Mojim: A Reliable and Highly-Available Non-Volatile Memory System

Yiying Zhang Jian Yang Amirsaman Memaripour Steven Swanson

Department of Computer Science and Engineering, University of California, San Diego

{yiyingzhang,jianyang,amemarip,swanson}@cs.ucsd.edu

Abstract

Next-generation non-volatile memories (NVMs) promise DRAM-like performance, persistence, and high density. They can attach directly to processors to form non-volatile main memory (NVMM) and offer the opportunity to build very low-latency storage systems. These high-performance storage systems would be especially useful in large-scale data center environments where reliability and availability are critical. However, providing reliability and availability to NVMM is challenging, since the latency of data replication can overwhelm the low latency that NVMM should provide.

We propose *Mojim*, a system that provides the reliability and availability that large-scale storage systems require, while preserving the performance of NVMM. Mojim achieves these goals by using a two-tier architecture in which the primary tier contains a mirrored pair of nodes and the secondary tier contains one or more secondary backup nodes with weakly consistent copies of data. Mojim uses highly-optimized replication protocols, software, and networking stacks to minimize replication costs and expose as much of NVMM's performance as possible. We evaluate Mojim using raw DRAM as a proxy for NVMM and using an industrial NVMM emulation system. We find that Mojim provides replicated NVMM with similar or even better performance than un-replicated NVMM (reducing latency by 27% to 63% and delivering between 0.4 to 2.7× the throughput). We demonstrate that replacing MongoDB's built-in replication system with Mojim improves MongoDB's performance by 3.4 to 4×.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..

ACM 978-1-4503-2835-7/15/03. . . $15.00.

http://dx.doi.org/10.1145/2694344.2694370

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Main memory

Keywords non-volatile memory; distributed storage systems; reliability; availability; data center; storage-class memory

1. Introduction

Fast, non-volatile memory technologies such as phase change memory (PCM), spin-transfer torque magnetic memories (STTMs), and the memristor are poised to radically alter the performance landscape for storage systems. They will blur the line between storage and memory, forcing designers to rethink how volatile and non-volatile data interact and how to manage non-volatile memories as reliable storage.

Attaching NVMs directly to processors will produce non-volatile main memories (NVMMs), exposing the performance, flexibility, and persistence of these memories to applications. However, taking full advantage of NVMMs' potential will require changes in system software [3].

The need for such changes is especially acute in large-scale data center environments where storage systems require more than simple non-volatility. These environments demand reliability and availability in the face of hardware, software, and network failures. Without this reliability and availability, NVMM will only be suitable as a transient data store or as a caching layer—it will not be able to serve as a reliable primary storage medium.

Data centers traditionally provide both reliability and availability by adding redundancy using replication [5, 11, 15, 55, 56] or erasure coding schemes [18, 21]. These approaches rest on the assumption that storage is slow, so the cost of the network and software protocols required to implement replications is acceptable.

NVMM will change this situation completely, since the networking and software overhead of existing replication mechanisms will squander the low latency that NVMM can provide. The interface with NVMM is also different from traditional storage: applications access NVMM directly with fine-grained memory operations.

We propose *Mojim*[1], a system that provides replicated, reliable, and highly-available NVMM as an operating system service. Applications can access data in Mojim using normal load and store instructions while controlling when and how updates propagate to replicas using system calls. Mojim allows applications to build data persistence abstractions ranging from simple log-based systems to complex transactions.

Mojim uses a two-tier architecture that allows flexibility in choosing different levels of reliability, availability, consistency, and monetary cost, while minimizing performance overhead. The primary tier includes one *primary node* and one *mirror node*. Mojim can, depending on the configuration, keep these nodes strongly or weakly consistent. An optional secondary tier provides an additional level of redundancy with one or more *backup node*s that are weakly consistent with the primary tier.

Mojim efficiently replicates fine-grained data from the primary node to the mirror node using an optimized RDMA-based protocol that is simpler than existing replication protocols. The mirror node replicates data to the backup nodes in the background, thus keeping the secondary tier off the performance-critical path. This design offers good performance and two strongly consistent copies of data plus more copies of weakly consistent data. To further improve availability and reliability, Mojim also provides a fast recovery process and atomic semantics that guarantee data integrity.

In building Mojim, we explore the performance and monetary cost impacts of providing availability, reliability, and consistency with NVMM, and we explore trade-offs among replication protocols for NVMM. Interestingly, we find that adding availability, reliability, and consistency does not necessarily impair NVMM performance, as long as the replication protocols and software layers are optimized for NVMM.

We evaluate Mojim using raw DRAM as a proxy for future NVMMs and with an industrial NVMM emulation system. Our evaluation shows that, surprisingly, Mojim reduces the average latency of the un-replicated system by 27% to 63%, even when it provides strongly consistent copies of data. Mojim's performance gain is mainly due to inefficiencies in the current instruction sets the un-replicated system uses to enforce data persistence. Mojim provides 0.4 to 2.7× the throughput of the un-replicated system. We also run several popular applications including a file system [12], the Google Hash Table [16], and MongoDB [40] on Mojim. The MongoDB results are the most striking: Mojim is 3.4 to 4× faster than the MongoDB replication mechanism and 35 to 741× faster than un-replicated MongoDB.

The rest of the paper is organized as follows. Section 2 provides some background on persistent memory and replicated storage systems. We present Mojim and its implementation in Sections 3 and 4. Section 5 describes our experience adapting applications to use Mojim. We then present the evaluation results of Mojim in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

[1] Mojim (魔镜) is the Chinese word for "magic mirror."

2. NVMM in the Data Center

NVMM blurs the line between memory and storage, and it poses new challenges for system designers and architects. Previous research on NVMM has focused on how to use these memories in a single machine [3, 8, 9, 12, 39, 58, 59], while most mission-critical data resides in distributed, replicated storage systems (e.g., in data centers). For NVMM to succeed as a first-class storage technology, it must provide the reliability and availability that these storage systems require [26].

Mojim's goal is to make NVMM a reliable and highly-available storage layer suited to these data center environments. Achieving this goal will require designers to address trade-offs between performance, reliability, monetary cost, and consistency. Below, we introduce NVMM and discuss why it demands new approaches to providing availability, reliability, and consistency in storage systems.

2.1 Next-Generation Non-Volatile Memory

NVM technologies are closing the performance, cost-per-bit, and capacity gap between low-latency, volatile memory technologies and high-capacity, persistent storage technologies [25, 32, 49, 62]. Next-generation NVM technologies (like PCM, the memristor, and STTM) are byte-addressable and projections show that their performance may approach that of DRAM [20, 27, 35, 60]. For example, PCM, the most mature next-generation NVM technology, has access latencies within a small factor of DRAM [33, 34, 38, 48].

Attaching next-generation NVMs to the main memory bus provides a raw storage medium that is orders of magnitude faster than modern disks and SSDs. NVMM presents many new technical challenges and has inspired a host of research projects on topics including OS management of NVMM [3], user-space libraries and programming models [8, 58], specialized NVMM file systems [9, 12, 59], and hybrid main memory and heterogeneous memory allocation [39, 41, 50]. This work focuses on the problems of replicating NVMM content so that NVMM can be applied in a distributed data center context.

2.2 NVMM Availability, Reliability, and Consistency

Although NVMM protects against power failure by making the contents of memory persistent, it does not address the other ways that systems fail, including software, hardware, and networking errors that are common in data centers [14, 42]. Providing availability and reliability in such environments is important to meet client SLAs [55] and application requirements. Strong consistency is also desirable in storage systems, since it makes it easier to reason about system correctness.

Adding redundancy or replication is a common technique for providing reliability and availability [1, 5, 11, 15, 17,

18, 29, 46, 47, 51, 52, 56, 61]. Various protocols exist to provide different consistency levels among redundant copies of data [2, 4, 11, 22, 31, 47]. For traditional storage systems with slow hard disks and SSDs, the performance overhead of replication is small relative to the cost of accessing a hard drive or SSD, even with complex protocols for strong consistency. With NVMMs, however, the networking round trips and software overhead involved in these techniques [4, 22, 31, 56] threaten to outstrip the low-latency benefit of using NVMMs in the first place. Even for systems with weak consistency [11, 47], increasing the rate of reconciliation between inconsistent copies of data can threaten performance [17, 28].

Since NVMM is vastly faster than existing storage technologies, it presents new challenges to data replication. First, NVMM-based systems must deliver high performance to justify their increased cost relative to disks or SSDs. Existing replication mechanisms built for these slower storage media have software and networking performance overhead that would obscure the performance benefits that NVMM could provide.

Second, NVMM is memory, and applications should be able to use it like memory (*i.e.*, via load and store instructions without operating system overheads for most accesses) rather than as a storage device (*i.e.*, via I/O system calls).

3. Mojim Design

Mojim provides an easy-to-use, generic layer of replicated NVMM that ensures reliability, availability, and consistency, while sacrificing as little of NVMM's performance as possible. Mojim uses a two-tier architecture and supports several operating modes to let applications tune Mojim's reliability, availability, and consistency to match their particular needs.

This section discusses Mojim's interfaces and architecture and the different modes Mojim provides.

3.1 Mojim's Interfaces

Mojim is an operating system service that provides reliable and highly-available NVMM. This section describes Mojim's typical usage scenario and the interface it provides.

To use Mojim, a system configuration file specifies a set of *Mojim regions* on the *primary node* to be replicated, along with a *mirror node* and a list of *backup nodes* where the replicas should reside. The primary node supports reads and writes to the replicated data. The mirror node and backup nodes support reads only. Kernel modules can access these regions and use them to build complex, replicated, memory-based services such as a kernel-level persistent key value store, a persistent disk cache, or a file system. The kernel could also make these services available to applications via a `malloc()`-like interface.

While Mojim can serve as the basis for many memory-based services, deploying an NVMM-aware file system to manage the replicated NVMM region would provide the most flexibility in application usage models. The file system would provide familiar file-system-based mechanisms of allocation and naming as well as conventional file-based access for non-performance-critical applications. The key requirement of the file system is that, for an *mmap*'d file, it maps the the NVMM pages corresponding to the file directly into the applications' address spaces rather than paging them in and out of the kernel's buffer cache. In our experiments, we use PMFS [12] for this purpose.

With a file system in place, applications can create files in the Mojim-backed file system and map them into their address space using *mmap*. We call the NVMM area mapped by applications the *data area*. After an *mmap*, applications can perform direct memory accesses to the data area using load and store instructions on the primary node and load instructions on the mirror node.

Mojim provides a mechanism called a *sync point* that allows applications to control when and what updates in the data area propagate to the replicas. At each sync point, Mojim atomically replicates all memory regions specified by an application.

Two APIs allow applications to create sync points: *msync* and *gmsync*.

Mojim leverages the existing *msync* system call to specify a sync point that applies to a single, contiguous address range. The semantics of Mojim's *msync* correspond to conventional *msync*, and applications that use *msync* will work correctly without modification under Mojim. Mojim allows an application to specify a fine-grained memory region in the *msync* API and replicates it atomically, while traditional *msync* flushes page-aligned memory regions to persistent storage and does not provide atomicity guarantees [45].

Mojim's *gmsync* adds the ability to specify multiple memory regions for the sync point to replicate, allowing for more flexibility than *msync*.

Mojim provides a mechanism to allow applications to make their data persistent atomically, but it does not provide primitives for synchronization. It would be possible to add synchronization primitives to Mojim, but this would increase the complexity of the system and require selecting a set of synchronization mechanisms to support. A better approach would be to build synchronization mechanisms that leverage Mojim's mechanisms.

Figure 1 shows a simple example in C of how to use Mojim to manage an append-only log on Mojim. The program first opens and *mmap*s a file in a Mojim region. It then updates the access count of the log and makes this value persistent with the conventional *msync* API. Next, it appends a log entry and updates the size of the log. It makes both these data persistent with an *gmsync* call. The atomicity that *gmsync* provides guarantees that the log size is consistent with the log content on the replica nodes.

```
int fd = open("/mnt/mmapfile", O_CREAT|O_RDWR);            // open a file in mounted Mojim region
void *base = mmap(NULL, 40960, PROT_WRITE,
                  MAP_SHARED, fd, 0);                      // mmap a 40KB area in the file

unsigned long *access_count_p = base;                      // access count of the log
unsigned long *log_size_p = base + sizeof(unsigned long);  // size of the log
int *log = base + 2*sizeof(unsigned long);                 // the log

*access_count_p = *access_count_p + 1;                     // memory load and store
msync(access_count_p, sizeof(unsigned long), MS_SYNC);     // call conventional msync

int beautiful_num = 24;
unsigned long curr_log_pos = *log_size_p;                  // memory load and store
log[curr_log_pos] = beautiful_num;
*log_size_p = *log_size_p + 1;
struct msync_input { void *address; int length; };
struct msync_input input[2];
input[0].address = &(log[curr_log_pos]);
input[0].length = sizeof(int);
input[1].address = log_size_p;
input[1].length = sizeof(unsigned long);
gmsync(input, 2, MS_MOJIM);                                // call gmsync to commit the log append
```

Figure 1. Sample code to use Mojim. *Code snippet that implements a simple log append operation with Mojim.*

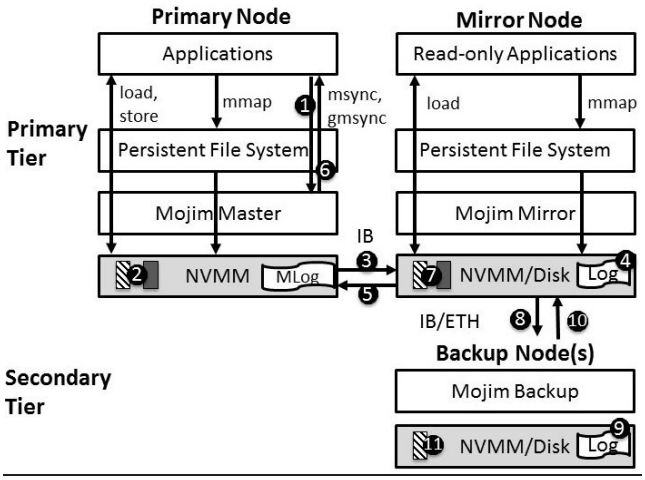

Figure 2. Mojim Architecture. *The numbered circles represent different steps in the Mojim replication process. MLog stands for the metadata log.*

3.2 Architecture

Mojim uses a two-tier architecture. The primary tier contains a primary node and its read-only mirror node; the secondary tier includes one or more backup nodes with weakly consistent, read-only copies of data. Figure 2 depicts the architecture of Mojim.

Mojim's primary tier contains a pair of mirroring nodes: a primary node replicates data to its mirror node at each *sync point* (i.e., call to *msync* or *gmsync*). The application can read and write data on the primary node, but Mojim only allows reads from the mirror node.

The primary tier offers good performance even when guaranteeing strong consistency, since it requires only one networking round trip for each sync point. Existing architectures that allow writes to all replicas (*E-writeall*) [31], or that use one primary and multiple secondary nodes (*E-chain and E-broadcast*) [2, 56], require multiple networking round trips or other performance overhead to guarantee strong consistency. We will discuss how Mojim differs from these existing schemes in more detail in Section 7.

To further improve performance, we connect the primary node and the mirror node with a high-speed Infiniband link and use an efficient software and networking layer to replicate data between them.

To improve reliability, we place the primary node and the mirror node on different racks, since failure bursts often happen within the same rack [14, 42].

The optional secondary tier includes one or more backup nodes to maintain additional copies of data. It provides additional reliability and availability, so that failure bursts will not be catastrophic. The mirror node replicates data to the backup nodes in the background. Thus, data in the backup nodes is not strongly consistent with data in the primary tier. By keeping the replication to the secondary tier in the background and off the performance-critical path, Mojim ensures good application performance.

With both tiers in operation and a total of N nodes, Mojim can tolerate $N - 1$ node failures. In most environments, one or a few backup nodes are enough to prevent data loss, since failure bursts are more likely to involve only a small number of nodes [14, 42]. Also, in most failure bursts, the nodes do not all fail at the same time; failures are usually separated by a few seconds. A fast recovery can thus prevent data loss even with few copies of replicated data. We discuss recovery optimizations in Section 4.3.

Scheme	R	A	C	$
S-unreplicated	0	Worst	N/A	Low
M-async	1	Good	Weak	Fair
M-sync	1	Good	Strong	Fair
M-syncdisk	1	OK	Strong	Low
M-syncsec	$N-1$	Best	Strong+Weak	High
M-syncseceth	$N-1$	Good	Strong+Weak	Fair
E-writeall	$N-1$	Best	Strong	High
E-chain	$N-1$	Best	Strong	High
E-broadcast	$N-1$	Best	Strong	High

Table 1. Replication Schemes. *Mojim supports a wide range of reliability, availability, consistency, and monetary cost levels (columns 2-5). The reliability column represents the number of node failures that can be tolerated in a system of N nodes. The last three rows compare Mojim to other existing replication schemes.*

3.3 Mojim Modes and Replication Protocols

Mojim supports several replication modes and protocols that allow users to choose different levels of performance, reliability, consistency, availability, and monetary cost depending on application requirements.

Table 1 summarizes these different modes and their properties, and we discuss them below using the numbered circles in Figure 2 to illustrate the replication process in each mode.

Across all the modes Mojim provides, Mojim achieves most of its performance by adopting a different architecture than most replicated storage systems. Instead of supporting multiple consistent replicas, Mojim only supports strong consistency at a single mirror node. This decision makes our replication protocols much simpler (e.g., there's no need for multi-phase commit or a complex consensus protocol) and, therefore, allows for much higher performance.

Mojim achieves the goal of providing its atomic data persistence interface by ensuring that atomic operations are replicated atomically to the mirror node and the backup nodes, by appending replicated data to logs on the mirror node and the backup nodes.

Un-replicated without Mojim: A single machine without Mojim (*S-unreplicated*) must flush an *msync*'d memory region from the processor's caches to ensure data persistence. S-unreplicated has poor availability and is only as reliable as the NVM devices. Moreover, even if the NVMM is recovered after a crash, data can still be corrupted. For example, if a crash occurs after a pointer is made persistent but before the data it points to becomes persistent, the system will contain corrupted data.

Sync: Mojim's *M-sync* mode guarantees strong consistency between the primary and the mirror node. It provides improved reliability and availability over S-unreplicated, since in the case of a failure the mirror node can take the place of the primary node without losing data.

In M-sync, when an application calls *msync* or *gmsync* (① in Figure 2), Mojim pushes data from the primary node to the mirror node via RDMA (③) and writes the data in the mirror node log (④). The primary node waits for the acknowledgment from the mirror node (⑤), and then returns the *msync* or *gmsync* call (⑥). The mirror node later takes a checkpoint to apply the log contents to the data area (⑦). Mojim stores both the mirror node logs and its data area in NVMM for high performance and fast recovery.

In M-sync, Mojim does not flush data from the primary node's caches (②). Modern RDMA devices are cache-coherent, so they will send the most up-to-date data [24, 30]. Thus, the mirror node always gets the latest data and pushing data to the mirror node is sufficient to ensure persistence. If the primary node crashes, the mirror node has the most up-to-date data. If the mirror node crashes, the primary node has all the data, but it may not be persistent, so the primary node immediately flushes its caches to prevent data loss. This means there is a small "window of vulnerability" after a mirror node failure during which a primary node failure could result in data loss. On our system, this window lasts for 450 μs, the time required to flush the processor caches.

Surprisingly, our evaluation results show that M-sync offers performance comparable to or better than S-unreplicated because flushing CPU caches is often more expensive than pushing the data over RDMA. The current *clflush* instruction is strongly ordered and cannot utilize the parallelism offered in modern processor architecture. Intel recently announced two instructions that are more efficient than *clflush* and that will be available on future systems [23], which should help resolve this problem.

Sync with cache flush: To close the window of vulnerability mentioned above, Mojim can flush data from the primary node's caches (②) before returning to applications' *msync* or *gmsync* calls (⑥). This mode is called *M-syncflush*, and with M-syncflush, all data can survive simultaneous failures of the primary node and the mirror node.

Async: *M-async* provides weaker consistency between the primary node and the mirror node. M-async ensures that data is persistent on the primary node for each sync point (②) and pushes the data to the mirror node (③), but it does not wait for the mirror node's acknowledgment (⑤) to complete the application's *msync* or *gmsync* call (⑥). Thus, data on the mirror node can be out of date relative to the primary node. M-async must flush the primary node CPU caches at each sync point to ensure that the latest data is persistent.

Sync with slow storage: To reduce the monetary cost of M-sync, Mojim supports a mode that stores the log on the mirror node in NVMM, but stores the mirror node's data area on a hard disk or SSD (*M-syncdisk*). M-syncdisk has a slower recovery process than M-sync, since Mojim needs to read data from hard disk or SSD to NVMM before applications can access them.

Sync with the secondary tier: *M-syncsec* adds the secondary tier to M-sync and increases reliability and availability by adding more copies of data. Mojim replicates data from the mirror node to the backup node in the background

(⑧-⑪). M-syncsec provides two strongly-consistent copies of the data at the primary and mirror nodes and one weakly-consistent data copy at each backup node. The amount of inconsistency between the mirror node and backup nodes is tunable and affects the recovery time. Even though the data at each backup node may be out-of-date, it still represents a consistent snapshot of application data because of the atomic semantics Mojim provides. Our evaluation results show that M-syncsec delivers performance similar to M-sync because replication to the backup nodes takes place in the background.

Sync with low-cost secondary tier: M-syncsec requires fast networks between the mirror node and backup nodes, which increases the monetary cost and networking bandwidth consumption of the system. A lower cost option, *M-syncseceth*, uses Ethernet between the mirror node and backup nodes. M-syncseceth has the worst performance of all the Mojim modes, but it still provides the same reliability, availability, and consistency guarantees as M-syncsec.

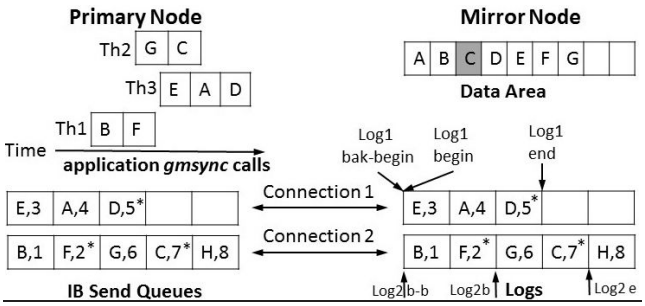

Figure 3. Example of Mojim Replication. *An example of Mojim's replication process. Each cell represents a request. The letter in the cell stands for the memory address and the number in the cell represents its unique ID. The upper-left part shows three threads placing three* gmsync *calls. The upper-right part shows the data area on the mirror node. The * represents the end mark of a* gmsync *operation. The gray cell in the mirror node data area represents the data that is recovered after a crash.*

4. Implementation

This section describes our implementation of Mojim in the Linux kernel. The core of Mojim comprises an optimized network stack and the replication and recovery code.

4.1 Networking

The networking delay of data replication is the most important factor in determining Mojim's overall performance. Mojim uses Infiniband (IB), a high-performance switched network that supports RDMA. RDMA is crucial because it allows the primary node to transfer data directly into the mirror node's NVMM without requiring additional buffering, copying, or cache flushes.

Mojim uses *IB-Verbs*, a set of native IB APIs based on send, receive, and completion queues [37]. IB-Verbs requires the application to post send (receive) requests to send (receive) queues. It uses completion messages in the completion queue to indicate the completion of requests and supports both polling and interrupts to detect completions. IB-Verbs offers native IB performance and outperforms alternative IB protocols such as IPoIB and RDS (see Section 6.2). Existing IB-Verbs implementations are userspace libraries that bypass the kernel. We created a kernel version of IB-Verbs for Mojim.

Mojim uses a thin protocol based on the reliable transportation mode of IB-Verbs. The Mojim protocol directly fetches data from the primary node and writes it to NVMM on the mirror node. For each sync point, the primary node posts a send request on the IB send queue and polls for its completion. The mirror node posts a set of receive requests in advance and polls for the arrival of incoming messages. Our measurements show that polling is more efficient than interrupts.

The protocol does not require explicit acknowledgment messages from the mirror node to the primary node, since we configure the IB link to provide a successful completion notification for the primary node's send request only once the data transfer succeeds. In the event of an error or a timeout, the primary node resends the message to the backup node. After a set number of unsuccessful re-send attempts, Mojim invokes its recovery process.

To sustain high bandwidth, Mojim creates multiple IB connections to handle client requests. For each connection, we assign one thread on the mirror node to poll for incoming messages. On the primary node, we let the application thread perform IB send operations for M-sync and use a background thread to post these operations for M-async.

4.2 Replication

We now describe the Mojim replication process and the techniques that we use to enable reliable, atomic, and consistent data replication.

Primary tier replication: At each sync point, the primary node posts IB send requests containing the target memory regions. Mojim ensures that all requests belonging to the same atomic operation are consecutive and on the same IB connection and marks the last request to let the mirror node know the end of an atomic operation. Since Mojim's reliable IB protocol ensures ordering in each IB connection, these requests will appear in the same order on the mirror node. A unique ID on each send request allows the mirror node to keep updates ordered across IB connections. For recovery purposes, the primary node stores the memory addresses of the most recent requests in a *metadata log*.

For each IB connection, the mirror node maintains a circular log and a thread that polls incoming requests. Mojim places the logs in NVMM for good performance and persistence and pre-allocates fixed-size buffers on the logs for RDMA accesses. With pre-allocated memory slots, Mojim

only needs one IB roundtrip to replicate data from the primary node to the mirror node. Because the receive buffer size is fixed, we limit the size of each send request on the primary node and break original memory regions into multiple send requests if needed. Since RDMA writes directly to NVMM, there is no need to flush the cache on the mirror node.

After all the data for a sync point has arrived on the log, the mirror node can write them to their permanent locations in the data area. This checkpointing happens periodically after a configurable number of requests (*CHECKPOINT_THRESH*) have been received, as well as when the system is idle and when a log is full. Mojim maintains global pointers to the beginning and end of each log to indicate data available for checkpointing.

To ensure that read-only applications on the mirror node see a consistent view of their data, Mojim removes the page table entries of the affected memory locations before a checkpointing operation. During the checkpointing, an application reading from those pages will generate a page fault. We changed the page fault handler to wait until Mojim completes the checkpointing and then restore the page table entries and return the application read.

Secondary tier replication: Replication to the secondary backups occurs in the background when there is data on the mirror node's logs. The protocol for replication to the backup node mimics the replication to the mirror node. The mirror node maintains a pointer for each log to indicate the amount of data that has not yet been replicated to the backup node. Mojim uses a threshold (*SECONDARY_TIER_THRESH*) to limit the amount of such unreplicated data on the mirror node and stalls further replication to the mirror node until un-replicated data drops below SECONDARY_TIER_THRESH.

Example: Figure 3 illustrates an example of Mojim's data structures and its replication process. In this example, Mojim uses two IB connections and two mirror node logs. Three application threads post three *gmsync* calls to the two IB send queues. To guarantee atomicity, Mojim serializes thread 2's requests after thread 1's requests on the second send queue. Mojim then sends these requests to the mirror node's logs. The mirror node threads poll for the completion of these writes and update the log-end pointers when they have received all requests belonging to one *gmsync* call. The checkpointing service processes the logs from the log-begin pointer to the log-end pointer. The mirror node replicates the log content between the log-bak-begin pointer and the log-end pointer to the backup node.

4.3 Recovery

Fast recovery is crucial to providing high availability and preventing data loss in the event of a failure. There are three types of failure scenarios: primary, mirror, and backup node failures. Mojim uses heartbeats to detect failures, but other techniques [7, 36] are possible.

When the primary node fails, the mirror node becomes the new primary node and a backup node becomes the new mirror node. The new primary node first sends the un-replicated data in its logs to the new mirror node and checkpoints its log content to its data area after the failure. For M-syncdisk, the new primary node needs to load data from the data file on disk to the NVMM. After these operations, applications can restart on the new primary node. Until these operations complete, the Mojim contents will be unavailable.

One option for activating a new backup node is to wait for the failed node to come back online. Rebooting the machine is often sufficient and more efficient than constructing a new node [14]. When the crashed primary node restarts, it receives the new data accumulated during its down time from the new primary node. When the failed node cannot reboot fast enough, a human operator or a system monitoring service selects a new backup node based on its available NVMM size, its networking topology, and other criteria [6, 53]. The new node receives a complete copy of the memory region and begins processing updates from the new mirror node.

When the mirror node or the backup node fails, the recovery process is similar. If the mirror node fails, the primary node first flushes its CPU caches. It also uses its metadata log to locate un-replicated data and sends them to the backup node. To restart the mirror node or the backup node, Mojim replays the logs and writes only the completed atomic operation content to the data area, with the help of the atomic operation end mark and unique IDs. In the example in Figure 3, the mirror node crashes after Mojim checkpoints G. The recovery process will checkpoint C and discard H. If the failed node cannot restart, a newly chosen node receives replicated data from the primary node as described above.

When both the primary node and the mirror node fail in quick succession, Mojim falls back to the backup node. Now Mojim needs to reconstruct two new nodes that the administrating node selects. This recovery process is more costly than the recovery of a single node failure. We reduce the risk of this situation by placing nodes on different racks and by setting a small SECONDARY_TIER_THRESH, thus speeding up the recovery process of a single node failure.

5. Mojim Applications

We have ported several existing systems to Mojim to illustrate how applications can use Mojim's interface. The applications include the PMFS file system [12], the Google hash table [16], and MongoDB [40].

5.1 PMFS

The Persistent Memory File System [12] (PMFS) provides a conventional file-system-like interface to NVMM, allowing applications to allocate space with file creation, limit access to data via file permissions, and name portions of the

NVMM using file names. The key difference between PMFS and a conventional file system is that its implementation of *mmap* maps the physical pages of NVMM into the applications' address spaces rather than moving them back and forth between the file store and the buffer cache.

PMFS ensures persistence using *sfence* and *clflush* instructions. Mojim invokes its replication when PMFS performs its persistence procedure. Mojim's M-sync also removes *clflush* and only performs *sfence* on the primary node. Mojim's change required modifications to just 20 lines of PMFS source code. Applications can use *mmap* to gain load-/store access to a file's contents and then use *fsync*, *msync*, or *gmsync* to manage replication and data consistency.

5.2 Google hash table

Google hash table [16] is an open source implementation of sparse and dense hash tables. Our Mojim-enabled version of the hash table stores its data in *mmap*'d PMFS files and performs *msync* at each insert and delete operation to let Mojim replicate the data. Porting the Google hash table to Mojim requires changes to just 18 lines of code.

5.3 MongoDB

MongoDB [40] is a popular NoSql database. Several aspects of MongoDB make it a good comparison point for Mojim. First, MongoDB stores its data in memory-mapped files and performs memory loads and stores for data access—a perfect match for Mojim's NVMM interface. Second, MongoDB supports both single node and replication in a set of nodes in several different modes (called "write concerns") that trade off among performance, reliability, and availability. Mojim provides similar functionality with a more general mechanism.

By default, MongoDB logs data in a journal file and checkpoints the data to the memory-mapped data file in a lazy fashion. With the *JOURNALED* write concern, MongoDB blocks a client call until the updated data is written to the journal file. With the *FSYNC_SAFE* write concern, MongoDB flushes all the dirty pages to the data file after each write operation and blocks the client call until this operation completes.

MongoDB supports data replication across a set of machines. A primary node in a MongoDB replica set serves all write requests and pushes *operation logs* to the secondary nodes. Secondary nodes can serve read requests but may return stale data. The MongoDB write concern *REPLICAS_SAFE* returns the client request after at least two secondary nodes have received the corresponding operation log. The REPLICAS_SAFE write concern does not wait for journal writes or checkpointing on the primary node.

Mojim offers another way to provide reliability and availability to MongoDB. With the help of Mojim's *gmsync* API and its reliability guarantees, we can remove journaling from MongoDB and still achieve the same consistency level. To guarantee the same atomicity of client requests as available through MongoDB, we modify the storage engine of MongoDB to keep track of all writes to the data file and group the written memory regions belonging to the same client request into a *gmsync* call. In total, this change requires modifying 117 lines of MongoDB.

An alternative way of using Mojim is to run unmodified MongoDB on Mojim by configuring MongoDB to place both its data file and journal file in Mojim's *mmap*'d data area. When MongoDB commits data to the journal or checkpoints the data to the data file, it performs an *msync* operation, which will trigger Mojim's data replication transparently.

6. Evaluation with DRAM

In this section, we study the performance of Mojim under each of the configurations and applications we described in Sections 3 and 5. Specifically, we first evaluate the performance of different Mojim modes and compare them to existing replication methods. We then evaluate the effects of different application parameters and Mojim configurations, the performance of applications ported to Mojim, and Mojim's recovery costs.

6.1 Test Bed Systems

We use two different systems to evaluate Mojim. The first is an industrial NVMM emulation system from Intel called PMEP [12]. PMEP augments an off-the-shelf, dual-socket server platform with special CPU microcode and custom firmware. It partitions the system's DRAM into emulated NVMM and regular DRAM. PMEP emulates NVMM read latency, read and write bandwidth, and data persistence costs. For read latency and read/write bandwidth, PMEP modifies the CPU and the memory controller. The PMEP platform uses write-back CPU caches and does not emulate NVMM write latency. It uses software to emulate the cost of data persistence: the kernel running on PMEP issues *clflush* instructions followed by an *sfence*, and adds a *write barrier* delay to model the cost of ensuring data persistence in NVMM. In our experiments, we emulate NVMM by setting the read latency to 300 ns, read and write bandwidth to 5 GB/s and 1.6 GB/s (1/8 of DRAM bandwidth), and the write barrier delay to 1 μs, the configuration used in Intel's PMFS project [12].

Each PMEP node has two 2.6 GHz 8-core Intel Xeon processors, 40 MB of aggregate CPU cache, 8 GB of DDR3 DRAM used as normal DRAM, 128 GB of DRAM used as emulated NVMM, and a 7200 RPM 4 TB hard disk. They also have 40 Gbps Mellanox Infiniband NICs and are directly connected to each other via Infiniband without a switch. The platforms run Ubuntu 13.10 and the 3.11.0 Linux kernel.

We have access to only two PMEP machines (located at an Intel facility), so to evaluate Mojim modes that require more than two machines, we use similar machines in our

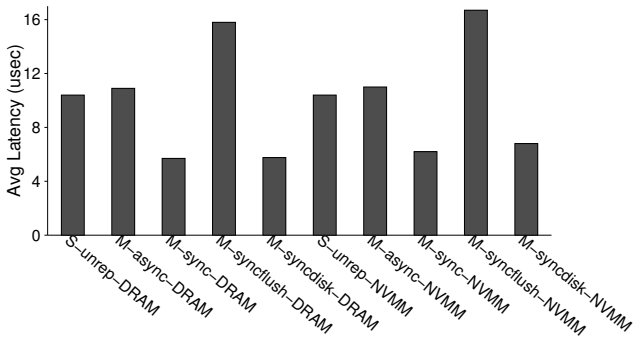

Figure 4. msync Latency with DRAM and NVMM. *The average 4 KB* msync *latency with PMEP's DRAM and NVMM modes.*

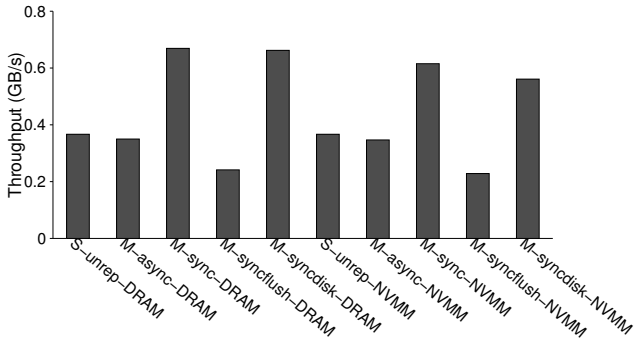

Figure 5. msync Throughput with DRAM and NVMM. *The 4 KB* msync *bandwidth with PMEP's DRAM and NVMM modes.*

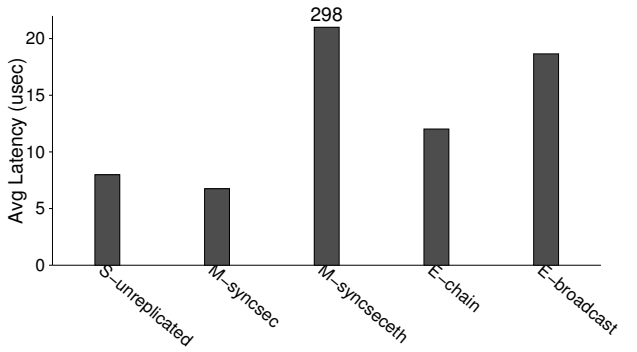

Figure 6. msync Latency with DRAM-based machines. *The average 4 KB* msync *latency with S-unreplicated and Mojim two-tier architecture.*

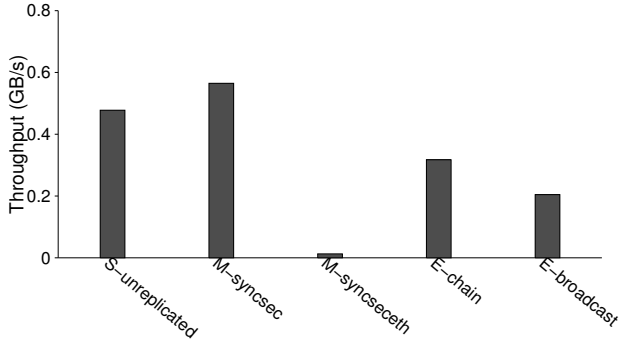

Figure 7. msync Throughput with DRAM-based machines. *The 4 KB* msync *throughput with S-unreplicated and Mojim two-tier architecture.*

lab that do not include PMEP functionality and use ordinary DRAM as a proxy for NVMM. Each of these machines has two Intel Xeon X5647 processors, 48 GB DRAM, one 40 Gbps Mellanox Infiniband NIC, and a 1000 Mbps Ethernet. A QLogic Infiniband Switch connects these machines' IB links. All machines run the CentOS 6.4 distribution and the 3.11.0 Linux kernel.

In all experiments, unless otherwise specified, we set CHECKPOINT_THRESH (the frequency of checkpointing the mirror node logs) to 1 (after each log write) and SECONDARY_TIER_THRESH (the threshold for sending unreplicated data to the backup nodes) to 40 MB.

6.2 Overall Replication Performance

We first compare the microbenchmark performance of Mojim modes that only involve two nodes using the PMEP platforms. To evaluate the impact of NVMM vs. DRAM, we run the same experiments with both PMEP's DRAM mode and its emulated NVMM.

Figures 4 and 5 present the average latency and throughput of *msync* calls with S-unreplicated, M-async, M-sync, M-syncflush, and M-syncdisk. For each experiment, we perform 10000 random 4 KB *msync* calls in a 4 GB *mmap*'d file.

Surprisingly, M-sync outperforms S-unreplicated significantly for both DRAM and emulated NVMM (reducing latency by 45% and 40% respectively). Even though M-sync waits for a networking round trip between the primary node and the mirror node, it still outperforms S-unreplicated because it does not need to flush data from processors' caches, while S-unreplicated must flush data on each *msync*. M-async's performance is similar to S-unreplicated, as it also needs to flush primary node's caches. M-syncflush has higher latency than S-unreplicated, since it performs both cache flushes and networking round trips.

Placing the mirror node data on disk adds only 1% to 10% overhead. However, M-syncdisk does not support read applications on the mirror node and adds an overhead in recovery time (see Section 6.5).

Comparing DRAM and emulated NVMM, the performance with emulated NVMM for all schemes is close to that with DRAM, indicating that the performance degradation of NVMM over DRAM only has a very small effect over application-level performance.

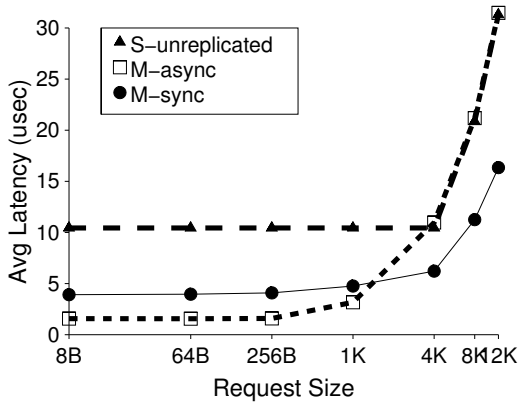

Figure 8. Average *msync* Latency with Different *msync* Sizes on Emulated NVMM. *The average latency of* msync *operation on NVMM with request sizes from 8 bytes to 12 KB.*

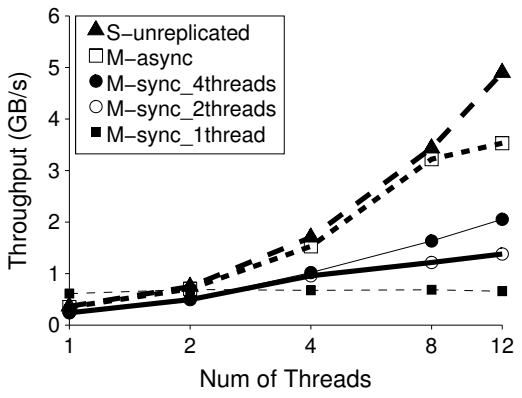

Figure 9. Throughput with Different Application Threads on Emulated NVMM. *The* msync *throughput with 1 to 12 threads performing* msync*.*

Next, to augment the PMEP results with more machines and to test Mojim's two-tier architecture, we use three DRAM-based machines in our lab to evaluate the performance of Mojim's two-tier modes and two existing schemes that use a one-primary, multiple-secondary architecture (Table 1). One of these existing schemes, *E-chain*, allows writes only at the primary node and propagates data replication from the primary node to the secondary nodes in a serialized order [2, 56]. The other existing scheme, *E-broadcast* [15], is similar to E-chain but broadcasts updates to the secondary nodes. E-chain and E-broadcast use one primary node and two secondary nodes interconnected by IB. They use the same IB protocol that we implemented for Mojim.

Figures 6 and 7 plot the average latency and throughput of using our lab machines to run the experiments shown in Figures 4 and 5. Compared to S-unreplicated, Mojim with the secondary tier does not degrade performance if a fast network connects the backup node. However, the lower-cost Ethernet configuration degrades performance by 37×, because the mirror node cannot drain its circular log fast enough and has to stall the primary tier replication.

Both E-chain and E-broadcast are slower than Mojim, increasing latency by 1.8× and 2.8× respectively, compared to M-syncsec.

Finally, we compare Mojim with two existing IB kernel protocols, RDS and IPoIB. We find that they both have worse performance than Mojim's networking protocol on IB-Verbs, with 4.9× and 31× slowdown.

Overall, Mojim delivers performance similar to or better than no replication while adding reliability and availability. Mojim's good performance is due to its efficient replication protocol, its ability to avoid expensive cache flush operations, and its optimized software and networking stacks.

6.3 Sensitivity Analysis

Both Mojim's configuration parameters and application-level behavior can affect performance. In this section, we measure their impact on Mojim's performance.

6.3.1 *msync* Size

The amount of data per *msync* has a strong impact on performance. Figure 8 plots the average latency of performing *msync* calls to random memory regions of 8 bytes to 12 KB with S-unreplicated, M-async, and M-sync using PMEP's emulated NVMM.

For smaller request sizes, M-async performs much better than S-unreplicated. S-unreplicated underperforms because of the current way *msync* call are implemented in Linux, with the *msync* call handler checking for the range of the *msync* memory and rounding it to memory pages (4 KB page for the default Linux kernel). With Mojim, we modify the *msync* call handler to allow any memory address range and only flush and replicate the application-specified memory regions.

M-sync does not perform *clflush* (since transferring the data to the mirror node guarantees persistence). As a result, its performance is always better than S-unreplicated and is better than M-async when *msync* size is bigger than 1 KB.

6.3.2 Application Threads and Networking Connections

Application thread count and the number of network connections Mojim uses also impact performance. Figure 9 presents the 4 KB *msync* throughput with one to 12 application threads for S-unreplicated, M-async, and M-sync using PMEP's emulated NVMM.

Both M-async and S-unreplicated scale well with the number of application threads, while M-sync with one IB connection (and thus one log) scales poorly. With more con-

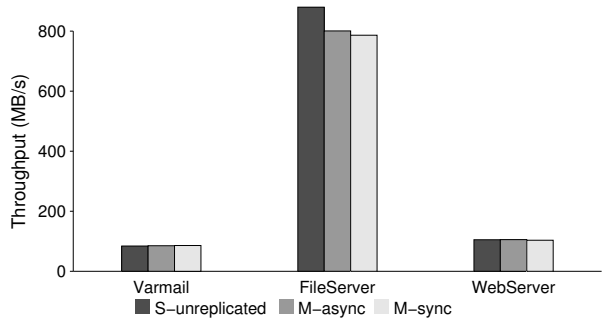

Figure 10. Filebench Throughput with Emulated NVMM. *The throughput of three Filebench workloads with single machine and no replication, the M-async mode, and the M-sync mode.*

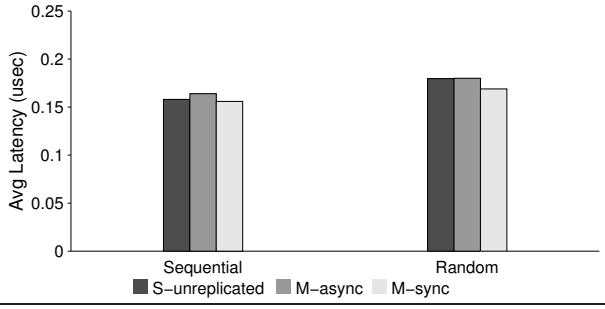

Figure 11. Google Hash Table Average Latency with Emulated NVMM. *The average latency of sequentially and randomly inserting key-value pairs to the Google dense hash table.*

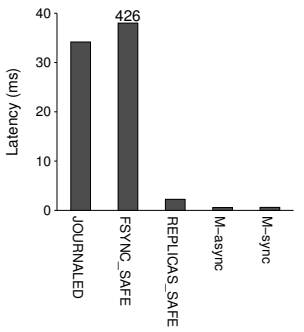

Figure 12. Insert Avg Latency. *Average latency of inserting key-value pairs on emulated NVMM.*

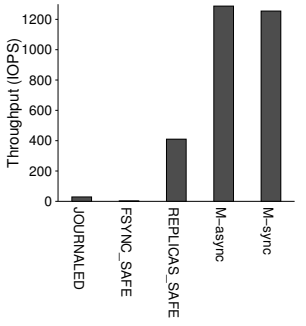

Figure 13. Insert Throughput. *Throughput of inserting key-value pairs on emulated NVMM.*

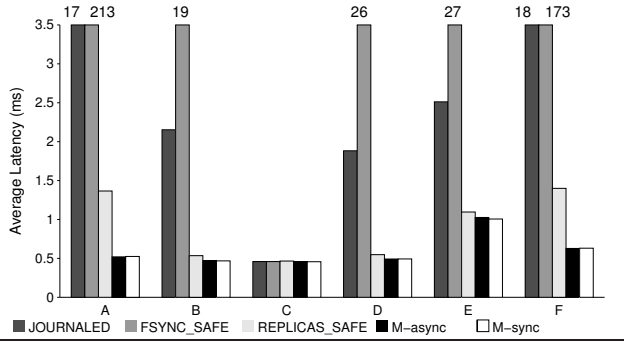

Figure 14. YCSB Average Latency. *Average latency of YCSB workloads on emulated NVMM.*

nections, M-sync's scaling improves. A tradeoff with increasing networking connections is that Mojim uses more threads to poll for receiving messages, consuming more CPU cycles.

6.3.3 Checkpoint and Secondary Tier Replication Thresholds

We change the two thresholds Mojim uses in its configurations: we change CHECKPOINT_THRESH, the frequency of checkpointing the mirror node logs, from 1 to 10000, and we change SECONDARY_TIER_THRESH, the amount of un-replicated data to the backup node, from 40 KB to 400 MB. We find that neither CHECKPOINT_THRESH nor SECONDARY_TIER_THRESH affects the application performance, because both the checkpointing process and the secondary tier replication via IB are fast enough not to block the primary tier replication.

6.4 Application Performance

In this section, we present the evaluation results for three applications: a file system, a hash table, and a NoSql database.

6.4.1 PMFS

We use the FileServer, WebServer, and VarMail workloads in the Filebench suite [54] to evaluate different Mojim modes under PMFS using emulated NVMM. Figure 10 presents the throughput of the three workloads of Filebench. For WebServer and Varmail, both M-async and M-sync yield performance similar to S-unreplicated. For FileServer, M-async and M-sync have slightly worse performance than S-unreplicated.

6.4.2 Google hash table

We perform sequential and random key-value insertion to the Google Dense Hash Table [16]. Each key-value pair contains an integer key and a random integer value. Figure 11 plots the average latency of S-unreplicated, M-async, and M-sync with emulated NVMM. For both workloads, all three schemes have similar performance, showing that Mojim has small performance overhead when it comes to hash table operations.

Workload	Read	Update	Scan	Insert	Read&Update
A	50	50	-	-	-
B	95	5	-	-	-
C	100	-	-	-	-
D	95	-	-	5	-
E	-	-	95	5	-
F	50	-	-	-	50

Table 2. YCSB Workload Properties. *The percentage of different operations in each YCSB workload.*

6.4.3 MongoDB

MongoDB is a natural fit for Mojim. We evaluate how MongoDB and Mojim compare using micro- and macro-benchmarks.

Microbenchmark: Our microbenchmark inserts key-value pairs to MongoDB. Each insert operation contains 10 key-value pairs, with each pair containing 100 bytes of randomly generated data. Figures 12 and 13 present the average latency and throughput of key-value pair insertions with PMEP's emulated NVMM. We set the MongoDB replication method to use two replicas (the primary node and the secondary node) and connect these nodes with IB.

MongoDB with Mojim outperforms the MongoDB replication method REPLICAS_SAFE by 3.7 to 3.9×. This performance gain is due to Mojim's efficient replication protocol and networking stack.

Mojim also outperforms the un-replicated JOURNALED MongoDB by 56 to 59× and the un-replicated FSYNC_SAFE by 701 to 741×. JOURNALED flushes journal content for each client write request. FSYNC_SAFE performs *fsync* of the data file after each write operation to guarantee data reliability without journaling. Both these operations are expensive.

To evaluate Mojim's two-tier architecture with MongoDB, we perform the same set of experiments using three DRAM-based machines in our lab. Similar to the PMEP results, Mojim's M-syncsec outperforms MongoDB's replication method by 3.4 to 4×, the un-replicated JOURNALED MongoDB by 35 to 43×, and the un-replicated FSYNC_SAFE by 238 to 311×, suggesting that Mojim's replication is better than MongoDB replication.

Finally, MongoDB can run unmodified on Mojim by configuring both its journal and data file to be in a *mmap*'d NVMM region. In this case, its performance is similar to JOURNALED, with a performance overhead of 0.2% to 6%. However, Mojim provides better reliability and availability than the un-replicated MongoDB.

Macrobenchmark: YCSB [10] is a benchmark designed to evaluate key-value store systems. YCSB includes six workloads that imitate web applications' data access models. The workloads contain a combination of read, update, scan, and insert operations. Table 2 summarizes the number of these operations in the YCSB workloads. Each workload performs 1000 operations on a database with 1000 1 KB records.

Figure 14 presents the latency of MongoDB and Mojim using the six YCSB workloads on emulated NVMM. For most workloads, both M-async and M-sync outperform the un-replicated and replicated MongoDB schemes. The performance improvement is especially high for write-heavy workloads. We also find similar results with three DRAM-based machines.

6.5 Recovery

Recovery performance is important because it directly affects availability and may impact reliability, since Mojim is vulnerable to additional node failures during some recovery scenarios. To test the robustness of the system, we stop a Mojim node at random and find that the rest of the system can continue serving client requests correctly. We further measure the recovery time in the event of a node failure.

We use a typical recovery scenario to illustrate Mojim's recovery performance. When a mirror node fails with M-syncsec, the recovery process requires sending the remaining, un-replicated data to the backup node, flushing the CPU caches on the primary node, and copying all the data areas to the new mirror node. Mojim performs these operations in parallel. We set SECONDARY_TIER_THRESH to 40 MB and use three machines in our lab to perform the recovery performance evaluation.

Mojim takes 450 μs to flush the 26 MB CPU caches on the primary node. Before the primary node flushes all its caches, if it also fails, there will be data loss. The window of vulnerability also depends on how soon the failure can be detected, thus in practice it will be longer than 450 μs [7]. It takes 14 ms to send 40 MB of data to the backup node and 1.9 seconds to send a 5 GB data area to the new mirror node. The whole recovery process completes in 1.9 seconds for a 5 GB NVMM. Even for a 1 TB NVMM, the recovery process will only take 6.5 minutes. Notice that the vulnerability window depends on how fast primary node detects a failure and flushes its caches, not on NVMM size.

For M-syncdisk, Mojim also needs to read the data file from the disk to the NVMM before applications can access the data. In this case, recovery takes 17 seconds for a 5 GB data file, a much higher cost in availability than when we use NVMM for the data area.

7. Related Work

This section places Mojim in context with other related research projects and systems.

Non-Volatile Main Memory: Recent years have seen increased interest in NVMM. Researchers have focused on NVMM-related problems, such as building NVMM file systems [9, 12, 59], hybrid DRAM/NVMM memory systems [39], memory allocators [41], memory management and paging mechanisms [3], and programming models [8, 58]. While previous research has focused on un-replicated NVMM in a single machine, Mojim focuses on providing

reliability, high availability, and redundancy to NVMM in data center environments.

Redundancy and Replication: To provide data reliability and availability, many systems use data redundancy or replication [1, 5, 11, 15, 17, 18, 29, 46, 47, 51, 52, 56, 61]. Several previous systems adopt the architecture of one primary and multiple backups [2, 15, 56]. These systems either use a total ordering of node to serialize data replication [2, 56] or broadcast the replicated data and have the primary wait for all the backups [15] to guarantee strong consistency. Mojim uses a two-tier architecture containing one primary node, one mirror node, and multiple backup nodes. Mojim's replication between the primary node and mirror node and the background replication to the backup nodes is more efficient than replication among one primary and multiple backups.

Other architectures allow writes to all replicas (E-writeall) [11, 29, 47, 55]. Systems that require strong consistency among the replicas use Paxos-like protocols [4, 22, 31]. With such architectures, at least two networking round trips are needed to deliver strong consistency. Moreover, the round trips are necessary at each write (memory store) rather than for the less frequent sync points. In contrast, Mojim only replicates data at sync points. Ensuring strong consistency for the Mojim primary tier is also much simpler and has much smaller performance cost. There are also systems that implement weak consistency protocols for the E-writeall architecture [11, 29, 47]. These systems require a reconciliation process for conflicts, which can increase performance overhead [17]. Mojim does not involve any reconciliation and supports strong consistency in its primary tier.

Mojim's architecture is similar to the two-tier architecture proposed [17], which reduced the locking and reconciliation overhead in mobile, disconnected environments. However, we focus on data center environments where nodes are mostly connected. Moreover, Mojim's primary tier only uses one primary and one mirror node for good performance.

To reduce system downtime, commercial storage systems often maintain a pair of interconnected nodes (called high-availability pairs) [13, 19, 43, 57]. When one node fails, the other member of the pair takes over its duties. Most of these high-availability pair schemes rely on shared storage and do not replicate storage data, whereas Mojim replicates data in NVMM. Moreover, Mojim can also provide additional redundancy with its secondary tier.

Finally, RAMCloud is a low-latency key-value store that keeps all data in DRAM [44]. While Mojim and RAMCloud both provide reliable memory-based storage systems, RAMCloud provides a key-value interface rather than a memory-like interface to applications. The key-value software layer adds significant latency to accesses and obscures much of the performance of the underlying memory. Mojim offers a memory-based interface (*i.e.*, applications access Mojim using memory load and store instructions). Based on the interface it supports and its performance targets, Mojim makes design decisions differently from RAMCloud. Mojim replicates NVMM to NVMM at application sync points, while RAMCloud replicates key-value contents on each put operation to slow storage devices. Mojim does not shard memory, because sharding would result in portions of applications' NVMM being on a remote node, vastly increasing latency. As a result, Mojim uses fail-over to recover from failures instead of RAMCloud's approach that relies on sharded data storage to achieve fast recovery.

8. Conclusions

We have described Mojim, a system for providing reliable and highly-available NVMM. Mojim uses a two-tier architecture and efficiently replicates data in NVMM. Our results demonstrate that Mojim can provide replication with small cost, in many cases even outperforming the un-replicated system. In doing so, Mojim paves the way for deploying NVMM in data centers that wish to take advantage of NVMM's enhanced performance but require strong guarantees about data safety.

Acknowledgments

We thank the anonymous reviewers for their enormously valuable feedback and comments, which have substantially improved the content and presentation of this paper. We also thank Dulloor Subramanya, Jeff Jackson, and the vLab team from Intel Corp. for their help with the PMEP platforms. Finally, we thank the members of the NVSL research group for their insightful comments.

This work was supported in part by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of C-FAR or other institutions.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[2] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*, San Francisco, California, October 1976.

[3] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systemsi (HotOS '13)*, Napa, California, May 2011.

[4] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[5] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[6] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*, Hong Kong, China, August 2013.

[7] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, California, May 2006.

[8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, New York, New York, March 2011.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Doug Burger, Benjamin C. Lee, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, New York, New York, June 2010.

[11] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[12] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the EuroSys Conference (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[13] EMC Corporation. EMC VNXe High Availability. `https://www.emc.com/collateral/hardware/white-papers/h8276-emc-vnxe-high-availability-wp.pdf`.

[14] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[16] Google Inc. Google Sparse Hash. `http://goog-sparsehash.sourceforge.net`.

[17] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*, New York, New York, June 1996.

[18] Lisa Hellerstein, Garth A. Gibson, Richard M. Karp, Randy H. Katz, and David A. Patterson. Coding Techniques for Handling Failures in Large Disk Arrays. *Algorithmica*, 12(2):182–208, August 1994.

[19] Hewlett Packard. HP NonStop operating system. `http://h17007.www1.hp.com/us/en/enterprise/servers/integrity/nonstop/nonstop-os.aspx`.

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

[21] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proceedings*

of the USENIX Annual Technical Conference (USENIX '12), Boston, Massachusetts, June 2012.

[22] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, Massachusetts, June 2010.

[23] Intel. Add Support for New Persistent Memory Instructions. `http://www.lwn.net/Articles/619851`.

[24] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[25] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, Pittsburgh, Pennsylvania, March 2010.

[26] James Pinkerton. The Future of Computing: The Convergence of Memory and Storage through Non-Volatile Memory (NVM). Storage Industry Summit, San Jose, California, Jan 2014.

[27] Brian G Johnson and Charles H Dennison. Phase change memory, September 2004. US Patent 6,791,102.

[28] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*, Providence, Rhode Island, May 2003.

[29] John Kubiatowicz, David Bindel, Patrick Eaton, Yan Chen, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Westley Weimer, Chris Wells, Hakim Weatherspoon, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, Cambridge, Massachusetts, November 2000.

[30] Amit Kumar and Ram Huggahalli. Impact of cache coherence protocols on the processing of network traffic. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '07)*, Chicago, Illinois, Dec 2007.

[31] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 32(4):18–25, November 2001.

[32] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Austin, Texas, June 2009.

[33] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Phase change memory architecture and the quest for scalability. *Commun. ACM*, 53(7):99–106, 2010.

[34] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE micro*, 30(1):143, 2010.

[35] Myoung-Jae Lee, Chang Bum Lee, Dongsoo Lee, Seung Ryul Lee, Man Chang, Ji Hyun Hur, Young-Bae Kim, Chang-Jung Kim, David H Seo, Sunae Seo, et al. A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta2o5- x/tao2- x bilayer structures. *Nature materials*, 10(8):625–630, 2011.

[36] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[37] Mellanox Technologies. Rdma aware networks programming user manual. `http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf`.

[38] Micron Technology Inc. P8p parallel phase change memory (pcm). `http://www.micron.com/~/media/Documents/Products/Data%20Sheet/PCM/p8p_parallel_pcm_ds.pdf`.

[39] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm+dram hybrid main memory. In *The Twelfth Workshop on Hot Topics in Operating Systems (HotOS XII)*, Monte Verita, Switzerland, May 2009.

[40] MongoDB Inc. MongoDB. `http://www.mongodb.org/`.

[41] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Conference on Timely Results in Operating Systems (TRIOS '13)*, Farmington, Pennsylvania, November 2013.

[42] Suman Nath, Haifeng Yu, Philip B. Gibbons, and Srinivasan Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, California, May 2006.

[43] NetApp Inc. NetApp SnapMirror Data Replication. `http://www.netapp.com/us/products/protection-software/snapmirror.aspx`.

[44] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[45] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync (): a simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the EuroSys Conference (EuroSys '13)*, Prague, Czech Republic, April 2013.

[46] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceed-*

ings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88), Chicago, Illinois, June 1988.

[47] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, October 1997.

[48] Moinuddin K Qureshi, Michele M Franceschini, Luis A Lastras-Montaño, and John P Karidis. Morphable memory system: a robust architecture for exploiting multi-level phase change memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '07)*, June 2010.

[49] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude 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)*, Austin, Texas, June 2009.

[50] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*, Tucson, Arizona, 2011.

[51] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[52] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[53] David Spence, Jon Crowcroft, Steven Hand, and Tim Harris. Location based placement of whole distributed systems. In *Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Technology (CoNEXT '05)*, Toulouse, France, October 2005.

[54] Sun Microsystems. Solaris Internals: FileBench. `http://filebench.sourceforge.net/`.

[55] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, November 2013.

[56] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[57] VMWare Inc. VMware High Availability. `http://www.vmware.com/files/pdf/VMware-High-Availability-DS-EN.pdf`.

[58] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, New York, New York, March 2011.

[59] Xiaojian Wu and A.L.N. Reddy. Scmfs: A file system for storage class memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, Nov 2011.

[60] J Joshua Yang, Dmitri B Strukov, and Duncan R Stewart. Memristive devices for computing. *Nature nanotechnology*, 8(1):13–24, 2013.

[61] Ming Zhong, Kai Shen, and Joel Seiferas. Replication degree customization for high availability. In *Proceedings of the EuroSys Conference (EuroSys '08)*, Glasgow, Scotland UK, March 2008.

[62] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Austin, Texas, June 2009.