Mnemosyne: Lightweight Persistent Memory

A memory is what is left when something happens and does not completely unhappen.
– Edward de Bono

Haris Volos[1], Andres Jaan Tack[2] *, Michael M. Swift[1]

[1] Computer Sciences Department, University of Wisconsin–Madison
[2] Skype Limited

[1] {hvolos, swift}@cs.wisc.edu, [2] andres.jaan.tack@skype.net

Abstract

New storage-class memory (SCM) technologies, such as phase-change memory, STT-RAM, and memristors, promise user-level access to non-volatile storage through regular memory instructions. These memory devices enable fast user-mode access to persistence, allowing regular in-memory data structures to survive system crashes.

In this paper, we present *Mnemosyne*, a simple interface for programming with persistent memory. Mnemosyne addresses two challenges: how to create and manage such memory, and how to ensure consistency in the presence of failures. Without additional mechanisms, a system failure may leave data structures in SCM in an invalid state, crashing the program the next time it starts.

In Mnemosyne, programmers declare global persistent data with the keyword "pstatic" or allocate it dynamically. Mnemosyne provides primitives for directly modifying persistent variables and supports consistent updates through a lightweight transaction mechanism. Compared to past work on disk-based persistent memory, Mnemosyne reduces latency to storage by writing data directly to memory at the granularity of an update rather than writing memory pages back to disk through the file system. In tests emulating the performance characteristics of forthcoming SCMs, we show that Mnemosyne can persist data as fast as 3 microseconds. Furthermore, it provides a 35 percent performance increase when applied in the OpenLDAP directory server. In microbenchmark studies we find that Mnemosyne can be up to 1400% faster than alternative persistence strategies, such as Berkeley DB or Boost serialization, that are designed for disks.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management; D.4.5 [*Operating Systems*]: Reliability; D.4.8 [*Operating Systems*]: Performance

General Terms Design, Languages, Performance, Reliability

Keywords Persistent memory, storage-class memory, persistence, memory transactions, performance

* Work done while a student at the University of Wisconsin – Madison

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

1. Introduction

Fast, cheap, and persistent memory has long been a dream for computer designers. Until recently, non-volatile storage was either slow (*e.g.*, disks) or expensive (*e.g.*, NVRAM). However, several new technologies promise cheap and fast storage that survives across system boot. For example, phase-change memory (PCM) provides near-DRAM speeds and is currently available in sizes up to 64 MB, and memristors may enable multiple terabytes of non-volatile memory to be placed on-chip [61]. These devices are termed *storage-class memory* (SCM) because they provide the interface of memory (load and store instructions) but the persistence of disks.

Existing operating systems are designed for a strict bifurcation of devices into *memory*, volatile, fast devices, and random-access devices erased on reboot, and *storage*, persistent, slow block-based devices. Thus, research on SCM tends to follow the same path. For example, recent work investigates use of phase-change memory within file systems [17], or as a low-power volatile DRAM replacement [39, 34]. Neither of these approaches exposes the full power of SCM to programmers.

We propose that operating systems should expose SCM as a *persistent memory* abstraction to provide direct access to the durability of SCM technologies. This abstraction enables programmers to make in-memory data structures persistent without first converting them to a serialized format. Thus, trees, lists, and hashes can survive program and system failures. Furthermore, direct access reduces latency to storage because it bypasses many software layers, including system calls, file systems, and device drivers.

We see persistent memory not as a replacement for files, but as a fast mechanism to store moderate amounts of data. For example, Firefox 3 had a problem with calling `fsync` too frequently [19], bringing the system to a crawl by frequent flushes to disk. Persistent memory could address this problem by providing low-latency storage of moderate amounts of data. Other possible uses are configuration changes, snapshots of in-progress edits, and logs, such as in distributed agreement protocols [12]. Applications should still use files for interchanging data and for compatibility between program versions when internal data-structure layouts change.

We have three goals for our system to expose persistent memory. First, it must be simple for a programmer to declare data as persistent, and persistence must fit naturally into existing programming models for volatile data structures. Second, and more importantly, the system must support *consistent modifications* of data structures. The system must enable programmers to move data structures between consistent states, automatically recovering to such a state after a failure. Finally, we seek a design that is compatible with existing commodity processors: this enables the adoption of SCM without the processor support required by other pro-

posals [17, 16], and avoids the chicken-and-egg problem of which comes first, SCM or processor support.

This paper presents *Mnemosyne*[1], a lightweight system for exposing persistent memory to user-mode programs. It provides a low-level programming interface, similar to C, for accessing persistent memory. Mnemosyne provides three key services that simplify programmer use of persistence. First, Mnemosyne provides *persistent memory regions*, segments of virtual memory stored in SCM rather than volatile memory. Regions can be created automatically to hold variables labeled with the keyword `pstatic` or allocated dynamically. Mnemosyne virtualizes persistent regions by swapping SCM pages to a backing file. Second, Mnemosyne provides *persistence primitives*, low-level operations that support consistently updating data. Finally, Mnemosyne provides a *durable memory transaction* mechanism that enables consistent in-place updates of arbitrary data structures.

Compared to past work on persistent memory and persistent object stores such as ObjectStore, Thor, Texas, LRVM, and QuickStore [36, 40, 56, 58, 64], Mnemosyne provides a low-level interface allowing both simple consistent updates (Section 3.2) as well as high-level transactions (Section 3.3). Upon this base, higher-level services such as garbage collection and safe references, to ensure that persistent data does not point to volatile data, can be provided by language frameworks. Furthermore, SCM enables the implementation to be much simpler as data can be made persistent without writing it out through the file system, and much lower latency by persisting data directly to memory at the granularity of an update rather than a page. The concurrently-developed NV-heaps project [16] also provides user-level transactional updates to persistent data. NV-heaps provide some features not currently found in Mnemosyne, such as type-safe pointers and garbage collection via reference counting. However, NV-heaps force the programmer to use a specific object framework and requires modifications to the processor.

We implement Mnemosyne as a pair of libraries and a small set of modifications to the Linux kernel for allocating and virtualizing SCM pages. We designed Mnemosyne to run on conventional processors, requiring no special support beyond the necessary memory controller for SCM, and implement it using regular x86 instructions with a performance emulator for SCM accesses.

In the evaluation, we show that Mnemosyne provides a simple abstraction for programmers to make data structures persistent. We compare Mnemosyne performance against Berkeley DB running on a RAM disk with the performance of phase-change memory. For small data sizes, Mnemosyne transactions perform 6–14 times better than Berkeley DB. We also convert two applications, OpenLDAP and Tokyo Cabinet, to use persistent memory and find the performance of moving existing in-memory data structures to persistent memory is 35–117 percent faster than Berkeley DB's optimized storage or flushing the whole structure to a file.

The contributions of our work are:

- A complete layered architecture for managing and exposing storage-class memory to programs.
- An exploration of mechanisms for consistent updates to in-memory data, and a library of primitives supporting consistent updates to data storage-class memory.
- A software transactional memory system that provides durability with SCM.
- A novel mechanism for atomic log writes without multiple barriers.

The rest of this paper begins with a short overview of storage-class memory technologies, followed by a high-level description of our design in Section 3. Section 4 describes the implementation of Mnemosyne, and Section 5 describes how we implement transactions in persistent memory. Section 6 demonstrates the performance and programming benefits of Mnemosyne, and we finish with related work and conclusions.

[1] Mnemosyne is the personification of memory in Greek mythology, and is pronounced *nee–moss–see–nee*.

	Technology	Read	Write	Endurance
Present	DRAM	60 ns	60 ns	$> 10^{16}$
	NAND Flash	25 μs	200–500 μs	10^4 - 10^5
	PCM	115ns	120 μs	10^6
Future	PCM	50 – 85 ns	150 – 1000 ns	10^8–10^{12}
	STT-RAM	6 ns	13 ns	10^{15}

Table 1. Access latency and endurance (in number of overwrites) of current and future DRAM, PCM, STT-RAM, and Flash memories. Prospective characteristics are based on demonstrated prototypes. Memristor figures are not available.

2. Storage-Class Memory

We use the term *storage-class memory* to refer to technologies that allow persistent storage to be attached to the memory bus and accessed through load and store instructions [25]. We do not consider flash as storage-class memory because it is only accessible as a block device through a driver.

Three recent technologies are promising implementations of SCM: phase-change memory (PCM) [39], spin-torque-transfer RAM (STT-RAM) [32], and memristors [61]. While the performance and reliability details differ, they all provide byte-granularity access and the ability to store data persistently across reboots without battery backing.

We assume, like others [17], that SCM is placed directly on the memory bus side-by-side with DRAM, allowing access through normal load/store instructions. We believe this is a reasonable design choice given that computer architects have already studied phase-change memory, a form of SCM, as a DRAM replacement in general-purpose systems [39, 53, 67].

While SCM is currently only available in small sizes, scaling projections indicate that within a few years larger sizes (*e.g.*, 1 GB) will become available. Eventually, it may be economically feasible, and perhaps cheaper, to outfit a system completely with SCM and no DRAM [39, 53, 67].

Phase-Change Memory. We prototype Mnemosyne with the characteristics of PCM because it is closest to commercial deployment: Samsung recently announced that it is shipping 512 Mbit PCM chips for use in mobile devices [55]. Furthermore, its scaling properties and low power make it viable as a DRAM replacement. Table 1 shows the performance and reliability properties of PCM and flash for comparison. Compared to DRAM, currently available PCM [49] is approximately 2 times slower for read accesses and 2000 times slower for write accesses. However, access latencies for PCM, as demonstrated by research prototypes [5, 63], are expected to match those of DRAM for reads and be 2-17 times slower for writes. Furthermore, current technologies suffer from wear-out after 10^8 writes, although many solutions for this problem have been proposed [34, 39]. These solutions change the mapping of physical address to locations in SCM to level writes across more locations. We assume such wear leveling is present and do not address it in Mnemosyne.

While memory systems for SCM are not yet available, we make several assumptions about the features they provide. First, we assume they can support an atomic write of at least 64 bits [17]. Second, we assume that it is possible to stall execution until a write

has made it all the way to PCM, similar to the `fsync` call for file systems.

Failure Models. While data stored in SCM is persistent, data stored in a processor cache is not. Thus, after a failure, only data actually resident in SCM survives. A system using SCM could reduce this restriction with low-level software that flushes data from the processor cache on application or OS failure. However, without battery backing, sudden loss of power or hardware faults could still cause data loss. We therefore assume that on a system failure, in-flight memory operations may fail, and that atomic updates either complete or do not modify memory.

3. Design

The purpose of Mnemosyne is to reduce the cost of making data persistent. Current program devote large bodies of code to formatting data for persistence, either in a file system or database, managing consistency of persistent data. They also carefully decide when to make data persistent, as writing data out to disk frequently leads to poor performance. In contrast, with Mnemosyne, a program should be able to make any data persistent, at any time, with little extra effort.

Mnemosyne presents an abstraction of persistent memory to programmers. We have three goals for the system:

1. *User-mode access* to persistence avoids the latency of entering the kernel and provides flexibility in how persistence is achieved.
2. *Consistent updates* modify persistent data without jeopardizing correctness in the presence of failures.
3. *Conventional hardware* lowers the barrier to adoption by allowing existing processors to work with new memory technologies.

Mnemosyne is designed as a low-level interface to persistent memory, providing necessary methods for consistency. It leaves the separation of volatile and persistent data and prevention of memory leaks to higher levels of software.

Mnemosyne achieves the first goal with *persistent regions*, a segment of memory that is created and virtualized by the kernel but may be accessed directly from user mode. Mnemosyne provides a two-layer solution to the second goal. The lower layer of Mnemosyne provides *persistence primitives*, which are support routines for programming with persistent memory. The upper layer of Mnemosyne provides *durable memory transactions* supported by a compiler to enable general-purpose code to create and modify persistent data structures. To satisfy the third goal, our consistent update mechanism relies on hardware primitives available in existing architectures. Figure 1 shows how the components are divided between user and kernel modes. While we designed Mnemosyne for a system with both DRAM and SCM, it applies equally well to a system that uses SCM for volatile storage as well.

3.1 Persistent Regions

Mnemosyne exposes storage-class memory directly to application programmers through the *persistent region* abstraction: a segment of data that user-mode code can read or write, and that survives application or system crashes. Persistent regions are mapped at fixed virtual addresses to support pointer-based data structures. This abstraction makes persistence explicit: only a portion of a process address space persists across restarts. In addition, programs must take explicit steps to guarantee persistence, such as writing data with special instructions or within a transaction to ensure data makes it all the way to SCM. However, data in a persistent region can be read and cached with regular instructions.

Mnemosyne supports both static and dynamic creation of persistent regions. A programmer can declare a C/C++ variable `pstatic`, which tells the linker to place it in a persistent region.

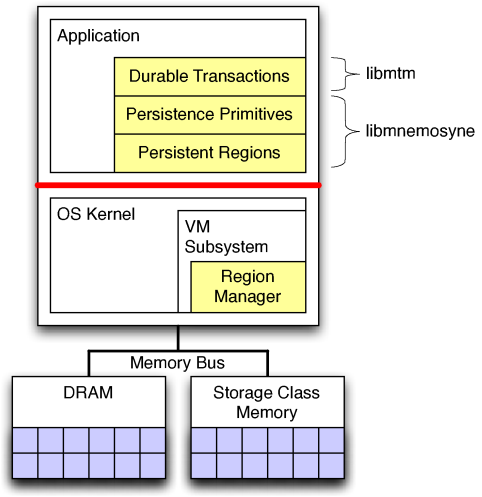

Figure 1. Mnemosyne architecture. User-mode components layer on top of a kernel region manager.

Similar to static variables, persistent static variables are initialized once when the program first runs, and then retain their value across invocations. A programmer can also explicitly create a persistent region with the `pmap` function, which provides functionality similar to `mmap`. While dynamic persistent regions offer programmers a generic way to store data of any size and structure, Mnemosyne also provides a persistent heap (Section 3.2.2), which enables dynamically allocated persistent variables.

Mnemosyne virtualizes regions by (i) recording the virtual–physical mapping of persistent regions in SCM, and (ii) swapping SCM pages to *backing files* that it allocates when creating a region. Thus, multiple applications can time-share or space-share access to SCM. As we discuss in Section 3.4, virtualization prevents a memory leak in one program from monopolizing a finite amount of SCM.

Static persistent regions are most appropriate for programs that allow only a single instance of the program to run per user, such as most office productivity applications and some web browsers. For programs that allow multiple simultaneous instances, Mnemosyne provides an environment variable to indicate which backing file contains the data for this instance.

Mnemosyne lets programmers annotate the target of a pointer type as persistent with the `persistent` keyword, which functions similar to the `const` keyword. The annotation has a shallow effect on the target: annotating a target as persistent does not indicate that its members persistent. The annotation also has no control over whether the data is allocated in volatile or persistent memory. It only serves as an indication to the compiler of the persistence type of the target, which can be used to identify potentially dangerous assignments of volatile address to a `persistent` pointer, and vice versa. This ensures that persistent data, which survives restarts, does not refer to volatile data that is lost. However, this cannot detect when the only pointer to persistent data is *stored in* volatile memory; in this case, the persistent data could be leaked if the program crashes.

3.2 Consistent Updates

Persistence requires a mechanism for application programmers to update persistent data in without risking corruption after a failure. File systems use a variety of techniques to ensure consistency, such as shadow updates [8, 54, 17], journaling [62], soft updates [42],

and post-reboot checking [43]. We seek to give programmers similar flexibility in implementing consistency.

A *consistent update* is a transformation that takes data from one consistent state to another consistent state; an update may contain any number of store and load instructions. For example, in a hash table, a consistent update may be the addition of a new value.

3.2.1 Ensuring consistency

The primary mechanism for ensuring consistency is *ordering writes*, for example to ensure that new data exists before changing a pointer to reference the new data. We identify four common mechanisms for consistently updating data in Table 2 in increasing order of flexibility. The more specific mechanisms can provide higher performance for certain data structures, while the more general mechanisms support a wider range of usage patterns. Mnemosyne supports all four methods.

Single variable update. The simplest update is atomically writing to a single variable. This is useful for recording when a program has been initialized or for storing statistics such as counters. These updates are totally ordered with respect to each other.

Append updates. An append update is used by logs and writes new data to empty space after the previous update, thus never modifying existing data. The individual stores comprising an append update are unordered, but separate appends must complete in order. After a failure, an incomplete append (there can be only one) is discarded.

Shadow updates. Similar to append updates, a shadow update writes all data to a new location. Once new data is persistent, the program atomically modifies the reference to the old data to refer to the new data. While there are no ordering constraints between the stores writing the new data, the reference can only be modified after the new data has completed writing. Shadow updates work best for tree-like structures where data is reachable through a single pointer, and must allocate new memory for every update. After a failure, a program must find and release unreferenced new data.

In-place updates. An in-place update can modify any data structure, such as a doubly linked list or a B-tree. It ensures consistency with transactions that undo or complete changes after a failure. As a result, the program must make a copy of either the old data for rolling back, or the new data for rolling forward. Stores updating a data structure must be ordered after stores that create the copy. Unlike the preceding consistency mechanisms, in-place updates can be used to modify any data structure and therefore enable existing volatile structures to be persistent. However, they perform worse than other consistency methods because they copy data for recovery.

3.2.2 Persistence Primitives

Mnemosyne provides a set of *persistence primitives*, which are low-level operations that enable programmers to implement these consistency mechanisms. At the lowest level, Mnemosyne provides hardware primitives for writing data persistently and ordering writes (described next), which is useful for single variable updates. The system also provides a *persistent heap* for allocating small blocks of memory. Allocated memory and the size of allocations persist across program invocations, so memory can be allocated during one invocation and freed during the next. The persistent heap supports shadow updates by providing space for new data. Mnemosyne also provides a *log facility* to support append-only updates. The log provides a simple double-ended buffer, allowing consistent appends to the tail of the log and truncation at the head of the log.

3.2.3 Mapping consistency onto hardware

A key challenge in implementing consistent updates is ensuring that data reaches SCM in the right order. Unlike disks, which are only accessed through software, SCM is attached to the memory bus and is subject to caching by the processor. While the cache can greatly improve read and write performance, it can also reorder updates to SCM: the cache may evict cached data at any time and in any order. Thus, if a program accesses SCM like normal memory, updates may not be consistent. Prior work on SCM addresses this problem with hardware support for programs to express ordering constraints [17]. However, the proposed hardware requires extensive modifications to processor caches, including an additional *epoch* tag per cache line and the ability to globally flush all caches lines in an epoch.

Mnemosyne's consistent updates mechanism exposes explicit commands to write data persistently and consistently. It relies on three hardware primitives found in most processors [33]: (i) *write-through stores*, which write data directly to memory rather than to the cache, (ii) *fences*, which prevent subsequent writes from completing before preceding writes, and (iii) *flushes*, which writes a cache line out to memory. Durable updates to persistent memory are implemented as write-through stores, ensuring that they will not be delayed by caching. If a program wishes to separate durability from writing data, so it may re-read data before it becomes durable, it can use a regular store and then flush the data later. Mnemosyne guarantees ordering constraints by issuing a fence between ordered updates, ensuring that a later update will not become persistent before an earlier update. In addition, a program wishing to stall until data is persistent can also use a fence.

3.3 Durable Memory Transactions

Mnemosyne provides *durable memory transactions* to support in-place updates. Leveraging recent advances in transactional memory, Mnemosyne provides a compiler to convert regular C/C++ code into transactions. A programmer places the `atomic` keyword before a block of code that updates a persistent data structure, and the compiler produces code that passes all memory references to a transaction system.

The transaction system ensures that all modifications are *atomic* and *durable*, so updates from committed transactions will survive restarts and uncommitted transactions roll back, and *isolated*, so no transaction can see intermediate states of other transactions. Unlike the checkpoint mechanism used by some previous persistent memory systems [17, 58], transactions thus allow multiple threads to concurrently update different data structures.

3.4 Persistent Memory Leaks

Memory leaks of volatile data may cause a program to crash, but can often be repaired by restarting the program. In contrast, leaks of persistent memory may use all the SCM in a system and be fatal to a program. Mnemosyne provides two mechanisms to help prevent leaks. During allocation, Mnemosyne requires that programs provide a persistent pointer to receive the memory, thereby ensuring that memory is not lost if a crash happens. Second, Mnemosyne virtualizes persistent memory by swapping it to files. This ensures that a leak in one program only affects that program and does not reduce the availability of persistent memory to other programs. When leaks do occur, a program can recover by allocating a new persistent region and then copying live data from the existing regions into the new region.

In addition, there are a variety of language-level techniques for preventing leaks, including conservative garbage collection [7], and smart pointers that perform reference counting as demonstrated by other persistent stores [40, 58, 64, 16].

Method	Ordering constraints within update		Data structures
	Count	Description	
Single variable update	0	None	flag, pointer
Append update	0	None	log, extent
Shadow update	1	Store modifying reference ordered after stores writing data	tree, bitmap
In-place update	N-1	Stores modifying original data ordered after stores making copy	any

Table 2. Methods for consistently updating persistent memory and the number of ordering requirements within an update.

4. Implementation

This section describes the implementation of Mnemosyne for Linux. The system consists of (i) kernel modifications to expose and virtualize storage-class memory, (ii) libraries to implement persistent regions, persistence primitives and the transaction system, and (iii) a compiler/linker that supports persistent variables and transactions. We implement our system for x86-64 Linux version 2.6.33.

The total implementation is comprised of about 450 new lines of kernel code and 10,050 lines of user-mode library code, not including comments. The user-mode code is further broken down into 3,500 lines for implementing persistent regions and persistence primitives, 3,100 lines for implementing the persistent heap, and 3,450 lines for implementing the transaction system.

4.1 Hardware Platform

Mnemosyne provides four memory primitives for consistently updating persistent data. As described in Section 3.2, we rely on two fundamental operations, *write through* and *fence*, to order updates and to guarantee that updates have reached memory. In addition, Mnemosyne provides a *store* operation, to update persistent memory in the cache, and *flush* to force cached data out to SCM. These allow data written to the cache immediately so it can be returned by later loads, and asynchronously flushed to SCM.

Given the long latency of PCM operations, a goal for the primitives is high throughput by batching multiple operations. We rely on the x86 *write-combining buffers*, which are provided for high-speed data streaming [2]. Write combining allows data to be stored temporarily in a buffer and merged with other data writes to the cache block before the processor writes them to memory. These buffers allow the processor to write multiple words in a single bus transaction to effectively utilize memory bandwidth. The block size is usually that of the cache line, and is 64 bytes on our platform. Without using these buffers, writes to the same cache line would require separate bus transactions and sacrifice performance.

The write-through operation issues streaming writes to the write-combining buffers with the `movntq` instruction. Fences use the `mfence` instruction, which delays execution until write-combining buffers have been flushed. Thus, any memory access after the fence waits until the data has been written stably to SCM. For regular writes, `store()` just invokes `mov`, and `flush()` issues a `clflush` (flush cache line). These operations are all implemented as macros for performance.

4.2 Persistent Regions

The Mnemosyne persistent region abstraction is provided by a combination of kernel support, library support, and compiler support for declaring persistent variables. As shown in Table 3, Mnemosyne provides both static and dynamic persistent regions. Static regions contain persistent global variables that are declared and initialized by a programmer at compilation time. A programmer declares a global variable as persistent by annotating it using the `pstatic` keyword. This keyword inserts a compile-time annotation with the `__attribute__ ((section("persistent")))`. The static linker coalesces all persistent variables into a single `.persistent` ELF section in the executable.

Programmers create dynamic persistent regions at runtime by calling the `pmap` function, similar to memory mapping files with `mmap`. Mnemosyne automatically maps dynamic regions created by the program on previous invocations into the address space when it initializes. To prevent newly created sections from being lost if the application crashes, the `pmap` function takes as an in/out parameter a persistent variable to receive the region's address. A programmer deletes a persistent region by calling the `punmap` function, which takes the starting address and the length of the region.

Mnemosyne lets programmers annotate pointer targets with the `persistent` keyword. This keyword inserts a compile-time annotation with the `__attribute__ ((address_space(1)))`. This annotation is interpreted by the Sparse semantic parser [1], which designates such annotated pointer targets in address space *1* and all other pointer targets in address space *0*. Sparse essentially treats pointers with identical target types but different address spaces as distinct types, and warns about code that mixes pointers to different address spaces. This allows Mnemosyne to identify code that accidentally assigns a pointer to persistent data to volatile data instead (and vice versa).

Persistent regions present two challenges: how to ensure that mappings are persistent, and how to virtualize a finite amount of SCM to support use by many applications. For regular volatile memory, pages are swapped to a shared page file or swap partition and mappings are deleted when a program terminates. However, for persistent regions the mappings of virtual addresses to physical SCM pages must survive system restarts. In addition, data swapped out of SCM must also survive system restarts.

Our implementation follows a two-layered approach. A kernel-mode layer, the *region manager*, exposes SCM to user-mode code as a memory-mapped file. A user-mode layer, *libmnemosyne*, associates each persistent region with a specific file and records the address of each region.

Region manager. The region manager is an extension of the existing Linux virtual memory system. The region manager creates a new zone of memory for SCM, and all allocations for persistent regions come from this zone using the existing Linux page allocator. We add a new flag, `MAP_PERSIST`, to the `mmap` system call implementation to indicate that a file should be mapped to SCM and not DRAM. The region manager records the list of virtual pages currently stored in SCM in the *persistent mapping table*. This table, stored at the base of physical SCM, stores triples of the form $\langle scm_{fn}, i, p_{off} \rangle$, which associate an SCM frame's number scm_{fn} with a page offset p_{off} in the file identified by inode i[2].

The region manager reconstructs persistent regions when the OS boots. It scans the persistent mapping table and (i) updates the Linux page descriptor for each mapped SCM page, and (ii) creates a VFS inode for the backing file of every mapping. The manager places SCM frames not in the table on a free list. After boot, the page descriptors and inodes enable the kernel to evict SCM pages to their proper files. When starting a process with persistent regions, all accesses to SCM pages already in memory are treated as soft page faults that update the page table without copying data from

[2] Uniquely identifying a file requires two extra pieces of information: device and inode generation numbers, which we omit for clarity of discussion.

Class	API	Description	Section
H/W primitives	flush(addr)	Writes back and invalidates the cache line that contains the linear address *addr*.	4.1
	store(addr, val)	Writes value *val*.	4.1
	wtstore(addr, val)	Writes value *val* to SCM.	4.1
	fence()	Prevents subsequent writes from completing before preceding writes.	4.1
Persistent regions	`pstatic` var	Allocates the variable *var* in the static region.	4.2
	pmap(addr, len, prot, flags)	Creates a dynamic region.	4.2
	punmap(addr, len)	Deletes part or all of a dynamic region.	4.2
	type `persistent` * ptr	Declares the target of the pointer *ptr* as persistent.	4.2
Persistent heap	pmalloc(sz, ptr)	Sets *ptr* to point to a newly allocated persistent memory chunk of size *sz*.	4.3
	pfree(ptr)	Deallocates the persistent memory chunk pointed by *ptr* and then nullifies *ptr*.	4.3
Log	log_create(flags, cbf)	Creates a log.	4.4
	log_append(rec)	Writes record *rec* by appending it at the end of the log.	4.4
	log_flush()	Blocks until all prior writes to the log reach SCM.	4.4
	log_truncate()	Drops any records written to the log.	4.4
Durable transactions	atomic {...}	Atomically updates persistent state.	5

Table 3. Summary of Mnemosyne's programming interface, including low-level operations implemented with processor instructions and high-level APIs for managing persistent regions.

the backing file. During process execution, the region manager may swap persistent memory pages back to the file system if there is memory pressure.

libmnemosyne. The libmnemosyne library creates and records the persistent regions for a process. Mnemosyne allocates all regions in a one terabyte *reserved range* of virtual address space (easily changed to any power-of-two sized region). This allows a quick determination of whether an address refers to persistent data and prevents persistent regions from conflicting with dynamically allocated address space. The library reserves 16KB in the static persistent region to store a *region table* containing the process's persistent regions. If the region table exceeds 16KB, it can overflow to a dynamically allocated page.

To create a dynamic region, the `pmap` function in libmnemosyne creates an empty backing file and invokes `mmap` to map the file into SCM. In the region table, libmnemosyne records the tuple $\langle r_{addr,len}, b, m \rangle$ that associates the region's starting address *addr* and length *len*, with the backing file *b*, and metadata *m*, which includes protection flags. The region table also serves as an intention log: libmnemosyne writes a flag indicating the successful completion of a `pmap` operation. When an application starts, libmnemosyne recreates previously allocated persistent regions and destroys partially created ones.

For the static regions containing variables labeled `pstatic`, libmnemosyne creates a new backing file the first time the program executes. It populates the new region with the initial values found in the executable. If a program wishes to discard existing contents of a static persistent region and revert to the data in the executable, a program can delete the backing file and restart. Static persistent variables serve as pointers into dynamically allocated persistent regions.

All the region backing files, including the region table's file, are stored by default in the program's current working directory. This location can be changed via the environment configuration variable `MNEMOSYNE_REGION_PATH`, thus allowing multiple concurrent instances of the same program to use separate backing files.

4.3 Memory allocation

Mnemosyne provides a persistent heap [3] in libmnemosyne. Table 3 lists the programming interface to this heap. Similar to `pmap`, the `pmalloc` call takes a persistent pointer as an argument to ensure that memory is not leaked if the system fails just after an allocation. The `pfree` call takes a pointer to a persistent pointer as an argument to ensure that the persistent pointer does not continue to point to the deallocated chunk of memory if the system fails just after a deallocation.

We base our implementation on two popular volatile memory allocators, Hoard [6] and dlmalloc [38], which we modify to allocate from a persistent region. While Hoard was originally designed for use in multiprocessor environments, we leverage its superblock-based structure to minimize the persistent state required to track allocations. Hoard splits the heap into superblocks, which are fixed-size regions containing an array of fixed-size blocks (different superblocks may have different block sizes). We modify Hoard to store a *persistent bitmap vector* per superblock to track allocated blocks; allocating memory requires only one write to SCM to set a bit in the superblock's vector. We separate bitmap vectors from allocated data to reduce the risk of corruption [41]. Hoard's indexes, which speed allocation, are in volatile memory and must be regenerated when a program starts. The allocator guarantees atomicity of its operations by logging the write to the bitmap vector and the destination/source pointer.

Mnemosyne uses the modified Hoard allocator for requests smaller than a superblock (8 KB). If the requested block is larger, Mnemosyne falls back to dlmalloc, which we chose for its scalability to large block sizes. Since we expect dlmalloc to be infrequently used, we have not modified it except to add logging to ensure allocations are atomic.

4.4 Logging

Mnemosyne relies on a log to make memory allocations and transactions atomic. This log is exposed to programmers by libmnemosyne for implementing append-only data structures. Table 3 lists the log programming interface. A program writes to the log with `log_append`, which writes data but does not guarantee persistence. The `log_flush` call ensures all prior log writes are persistent. Finally, logs are truncated with `log_truncate`. A program can synchronously truncate the log by interspersing truncates with appends in a single thread. Alternatively, the application can asynchronously truncate the log from another thread, which moves the latency of truncating off the critical path. In addition, a program may truncate logs at startup. Mnemosyne's durable transaction mechanism supports all three uses.

We implemented a high-performance *raw word log* (RAWL) and associated manager that logs uninterpreted word-size values. Such a semantic-free log can be the basis for other semantic-rich logs, such as journals. The log is implemented as a fixed size single-consumer/single-producer Lamport circular buffer, which allows simultaneous appends and truncations without locking [37]. The log manager implements functions to access the log and recover log contents after a failure.

The main challenge in implementing a log is maximizing performance while ensuring that appends are atomic. As logs are

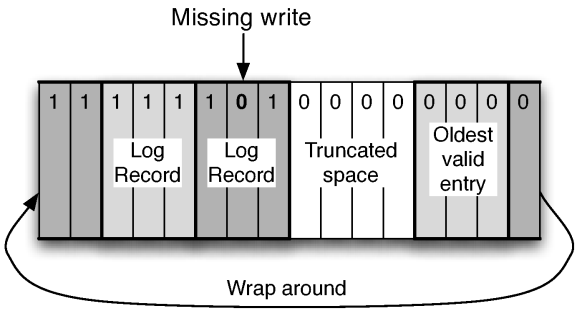

Figure 2. Example of missing write detection in the RAWL. Entries written on the current pass through the log have a torn bit of '1', while entries written the previous pass have a '0'. A missing write is detected by an incorrect torn bit.

always written sequentially, Mnemosyne uses the x86 streaming write instruction `movntq` to write log data. To achieve high performance, these instructions do not guarantee that writes are executed in program order. If the system crashes, later writes may have completed while earlier ones did not. When restarting, the log manager must identify whether all the writes that comprise an append operation completed. This problem is similar to the torn-write problem in databases and file systems, where only part of a multi-sector write completes. The common solution in file systems is to use two fences: write the data, wait for the data writes to complete with a fence, then write a commit record, and wait for the commit record to complete with a fence. A second option is to append a checksum of the data [51]. However, both these techniques have high overhead. Commit records require two long-latency fences, and checksumming adds substantial overhead to every write.

Mnemosyne instead implements a novel *tornbit RAWL* that requires only one fence. The tornbit RAWL reserves a single *torn bit* in every 64-bit word. This bit has the same value for all writes in one pass over the log buffer, and reverses sense when the log is overwritten. Thus, completely written log entries will have the same torn bit, while incomplete entries will mix values. Figure 2 demonstrates the use of torn bits. When Mnemosyne creates a log, it initializes the memory to zeroes. The log manager treats the incoming 64-bit words to be written to the log as a stream of bits. It forms and writes out to the log 64-bit words that are composed of 63 bits taken from the head of the stream and the proper torn bit. On a `log_flush` the log manager writes out to the log any bits left in the stream. Upon recovery, Mnemosyne locates the head of the log and scans forward until it reaches the end of the log, where the torn bit reverses, or until it finds a log word with an out-of-sequence torn-bit, indicating a partial write.

4.5 Limitations

Shared Memory. The recovery mechanisms used for logging and memory allocation execute when a process starts. Furthermore, the allocator caches index information in volatile memory for performance. Therefore, these mechanisms may be unsafe if a persistent region is shared between processes. However, sharing is safe if the processes cooperate to ensure that (i) within each region, only one process writes to a log or allocates from a heap, and (ii) both processes have started and completed recovery before accessing shared data. Thus, producer-consumer style communication, where a single process is responsible for creating and later deleting work items, can be implemented safely.

Wearout. Our prototype implementation does not address wearout of storage-class memory. However, virtualization enables remapping heavily used virtual pages to spread writes to different physical PCM frames. Additionally, our system's structures, such as mapping tables and RAWL, may be relocated to spread writes around. For example, RAWL's tornbits may periodically be shifted to avoid writing 0's and 1's continuously to the same bits. Finally, our system could benefit from work on wear-leveling mechanisms that reduce wearout of PCM when that technology is used as DRAM replacement in main-memory systems [34, 52].

```
update_hash(key, value) {
  atomic {
    pmalloc(&bucket, sizeof(*bucket));
    bucket->key = key;
    bucket->value = value;
    insert(bucket);
  }
}
```

Figure 3. Example of a durable memory transaction.

5. Durable Memory Transactions

Mnemosyne supports in-place updates with *durable memory transactions*. Updates within a transaction execute to completion, at which point the changes persist across failures, or the changes are rolled back upon restart. Figure 3 shows an example use of the `atomic` block language construct to form a transaction (Table 3).

Memory transactions are implemented by three components: a compiler to create transactions from ordinary C or C++ code, a transaction system to store data necessary for recovery, and a transaction log. The *libmtm* library provides the transaction system on top of the persistence primitives in libmnemosyne.

Compiler. We use Intel's STM Compiler [48] to automatically generate object code with calls into a transaction system. The compiler emits compiled code that invokes the transaction system when a transaction begins, commits, and on every memory reference within the transaction.

Transaction system. Mnemosyne transactions are based on TinySTM [24] a lightweight software transactional memory (STM) implementation. The Mnemosyne transaction system implements *lazy version management* [46] with write-ahead redo logging, and *eager conflict detection* [46] with encounter-time locking.

With write-ahead redo logging, new values written during the transaction and their addresses are added to a log and also buffered in volatile memory. The transaction system performs a quick range check against the reserved persistent address range and logs only writes to persistent memory. At commit, the log is flushed to SCM, and the new values can optionally be written back. During a transaction, memory at a variable's address still contains unmodified values. When called to read data, the transaction system checks whether the data was modified. If so, it returns the new data from the buffer and if not returns the value from memory.

We implement write-ahead redo logging because it reduces the ordering constraints on writes to SCM: the only requirement is that the log is written completely before any data values are updated. In contrast, undo logging, where old values are written to a log and new values are written to memory, would require ordering a log write before every memory update. However, buffering writes has a performance cost: transaction commits are slower because the new values have to be written out to memory, and reads within transactions have to check if new values exist.

For encounter-time locking, we use a global array of volatile locks, with each lock covering a portion of the address space. When accessing a memory location, the transaction first identifies the lock that covers the memory address, and if it does not own

the lock it tries to acquire it. If successful, the transaction brings the lock in its lock-set and continues with the access. Otherwise, the transaction aborts by releasing all locks acquired, discarding any buffered updates, and writing to the log a mark to indicate the transaction as aborted. When a program starts, Mnemosyne replays all completed transactions by writing the data at the logged address.

Transaction log. We implement the redo log in persistent memory using a RAWL. The transaction log can be truncated after data values are forced to SCM. We implement both *synchronous* and *asynchronous* truncation. Synchronous truncation forces new values to memory during transaction commit. After flushing the log, the transaction system walks the buffer of modified data and calls `flush` on every address written. It then truncates the entire log. Under heavy load, this prevents the transaction log from growing too large. Asynchronous truncation retains the log after transaction commit, so the latency of committing is shorter. A separate log manager thread consumes the log and forces values out to memory before truncating the log. However, if the log manager thread is unable to execute, program threads may stall until there is free log space.

For better multiprocessor scalability, Mnemosyne keeps a per-thread log. This slightly complicates recovery as Mnemosyne must ensure that transactions are redone in the order executed by the program. Mnemosyne relies on TinySTM's existing global timestamp counter, which is incremented at every transaction completion. Mnemosyne captures a total order over transactions by storing this global counter along with each transaction in the log. During recovery, transactions from different threads are replayed in counter order.

Discussion. The cost of durable transactions is two writes to SCM for every update: once to store recovery information in the log, and once to write the data itself. Other consistent-update mechanisms may perform better. But, they come at the cost of increased complexity, such as recovery code to replay logs for append-only updates, garbage collection for memory lost during shadow updates, and explicit fences to order updates.

6. Evaluation

The goal of Mnemosyne is to abstract storage-class memory at low latency for program use. We evaluate three questions about the system:

1. *Does it work?* Can existing programs benefit from persistent memory, and does Mnemosyne provide persistence across crashes?
2. *How fast is it?* How does Mnemosyne perform, both in latency and throughput, across a range of workloads, and how beneficial are Mnemosyne's mechanisms?
3. *With which technologies can it work?* How does Mnemosyne's performance depend on the latency of the underlying SCM technology?

While programmers may program directly with persistence primitives, they require a sophisticated understanding of recovery protocols similar to databases or file systems [8, 45]. Durable transactions provide a much simpler interface to persistent memory and require few application changes, so we concentrate our evaluation on transaction performance.

6.1 Methodology

Because real memory systems based on PCM are not available, we developed a simple performance emulator based on DRAM to evaluate performance of our system. There are a wide variety of projections for PCM's performance, and the specific design of the memory system can have a great impact on performance [39]. So, we limit our emulation to the most important aspect of performance: slow writes.

To account for PCM's slower writes relative to DRAM, we introduce a delay after each write into the hardware access macros. The emulator adds delays on operations that already go to DRAM (not the cache), so the delay is only for the *additional latency* of PCM. For cacheable writes (`store` operations) we insert the delay on the subsequent `flush`, while for non-cacheable ones we insert the delay in-place. We also insert the delay after each memory fence to account waiting for outstanding writes to PCM. For sequential write-through (`wtstore`) we model write bandwidth by inserting a proper delay after the write sequence completes to limit the effective bandwidth.

In all cases we implement the delay with a loop that reads the processor's timestamp counter (TSC) in each iteration. The loop continues until the requested delay has elapsed. In calibration tests, we found that inserted delays are at least equal to the target delay, and that our bandwidth model is accurate to within 4%.

Our emulator does not account for additional latency on loads. However, the primary benefit of SCM is fast, durable updates, and our workloads focus on write rather than read performance. Furthermore, many loads will hit the cache, so only misses will incur a penalty from PCM. Also, our model does not account for the effect of cache evictions or read-after-write bank conflicts, where reads may be queued behind long writes to the same bank, thus increasing read latency.

To compare Mnemosyne against other uses of PCM, we constructed an emulator, *PCM-disk*, for a PCM-based block device. Base on Linux's RAM disk (`brd` device driver), PCM disk introduces delays when writing a block. We model block writes using sequential write-through operations as described above, and mount an ext2 file system.

We performed our experiments on an Intel Core 2 quad-core based machine equipped with 4GB of physical DRAM (accessible via the DDR2-800 interface) running our modified x86-64 Linux version 2.6.33 kernel at 2.5GHz. All tests add 150 ns of extra latency and are limited to 4GB/s of write bandwidth unless otherwise noted. We estimated write bandwidth based on projections provided by Numonyx [21]. For all our experiments we report averages of at least five runs.

6.2 Applications use of Persistent Memory

Applications that can immediately benefit from persistent memory are difficult to find: the long latency of writing to disk has taught programmers to avoid frequently committing data. We expect that a common use will be for applications to create a useful in-memory data structure, such as a tree or hash table, and then make it persistent by allocating it from a persistent heap and wrapping updates in transactions. For reliability and to support conversion between program versions, the program should periodically export serialized version of the structure.

We model this use of persistent memory by adapting programs that *already* maintain a fast in-memory data structure for frequent access and a separate version for consistent, durable updates. We converted two programs that take different approaches to persistence for use with Mnemosyne: OpenLDAP and Tokyo Cabinet. One program frequently commits data to disk using a storage manager, and the other periodically snapshots a whole data structure to disk.

OpenLDAP. OpenLDAP is an implementation of the Lightweight Directory Access Protocol (LDAP). OpenLDAP supports a number of storage backends; the default is *back-bdb*, which provides transactional storage using Berkeley DB. An alternative, *back-ldbm*, also uses Berkeley DB but without transactions; instead, it periodically asks Berkeley DB to flush dirty data to disk to mini-

Application	Backend	Workload	Updates/s
OpenLDAP	back-bdb on PCM-disk	SLAMD	5428
	back-ldbm on PCM-disk		6024
	back-mnemosyne		7350
Tokyo Cabinet	`msync` on PCM-disk	64B	19,382
		1024B	2,044
	Mnemosyne	64B	42,057
		1024B	30,361

Table 4. Update throughput for OpenLDAP and Tokyo Cabinet.

mize the window of vulnerability. To improve query performance, each backend maintains its own cache of data outside Berkeley DB [14]. We believe that such read-mostly caches can benefit from lightweight persistent memory: the backing store can be removed, leaving only a persistent cache.

To test this hypothesis, we modified the back-ldbm backend to remove Berkeley DB and to make the cache persistent with durable transactions. The cache is organized using an AVL tree, which we make persistent by allocating nodes with `pmalloc` and placing `atomic` blocks around updates in four places.

While we do not generally encourage keeping pointers to volatile memory in a persistent region, we found this useful in OpenLDAP. The cache entries of the original back-ldbm and our back-mnemosyne store pointers to a description of each attribute, which is kept by the front end in volatile memory. Instead of modifying the frontend to keep those descriptions in persistent memory, we found it more convenient to keep the descriptions in volatile memory and have the persistent cache entry keep pointers to those volatile descriptions. Because the volatile descriptions become stale after a restart, we augmented each cache entry with a version number that is used to determine whether the persistent pointer is up-to-date.

We use the SLAMD distributed load generation engine to exercise three versions of the server: (1) *back-bdb*, the default unmodified transactional backend with the cache and Berkeley DB, (2) *back-ldbm*, the unmodified back-ldbm backend with the cache and Berkeley DB, and (3) *back-mnemosyne*, our modified backend based on back-ldbm. We set the cache sizes large enough to avoid evictions due to capacity. We used a LDIF template to generate a workload of 100,000 directory entries.

TokyoCabinet. Tokyo Cabinet is a high-performance key-value store [31]. It stores data in a B+ tree and periodically calls `msync` on a memory-mapped file to flush modified pages to disk. These syncs can reduce performance by 96 percent [47], so they are rarely invoked. As a result, the application loses unsaved data after a crash.

We modified Tokyo Cabinet to allocate its B+ tree in a persistent region and perform updates in durable transactions. While we re-used the existing update code, in several cases we had to duplicate functions to create separate persistent and volatile versions with different allocators. We completely removed the persistence code that calls `msync`. We also removed the locks used for synchronizing concurrent accesses to the tree and relied on transactions for concurrency control.

Performance. Table 4 lists throughput of the OpenLDAP server for a workload that adds new records. OpenLDAP is configured to run with 16 threads (4 threads per core) as advised by its tuning manual. While the three backends perform similarly, back-ldbm offers a lower level of reliability than the other two backends. This close performance arises because PCM is fast enough that the time to write updates is a small fraction of the total time to service a request. These results demonstrate that with Mnemosyne it may not be necessary to create specialized structures optimized for persistence, such as Berkeley DB's tables. Instead, standard in-memory data structures (in this case an AVL tree) provide simpler programming, in addition to durability and consistency, all with similar or better performance than a highly tuned storage engine.

Table 4 also shows the throughput of 64-byte and 1024-byte insert/delete queries with Tokyo Cabinet for single-thread configurations. As a comparison, we ran the standard implementation of Tokyo Cabinet on our PCM-disk emulator and configured it to save data with `msync` after every update. Mnemosyne was about 2 – 15 times faster in these tests, while at the same time providing *stronger* consistency guarantees than the `msync` version, which can suffer from torn writes if the system fails while flushing pages. For multiple-thread runs, we found that the throughput of Tokyo Cabinet Mnemosyne degrades by 9% because of increased contention on the tree, causing transactions to abort. The throughput of Tokyo Cabinet on PCM-disk increases by 10%, but is still far below Mnemosyne.

Reliability. We validate that Mnemosyne works correctly by injecting synthetic failures. We intentionally crashed OpenLDAP in the middle of a transaction, and verified that after every restart, the data affected by the transaction were still available. In addition, we wrote a crash stress program, which uses transactions to perform random updates to memory using a known seed. We verified that after a crash, memory contains the correct random values. Finally, we tested the torn-bit feature of the RAWL by injecting bit flips into the log before a crash. In all cases, Mnemosyne correctly recovered after the crash.

6.3 Microbenchmark Performance

Programmers wishing to persist in-memory data structures today have a wide variety of choices. We evaluate two common approaches: (i) use a storage manager for data storage and caching, and (ii) serialize the data structure to a file. With a storage manager such as Berkeley DB [59], a program can commit small changes to a data structure, such as adding a record to a hash table. We compare the performance of a simple hash table [15]. using Mnemosyne transactions for persistence against using Berkeley DB's hash table. We store the database files on our PCM-disk emulator. In both cases, data is committed to storage on every update.

Figures 4 and 5 compare write latency and update throughput of Mnemosyne to Berkeley DB for different number of threads and record sizes. Deletes are introduced at the same rate as writes to ensure steady progress. Update throughput is aggregate throughput of writes and deletes. For single threaded runs and transactions that update records smaller than 2048B, Mnemosyne's direct access of memory achieves a write latency that is almost six times better than Berkeley DB. Mnemosyne's performance for small transactions is limited by the cost of (i) transaction instrumentation in the code, which adds a function call to every load and store, (ii) fences to force the log to memory and (iii) flushes to force data to memory. With a microbenchmark, we measured the cost of instrumenting and logging each word written as 190 ns when the transaction's write set size is smaller than 128 cache lines. For larger write sets, this cost increases linearly with the number of distinct cache lines written. The cost of committing a transaction, which consists of a fence and flushing data out to SCM, adds up to 250 ns per distinct cache line flushed. These costs represent the overhead of supporting in-place updates. A hash table insert of 64 bytes requires on average 15 updates to 5 distinct cache lines, for a total cost of 4.3 μs. With larger data sizes, Berkeley DB's optimizations for disk-like performance, such as large sequential writes and infrequent fences (once per block in PCM-disk), give it better write latency.

With multiple threads, Mnemosyne achieves 10-14x improvement in update throughput compared to Berkeley DB. Mnemosyne improves throughput almost linearly with the number of threads,

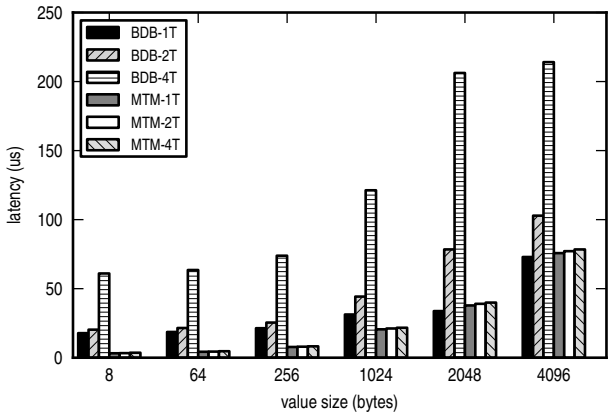

Figure 4. Write latency for a hashtable with durable transactions compared to Berkeley DB.

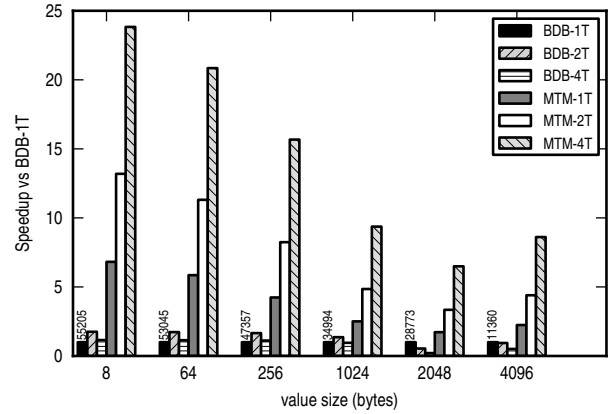

Figure 5. Update throughput for a hashtable with durable transactions compared to Berkeley DB. Numbers above the bars are the absolute updates per second for the Berkeley DB single-threaded version.

without hurting write latency. The slight increase in write latency is due to contention on the global timestamp counter in the transaction system, which is relatively more expensive for short transactions. In contrast, Berkeley DB does not scale beyond 2 threads. We found this is due to contention on the centralized log buffer, which becomes the serialization bottleneck as I/O latency becomes shorter. Also, Berkeley DB's throughput improvement with 2 threads comes at the cost of increasing write latency, possibly due to group commit, which is not necessary with Mnemosyne's fine-grain memory transactions. Finally, while Berkeley DB's write latency is lower than Mnemosyne's for for values larger than 2048 bytes, the Mnemosyne's throughput is higher because it is the aggregate of writes and deletes, and delete latency remains almost constant for Mnemosyne as value size increases.

An alternative approach, often used for less structured data, is to serialize the data into a buffer and write it to a file. For example, productivity applications including word processors use this approach for periodic *fast saves*. We compare the cost of maintaining a red-black tree with 128 byte nodes in persistent memory against the cost of keeping it in DRAM and periodically serializing it and storing it in a file. Using small data sizes maximizes the overhead of transactions, because there is not much data to write out between fences.

Tree Size	Insert Latency	Serialize Latency	Inserts per Serialization
1 K	4.7 μs	517 μs	189
8 K	5.1 μs	3,413 μs	1,345
64 K	5.5 μs	33,859 μs	9,202
256 K	5.8 μs	143,776 μs	24,788

Table 5. Performance of Mnemosyne updates and Boost serialization of red-black trees. The right-most column shows the number of Mnemosyne updates that can be performed in the time for a single Boost serialization of the tree.

Record Size (B)	8	64	256	1024	2048	4096
Base (MB/s)	17	128	416	881	1088	1244
Tornbit (MB/s)	34	227	591	929	1045	1093

Table 6. Throughput of base and tornbit RAWLs..

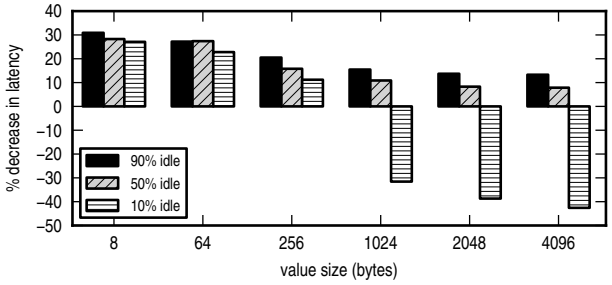

Figure 6. Decrease in write latency of hashtable with asynchronous over synchronous truncation.

Table 5 compares the cost of updating the tree with Mnemosyne transactions against the cost of periodically serializing it using Boost [9] and storing it on PCM-disk. Tree updates cost 5-6 μs with Mnemosyne, and on average 10 percent of the tree can be updated for the cost of serializing and storing the tree just once. Thus, even data structures with high rates of change can more efficiently be made persistent with transactions than by serializing and storing.

6.3.1 Optimizations

Mnemosyne implements two optimizations to reduce the cost of transactions: the torn bit in the RAWL, and asynchronous log truncation. Table 6 compares performance of torn bit against an implementation of the RAWL that writes a commit record, with a separate fence. For log records smaller than 2048 bytes, the torn-bit log performs up to 100 percent better. Above 2048 bytes, the torn bit log performs *worse* than a separate commit record. The cost of the fence is fixed, while the cost of bit manipulation to implement torn bits scales with the amount of data. Thus, for large records, the cost of manipulating bits to make space for the torn bit is larger than the cost of a single extra fence. As Mnemosyne targets smaller transactions, the torn bit log is a valuable optimization but should be omitted if large transactions are expected.

Asynchronous log truncation may reduce latency and improve performance under low load because it moves flushing modified data off the critical path. Instead, a separate thread processes the log to flush all modified data to PCM, after which it truncates the log. Figure 6 compares performance when the hash table thread is idle 90, 50, and 10 percent of the time. We find that for 50 and 90 percent idle time, the truncation thread can keep up with the active thread and achieves a reduction of 7-31% in write latency.

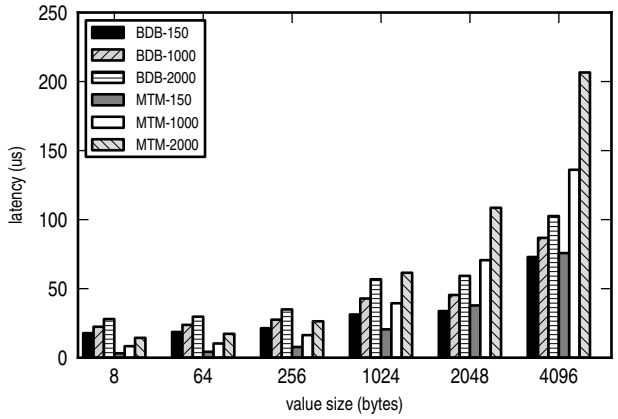

Figure 7. Mnemosyne's performance relative to Berkeley DB for different memory access latencies.

However, with only 10 percent idle time, the active thread stalls for extended periods while the truncation thread flushes data, which can increase latency by up to 42% for 4KB writes. Thus, this optimization is most helpful for transactions with a moderate duty cycle.

6.3.2 Reincarnation Cost

There are two reincarnation costs associated with Mnemosyne: (i) the cost to reconstruct persistent regions when the OS boots, and (ii) the cost to remap persistent regions to a process' address space, scavenge the persistent heap, and replay all completed but not flushed transactions when a program starts. For (i), we measured the worst case scenario cost, which is to reconstruct a persistent region for each SCM frame. This takes approximately 734 ms for 1GB of SCM, indicating that reincarnation contributes less than 1s overhead to the OS boot process. For (ii), we found the overall cost in our workloads to be less than 100 ms. Specifically, we measured the cost to remap the persistent regions to be about 1.1 ms, while the cost to scavenge the persistent heap region and reconstruct the heap's volatile indexes was about 89 ms. This high cost is due to the incremental allocation of memory needed by the volatile indexes, which results to a large number of `brk` system calls. We believe this cost could be reduced via bulk allocation or lazy construction of the indexes but have not implemented any of these optimizations. Finally, we found the cost to replay a completed but not flushed transaction ranges between 3 to 76 μs. In the case of synchronous truncation, the number of completed but not flushed transactions is bounded by the number of threads, which for 4 threads results in a worst case cost of about 300 μs.

6.4 Sensitivity to Memory Performance

Mnemosyne exposes persistent memory directly to programs as memory. If however, SCM latencies are long, then applications may perform better treating SCM like disk, with associated optimizations to overlap I/O and computation and to optimize for sequential access. We evaluate the impact of different memory latencies by comparing the performance of Mnemosyne transactions on a hash table to Berkeley DB, which optimizes for disk-like performance.

Figure 7 shows the relative performance of Mnemosyne over Berkeley DB as a function of data element size for three different latencies: 150 ns, 1000 ns, and 2000 ns. For small data sizes, Mnemosyne is always faster because it needs to write much less data. However, for longer latencies the benefit is much lower: 200 percent better performance for 1000 ns latencies and 100 percent for 2000 ns latencies. Furthermore, the benefit drops off faster with larger data sizes: at 2000 ns, Berkeley DB and Mnemosyne perform the same for 1024 byte inserts.

Thus, Mnemosyne is most useful when SCM latencies are close to those of DRAM, because the benefit of modifying small amounts of data outweighs the latency of access. For larger latencies, SCM may best be treated as a disk and accessed through the file system. Should someone prefer using Mnemosyne though, then our optimizations that remove latency off the critical path become even more evident. For example, under low load (10 percent idle) and 2000 ns latency to SCM, asynchronous truncation would be able to reduce write latency for 1024 and 2048 bytes by 70% and 60% respectively, enabling Mnemosyne to still perform better than the disk-based interface.

7. Related Work

NV-heaps [16] is a concurrently developed, project that also seeks to provide transaction updates to data in storage-class memory. However, the focus of the two projects is different: NV-heaps focus on on persisting user-defined *objects*, so they support transactions via an object-based transactional memory system. In contrast, Mnemosyne supports transactions via a word-based transactional memory system, which can be used to transactionally update any data kept in persistent memory. Nevertheless, NV-heaps provide some features not currently found in Mnemosyne, such as type-safe pointers and garbage collection via reference counting. Finally, NV-heaps require processor support, while Mnemosyne can be used with existing commodity processors.

We next discuss past projects on persistent objects/languages, recoverable memory, transactional memory, and transaction processing.

Persistent memory. Several projects have investigated the use of a large battery-backed memory for persistent storage [10, 18, 23]. Unlike Mnemosyne, the memory in these systems is accessible only through a driver to prevent corruption, and hence only available through a DBMS or file system. The eNVy system [65] focuses on the architecture of attaching flash to the memory bus as persistent memory, and uses a battery-backed SRAM buffer to hide the block-addressable nature of flash. Mnemosyne instead focuses on the programming model and can be used with eNVy's architecture.

Persistent objects/languages. Previous research on persistent object systems attempted to provide language level persistence through the integration of database systems and programming languages [3, 4, 11, 36]. Virtual memory-based persistent object systems, such as Texas [58], QuickStore [64] are most similar. However, these systems rely on a C++ object-oriented programming model, which limits the applicability of the system to programs non-OO programs. Furthermore, Texas provides a checkpoint mechanism for consistency, which is insufficient in the presence of multiple threads. Finally, these systems operate on virtual memory pages, which makes the system heavier weight, as they must read or write more data at a time and rely on optimizations such as page-diffing to reduce the amount of data written. However, the higher-level interface to these systems and others, such as Thor [40], allows them to provide better safety properties, such as only allowing pointers in persistent data structures to reference other persistent data (or, in the case of Thor, to make persistent any volatile structures referenced). Their knowledge of all pointers in a data structure also enable garbage collection to prevent memory leaks. These mechanisms could be layered on top of Mnemosyne's primitives to provide the same guarantees.

Durable memory transactions. Mnemosyne is similar to lightweight recoverable virtual memory (RVM) [56] and Rio Vista transactions [41]. Mnemosyne borrows the notion of persistent re-

gions from RVM, but commits data at word granularity rather than at page granularity. Unlike Rio Vista, Mnemosyne transactions operate completely at user level and do not require a battery back-up system. Several IBM systems have provided transactions in memory by marking pages accessed by a transaction in a TLB-like structure [13, 60]. Similar to RVM, this provides coarse-grained transactions at the page granularity, and relies on a disk for durability.

Stasis is a user-mode library that generalizes write-ahead-logging (WAL) to provide transactional storage to applications [57]. Unlike Mnemosyne, Stasis does not provide compiler support for transactions [57], so programmers must explicitly code transactional writes and reads.

Transactional memory. Recent work on transactional memory focuses on the concurrency of transactions, through automatic detection of conflicts, rather than on durability [30, 29]. Mnemosyne relies on software transactional memory techniques to automatically annotate program code in a compiler, and to implement efficient commit processing. As a result, our system benefits from recent research in the area [66]. Recent work by FusionIO on atomic updates to flash storage uses a technique similar to the torn-bit RAWL [50], but does not need to use sense reversing.

Transactional processing. The use of transactions as the basic unit of atomicity and durability has been the focus of the database and systems research community for almost three decades [27, 28, 45]. Similar to many database systems, Mnemosyne relies on write-ahead logging for recovery. Mnemosyne differs from disk-based transactions in that it cannot control when data is evicted from the cache, and as a result must force log writes early. In addition, the low cost of SCM access raises the cost of sophisticated logging and recovery mechanisms [45]. Similar to Mnemosyne, databases must deal with partial writes, but often solve it with checksums [44]. Main-memory databases [26, 20] store all data in memory rather than caching data, but still rely on a disk for persistently storing a log of committed updates.

Storage-class memory for files. Other work focuses on SCM as a replacement/acceleration for flash or magnetic storage media. One approach is to store just metadata or frequently accessed data in SCM [22, 35], treating it as a block device. BPFS instead leverages SCM's byte addressability to reduce the amount of metadata written during an update and achieves consistency through shadow updates [17]. These approaches all improve performance for file access, but do not provide fine-grained persistence to programs.

8. Conclusions

Programmers trained in the era of disks have learned that frequently updating persistent state should either be avoided or be handled by a database engine. Storage class memory presents new opportunities for fast data persistence that obsolete these rules. Mnemosyne provides lightweight persistent memory, and common in-memory data structures can be made persistent using durable transactions. Thus, programmers can create a single data structure, optimized for memory, rather than designing separate in-memory and update-optimized persistent representations. Compared to past work on persistent object systems, Mnemosyne provides greater flexibility, by not requiring C++ objects, and lower latency by persisting data directly to memory at the granularity of a a single update rather than a whole page.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) grants CNS-083447 and CNS-0720565. We would like to thank Mark Hill and Arkaprava Basu for valuable discussions during the initial stages of the project. We would also like to thank our shepherd, Emery Berger, and the anonymous reviewers for their invaluable feedback. Swift has a significant financial interest in Microsoft.

References

[1] Sparse - a semantic parser for C. `sparse.wiki.kernel.org`.

[2] AMD, Inc. Software optimization guide for AMD64 processors. `http://support.amd.com/us/Embedded\_TechDocs/25112.PDF`, 2005.

[3] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, Nov 1983.

[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] 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. In *VLSI Circuits*, 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 9*, Nov. 2000.

[7] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw., Pract. Exper.*, 18(9):807–820, 1988.

[8] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. Technical report, Sun Microsystems.

[9] Boost C++ Libraries. Serialization overview. `http://www.boost.org/doc/libs/1\_42\_0/libs/serialization/doc/index.html`, Nov. 2004.

[10] T. C. Bressoud, T. Clark, and T. Kan. The design and use of peristent memory on the DNCP hardware fault-tolerant platform. In *DSN*, 2001.

[11] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. *SIGMOD Rec.*, 23(2):383–394, 1994.

[12] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC 26*, Aug. 2007.

[13] A. Chang and M. F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1), Feb. 1988.

[14] J. H. Choi, H. Franke, and K. Zeilenga. Enhancing the performance of OpenLDAP directory server with multiple caching. In *International Symposium on Performance Evaluation of Computers and Telecommunications Systems (SPECTS)*, July 2003.

[15] Christopher Clark. C hash table. `http://www.cl.cam.ac.uk/~cwc22/hashtable/hashtable.c`.

[16] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS 16*, Mar. 2011.

[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 *SOSP 22*, pages 133–146, Oct. 2009.

[18] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In *VLDB 15*, Aug. 1989.

[19] J. Corbet. Fsyncers and curveballs (the firefox 3 fsync() problem). `http://lwn.net/Articles/283745/`, May 2008.

[20] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[21] E. Doller. Phase change memory and its impacts on memory hierarchy. `http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf`, 2009.

[22] N. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt. MRAMFS: A compressing file system for non-volatile RAM. In *Proceedings of thje IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 596–603, Oct. 2004.

[23] F. Eskesen, M. Hack, A. Iyengar, R. P. King, and N. Halim. Software exploitation of a fault-tolerant computer with a large memory. In *FTCS*, 1998.

[24] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP 13*, Feb. 2008.

[25] R. F. Freitas and W. W. Wilcke. Storage-class memory: the next storage system technology. *IBM J. Res. Dev.*, 52(4):439–447, 2008.

[26] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, 1992.

[27] J. Gray. The transaction concept: Virtues and limitations. In *VLDB 7*, Sept. 1981.

[28] J. Gray, P. Mcjones, M. Blasgen, B. Lindsay, R. Lorie, and T. Price. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13:223–242, 1981.

[29] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP 10*, June 2005.

[30] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA 20*, May 1993.

[31] M. Hirabayashi. Tokyo cabinet: a modern implementation of DBM. `http://1978th.net/tokyocabinet/`, 2010.

[32] Y. Huai. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS Bulletin*, 18(6):33–40, Dec. 2008.

[33] Intel Corp. Intel 64 and ia-32 architectures software developers manual volume 1: Basic architecture. `http://www.intel.com/assets/pdf/manual/253665.pdf`, Mar. 2010.

[34] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *ASPLOS 15*, Mar. 2010.

[35] J. Jung, Y. Won, E. ki Kim, H. Shin, and B. Jeon. Frash: Exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Transactions on Storage*, 6(1), 2010.

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

[37] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2), Mar. 1977.

[38] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[39] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA 36*, June 2007.

[40] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *SIGMOD Conference*, pages 318–329, 1996.

[41] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *SOSP 16*, Oct. 1997.

[42] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of FREENIX*, June 1999.

[43] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - the unix file system check program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, Apr. 1986.

[44] Microsoft Corp. SQL server 2008 books online: Memory management architecture: Buffer management. `http://msdn.microsoft.com/en-us/library/aa337525.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] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *HPCA 12*, pages 258–269, Feb. 2006.

[47] MySQL Performance Blog. Tokyo tyrant the extras part i : Is it durable? `http://www.mysqlperformanceblog.com/2009/11/10/tokyo-tyrant-the-extras-part-i-is-it-durable/`, Nov. 2009.

[48] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA 23*, Oct. 2008.

[49] Numonyx. Omneo P8P PCM 128-Mbit Parallel PCM. `www.numonyx.com/Documents/Datasheets/316144_P8P_DS.pdf`, Aug. 2010.

[50] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *HPCA 17*, Feb. 2011.

[51] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP 20*, pages 206–220, Brighton, United Kingdom, Oct. 2005.

[52] 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*, Dec. 2009.

[53] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA 36*, June 2007.

[54] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[55] Samsung. Samsung ships industry's first multi-chip package with a pram chip for handsets. `http://www.samsung.com/us/business/semiconductor/newsView.do?news\_id=1149`, Apr. 2010.

[56] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *SOSP 14*, Dec. 1993.

[57] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *OSDI 8*, Dec. 2008.

[58] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: good, fast, cheap persistence for c++. *SIGPLAN OOPS Mess.*, 4(2):145–147, 1993.

[59] Sleepycat Software. Sleepycat software: Berkeley DB database. http://www.sleepycat.com.

[60] F. G. Soltis. *Inside the AS/400*. Duke Press, second edition, 1997.

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

[62] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[63] C. Villa, D. Mills, G. Barkley, H. Giduturi, S. Schippers, and D. Vimercati. A 45nm 1Gb 1.8V phase-change memory. In *ISSCC 2010*, pages 270 –271, Feb. 2010.

[64] S. J. White and D. J. DeWitt. Quickstore: A high performance mapped object store. *VLDB Journal*, 4(4):629–673, 1995.

[65] M. Wu and W. Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *ASPLOS 6*, Oct. 1994.

[66] P. Wu, M. M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. W. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, M. F. Mergen, X. Shen, M. F. Spear, H. Wang, and K. Wang. Compiler and runtime techniques for software transactional memory optimization. *Concurrency and Computation: Practice and Experience*, 21(1):7–23, 2009.

[67] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA 36*, June 2007.