. Many wear-leveling schemes are available [15, 19, 35, 48, 62] and some can provide excellent wear-leveling at the memory controller level for less than 1% overhead. NV-heaps (like BPFS [17]) assume that the system provides this service to all the applications that use the storage.

3.2 System-level support

NV-heaps require a few simple facilities from the system. To the file system, NV-heaps are normal files. To access them efficiently, the file system should be running on top of byte-addressable non-volatile memory that appears in the CPU's physical address space. To open an NV-heap, the system uses `mmap()`. Normally, the kernel copies `mmap()`'d data between a block device and DRAM. In a system in which byte-addressable non-volatile memories appear in the processor's address space, copying is not necessary. Instead, `mmap()` maps the underlying physical memory pages directly into the application's virtual address space. In our kernel (2.6.28), the brd ramdisk driver combined with ext2 provides this capability.

The second requirement is a mechanism to ensure that previous updates to non-volatile storage have reached the storage and are permanent. For memory-mapped files, `msync()` provides this functionality, but the system call overhead is too high for NV-heaps. Instead, NV-heaps rely on architectural support in the form of the atomic 8-byte writes and epoch barriers developed for BPFS [17] to provide atomicity and consistency. Epoch barriers require small changes to the memory hierarchy and a new instruction to specify and enforce an ordering between groups of memory operations. Section 4 describes how we model the overhead for these operations. BPFS also provides durability support by incorporating capacitors onto the memory cards to allow in-progress operations to finish in the event of a power failure. We assume similar hardware support.

3.3 Memory management

The NV-heap memory management system implements allocation, automatic garbage collection, reference counting, and pointer assignments as simple, fixed-size ACID transactions. These basic operations form the foundation on which we build full-blown transactions.

Atomicity and Durability The allocator uses fixed-size, non-volatile, redo-logs called *operation descriptors* (similar to the statically-sized transactions in [56]) to provide atomicity and durability for memory allocation and reference-counted pointer manipulation. There is one set of operation descriptors per thread and the design ensures that a thread only ever requires one of each type of

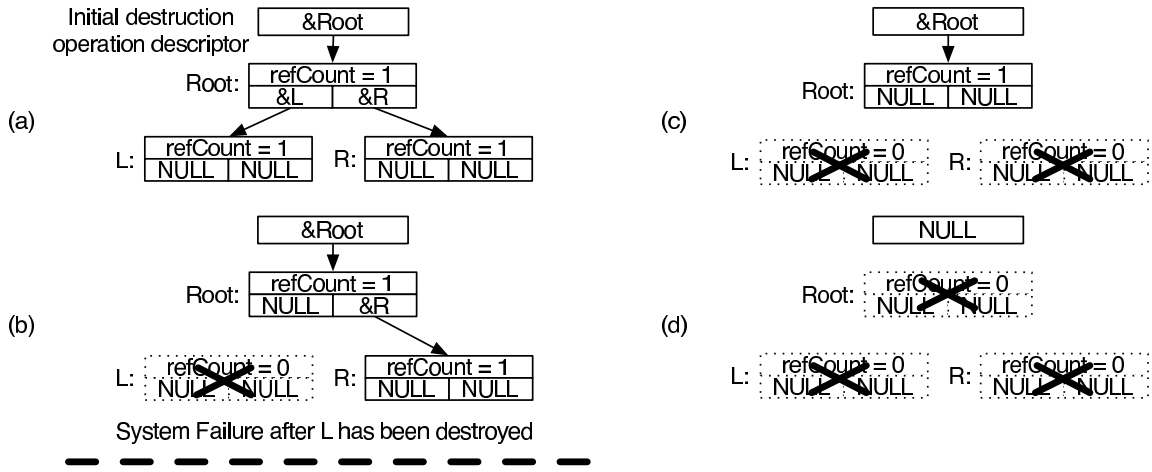

Figure 3. Restartable object destruction Durable, atomic reference counting requires being able to restart the recursive destruction of a potentially large numbers of objects. In this example the root node of a tree is destroyed and triggers the deletion of the entire tree. After a system failure, the process resumes and completes during the recovery process.

descriptor for all operations. Epoch barriers ensure that the descriptors are in a consistent state before the operation logically commits.

Concurrency To provide support for concurrent accesses to objects, each persistent object contains a lock that protects its reference count. The transaction system protects the other data in the object.

Locks are volatile by nature because they only have meaning during run-time. Therefore, to operate correctly, all locks must be released after a system failure. This could necessitate a scan of the entire storage array, violating our scalability requirement.

We avoid this problem by using *generational locks*: We associate a current *generation number* with each NV-heap, and the system increments the generation number when the NV-heap is reloaded. A generational lock is an integer. If the integer is equal to the current generation number, a thread holds it, otherwise, it is available. Therefore, incrementing the NV-heap's generation instantly releases all of its locks.

Allocation The allocator uses per-thread free lists and a shared global free list to reduce contention. If free space is not available, the NV-heap library expands the file that holds the NV-heap and maps that storage into the address space.

Prior work in memory management for multi-core systems played a part in the design of the NV-heap 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 [28].

Deallocation When the reference counting system discovers that an object is dead, it deallocates the storage. The deallocation routine atomically calls the destructor, deallocates the memory, and sets the requester's pointer to NULL. Deallcoation 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.

To implement atomic recursive destruction, the NV-heap records the top level object to be destroyed in the *root deletion* operation descriptor and then calls its destructor. When the destructor encounters a pointer to another object, it checks that object's reference count. If the reference count is one (i.e., this is the last pointer to the object), it performs that deallocation using the *non-root deletion* descriptor. If that deletion causes a further recursive deletion, the non-root descriptor is reassigned to that deletion. The combination of the root and non-root descriptors provide a snapshot of the recursive deletion process that 1) requires only two descriptors regardless of recursion depth and 2) allows the process to restart in the case of failure.

On recovery, the NV-heap processes the non-root descriptor first to restore the invariant that all non-null pointers in the structure point to valid objects. It then restarts the root deletion, which 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.

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 (b), 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 is not a problem, but some idioms that require the destructor to, for instance, track the number of live instances of an object will be more difficult to implement. In our experience using the system, however, this has not been a significant problem.

Recovery To recover from a system failure, the memory allocator performs the following two steps. First, it replays any valid operation descriptors for basic storage allocation and deallocation operations. Then, it replays any reference count updates and reference counting pointer assignments, which may include the recursive destruction process described above. When this is complete, all the reference counts are valid and up-to-date and the recovery process moves on to the transaction system.

3.4 Pointers in NV-heaps

The NV-heap library uses operator overloading to implement pointer types for NV-to-NV, V-to-NV, and weak NV-to-NV pointers. Their assignment operators, copy constructors, and cast operators work together with the memory allocator to enforce correct semantics for each pointer type.

The pointers play a key role in preventing the creation of inter-heap NV-to-NV pointers and NV-to-V pointers. NV-to-NV pointers are "wide" and include a pointer to the NV-heap they belong to. Non-volatile objects contain a similar pointer. The assignment operators for the pointer check that the assignment is valid (i.e., that the pointer and the object belong to the same NV-heap).

The smart pointer types also allow NV-heaps to be relocatable. Instead of holding the actual address (which would change from application to application or execution to execution), the pointer holds an offset from the pointer's address to the data it points to.

Below we describe the implementation of each pointer type.

NV-to-NV pointers NV-heaps support two types of NV-to-NV pointers. Normal NV-to-NV pointers affect reference counts and are the most common type of pointer in the applications we have written. They reside in the NV-heap and point to data in the same NV-heap.

Weak NV-to-NV pointers are similar but they do not affect an object's reference count. Weak pointers are required to implement cyclic data structures (e.g., doubly-linked lists) without introducing memory leaks. Ensuring that non-null weak NV-to-NV pointers point to valid data requires that when the object they refer to becomes dead (i.e., no more non-weak pointers refer to it), all the weak pointers should atomically become NULL. This may lead to an unexpected NULL pointer dereference, but it cannot result in corrupted data.

We use proxy objects to implement this behavior. Weak NV-to-NV pointers refer indirectly to the object via a proxy that contains a pointer to the actual object. When an object dies, its destructor sets the pointer in its proxy to NULL, instantly nullifying all the weak pointers. The system manages proxy objects with reference counting, similar to regular objects, because they must survive as long as there are weak pointers that refer to them.

V-to-NV pointers There are three key requirements for V-to-NV references. First, a V-to-NV reference must be sufficient to keep an object alive. Second, when a program exits and all the V-to-NV references are destroyed, the objects' reference counts must be adjusted accordingly. Third, the system must eventually reclaim any objects that become dead when a program exits and destroys all of the V-to-NV pointers.

To address the these issues, we add a second count of V-to-NV references to each object. One attractive solution is to store these counts in volatile memory. This would ensure they were reset correctly on exit, but, in the event of a system or application failure, the V-to-NV reference counts would be lost, making any dead objects now impossible to reclaim.

Our implementation stores the count of volatile references in non-volatile memory and uses the generation number technique we used for non-volatile locks to reset it. To locate orphaned objects, the NV-heap maintains a per-thread list of objects that only have V-to-NV references. On startup, the NV-heap reclaims everything in the list.

3.5 Implementing transactions

NV-heaps provide fine-grain consistency through atomic sections that log all updates to the heap in non-volatile memory. Atomic sections for NV-heaps build on previous work developing transactional memory systems for volatile memories [8, 22, 24, 26, 27, 50, 56]. The NV-heap transaction system is software-only (except for epoch barrier support), and it relies heavily on the transactional memory allocator and reference counting system described in Sections 3.3 and 3.4.

Transactions in NV-heaps Our implementation of NV-heaps provides ACID semantics for programs that meet two criteria: First, a program must use the accessor functions to access object fields and it must not use unsafe casts to circumvent the C++ type system. Second, transactions should not access shared *volatile* state, since the NV-heap transaction system does not protect it. We have not found this to be a significant problem in our use of NV-heaps, but we plan to extend the system to cover volatile data as well. Failure to adhere to the second condition only impacts isolation between transactions.

NV-heap's log processing is the main difference relative to conventional volatile transactional memory systems. NV-heaps maintain a volatile read log and a non-volatile write log for each transaction. When a transaction wants to modify an object, it must be opened for writing, meaning the system will make a copy of the object so that any changes can be rolled back in case of an abort or system/application failure. When this happens, the running atomic section must take ownership of the object by acquiring a volatile lock in a table of ownership records indexed by the low-order bits of the object's unique ID. If another transaction owns the object, a conflict has occurred, and the atomic section retries. Once the atomic section becomes the owner, the NV-heap copies the object into the log. To open an object for reading, NV-heaps store a pointer to the object and its current version number in the read log.

After the system makes a copy of the object, the application can safely modify the original object. If a system or application failure occurs or the atomic section aborts, the NV-heap rolls back the object to its original state by using the copy stored in the write log.

We choose to copy entire objects to the log rather than individual object fields, and this is a common trade-off in transactional memory systems. In NV-heaps, each log operation requires an epoch barrier which has some performance cost. Copying the entire object helps to amortize this cost, and it pays off most when transactions make multiple updates to an object, which is a common feature of the data structures that we studied.

NV-heaps borrow ideas from DSTM [26], RSTM [40], DracoSTM [21], and McRT-STM [50]. The system logs entire objects as opposed to individual fields, and uses eager conflict detection for writes. It detects read conflicts by validating objects at access time using version numbers. It stores undo logs for outstanding transactions in non-volatile memory for recovery. The contention management scheme [52] backs off and retries in case of conflict. NV-heaps flatten nested transactions into a single transaction.

Transaction abort and crash recovery The processes for aborting a transaction and recovering from a system or application failure are very similar: NV-heaps roll back the transaction by restoring the backup copies from the log. In the case of a crash, the system first follows the recovery procedure defined in Section 3.3 to ensure that the memory allocator is in a consistent state and all reference counts are up-to-date.

An additional concern with crash recovery is that recovery must be restartable in the case of multiple failures. NV-heaps recover from failure by only rolling back valid log entries and using an epoch barrier to ensure that an entry's rollback is durably recorded in non-volatile storage before marking the entry invalid.

3.6 Storage and memory overheads

The storage overheads of NV-heaps are small. Each NV-heap contains a control region that holds the root pointer, pointers to the free lists, the operation descriptors, version information, and the current generation number. The storage requirement for the control area is 2 KB plus 1.5 KB per thread for operation descriptors. The NV-heap also uses 0.5 KB of volatile memory.

NV-to-NV and V-to-NV pointers are 128-bits which includes a 64-bit relative pointer and dynamic type information to prevent assignments that would create unsafe pointers. Each object includes 80 bytes of metadata including reference counts, a unique ID,

ownership information, a generational lock, and other state. For small objects such as primitive data types and pointers, we provide an array object type to amortize the metadata overhead across many elements.

Supporting transactions requires 80 bytes of per-thread transaction state (e.g., pointers to logs) in addition to storage for the write logs.

3.7 Validation

To validate our implementation of the NV-heap allocator and transaction system, we created a set of stress tests that create complex objects and perform concurrent transactions on them. These tests exercise the concurrency and safety of the memory allocation operations, pointer assignments, and the user-defined transactions which run on top of them. We run these tests with up to eight threads for long periods (many hours) and observe no deadlock or data corruption.

To test recovery from failures, we run the tests and kill the program with `SIGKILL` at random intervals. During the recovery process we record which logs and operation descriptors the recovery system processes. Then we perform a consistency check. After killing our test programs thousands of times, we have observed numerous successful recoveries involving each descriptor and log type. In all cases, the recoveries were successful and the consistency checks passed.

4. Modeling fast non-volatile storage

NV-heaps aim to support systems with many gigabytes of high-performance non-volatile memory, but mature products based on those memories will take several years to appear. In the meantime, we use two emulation systems to run applications for many billions of instructions while simulating the performance impact of using advanced non-volatile memories.

4.1 Modeling byte-addressable storage

The first emulation system models the latency for memory-level load and store operations to advanced non-volatile memories on the processor's memory bus. The system uses Pin [39] to perform a detailed simulation of the system's memory hierarchy augmented with non-volatile memory technology and the atomicity support that NV-heaps require. The memory hierarchy simulator accounts for both the increased array read time and the added delay between a write and the operations that follow, allowing it to accurately model the longer read and write times of PCM and STTM memories. For PCM we use the performance model from [35] which gives a PCM read time of 67 ns and a write time of 215 ns. We model STTM performance (29 ns reads and 95 ns writes) based on [59] and discussion with industry. The baseline DRAM latency for our system is 25 ns for reads and 35 ns for writes, according to the datasheet.

We run the simulation on the first 100 million instructions out of each one billion instructions executed. The simulation provides the average latency for last level cache hits and misses and for the epoch barriers. After the simulation phase, we use hardware performance counters to track these events on a per-thread basis, and combine these counts with the average latencies to compute the total application run-time. The model assumes that memory accesses execute serially, which makes our execution time estimates conservative.

To calibrate our system we used a simple program that empirically determines the last-level cache miss latency. We ran the program with the simulated PCM and STTM arrays and its estimates matched our target latencies to within 10%.

4.2 Modeling block device based on advanced non-volatile memory

In Section 5, we compare NV-heaps to two systems, Stasis [53] and BerkelelyDB [46], that target a block device (i.e., a disk) instead of a non-volatile main memory. To model a non-volatile memory-based block device, we modified the Linux RAM disk driver to let us insert extra delay on accesses to match the latency of non-volatile memories. Measurements with a simple disk latency benchmark show that the emulation is accurate to within about 2%.

5. Results

This section describes our evaluation of NV-heaps. We describe the test system and then present experiments that measure basic operation latency. Next, we evaluate its scalability and performance on a range of benchmarks. We examine the overheads that NV-heaps incur to provide strong safety guarantees. Then, we compare NV-heaps to Stasis [53] and BerkeleyDB [46], transactional storage systems that target conventional block devices. Finally, we evaluate performance at the application level by implementing a version Memcachedb [16], a persistent key-value store for dynamic Web applications, with NV-heaps.

5.1 System configuration

We present results collected on two-socket, Core 2 Quad (a total of 8 cores) machines running at 2.5 GHz with 64 GB of physical DRAM and 12 MB L2 caches. These machines are equipped with both a conventional 250 GB hard drive and a 32GB Intel Extreme flash-based SSD. We configure the machines with a 32GB RAM disk. We use 24 GB for emulated non-volatile memory and 8 GB for program execution. For the experiments that use disks and the SSD we report "wall clock" timing measurements.

5.2 Basic operation performance

Table 1 summarizes the basic operation latencies for several implementations of NV-heaps that isolate different overheads that NV-heaps incur. BASE, SAFE, TX, and C-TX represent four layers of the system that offer varying levels of safety and performance. The BASE layer provides manual memory management and checkpointing, yielding a system very similar to Rio Vista [38]. The SAFE layer adds automatic memory management and pointer safety. The third layer, TX, extends SAFE with transactions for atomicity and durability in single-threaded programs. Layer C-TX provides support for concurrency through multi-threaded transactions. Finally, the version NoDur is a volatile transactional memory system without support for durability. The table reports latencies for each implementation of NV-heaps running on DRAM. It also reports measurements for the C-TX version running on emulated STTM and PCM.

The value for "new/delete" is the time to allocate and deallocate a very small object. The three "ptr" rows give the time to assign to a pointer and then set it to NULL. The "nop tx" is the time to execute an empty transaction. "Log for read" and "Log for write" give the times to log an object before access.

For C-TX (i.e., full-fledged NV-heaps), the most expensive operation is logging an object for writing. This operation only occurs once per modified object per transaction and requires an allocation, a copy, one pointer manipulation, and several epoch barriers. The cost of an epoch barrier depends on the size of the epoch and whether or not the corresponding cache lines have been written back yet. Our operation descriptors tend to be small, so the cost is often the time to flush a only single cache line.

In contrast, V-to-NV pointer manipulation and read logging are extremely inexpensive, because they do not require durability guarantees or, therefore, epoch barriers. In fact, these operations

Layer	BASE	SAFE	TX	C-TX			NoDur
Storage technology	DRAM (μs)			DRAM (μs)	STTM (μs)	PCM (μs)	DRAM (μs)
new/delete	<0.1	<0.1	<0.1	<0.1	0.75	2.16	<0.1
V-to-NV ptr	0.03	0.05	0.11	0.13	0.13	0.13	0.08
NV-to-NV ptr	0.03	0.05	0.17	0.25	0.72	1.68	0.13
weak NV-to-NV ptr	n/a	0.05	0.20	0.25	0.78	1.84	0.15
nop tx	n/a	n/a	0.05	0.05	0.05	0.05	0.05
log for read	n/a	n/a	0.17	0.26	0.26	0.26	0.21
log for write	n/a	n/a	1.68	1.99	5.55	12.67	1.00

Table 1. Basic operation latency for NV-heaps Support for durability at various levels and the increased latency for PCM and STTM both exact a toll in terms of basic operation latencies. Latencies for "new/delete" are listed as <0.1 because of inconsistencies due to caching effects.

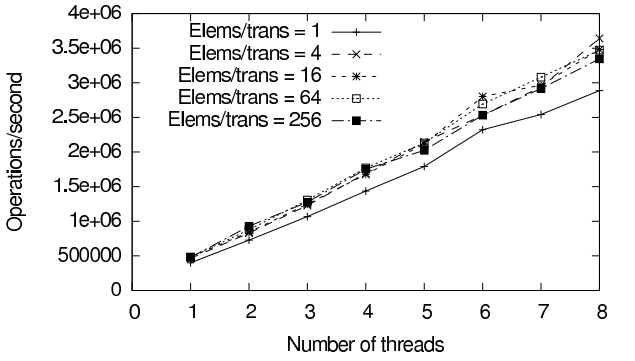

Figure 4. Throughput vs. transaction size. We scaled the number of elements a transaction modifies from one to 256 as we scaled the number of threads in order to show the effect of transaction size on throughput.

can occur entirely in the CPU's caches, so the impact of longer memory latencies is minimal.

The PCM and STTM data show the impact that slower underlying memory technology will have on basic operation performance. PCM's longer write latency increases the cost of write logging by 6.4× relative to DRAM. STTM shows a smaller increase — just 2.8×.

Figure 4 shows how performance scales with transaction size. It measures throughput for updates to an array of one million elements as we vary the number of updates performed in a single transaction from one to 256. 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. Overall, the scaling is good: Eight threads provide 7.5× the throughput of a single thread.

5.3 Benchmark performance

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 NV-heaps. Table 2 describes them.

Figure 5 shows the performance of the benchmarks. The numbers inside each bar are the operations (inserts/deletes for BTree, RBTree, and HashTable; path searches for SixDegs; updates or swaps for SPS; iterations through the outer loop of the final phase for SSCA) per second. The graph normalizes performance to NoDur with one thread.

Name	Footprint	Description
SPS	24 GB	Random swaps between entries in an 8 GB array of integers.
SixDegs	8 GB	Concurrently performs two operations: 1) Search for a path of length no more than six between two vertices in a large, scale-free graph 2) modify the graph by inserting and removing edges.
BTree	4 GB	Searches for an integer in a B-tree. Insert it if it is absent, remove it otherwise.
HashTable	8 GB	Searches for an integer in an open-chain hash table. Insert it if it is absent, remove it otherwise.
RBTree	24 GB	Searches for an integer in a 24 GB red-black tree. Insert it if it is absent, remove it otherwise.
SSCA	3 MB	A transactional implementation of SSCA 2.2 [5]. It performs several analyses of a large, scale-free graph.

Table 2. Workloads We use six workloads of varying complexity to evaluate NV-heaps.

The difference between the NoDur and DRAM bar in each group shows that adding durability to the transaction and memory management systems reduces performance by 40% on average. The cost is largest for SixDegs, because it executes complex transactions and requires frequent epoch barriers. The remaining bars show that the impact of increasing latency of the underlying memory technology is roughly proportional to the change in latency.

Differences in program behavior lead to varied scaling behavior. SPS, HashTable, and BTree scale very well (7.6×, 7.3×, and 7×, respectively, with 8 threads), while SixDegs and SSCA scale less well due to long transactions and additional conflicts between atomic sections with increasing thread count. We have found that much of this contention is due to the applications rather than NV-heaps. For instance, reducing the search depth in SixDegs from six to one improves scalability significantly.

5.4 The price of safety

To understand the cost of NV-heap's usability features, we have implemented our benchmarks in the layers BASE, SAFE, TX, and C-TX described previously. Figure 6 shows the performance of our benchmarks implemented in each NV-heaps layer. The absolute performance is given in operations per second in the BASE bar

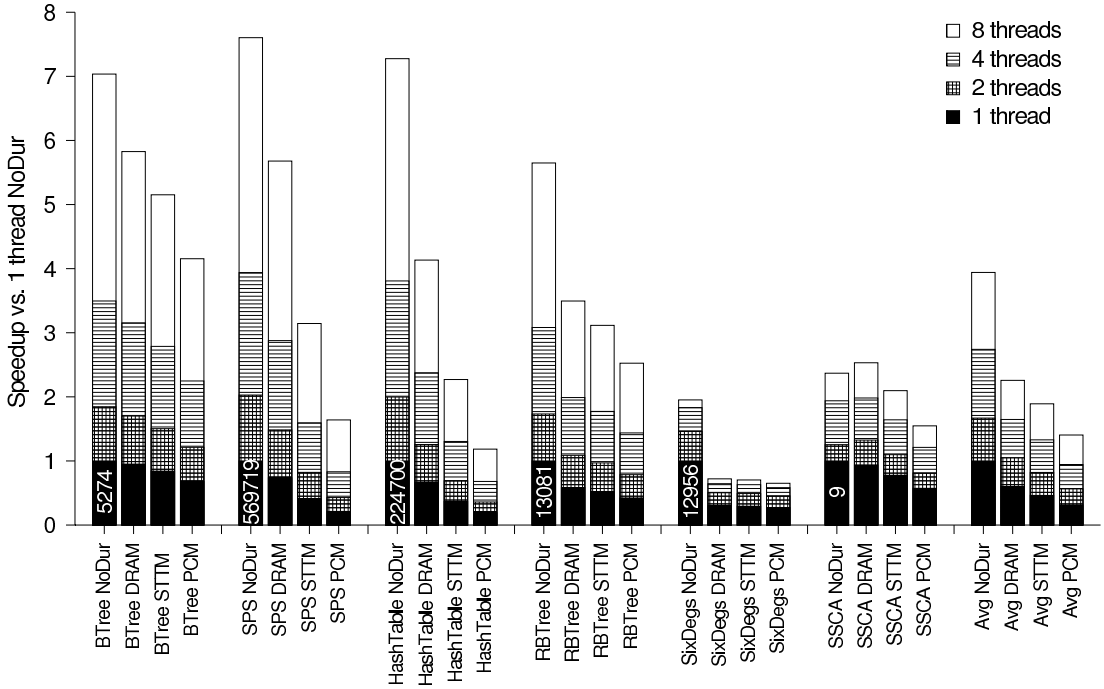

Figure 5. NV-heap performance Both adding durability and running on non-volatile memory affect NV-heap performance. Numbers in the bars are the throughput for the single-threaded version.

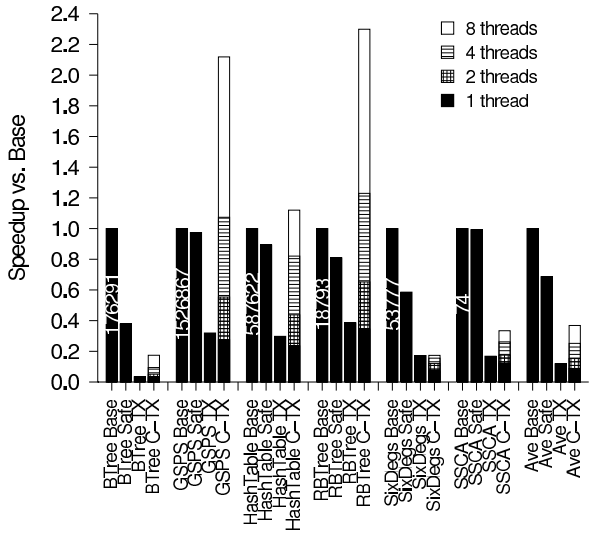

Figure 6. The price of safety in NV-heaps Removing safety guarantees improves performance by as much as 11×, but the resulting system is very difficult to use correctly. The different layers highlight the safety and performance trade-off.

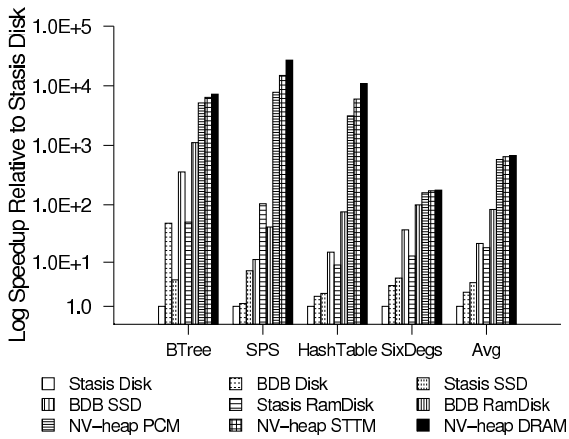

Figure 7. Comparison to other persistent storage systems NV-heaps outperform Berkeley DB and Stasis by 33 and 244× on average in large part because NV-heaps do not require system calls to provide durability guarantees.

for each benchmark. For a single thread, BASE provides the highest performance, but SAFE results in only a modest performance hit for most applications. One exception is BTree, which makes extensive use of NV-to-NV and weak NV-to-NV references, resulting in 62% lower performance for SAFE. This suggests that, for many applications, the extra safety that SAFE provides will be worth the performance penalty.

TX exacts a larger toll, reducing performance by 82% on average versus SAFE. The cost of durability is due to the copying required to log objects in non-volatile memory. The price is especially steep for BTree, because writing a node of the tree into the log requires many pointer copies.

Adding support for concurrency and conflict detection in C-TX has a small effect on peformance (30% on average relative to TX), and allows HashTable, RBTree, and SPS to reclaim much of their lost performance.

The gap in performance between BASE and C-TX, 11× on average for single-threaded programs, is the cost of safety in our system. The increase in performance of BASE is significant, but the price in terms of usability is high. The programmer must explicitly manage concurrency, atomicity, and failure recovery while avoiding the accidental creation of unsafe pointers. Whether this extra effort is worth the increased performance depends on the application and the amount of time the programmer is willing to spend testing and debugging the code. We found programming in this style to be tedious and error-prone.

5.5 Comparison to other systems

This section compares NV-heaps to two other systems that also provide transactional access to persistent state. We compare NV-heaps to Stasis [53], a persistent object system that targets disk and BerkeleyDB [54], a lightweight database. Both provide ACID transactions.

NV-heaps, Stasis, and BerkeleyDB have a similar high-level goal — to provide an easy-to-use interface to persistent data with strong consistency guarantees. Stasis and BerkeleyDB, however, target conventional spinning disks rather than byte-addressable storage. This means they must use system calls to force updates to disk and provide durability while NV-heaps take advantage of fast epoch barriers to enforce ordering.

Figure 7 compares NV-heap DRAM, NV-heap STTM, and NV-heap PCM to Stasis and BerkeleyDB running on three different hardware configurations: An enterprise hard disk, an Intel Extreme 32 GB SSD, and a RAM-disk. We implemented the data structures in Stasis ourselves. We used BerkeleyDB's built-in BTree and HashTable implementations and implemented SixDegs by hand. The vertical axis is a log scale. The data are for four threads. These results are for smaller, 4 GB data sets to keep initialization times manageable for the disk-based runs.

The first six bars in each group measure BerkeleyDB's and Stasis' performance on disks, SSDs, and the RAM-disk. The data show that while BerkeleyDB and Stasis benefit from running on fast non-volatile memory (e.g., BerkeleyDB on the RAM-disk is 3.2× faster than on the SSD and 24× than running on disk), the benefits are only a fraction of the raw speedup that non-volatile memories provide compared to SSDs and hard disks.

The next three bars in each group show that NV-heaps do a much better job exploiting that raw performance. NV-heap DRAM is between 2 and 643× faster than BerkeleyDB, and the performance difference for Stasis is between 13 and 814×. Two components contribute to this gap. The first is the `fsync()` and/or `msync()` required for durability on a block device. Removing this overhead by disabling these calls (and sacrificing durability) improves performance by between 2 and 10× for BerkeleyDB. The remaining gap in performance is due to other software overheads.

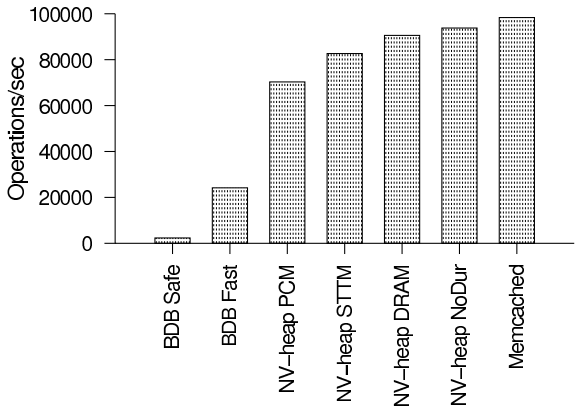

Figure 8. Memcached performance Using NV-heaps brings performance to within 8% of the original, non-durable Memcached, and the NV-heaps version achieves 39× higher throughput than Memcachedb (BDB Safe) which provides similar safety guarantees.

Comparing NV-heap performance to BerkeleyDB demonstrates that NV-heaps are both flexible and efficient. The NV-heap PCM BTree is 6.7× faster than BerkeleyDB running on the same technology, despite the fact that the BerkeleyDB version is a highly-optimized, specialized implementation. In contrast, the NV-heap implementation uses just the components that NV-heaps provide. We see similar performance for SPS and HashTable. For SixDegs, BerkeleyDB is nearly as fast as the NV-heaps, but this is, in part, because the BerkeleyDB version does not include reference counting. Removing dead nodes from the graph requires a scan of the entire table that stores them.

It is worth noting that our results for comparing BerkeleyDB and Stasis do not match the results in the original Stasis paper [53]. We suspect this is due to improvements in BerkeleyDB over the past five years and/or inefficiencies in our use of Stasis. However, assuming results similar to those in the original paper would not significantly alter the above conclusions.

5.6 Application-level performance

This section measures the impact of NV-heaps at the application level. We focus on Memcachedb [16], a version of Memcached [42] that provides a persistent key-value store. By default, Memcachedb uses BerkeleyDB to store the key-value pairs and provide persistence.

Our version of Memcachedb uses the NV-heap open-chaining hash table implementation we evaluated in Section 5.3 to hold the key-value store. All operations on the key-value store are transactional.

Figure 8 shows the throughput of insert/delete operations for the original Memcached application, the Berkeley DB implementation (Memcachedb), and our NV-heap implementation. All tests use 16 byte keys and 512 byte values. The multi-threaded client test program uses the libMemcached [1] library, and it runs on the same machine as the server to put maximum pressure on the key-value store. We measure performance of the BerekeleyDB-based version of Memcachedb with (BDB Safe) and without (BDB Fast) synchronous writes. Memcachedb uses the BerkeleyDB hash table implementation and runs on a RAM disk.

Compared to the original Memcached running in DRAM, adding persistence with NV-heaps results in only an 8 to 16% performance penalty depending on the storage technology. When we run NV-heaps without durability (NoDur), performance is within

just 5% of Memcached, indicating that the overhead for durability can be low in practice.

The data show that our hash table implementation provides much higher throughput than BerkeleyDB and that the overhead for providing transactional reliability is much lower with NV-heaps. NV-heap DRAM outperforms BDB Safe by up to 39× and NV-heap NoDur outperforms BDB Fast by 3.9×. BerkeleyDB provides many features that our hash table lacks, but for this application those features are not necessary. This highlights one of the advantages of NV-heaps — they allow programmers to provide (and pay the performance penalty for) only the features they need for a particular application.

6. Conclusion

We have described NV-heaps, a system for creating persistent data structures on top of fast, byte addressable, non-volatile memories. NV-heaps prevent several classes of well-known programming errors as well as several new types of errors that arise only in persistent data structures. As a result, NV-heaps allow programmers to implement very fast persistent structures that are robust in the face of system and application failures. The performance cost of the protections that NV-heaps provide is modest, especially compared to the overall gains that non-volatile memories can offer.

Acknowledgements

The authors would like to thank Geoff Voelker for his valuable feedback, Michael Swift and Haris Volos for their comments, and Amir Roth for his encouragement. We would also like to thank the reviewers for their feedback and suggestions that have improved this paper significantly.

References

[1] B. Aker. Libmemcached. http://libmemcached.org/.

[2] P. Allenbach. Java card 3: Classic functionality gets a connectivity boost, March 2009. http://java.sun.com/developer/technicalArticles/javacard/javacard3/.

[3] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 430–440, New York, NY, USA, 1987. ACM.

[4] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *SIGMOD Rec.*, 25(4):68–75, 1996.

[5] 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.

[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] R. Biswas and E. Ort. The java persistence api - a simpler programming model for entity persistence, May 2006. http://java.sun.com/developer/technicalArticles/J2EE/jpa/.

[8] 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.

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

[10] P. Butterworth, A. Otis, and J. Stein. The gemstone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[11] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The exodus extensible dbms project: an overview. pages 474–499, 1990.

[12] R. G. Cattell. *Object Data Management: Object-Oriented and Extended*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[13] A. M. Caulfield, J. Coburn, T. I. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[14] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, pages 385–395, New York, NY, USA, 2010. ACM.

[15] S. Cho and H. Lee. Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 347–357, New York, NY, USA, 2009. ACM.

[16] S. Chu. Memcachedb. http://memcachedb.org/.

[17] 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*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[18] A. Dearle, G. N. C. Kirby, and R. Morrison. Orthogonal persistence revisited. In *Proceedings of the Second international conference on Object databases*, ICOODB'09, pages 1–22, Berlin, Heidelberg, 2010. Springer-Verlag.

[19] G. Dhiman, R. Ayoub, and T. Rosing. Pdram: a hybrid pram and dram main memory system. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 664–469, New York, NY, USA, 2009. ACM.

[20] 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.

[21] 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.

[22] 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.

[23] D. M. Hansen, D. R. Adams, and D. K. Gracio. In the trenches with objectstore. *TAPOS*, 5(4):201–207, 1999.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] International technology roadmap for semiconductors: Emerging research devices, 2009.

[30] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalf: A high performance main memory storage manager. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 48–59, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[31] Jedec standard: Low power double data rate 2 (lpddr2), March 2009.

[32] 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.

[33] E. K. Kolodner and W. E. Weihl. Atomic incremental garbage collection and recovery for a large stable heap. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 177–186, New York, NY, USA, 1993. ACM.

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

[35] 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.

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

[37] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329, New York, NY, USA, 1996. ACM.

[38] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 92–101, New York, NY, USA, 1997. ACM.

[39] 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.

[40] 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.

[41] A. Marquez, J. N. Zigman, and S. M. Blackburn. Fast portable orthogonally persistent java. *Softw. Pract. Exper.*, 30(4):449–479, 2000.

[42] Memcached. http://memcached.org/.

[43] J. Metz. E-mail and appointment falsification analysis analysis of e-mail and appointment falsification on microsoft outlook/exchange. http://sourceforge.net/projects/libpff/files/.

[44] http://office.microsoft.com/en-us/outlook/ha010563001033.aspx.

[45] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[46] Berkeley db. http://www.oracle.com/technology/products/berkeley-db/index.html.

[47] J. O'Toole, S. Nettles, and D. Gifford. Concurrent compacting garbage collection of a persistent heap. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 161–174, New York, NY, USA, 1993. ACM.

[48] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[49] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[50] 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.

[51] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 146–160, New York, NY, USA, 1993. ACM.

[52] 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.

[53] 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.

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

[55] J. S. Shapiro and J. Adams. Design evolution of the eros single-level store. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 59–72, Berkeley, CA, USA, 2002. USENIX Association.

[56] 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.

[57] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An efficient, portable persistent store. In *POS*, pages 11–33, 1992.

[58] F. G. Soltis. *Inside the as/400*. Twenty Ninth Street Press, 1996.

[59] 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.

[60] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 2011. ACM.

[61] S. J. White and D. J. DeWitt. Quickstore: a high performance mapped object store. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 395–406, New York, NY, USA, 1994. ACM.

[62] 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.