Distributed Shared Persistent Memory

Yizhou Shan
Purdue University
shan13@purdue.edu

Shin-Yeh Tsai
Purdue University
tsai46@purdue.edu

Yiying Zhang
Purdue University
yiying@purdue.edu

ABSTRACT

Next-generation non-volatile memories (*NVMs*) will provide byte addressability, persistence, high density, and DRAM-like performance. They have the potential to benefit many datacenter applications. However, most previous research on NVMs has focused on using them in a single machine environment. It is still unclear how to best utilize them in distributed, datacenter environments.

We introduce *Distributed Shared Persistent Memory (DSPM)*, a new framework for using persistent memories in distributed datacenter environments. DSPM provides a new abstraction that allows applications to both perform traditional memory load and store instructions and to name, share, and persist their data.

We built *Hotpot*, a kernel-level DSPM system that provides low-latency, transparent memory accesses, data persistence, data reliability, and high availability. The key ideas of Hotpot are to integrate distributed memory caching and data replication techniques and to exploit application hints. We implemented Hotpot in the Linux kernel and demonstrated its benefits by building a distributed graph engine on Hotpot and porting a NoSQL database to Hotpot. Our evaluation shows that Hotpot outperforms a recent distributed shared memory system by 1.3× to 3.2× and a recent distributed PM-based file system by 1.5× to 3.0×.

CCS CONCEPTS

• Software and its engineering → Distributed memory; **Distributed systems organizing principles**; **• Computer systems organization →** *Reliability*; *Availability*;

KEYWORDS

Persistent Memory, Distributed Shared Memory

ACM Reference Format:
Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 15 pages.
https://doi.org/10.1145/3127479.3128610

1 INTRODUCTION

Next-generation non-volatile memories (*NVMs*), such as 3DX-point [36], phase change memory (*PCM*), spin-transfer torque magnetic memories (*STTMs*), and the memristor will provide byte addressability, persistence, high density, and DRAM-like performance [85]. These developments are poised to radically alter the landscape of memory and storage technologies and have already inspired a host of research projects [5, 15, 16, 22, 57, 62, 88, 89, 95]. However, most previous research on NVMs has focused on using them in a single machine environment. Even though NVMs have the potential to greatly improve the performance and reliability of large-scale applications, it is still unclear how to best utilize them in distributed, datacenter environments.

This paper takes a significant step towards the goal of using NVMs in distributed datacenter environments. We propose *Distributed Shared Persistent Memory (DSPM)*, a framework that provides a global, shared, and persistent memory space using a pool of machines with NVMs attached at the main memory bus. Applications can perform native memory load and store instructions to access both local and remote data in this global memory space and can at the same time make their data persistent and reliable. DSPM can benefit both single-node persistent-data applications that want to scale out efficiently and shared-memory applications that want to add durability to their data.

Unlike traditional systems with separate memory and storage layers [23, 24, 80, 81], we propose to use just one layer that incorporates both distributed memory and distributed storage in DSPM. DSPM's one-layer approach eliminates the performance overhead of data marshaling and unmarshaling, and the space overhead of storing data twice. With this one-layer approach, DSPM can potentially provide the low-latency performance, vast persistent memory space, data reliability, and high availability that many modern datacenter applications demand.

Building a DSPM system presents its unique challenges. Adding "Persistence" to Distributed Shared Memory (DSM) is not as simple as just making in-memory data durable. Apart from data durability, DSPM needs to provide two key features that DSM does not have: persistent naming and data reliability. In addition to accessing data in PM via native memory loads and stores, applications should be able to easily name, close, and re-open their in-memory data structures. User data should also be reliably stored in NVM and sustain various types of failures; they need to be consistent both within a node and across distributed nodes after crashes. To make it more challenging, DSPM has to deliver these guarantees without sacrificing application performance in order to preserve the low-latency performance of NVMs.

We built *Hotpot*, a DSPM system in the Linux kernel. Hotpot offers low-latency, direct memory access, data persistence, reliability, and high availability to datacenter applications. It exposes a global virtual memory address space to each user application and provides a new persistent naming mechanism that is both easy-to-use and efficient. Internally, Hotpot organizes and manages data in a flexible

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '17, September 24–27, 2017, Santa Clara, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3128610

way and uses a set of adaptive resource management techniques to improve performance and scalability.

Hotpot builds on two main ideas to efficiently provide data reliability with distributed shared memory access. Our first idea is to integrate distributed memory caching and data replication by imposing *morphable* states on persistent memory (*PM*) pages.

In DSM systems, when an application on a node accesses shared data in remote memory *on demand*, DSM caches these data copies in its local memory for fast accesses and later evicts them when reclaiming local memory space. Like DSM, Hotpot caches application-accessed data in local PM and ensures the coherence of multiple cached copies on different nodes. But Hotpot also uses these cached data as *persistent data replicas* and ensures their reliability and crash consistency.

On the other hand, unlike distributed storage systems, which *creates* extra data replicas to meet user-specified reliability requirements, Hotpot makes use of data copies that *already exist* in the system when they were fetched to a local node due to application memory accesses.

In essence, every local copy of data serves two simultaneous purposes. First, applications can access it locally without any network delay. Second, by placing the fetched copy in PM, it can be treated as a persistent replica for data reliability.

This seemingly-straightforward integration is not simple. Maintaining wrong or outdated versions of data can result in inconsistent data. To make it worse, these inconsistent data will be persistent in PM. We carefully designed a set of protocols to deliver data reliability and crash consistency guarantees while integrating memory caching and data replication.

Our second idea is to exploit application behaviors and intentions in the DSPM setting. Unlike traditional memory-based applications, persistent-data-based applications, DSPM's targeted type of application, have well-defined data *commit points* where they specify what data they want to make persistent. When a process in such an application makes data persistent, it usually implies that the data can be *visible* outside the process (*e.g.*, to other processes or other nodes). Hotpot utilizes these data commit points to also push updates to cached copies on distributed nodes to avoid maintaining coherence on every PM write. Doing so greatly improves the performance of Hotpot, while still ensuring correct memory sharing and data reliability.

To demonstrate the benefits of Hotpot, we ported the MongoDB [63] NoSQL database to Hotpot and built a distributed graph engine based on PowerGraph [28] on Hotpot. Our MongoDB evaluation results show that Hotpot outperforms a PM-based replication system [95] by up to 3.1×, a recent PM-based distributed file systems [57] by up to 3.0×, and a DRAM-based file system by up to 53×. Hotpot outperforms PowerGraph by 2.3× to 5×, and a recent DSM system [65] by 1.3× to 3.2×. Moreover, Hotpot delivers stronger data reliability and availability guarantees than these alternative systems.

Overall, this paper makes the following key contributions:

- We are the first to introduce the Distributed Shared Persistent Memory (DSPM) model and among the first to build distributed PM-based systems. The DSPM model provides direct and shared memory accesses to a distributed set of PMs and is an easy and efficient way for datacenter applications to use PM.
- We propose a one-layer approach to build DSPM by integrating memory coherence and data replication. The one-layer approach avoids the performance cost of two or more indirection layers.
- We designed two distributed data commit protocols with different consistency levels and corresponding recovery protocols to ensure data durability, reliability, and availability.
- We built the first DSPM system, Hotpot, in the Linux kernel, and open source it together with several applications ported to it.
- We demonstrated Hotpot's performance benefits and ease of use with two real datacenter applications and extensive microbenchmark evaluation. We compared Hotpot with five existing file systems and distributed memory systems.

The rest of the paper is organized as follows. Section 2 presents and analyzes several recent datacenter trends that motivated our design of DSPM. We discuss the benefits and challenges of DSPM in Section 3. Section 4 presents the architecture and abstraction of Hotpot. We then discuss Hotpot's data management in Section 5. We present our protocols and mechanisms to ensure data durability, consistency, reliability, and availability in Section 6. Section 7 briefly discusses the network layer we built underlying Hotpot, and Section 8 presents detailed evaluation of Hotpot. We cover related work in Section 9 and conclude in Section 10.

2 MOTIVATION

DSPM is motivated by three datacenter trends: emerging hardware PM technologies, modern data-intensive applications' data sharing, persistence, and reliability needs, and the availability of fast datacenter network.

2.1 Persistent Memory and PM Apps

Next-generation non-volatile memories (*NVMs*), such as 3DX-point [36], phase change memory (*PCM*), spin-transfer torque magnetic memories (*STTMs*), and the memristor will provide byte addressability, persistence, and latency that is within an order of magnitude of DRAM [33, 50–52, 57, 74, 85, 90]. These developments are poised to radically alter the landscape of memory and storage technologies.

NVMs can attach directly to the main memory bus to form Persistent Memory, or PM. If applications want to exploit all the low latency and byte-addressability benefits of PM, they should directly access it via memory load and store instructions without any software overheads [15, 41, 61, 70, 88, 95] (we call this model *durable in-memory computation*), rather than accessing it via a file system [16, 21, 22, 57, 69, 89].

Unfortunately, most previous durable in-memory systems were designed for the single-node environment. With modern datacenter applications' computation scale, we have to be able to scale out these single-node PM systems.

2.2 Shared Memory Applications

Modern data-intensive applications increasingly need to access and share vast amounts of data fast. We use PIN [58] to collect

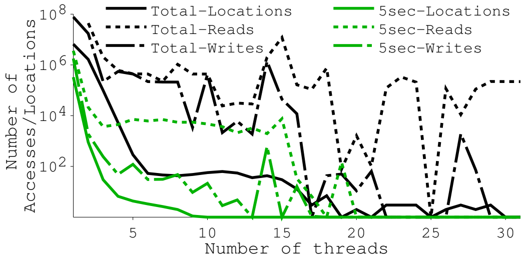


Figure 1: PowerGraph Sharing Analysis. *Results of running PageRank [49] on a Twitter graph [46]. Black lines represent total amount of sharing. Green lines represent sharing within five seconds.*

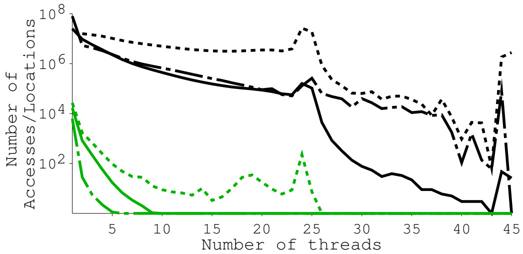


Figure 2: Tensorflow Sharing Analysis. *Results of running a handwriting recognition workloads provided by TensorFlow. Black lines represent total amount of sharing. Green lines represent sharing within five seconds.*

memory access traces of two popular data-intensive applications, TensorFlow [2] and PowerGraph [28]. Figures 1 and 2 show the total number of reads and writes performed to the same memory location by N threads and the amount of these shared locations. There are a significant amount of shared read accesses in these applications, especially across a small set of threads. We further divided the memory traces into smaller time windows and found that there is still a significant amount of sharing, indicating that many shared accesses occur at similar times.

Distributed Shared Memory (*DSM*) takes the shared memory concept a step further by organizing a pool of machines into a globally shared memory space. Researchers and system builders have developed a host of software and hardware DSM systems in the past few decades [6, 8, 9, 19, 25, 27, 39, 40, 43, 54, 76, 79, 82, 83, 96, 97]. Recently, there is a new interest in DSM [65] to support modern data-intensive applications.

However, although DSM scales out shared-memory applications, there has been no persistent-memory support for DSM. DSM systems all had to checkpoint to disks [66, 77, 84]. Memory persistence can allow these applications to checkpoint fast and recover fast [64].

2.3 Fast Network and RDMA

Datacenter network performance has improved significantly over the past decades. InfiniBand (*IB*) NICs and switches support high bandwidth ranging from 40 to 100 Gbps. Remote Direct Memory Access (*RDMA*) technologies that provide low-latency remote memory accesses have become more mature for datacenter uses in recent years [21, 32, 37, 38]. These network technology advances make remote-memory-based systems [7, 12, 31, 65, 72, 92] more attractive than decades ago.

2.4 Lack of Distributed PM Support

Many large-scale datacenter applications require fast access to vast amounts of persistent data and could benefit from PM's performance, durability, and capacity benefits. For PMs to be successful in datacenter environments, they have to support these applications. However, neither traditional distributed storage systems or DSM systems are designed for PM. Traditional distributed storage systems [3, 11, 18, 26, 45, 73] target slower, block-based storage devices. Using them on PMs will result in excessive software and network overheads that outstrip PM's low latency performance [95]. DSM systems were designed for fast, byte-addressable memory, but lack the support for data durability and reliability.

Octopus [57] is a recent RDMA-enabled distributed file system built for PM. Octopus and our previous work Mojim [95] are the only distributed PM-based systems that we are aware of. Octopus was developed in parallel with Hotpot and has a similar goal as Hotpot: to manage and expose distributed PM to datacenter applications. However, Octopus uses a file system abstraction and is built in the user level. These designs add significant performance overhead to native PM accesses (Section 8.2). Moreover, Octopus does not provide any data reliability or high availability, both of which are key requirements in datacenter environments.

3 DISTRIBUTED SHARED PERSISTENT MEMORY

The datacenter application and hardware trends described in Section 2 clearly point to one promising direction of using PM in datacenter environments — as distributed, shared, persistent memory (DSPM). A DSPM system manages a distributed set of PM-equipped machines and provides the abstraction of a global virtual address space and a data persistence interface to applications. This section gives a brief discussion on the DSPM model.

3.1 DSPM Benefits and Usage Scenarios

DSPM offers low-latency, shared access to vast amount of durable data in distributed PM, and the reliability and high availability of these data. Application developers can build in-memory data structures with the global virtual address space and decide how to name their data and when to make data persistent.

Applications that fit DSPM well have two properties: accessing data with memory instructions and making data durable explicitly. We call the time when an application makes its data persistent a *commit point*. There are several types of datacenter applications that meet the above two descriptions and can benefit from running on DSPM.

First, applications that are built for single-node PM can be easily ported to DSPM and scale out to distributed environments. These applications store persistent data as in-memory data structures and already express their commit points explicitly. Similarly, storage applications that use memory-mapped files also fit DSPM well, since they operate on in-memory data and explicitly make them persistent at well-defined commit points (*i.e.*, *msync*). Finally, DSPM fits shared-memory or DSM-based applications that desire to incorporate durability. These applications do not yet have durable data commit points, but we expect that when developers want to make their applications durable, they should have the knowledge of when and what data they want make durable.

3.2 DSPM Challenges

Building a DSPM system presents several new challenges.

First, *what type of abstraction should DSPM offer to support both direct memory accesses and data persistence (Section 4)*? To perform native memory accesses, application processes should use virtual memory addresses. But virtual memory addresses are not a good way to *name* persistent data. DSPM needs a naming mechanism that applications can easily use to retrieve their in-memory data after reboot or crashes (Section 4.2). Allowing direct memory accesses to DSPM also brings another new problem: pointers need to be both persistent in PM and consistent across machines (Section 4.3).

Second, *how to efficiently organize data in DSPM to deliver good application performance (Section 5)?* To make DSPM's interface easy to use and transparent, DSPM should manage the physical PM space for applications and handle PM allocation. DSPM needs a flexible and efficient data management mechanism to deliver good performance to different types of applications.

Finally, *DSPM needs to ensure both distributed cache coherence and data reliability at the same time* (Section 6). The former requirement ensures the coherence of multiple cached copies at different machines under concurrent accesses and is usually enforced in a distributed memory layer. The latter provides data reliability and availability when crashes happen and is implemented in distributed storage systems or distributed databases. DSPM needs to incorporate both these two different requirements in one layer in a correct and efficient way.

4 HOTPOT ARCHITECTURE AND ABSTRACTION

We built *Hotpot*, a kernel-level DSPM system that provides applications with direct memory load/store access to both local and remote PM and a mechanism to make in-PM data durable, consistent, and reliable. Hotpot is easy to use, delivers low-latency performance, and provides flexible choices of data consistency, reliability, and availability levels. This section presents the overall architecture of Hotpot and its abstraction to applications.

We built most of Hotpot as a loadable kernel module in Linux 3.11.0 with only a few small changes to the original kernel. Hotpot has around 19K lines of code, out of which 6.4K lines are for a customized network stack (Section 7). Hotpot is available at https://github.com/WukLab/Hotpot.

Hotpot sits in the kernel space and manages PMs in a set of distributed nodes, or *Hotpot nodes*. Hotpot provides applications with an easy-to-use, memory-based abstraction that encapsulates both memory and persistent data access in a transparent way. Figure 3 presents Hotpot's architecture. Hotpot uses a *Central Dispatcher (CD)* to manage node membership and initialization tasks (*e.g.*, create a dataset). All data and metadata communication after a dataset has been created takes place between Hotpot nodes and does not involve the CD.

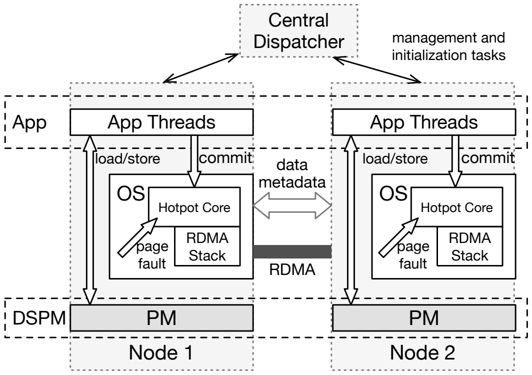


Figure 3: Hotpot Architecture.

4.1 Application Execution and Data Access Abstraction

Most data-intensive applications are multithreaded and distribute their data processing work across threads [28, 63]. Thus, Hotpot adopts a thread-based model to run applications on a set of Hotpot nodes. Hotpot uses application threads as the unit of deployment and lets applications decide what operations and what data accesses they want to include in each thread. Applications specify what threads to run on each Hotpot node and Hotpot runs an application by starting all its threads together on all Hotpot nodes. We give users full flexibility in choosing their initial thread and workload distributions. However, such user-chosen distributions may not be optimal, especially as workloads change over time. To remedy this situation, Hotpot provides a mechanism to adaptively move data closer to computation based on workload behavior, as will be discussed in Section 5.5.

Hotpot provides a global virtual memory address space to each application. Application threads running on a node can perform native memory load and store instructions using global virtual memory addresses to access both local and remote PM. The applications do not know where their data physically is or whether a memory access is local or remote. Internally, a virtual memory address can map to a local physical page if the page exists locally or generate a page fault which will be fulfilled by Hotpot by fetching a remote page (more in Section 5.3). Figure 4 presents an example of Hotpot's global virtual address space. Unlike an I/O-based interface, Hotpot's native memory interface can best exploit PMs' low-latency, DRAM-like performance, and byte addressability.

On top of the memory load/store interfaces, Hotpot provides a mechanism for applications to name their data, APIs to make their data persistent, and helper functions for distributed thread synchronization. Table 1 lists Hotpot APIs. We also illustrate Hotpot's programming model with a simple program in Figure 5. We will explain Hotpot's data commit semantics in Section 6.

4.2 Persistent Naming

To be able to store persistent data and to allow applications to re-open them after closing or failures, Hotpot needs to provide a naming mechanism that can sustain power recycles and crashes.

API	Explanation	Backward
open (*close*)	open or create (close) a DSPM dataset	same as current
mmap (*munmap*)	map (unmap) a DSPM region in a dataset to application address space	same as current
commit	commit a set of data and make *N* persistent replicas	similar to msync
acquire	acquire single writer permission	
thread-barrier	helper function to synchronize threads on different nodes	similar to pthread barrier

Table 1: Hotpot APIs. *Apart from these APIs, Hotpot also supports direct memory loads and stores.*

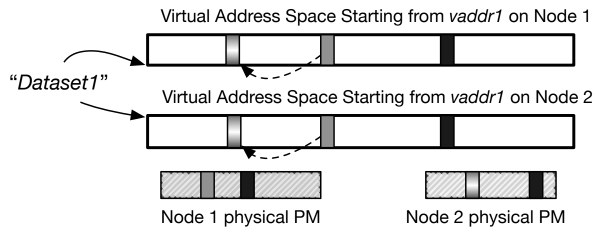

Virtual Address Space Starting from *vaddr1* on Node 1
"Dataset1"
Virtual Address Space Starting from *vaddr1* on Node 2
Node 1 physical PM
Node 2 physical PM

Figure 4: Hotpot Addressing. *Hotpot maps "Dataset1" to Node 1 and Node 2's virtual address space using the same base virtual addresses. The physical address mapping on each node is different. The grey blocks in the middle are pointers that point to the blocks on the left.*

```
/* Open a dataset in Hotpot DSPM space */
int fd = open("/mnt/hotpot/dataset", O_CREAT|O_RDWR);

/* Obtain virtual address of dataset with traditional mmap() */
void *base= mmap(0,40960,PROT_WRITE,MAP_PRIVATE,fd,0);

/* Size of the application log */
int *log_size = base;
/* The application log */
int *log = base + sizeof(int);

/* Append an entry to the end of the log */
int new_data = 24;
log[*log_size] = new_data;
*log_size += 1;

/* Prepare memory region metadata for commit */
struct commit_area_struct {void *address; int length;};
struct commit_area_struct areas[2];
areas[0].address = log_size;
areas[0].length = sizeof(int);
areas[1].address = &log[*log_size];
areas[1].length = sizeof(int);

/* Commit the two data areas, each with two replicas */
commit(areas, 2);
```

Figure 5: Sample code using Hotpot. *Code snippet that implements a simple log append operation with Hotpot.*

Many modern data-intensive applications such as in-memory databases [63] and graphs [28, 29] work with only one or a few big datasets that include all of an application's data and then manage their own fine-grained data structures within these datasets. Thus, instead of traditional hierarchical file naming, we adopt a flat naming mechanism in Hotpot to reduce metadata management overhead.

Specifically, Hotpot applications assign names by *datasets* and can use these names to open the datasets. A dataset is similar to the traditional file concept, but Hotpot places all datasets directly under a mounted Hotpot partition without any directories or hierarchies. Since under Hotpot's targeted application usage, there will only be a few big datasets, dataset lookup and metadata management with Hotpot's flat namespace are easy and efficient. We use a simple (persistent) hash table internally to lookup datasets.

The *open* and *mmap* APIs in Table 1 let applications create or open a dataset with a name and map it into the application's virtual memory address space. Afterwards, all data access is through native memory instructions.

4.3 Consistent and Persistent Pointers

Hotpot applications can use DSPM as memory and store arbitrary data structures in it. One resulting challenge is the management of pointers in DSPM. To make it easy to build persistent applications with memory semantics, Hotpot ensures that pointers in DSPM have the same value (*i.e.*, virtual addresses of the data that they point to) both across nodes and across crashes. Application threads on different Hotpot nodes can use pointers directly without pointer marshaling or unmarshaling, even after power failure. We call such pointers *globally-consistent and persistent pointers*. Similar to NV-Heaps [15], we restrict DSPM pointers to only point to data within the same dataset. Our targeted type of applications which build their internal data structures within big datasets already meet this requirement.

To support globally-consistent and persistent pointers, Hotpot guarantees that the same virtual memory address is used as the starting address of a dataset across nodes and across re-opens of the dataset. With the same base virtual address of a dataset and virtual addresses within a dataset being consecutive, all pointers across Hotpot nodes will have the same value.

We developed a new mechanism to guarantee that the same base virtual address is used across nodes and crashes. When an application opens a dataset for the first time, Hotpot uses a consensus protocol to discover the current available virtual address ranges on all nodes and select one for the dataset. Nodes that have not opened the dataset will reserve this virtual address range for possible future opening of the dataset. Since the total amount of virtual addresses for DSPM is bound to the total size of DSPM datasets, Hotpot can always find available virtual address ranges on 64-bit platforms. Hotpot records the virtual address range persistently and forces applications to use the same virtual address the next time it starts. To ensure that recorded persistent virtual address ranges are always available when opening datasets, we change the kernel loader and virtual memory address allocator (*i.e.*, *brk* implementation) to exclude all recorded address ranges.

5 DATA MANAGEMENT AND ACCESS

This section presents how Hotpot manages user data in DSPM. We postpone the discussion of data durability and reliability to Section 6.

5.1 PM Page Morphable States

One of Hotpot's design philosophies is to use one layer for both memory and storage and to integrate distributed memory caching and data replication. To achieve this goal, we propose to impose *morphable* states on PM pages, where the same PM page in Hotpot can be used both as a local memory cached copy to improve performance and as a redundant data page to improve data reliability and availability.

We differentiate three states of a PM page: active and dirty, active and clean, and inactive and clean, and we call these three states *dirty*, *committed*, and *redundant* respectively. A page being clean means that it has not been updated since the last commit point; committing a dirty page moves it to the clean state. A page being active means that it is currently being accessed by an application, while an redundant page is a page which the application process has not mapped or accessed. Several Hotpot tasks can change page states, including page read, page write, data commit, data replication, page migration, and page eviction. We will discuss how page states change throughout the rest of this section. Figure 6 illustrates two operations that cause Hotpot data state changes.

5.2 Data Organization

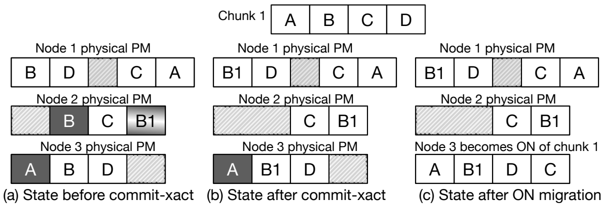
Hotpot aims to support large-scale, data-intensive applications on a fairly large number of nodes. Thus, it is important to minimize Hotpot's performance and scalability bottlenecks. In order to enable flexible load balancing and resource management, Hotpot splits the virtual address range of each dataset into *chunks* of a configurable size (*e.g.*, 4 MB). PM pages in a chunk do not need to be physically consecutive and not all pages in a chunk need to exist on a node.

Each chunk in Hotpot is owned by an *owner node (ON)*, similar to the "home" node in home-based DSM systems [97]. An ON maintains all the data and metadata of the chunk it owns. Other nodes, called *data node* or *DN*, always fetch data from the ON when they initially access the data. A single Hotpot node can simultaneously be the ON for some data chunks and the DN for other chunks. When the application creates a dataset, Hotpot CD performs an initial assignment of ONs to chunks of the dataset.

Two properties separate Hotpot ONs from traditional home nodes. First, Hotpot ON is responsible for the reliability and crash consistency of the pages it owns, besides serving read data and ensure the coherence of cached copies. Second, Hotpot does not fix which node owns a chunk and the location of ON adapts to application workload behavior dynamically. Such flexibility is important for load balancing and application performance (see Section 5.5).

5.3 Data Reads and Writes

Hotpot minimizes software overhead to improve application performance. It is invoked only when a page fault occurs or when applications execute data persistence operations (see Section 6 for details of data persistence operations).



(a) State before commit-xact (b) State after commit-xact (c) State after ON migration

Figure 6: Data State Change Example. *White, black, and striped blocks represent committed, redundant, and dirty states. Before commit, Node 2 and Node 3 both have cached copies of data page B. Node 2 has written to B and created a dirty page, B1. During commit, Node 2 pushes the content B1 to its ON, Node 1. Node 1 updates its committed copy to B1 and also sends this update to Node 3. Figure (c) shows the state after migrating the ON of chunk 1 from Node 1 to Node 3. After migration, Node 3 has all the pages of the chunk and all of them are in committed states.*

When a page fault happens because of read, it means that there is no valid local page. Hotpot first checks if there is any local redundant page. If so, it will move this page to the committed state and establish a page table entry (PTE) for it. Otherwise, there is no available local data and Hotpot will fetch it from the remote ON. Hotpot writes the received data to a newly-allocated local physical PM page. Afterwards, applications will use memory instructions to access this local page directly.

Writing to a committed page also causes a page fault in Hotpot. This is because a committed page can contribute towards user-specified degree of replication as one data replica, and Hotpot needs to protect this committed version from being modified. Thus, Hotpot write protects all committed pages. When these pages are written to (and generating a write page fault), Hotpot creates a local Copy-On-Write (COW) page and marks the new page as dirty while leaving the original page in committed state. Hotpot does not write protect this COW page, since it is already in the dirty state.

Following Hotpot's design philosophy to exploit hints from our targeted data-intensive applications, we avoid propagating updates to cached copies at other nodes on each write and only do so at each application commit point. Thus, all writes in Hotpot is local and only writing to a committed page will generate a page fault.

Not updating remote cached copies on each write also has the benefit of reducing write amplification in PM. In general, other software mechanisms and policies such as wear-aware PM allocation and reclamation and hardware techniques like Start-Gap [75] can further reduce PM wear. We do not focus on PM wear in this paper and leave such optimizations for future work.

5.4 PM Page Allocation and Eviction

Each Hotpot node manages its own physical PM space and performs PM page allocation and eviction. Since physical pages do not need to be consecutive, we use a simple and efficient allocation mechanism by maintaining a free page list and allocating one page at a time.

Hotpot uses an approximate-LRU replacement algorithm that is similar to Linux's page replacement mechanism. Different from Linux, Hotpot distinguishes pages of different states. Hotpot never evicts a dirty page and always tries to evict redundant pages before evicting committed pages. We choose to first evict redundant pages, because these are the pages that have not been accessed by applications and less likely to be accessed in the future than committed pages.

Since both redundant and committed pages can serve as a redundant copy for data reliability, Hotpot cannot simply throw them away during eviction. The evicting node of a page will contact its ON, which will check the current degree of replication of the candidate pages and prioritize the eviction of pages that already have enough replicas. For pages that will drop below the user-defined replication degree after the eviction, the ON will make a new redundant page at another node.

5.5 Chunk ON Migration

An ON serves both page read and data commit requests that belong to the chunks it owns. Thus, the location of ON is important to Hotpot's performance. Ideally, the node that performs the most reads and commits of data in a chunk should be its ON to avoid network communication.

By default, Hotpot initially spreads out a dataset's chunks to all Hotpot nodes in a round robin fashion (other static placement policies can easily replace round robin). Static placement alone cannot achieve optimal run-time performance. Hotpot remedies this limitation by performing online chunk migration, where one ON and one DN of a chunk can switch their identities and become the new DN and new ON of the chunk.

Hotpot utilizes application behavior in recent history to decide how to migrate ONs. Each ON records the number of page read requests and the amount of committing data it receives in the most recent time window.

ONs make their migration decisions with a simple greedy algorithm based on the combination of two criteria: maximizing the *benefit* while not exceeding a configurable *cost* of migration. The benefit is the potential reduction in network traffic during remote data reads and commits. The node that performs most data communication to the ON in recent history is likely to benefit the most from being the new ON, since after migration these operations will become local. We model the cost of migration by the amount of data needed to copy to a node so that it has all the chunk data to become ON.

Once Hotpot has made a decision, it performs the actual chunk migration using a similar method as process and VM migration [14, 20, 68] by temporary stopping commits to the chunk under migration and resume them at the new ON after migration.

6 DATA DURABILITY, CONSISTENCY, AND RELIABILITY

Being distributed shared memory and distributed storage at the same time, DSPM should ensure both correct shared memory accesses to PM and the persistence and reliability of in-PM data. Hotpot provides three guarantees: coherence among cached copies of in-PM data, recovery from various types of failures into a consistent state, and user data reliability and availability under concurrent failures. Although each of these three properties have been explored before, as far as we know, Hotpot is the first system that integrates all of them in one layer. Hotpot also has the unique requirement of low software overhead to retain the performance benefit of PM.

- *Cache coherence.* In Hotpot, application processes on different nodes cache remote data in their local PM for fast accesses. Hotpot provides two consistency levels across cached copies: ***R1.a***, multiple readers and single writer (*MRSW*) and ***R1.b***, multiple readers and multiple writers (*MRMW*). MRMW allows multiple nodes to concurrently write and commit their local cached copies. With MRMW, there can be multiple versions of dirty data in the system (but still one committed version), while MRSW guarantees only one dirty version at any time. An application can use different modes for different datasets, but only one mode with the same dataset. This design allows flexibility at the dataset granularity while guaranteeing correctness.
- *Crash consistency.* Data storage applications usually have well-defined *consistent* states and need to move from one consistent state to another atomically. When a crash happens, user data should be recovered to a consistent state (*i.e.*, *crash consistency*). Hotpot guarantees crash consistency both within a single node (***R2.a***) and across distributed nodes (***R2.b***). Note that crash consistency is different and orthogonal to cache coherence in **R1.a** and **R1.b**.
- *Reliability and availability.* To ensure that user persistent data can sustain $N - 1$ concurrent node failures, where N is a user defined value, Hotpot guarantees that ***R3***, once data has been committed, there are always N copies of clean, committed data.

This section first discusses how Hotpot ensures crash consistency within a single node, then presents the MRMW and MRSW modes and their atomic commit protocols, and ends with the discussion of Hotpot's recovery mechanisms under different crash scenarios.

6.1 Single-Node Persistence and Consistency

Before ensuring user data's global reliability and consistency in DSPM, Hotpot first needs to make sure that data on a single node can properly sustain power crashes (**R2.a**) [70]. Hotpot makes data persistent with the standard Intel persistent memory instructions [42], *i.e.*, `clflush`, `mfence` (note that we do not include the deprecated `pcommit` instruction [35]).

After a node crashes, if its PM is still accessible, Hotpot will use the PM content to recover; otherwise, Hotpot will use other nodes to reconstruct data on a new node (Section 6.4). For the former case, Hotpot needs to guarantee that user data in DSPM is in a consistent state after crash. Hotpot also needs to ensure that its own metadata is persistent and is consistent with user data.

Hotpot maintains metadata on a local node to find user data and record their morphable states (*i.e.*, committed, dirty, or redundant). Since these metadata are only used within a single node, Hotpot does not need to replicate them on other nodes. Hotpot makes these metadata persistent at known locations in PM — a pre-allocated beginning area of PM. Hotpot also uses metadata to record online state of the system (*e.g.*, ON maintains a list of active DNs that have a cached copy of data). These metadata can be reconstructed by re-examining system states after recovery. Thus, Hotpot does not make these metadata persistent.

Similar to traditional file systems and databases, it is important to enforce *ordering* of metadata and data persistence in order to recover to a consistent state. For single-node non-commit operations (we defer the discussion of commit operations to Sections 6.2 and 6.3),

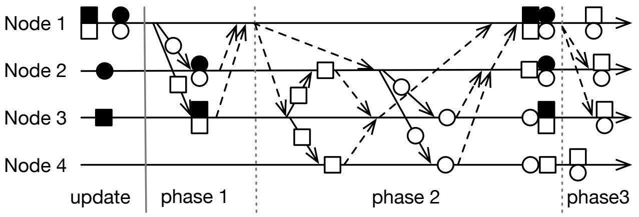


Figure 7: MRMW Commit Example. *Solid arrows represent data communication. Dashed arrows represent metadata communication. Node 1 (CN) commit data to ONs at Node 2 and 3 with replication degree four. Black shapes represent old committed states before the update and white shapes represent new states.*

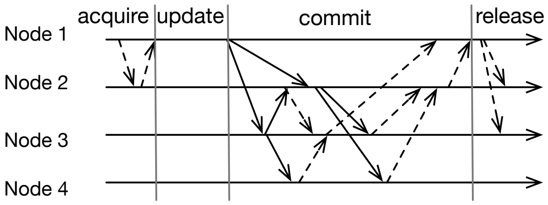


Figure 8: MRSW Example. *Node 1 (CN) first acquires write permission from Node 2 (MN) before writing data. It then commits the new data to ONs at Node 2 and 3 with replication degree four and finally releases the write permission to MN.*

Hotpot uses a simple shadow-paging mechanism to ensure that the consistency of metadata and data. Specifically, we associate each physical memory page with a metadata slot and use a single 8-byte index value to locate both the physical page and its metadata. When an application performs a memory store to a committed page, Hotpot allocates a new physical page, writes the new data to it, and writes the new metadata (*e.g.*, the state of the new page) to the metadata slot associated with this physical page. After making all the above data and metadata persistent, Hotpot changes the index from pointing to the old committed page to pointing to the new dirty page. Since most architectures support atomic 8-byte writes, this operation atomically moves the system to a new consistent state with both the new data and the new metadata.

6.2 MRMW Mode

Hotpot supports two levels of concurrent shared-PM accesses and uses different protocols to commit data. The MRMW mode allows multiple concurrent versions of dirty, uncommitted data to support great parallelism. MRMW meets **R1.b**, **R2.b**, and **R3**.

MRMW uses a distributed atomic commit protocol at each commit point to make local updates globally visible, persistent, and replicated. Since MRMW supports concurrent commit operations and each commit operation can involve multiple remote ONs, Hotpot needs to ensure that all the ONs reach consensus on the commit operation they serve. We designed a three-phase commit protocol for the MRMW mode based on traditional two-phase commit protocols [30, 48, 78] but differs in that Hotpot needs to ensure cache coherence, crash consistency, and data replication all in one protocol. Figure 7 illustrates an example of MRMW.

Commit phase 1. When a node receives a *commit* call (we call this node *CN*), it checks if data specified in the *commit* call is dirty and commits only the dirty pages. CN persistently records the addresses of these dirty pages for recovery reasons (Section 6.4). CN also assigns a unique ID (*CID*) for this *commit* request and persistently records the CID and its state of starting phase 1.

Afterwards, CN sends the committing data to its ONs to prepare these ONs for the commit. Each ON accepts the commit request if it has not accepted other commit request to the same pages, and it stores the committing data in a *persistent redo log* in PM. The ON also persistently records the CID and its state (*i.e.*, completed phase 1) persistently. The ON will block future commit requests to these data until the whole commit process finishes. The CN can proceed to phase 2 only when all ONs return successfully.

Commit phase 2. In commit phase 2, Hotpot makes the committing data persistent, coherent, and replicated. This is the phase that Hotpot differs most from traditional distributed commit protocols.

CN sends a command to all the involving ONs to indicate the beginning of phase 2. Each ON then performs two tasks in one multicast operation (Section 7): updating DNs' cached copies of the committing data and making extra replicas. ON looks up its metadata to find what DNs have a cached copy. If these DNs alone cannot meet the replication degree, ON will choose new DNs that do not have a copy of the data and send the data to them.

When a DN receives the committing data from an ON, it checks the state of its local data pages. If a local page is in the committed state or the redundant state, the DN will directly overwrite the local page with the received data. In doing so, the DN's cached PM data is updated. If the local page is dirty or if there is no corresponding local page, the DN allocates a new physical page and writes the new data to this page. The new physical page will be in the redundant state and will not affect the DN's dirty data. In this way, all DNs that receive updated data from the ON will have a clean, committed copy, either in the committed or the redundant state.

After all DNs have replied to the ON indicating that there are now N copies of the committing data, the ON commits data locally by checkpointing (copying) data from the redo log to their home locations. Unlike traditional databases and file systems that lazily checkpoint logged data, Hotpot checkpoints all committing data in this phase so that it can make the updated version of the data visible to applications immediately, a requirement of shared-memory cache coherence. During checkpointing, the ON will block both local and remote reads to the committing data to prevent applications from reading intermediate, inconsistent data.

After the CN receives successful replies from all the ONs, it deletes its old local data and moves to the new, committed version. At this point, the whole system has a coherent view of the new data and has at least N copies of it.

Commit phase 3. In the last phase, the CN informs all ONs that the *commit* operation has succeeded. The ONs then delete their redo logs.

Committing to a single ON and to local ON. When only one remote ON is involved in a *commit* operation, there is no need to coordinate multiple ONs and Hotpot performs the above commit protocol in a single phase.

The CN can also be the ON of committing data. In this case, the CN performs the *commit* operation locally. Since all local dirty pages are the COW of old committed pages, CN already has an undo and a redo copy of the committing data and does not need to create any other redo log as in remote ON's phase 1.

6.3 MRSW Mode

The MRSW mode allows only one writer to a PM page at a time to trade parallelism for stronger consistency. MRSW meets **R1.a**, **R2.b**, and **R3**.

Traditional MRSW protocols in DSM systems are usually invoked at every memory store (*e.g.*, to update cached read copies, to revoke current writer's write permission). Unlike DSM systems, DSPM applications store and manage persistent data; they do not need to ensure coherence on every memory store, since they have well-defined points of when they want to start updating data and when they want to commit. To avoid the cost of invoking coherence events on each memory store while ensuring only one writer at a time, Hotpot uses an *acquire* API for applications to indicate the data areas they want to update. Afterwards, applications can update any date that they have acquired and use the *commit* call to both commit updates and release corresponding data areas. Figure 8 shows an example of MRSW.

Acquire write permission. Hotpot uses a master node (*MN*) to maintain the active writer of each page. An MN can be one of the Hotpot node, the CD, or a dedicated node. When a node receives the *acquire* call, it sends the virtual addresses of the data specified in the call to the MN. If the MN finds that at least one of these addresses are currently being written to, it will reject the *acquire* request and let the requesting node retry later.

Commit and release data. MRSW's commit protocol is simpler and more efficient than MRMW's, since there is no concurrent commit operations to the same data in MRSW (concurrent commit to different data pages is still allowed). MRSW combines phase 1 and phase 2 of the MRMW commit protocol into a single phase where the CN sends committing data to all ONs and all ONs commit data on their own. Each ON individually handles commit in the same way as in the MRMW mode, except that it does not need to coordinate with any other ONs or the CN. ON directly proceeds to propagating data to DNs after it has written its own redo log.

At the end of the commit process, the CN informs the ONs to delete their redo logs (same as MRMW commit phase 3) and the MN to release the data pages.

6.4 Crash Recovery

Hotpot can safely recover from different crash scenarios without losing applications' data. Hotpot detects node failures by request timeout and by periodically sending heartbeat messages from the CD to all Hotpot nodes. We now explain Hotpot's crash recovery mechanism in the following four crash scenarios. Table 2 summarizes various crash scenarios and Hotpot's recovery mechanisms.

Recovering CD and MN. CD maintains node membership and dataset name mappings. Hotpot currently uses one CD but can be easily extended to include a hot stand-by CD (*e.g.*, using Mojim [95]).

	Node	PM	Time	Action
	any	Y	any	resume normal operation after reboot
	CD	N	any	reconstruct using mirrored copy
	ON	N	NC	promote an existing DN to ON
	DN	N	NC	reconstruct data to meet replication degree
MRMW Commit	CN/ON	N	p1	undo commit, ONs delete redo logs
	CN	N	p2	redo commit, ONs send new data to DNs
	ON	N	p2	redo commit, CN sends new data to new ON
	DN	N	p2	continue commit, ON sends data to new DN
	CN	N	p3	complete commit, ONs delete redo logs
	ON/DN	N	p3	complete commit, new chunk reconstructed using committed data
MRSW	CN	N	commit	undo commit, ONs send old data to DNs
	ON	N	commit	CN redo commit from scratch
	CN	N	release	complete commit, release data
	ON/DN	N	release	complete commit, new chunk reconstructed using committed data

Table 2: Crash and Recovery Scenarios. *Columns represent crashing node, if PM is accessible after crash, time of crash, and actions taken at recovery. NC represents non-commit time.*

MN tracks which node has acquired write access to a page under the MRSW mode. Hotpot does not make this information persistent and simply reconstructs it by contacting all other nodes during recovery.

Non-commit time crashes. Recovering from node crashes during non-commit time is fairly straightforward. If the PM in the crashed node is accessible after the crash (we call it *with-PM failure*), Hotpot directly restarts the node and lets applications access data in PM. As described in Section 6.1, Hotpot ensures crash consistency of a single node. Thus, Hotpot can always recover to a consistent state when PM survives a crash. Hotpot can sustain arbitrary number of with-PM failures concurrently.

When a crash results in corrupted or inaccessible PM (we call it *no-PM failure*), Hotpot will reconstruct the lost data using redundant copies. Hotpot can sustain $N - 1$ concurrent no-PM failures, where N is the user-defined degree of replication.

If a DN chunk is lost, the ON of this chunk will check what data pages in the chunk have dropped below user-defined replication degree and replicating them on the new node that replaces the failed node. There is no need to reconstruct the rest of the DN data; Hotpot simply lets the new node access them on demand.

When an ON chunk is lost, it is critical to reconstruct it quickly, since an ON serves both remote data read and commit operations. Instead of reconstructing a failed ON chunk from scratch, Hotpot promotes an existing DN chunk to an ON chunk and creates a new DN chunk. The new ON will fetch locally-missing committed data from other nodes and reconstruct ON metadata for the chunk. Our evaluation results show that it takes at most 2.3 seconds to promote a 1GB DN chunks to ON.

Crash during commit. If a with-PM failure happens during a *commit* call, Hotpot will just continue its commit process after restart. When a no-PM failure happens during commit, Hotpot takes different actions to recover depending on when the failure happens.

For MRMW commit, if no-PM failure happens before all the ONs have created the persistent redo logs (*i.e.*, before starting phase 2), Hotpot will undo the commit and revert to the old committed state by deleting the redo logs at ONs. If a no-PM failure happens after all ONs have written the committing data to their persistent redo logs (*i.e.*, after commit phase 1), Hotpot will redo the commit by replaying redo logs.

For MRSW, since we combine MRMW's phase 1 and phase 2 into one commit phase, we will not be able to tell whether or not an ON has pushed the committing data to DNs when this ON experience a no-PM failure. In this case, Hotpot will let CN redo the commit from scratch. Even if the crashed ON has pushed updates to some DNs, the system is still correct after CN redo the commit; it will just have more redundant copies. When the CN fails during MRSW commit, Hotpot will undo the commit by letting all ONs delete their redo logs and send old data to DNs to overwrite DNs' updated data.

During commit, Hotpot only supports either CN no-PM failure or ON no-PM failure. We choose not to support concurrent CN and ON no-PM failures during commit, because doing so largely simplifies Hotpot's commit protocol and improves its performance. Hotpot's commit process is fast (under $250\mu s$ with up to 16 nodes, see Section 8.4). Thus, the chance of CN and ON both fail and lose their PM during commit is very small. Hotpot always supports DN no-PM failures during commit regardless of whether there are concurrent CN or ON failure.

7 NETWORK LAYER

The networking delay in DSPM systems is crucial to their overall performance. We implement Hotpot's network communication using RDMA. RDMA provides low-latency, high-bandwidth direct remote memory accesses with low CPU utilization. Hotpot's network layer is based on LITE [87], an efficient RDMA software stack we built in the Linux kernel on top of the RDMA native APIs, *Verbs* [60].

Most of Hotpot's network communication is in the form of RPC. We implemented a customized RPC-like interface in our RDMA layer based on the two-sided RDMA send and receive semantics. We further built a multicast RPC interface where one node can send a request to multiple nodes in parallel and let them each perform their processing functions and reply with the return values to the sending node. Similar to the findings from recent works [38], two-sided RDMA works better and is more flexible for these RPC-like interfaces than one-sided RDMA.

To increase network bandwidth, our RDMA layer enables multiple connections between each pair of nodes. It uses only one busy polling thread per node to poll a shared ring buffer for all connections, which delivers low-latency performance while keeping CPU utilization low. Our customized RDMA layer achieves an average latency of $7.9\mu s$ to perform a Hotpot remote page read. In comparison, IPoIB, a standard IP layer on top of Verbs, requires $77\mu s$ for a round trip with the same size.

8 APPLICATIONS AND EVALUATION

This section presents the performance evaluation of two applications and a set of microbenchmarks. We ran all experiments on a cluster of 17 machines, each with two Intel Xeon CPU E5-2620 2.40GHz processors, 128 GB DRAM, and one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter; a Mellanox 40 Gbps InfiniBand switch connects all of the machines. All machines run the CentOS 7.1 distribution and the 3.11.1 Linux kernel.

The focus of our evaluation is to understand the performance of DSPM's distributed memory model, its commit protocols, and its data persistence cost. As there is no real PM in production yet, we use DRAM as stand-in for PM. A previous study [94] shows that even though PM and DRAM can have some performance difference, the difference is small and has much lower impact on application performance than the cost of flushing data from CPU caches to PM, which we have included in Hotpot and can measure accurately.

8.1 Systems in Comparison

We compare Hotpot with one in-memory file system, two PM-based file systems, one replicated PM-based system, and a distributed shared memory systems. Below we briefly describe these systems in comparison.

Single-Node File Systems. Tmpfs is a Linux file system that stores all data in main memory and does not perform any I/Os to storage devices. PMFS [22] is a file system designed for PM. The key difference between PMFS and a conventional file system is that its implementation of *mmap* maps the physical PM pages directly into the applications' address spaces rather than moving them back and forth between the file store and the buffer cache. PMFS ensures data persistence using `sfence` and `clflush` instructions.

Distributed PM-Based Systems Octopus [57] is a user-level RDMA-based distributed file system designed for PM. Octopus provides a set of customized file APIs including read and write, but does not support memory-mapped I/Os or provide data reliability and availability.

Mojim [95] is our previous work that uses a primary-backup model to replicate PM data over a customized IB layer. Similar to Hotpot, PMFS, and Octopus, Mojim maps PM pages directly into application virtual memory address spaces. Mojim supports application reads and writes on the primary node but only reads on backup nodes.

Distributed Shared Memory System. We compare Hotpot with Grappa [65], a recent DSM system that supports modern data-parallel applications. Grappa implements a customized network stack on InfiniBand. Different from traditional DSM systems, Grappa moves computation to data instead of fetching data to where computation is.

8.2 In-Memory NoSQL Database

MongoDB [63] is a popular distributed NoSQL database that supports several different storage engines including its own storage engine that is based on memory-mapped files (called MMAPv1). Applications like MongoDB can largely benefit from having a fast means to store and access persistent data. We ported MongoDB v2.7.0 to Hotpot by modifying its storage engine to keep track of all writes to the memory-mapped data file. We then group the written memory regions belonging to the same client request into a Hotpot *commit* call. In total, porting MongoDB to Hotpot requires modifying 120 lines of code.

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

Figure 9: YCSB Workload Properties. *The percentage of operations in each YCSB workload. R&U stands for Read and Update.*

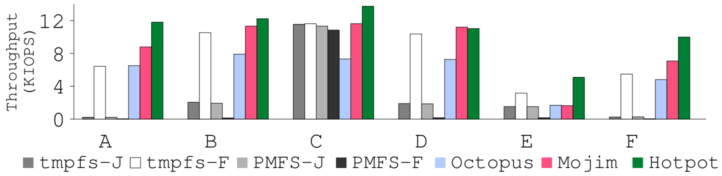


Figure 10: YCSB Workloads Throughput.

To use the ported MongoDB, administrators can simply configure several machines to share a DSPM space under Hotpot and run ported MongoDB on each machine. Applications on top of the ported MongoDB can issue requests to any machine, since all machines access the same DSPM space. In our experiments, we ran the ported MongoDB on three Hotpot nodes and set data replication degree to three.

We compare this ported MongoDB with the default MongoDB running on tmpfs, PMFS, and Octopus, and a ported MongoDB to Mojim on three nodes connected with IB. Because Octopus does not have memory-mapped operations and MongoDB's storage engine is based on memory-mapped files, MongoDB cannot directly run on Octopus. We run MongoDB on top of FUSE [1], a full-fledged user-level file system, which in turn runs on Octopus.

For tmpfs and PMFS, we use two consistency models (called MongoDB write concerns): the JOURNALED write concern and the FSYNC_SAFE write concern. With the JOURNALED write concern, MongoDB logs data in a journal file and checkpoints the data in a lazy fashion. MongoDB blocks a client call until the updated data is written to the journal file. With FSYNC_SAFE, MongoDB does not perform journaling. Instead, it flushes its data file after each write operation and blocks the client call until this operation completes. We run Octopus and Mojim with the FSYNC_SAFE write concern. None of Octopus, tmpfs, and PMFS perform any replication, while Mojim and Hotpot use their own replication mechanisms to make three replicas of all data (Mojim uses one node as the primary node and the other two nodes as backup nodes).

YCSB [17] is a key-value store benchmark that imitates web applications' data access models. Figure 9 summarizes the amount of different operations in the YCSB workloads. Each workload performs 10,000 operations on a database with 100,000 1 KB records. Figure 10 presents the throughput of MongoDB on tmpfs, PMFS, Octopus, Mojim, and Hotpot using YCSB workloads.

For all workloads, Hotpot outperforms tmpfs, PMFS, Octopus, and Mojim for both the JOURNALED and the FSYNC_SAFE write concerns. The performance improvement is especially high for write-heavy workloads. PMFS performs worst mainly because of its inefficient process of making data persistent with default MongoDB. The default MongoDB `fsync`s the whole data file after each write under FSYNC_SAFE, and PMFS flushes all cache lines of the file to PM by performing one `clflush` at a time. Hotpot and Mojim only commit dirty data, largely improving MongoDB performance over PMFS. Compared to tmpfs and PMFS under JOURNALED, Hotpot and Mojim use their own mechanisms to ensure data reliability and avoid the performance cost of journaling. Moreover, Hotpot and Mojim make three persistent replica for all data, while PMFS makes only one. Tmpfs with JOURNALED is slower than Hotpot even though tmpfs does not make any data persistent, because MongoDB's slower journaling mechanism.

Octopus performs worse than Hotpot and Mojim because it incurs significant overhead of additional *indirection layers*: each memory operation within the memory-mapped file goes through the FUSE file system and then through Octopus. Hotpot and Mojim both support native memory instructions and incurs no indirection overhead. Octopus's incurs performance overhead for all types of I/O operations; for the read-only workload, it is worse than all the rest of the systems in comparison. Finally, even though Mojim's replication protocol is simpler and faster than Hotpot's, Hotpot outperforms Mojim because Mojim only supports writes on one node while Hotpot supports writes on all nodes.

8.3 Distributed (Persistent) Graph

Graph processing is an increasingly important type of applications in modern datacenters [28, 29, 47, 55, 56, 59]. Most graph systems require large memory to run big graphs. Running graph algorithms on PM not only enables them to exploit the big memory space the high-density PM provides, but can also enable graph algorithms to stop and resume in the middle of a long run.

We implemented a distributed graph processing engine on top of Hotpot based on the PowerGraph design [28]. It stores graphs using a vertex-centric representation with random graph partitioning and distributes graph processing load to multiple threads across all Hotpot nodes. Each thread performs graph algorithms on a set of vertices in three steps: gather, apply, and scatter, with the optimization of delta caching [28]. After each step, we perform a global synchronization with *thread-barrier* and only start the next step when all threads have finished the previous step. At the scatter step, the graph engine uses Hotpot's MRSW *commit* to make local changes of the scatter values visible to all nodes in the system. We implemented the Hotpot graph engine with around 700 lines of code.

We compare Hotpot's graph engine with PowerGraph and Grappa [65] using two real datasets, Twitter (41 M vertices, 1 B directed edges) [46] and LiveJournal (4 M vertices, 34.7 M undirected edges) [53]. For space reason, we only present the results of the Twitter graph, but the results of LiveJournal are similar. Figure 11 shows the total run time of the PageRank [49] algorithm with Hotpot, PowerGraph, and Grappa under three system settings: four nodes each running four graph threads, seven nodes each running four threads, and seven nodes each running eight threads.

Hotpot outperforms PowerGraph by 2.3× to 5× and Grappa by 1.3× to 3.2×. In addition, Hotpot makes all intermediate results of graph persistent for fast restart. A major reason why Hotpot outperforms PowerGraph and Grappa even when Hotpot requires

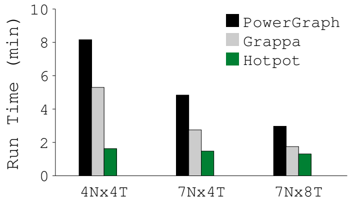


Figure 11: Pagerank Total Run Time. *N stands for the total number of nodes. T stands for the number of threads running on each node.*

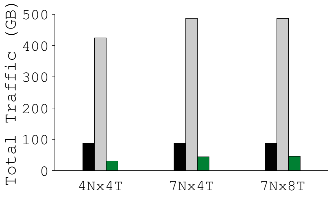


Figure 12: Pagerank Total Network Traffic.

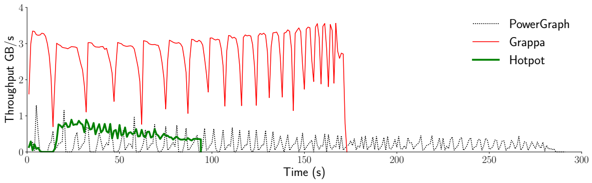


Figure 13: Pagerank Network Traffic Over Time.

data persistence and replication is because of Hotpot's efficient network stack. Compare to the IPoIB used in PowerGraph and Grappa's own network stack, Hotpot's RDMA stack offers much lower latency.

To further understand the performance differences, we traced the network traffic of these three systems. Figure 12 plots the total amount of traffic and Figure 13 plots a detailed trace of network activity of the 7Nx4T setting. Hotpot sends less total traffic and achieves higher bandwidth than PowerGraph and Grappa.

8.4 Micro-Benchmark Results

We now present our microbenchmark results that evaluate the effect of different system settings and parameters. Since Hotpot reads have a constant latency (around 7.9μs) and Hotpot writes do not go through network, Hotpot's performance is largely affected by its data commit process. Thus, we focus our microbenchmark experiments on the commit operations.

Scalability. Figure 14 shows the total commit throughput of Hotpot on 2 to 16 nodes with a workload that lets all nodes concurrently commit 32 random 4 KB areas with replication degree 1. Overall, both MRMW and MRSW commit scale. As expected, MRMW *commit* is more costly than MRSW.

Replication degree and committing size. We next evaluate the effect of replication degree and the total amount of data in a *commit* call. Our results show that with higher replication degree and with more committing data, *commit* takes longer for both MRMW and MRSW. Because of space reasons, we do not include figures for these experiments.

Chunk size. We use a controlled microbenchmark to showcase the effect of chunk size (Figure 15). Each run has one node in a cluster of four nodes committing 32 1 KB areas that spread evenly in a 32 MB region with replication degree 1. Since Hotpot distributes chunks in Round Robin, when chunk size is below 8 MB, the 32 MB region will be distributed equally to all four nodes. The *commit* performance stays similar with 1, 4, and 8 MB chunk size, since *commit* will always use all four nodes as ONs. When chunk size is 16 MB (or 32 MB), only two (or one) nodes are ON. We observe two different behaviors: when the CN happens to also be the ON of the chunk that contains the committing data, the *commit* performance is better than when chunk size is below 8 MB, since half (or all) commit happens locally at the CN. But when the CN is not ON, all commit traffic goes to only two (or one) remote nodes, resulting in worse performance than when chunk size is small. This result suggest that smaller chunk size has better load balancing.

ON migration. From the previous experiments, we find that the *commit* performance depends heavily on the location of ON and the initial Hotpot ON assignment may not be optimal. We now evaluate how effective Hotpot's ON migration technique is in improving *commit* performance (Figure 16). We ran a workload with Zipf distribution to model temporal locality in datacenter applications [4, 10] on four nodes with replication degree 1 to 4. Each node issues 100,000 *commit* calls to commit two locations generated by Zipf. With ON migration, the *commit* performance improves by 13% to 29% under MRSW and 38% to 64% under MRMW. ON migration improves performance because the node that performs most *commit* on a chunk becomes its ON after migration. The improvement is most significant with replication degree one, because when CN is ON and the replication degree is one, there is no need to perform any network communication. MRMW's improvement is higher than MRSW, because MRMW can benefit more from committing data locally (the MRMW commit process that involves remote ONs is more costly than that of MRSW).

Effect of conflict commits. Figure 17 compares the *commit* performance of when 1 to 4 nodes in a four node cluster concurrently commit data in two scenarios: all CNs commit the same set of data (32 sequential 1 KB areas) at the same time which results in commit conflict, and CNs use different set of data without any conflict. Commit conflict causes degraded performance, and the degradation is worse with more conflicting nodes. However, conflict is rare in reality, since *commit* is fast. Conflict only happens when different nodes commit the same data page at exactly the same time. In fact, we had to manually synchronize all nodes at every *commit* call using *thread-barrier* to create conflict.

9 RELATED WORK

There have been a host of distributed shared memory systems and distributed storage systems [3, 11, 13, 18, 26, 27, 44, 45, 73, 79, 82, 83, 86, 93, 96, 97] over the past few decades. While some of Hotpot's coherence protocols may resemble existing DSM systems, none of them manages persistent data. There are also many single-node PM systems [15, 16, 21, 22, 41, 42, 61, 69, 71, 88, 89], but they do not support distributed environments.

Octopus [57] is a user-level RDMA-based distributed PM file system developed in parallel with Hotpot. Octopus manages file system metadata and data efficiently in a pool of PM-equipped machines. Octopus provides a set of customized file APIs including read and write but not any memory-mapped interfaces. Octopus

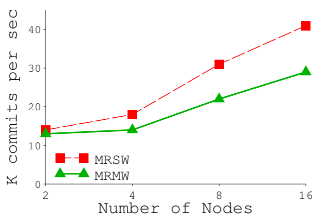


Figure 14: Hotpot Scalability. *commit throughput with 2 to 16 nodes.*

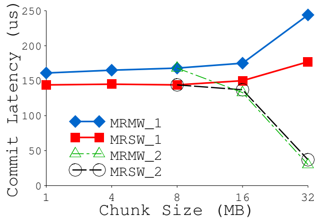


Figure 15: Chunk Size. *For 16 MB and 32 MB cases, 1 represents ON being remote and 2 represents CN being ON.*

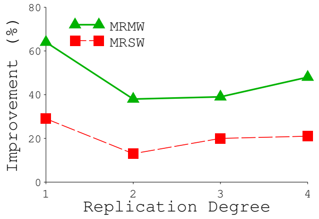


Figure 16: ON Migration. *The improvement of average commit latency with ON migration over no migration.*

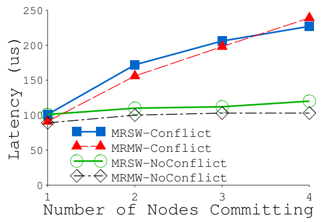


Figure 17: Commit Conflict. *Average commit latency with and without conflict.*

does not provide data reliability and high availability either. Hotpot's abstraction is memory-based rather than file-based, and it offers data reliability, availability, and different consistency levels.

Grappa [65] is a DSM system that supports modern data-parallel applications. Instead of fetching remote memory to a local cached copy, Grappa executes functions at the remote side. Hotpot is a DSPM system and lets applications store persistent data. It fetches remote data for both fast local access and data replication.

FaRM [21] is an RDMA-based distributed memory system. RAMCloud is a low-latency distributed key-value store system that keeps a single copy of all data in DRAM [67] and replicates data on massive slower storages for fast recovery. The major difference between Hotpot and FaRM or RAMCloud is that FaRM and RAMCloud both add software indirection layers for key-value stores which can cause significant latency overhead over native load/store operations and obscure much of the performance of the underlying PM. Hotpot uses a memory-like abstraction and directly stores persistent data in PM. Hotpot also performs data persistence and replication differently and uses a different network layer in the kernel.

Crail [34] is an RDMA-based high-performance multi-tiered distributed storage system that integrates with the Apache Spark ecosystem [91]. Crail mainly consists of a file system that manages tiered storage resources (*e.g.*, DRAM, flash, disk) with flexible allocation policies across tiers. Hotpot is a pure PM-based system that exposes a memory-like interface.

PerDis [81] and Larchant [23, 80] use a distributed file system below a DSM layer. Unlike these systems, Hotpot is a single-layer system that provides shared memory access, data persistence, and reliability.

Our own previous work, Mojim [95], provides an efficient mechanism to replicate PM over IB using a primary-backup protocol. Hotpot is a DSPM system that provides a shared-memory abstraction and integrates cache coherence and data replication.

10 CONCLUSION

We presented Hotpot, a kernel-level DSPM system that provides applications with a shared persistent memory abstraction. Our evaluation results show that it is easy to port existing applications to Hotpot and the resulting systems significantly outperform existing solutions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Eno Thereska for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper.

This material is based upon work supported by the National Science Foundation under the following grant: NSF CNS-1719215. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

REFERENCES

[1] Filesystem in Userspace. http://libfuse.github.io/.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. Technical report, 2015.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. 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.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, New York, NY, USA, 2012.

[5] K. Bailey, L. Ceze, S. D. Gribble, and H. 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.

[6] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the 2ed ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '90)*, Seattle, Washington, March 1990.

[7] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[8] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, Seattle, Washington, May 1990.

[9] D. L. Black, A. Gupta, and W.-D. Weber. Competitive Management of Distributed Shared Memory. In *Proceedings of the 34th IEEE Computer Society International Conference on COMPCON (COMPCON '89)*, San Francisco, California, March 1989.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1999.

[11] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. 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.

[12] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems (EUROSYS'16)*, London, UK, April 2016.

[13] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation (NSDI '06)*, San Jose, California, May 2006.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[15] 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 *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, New York, New York, March 2011.

[16] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. C. Lee, and D. 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.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. 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.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. 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.

[19] G. S. Delp. *The Architecture and Implementation of MEMNET: A High–speed Shared-memory Computer Communication Network*. PhD thesis, Newark, DE, USA, 1988. Order No. GAX88-24208.

[20] F. Douglis and J. K. Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[21] A. DragojeviÄĞ, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, Seattle, Washington, April 2014.

[22] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the EuroSys Conference (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[23] P. Ferreira and M. Shapiro. Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, 1996.

[24] P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Robert, F. Sandakly, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, and S. Krakowiak. *PerDiS: Design, Implementation, and Use of a PERsistent DIstributed Store*. 2000.

[25] B. D. Fleisch and G. J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*.

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[27] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '91)*, Hilton Head, South Carolina, July 1991.

[28] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI '12)*, Vancouver, Canada, October 2010.

[29] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in A Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[30] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, London, UK, UK, 1978.

[31] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. Shin. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Boston, MA, USA, April 2017.

[32] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*, Florianopolis, Brazil, August 2016.

[33] 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.

[34] IBM. Crail Distributed File System. http://www.crail.io/.

[35] Intel Corporation. Deprecating the PCOMMIT Instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[36] Intel Corporation. Intel Non-Volatile Memory 3D XPoint. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html?wapkw=3d+xpoint.

[37] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM conference on SIGCOMM (SIGCOMM '14)*, Chicago, Illinois, August 2014.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savanah, GA, November 2016.

[39] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, Queensland, Australia, May 1992.

[40] R. E. Kessler and M. Livny. An Analysis of Distributed Shared Memory Algorithms. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[41] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, New York, NY, USA, 2016.

[42] A. Kolli, J. Rosen, S. Diestelhorst, A. G. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016.

[43] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, J. Wagner Meira, S. Dwarkadas, and M. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, Denver, Colorado, June 1997.

[44] O. Krieger and M. Stumm. An Optimistic Algorithm for Consistent Replicated Shared Data. In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences (HICSS '90)*, Kailua-Kona, Hawaii, January 1990.

[45] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. 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.

[46] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, A Social Network or A News Media? In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*, Raleigh, North Carolina, April 2010.

[47] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, California, October 2012.

[48] B. W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, London, UK, UK, 1981.

[49] P. Lawrence, B. Sergey, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[50] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Phase Change Memory Architecture and the Quest for Scalability. *Commun. ACM*, 53(7):99–106, 2010.

[51] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE micro*, 30(1):143, 2010.

[52] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, S. Seo, et al. A Fast, High-Endurance and Scalable Non-Volatile Memory Device Made from Asymmetric Ta2O(5-x)/TaO(2-x) Bilayer Structures. *Nature materials*, 10(8):625–630, 2011.

[53] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[54] V. M. Lo. Operating Systems Enhancements for Distributed Shared Memory. *Advancies in Computers*, 39:191–237, December 1994.

[55] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework For Parallel Machine Learning. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI '10)*, Catalina Island, California, July 2010.

[56] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *VLDB Endowment*, 5(8):716–727, April 2012.

[57] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, 2017.

[58] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on*

Programming Language Design and Implementation (PLDI '05), Chicago, Illinois, 2005.

[59] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, Indianapolis, Indiana, June 2010.

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

[61] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017.

[62] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating System support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS '09)*, Monte Verita, Switzerland, May 2009.

[63] MongoDB Inc. MongoDB. http://www.mongodb.org/.

[64] D. Narayanan and O. Hodson. Whole-System Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, London, United Kingdom, March 2012.

[65] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC '15)*, Santa Clara, California, July 2015.

[66] N. Neves, M. Castro, and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 1994.

[67] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[68] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[69] J. Ou, J. Shu, and Y. Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016.

[70] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, Piscataway, NJ, USA, 2014.

[71] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, Minneapolis, Minnesota, USA, 2014.

[72] Peter X. Gao and Akshay Narayan and Sagar Karandikar and Joao Carreira and Sangjin Han and Rachit Agarwal and Sylvia Ratnasamy and Scott Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savanah, GA, November 2016.

[73] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. 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.

[74] M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Montaño, and J. 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.

[75] 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 *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, New York, NY, USA, 2009.

[76] U. Ramachandran and M. Y. A. Khalidi. An Implementation of Distributed Shared Memory. *Software:Practice and Experience*, 21(5):443–464, May 1991.

[77] G. C. Richard and M. Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proceedings of 1993 IEEE 12th Symposium on Reliable Distributed Systems*, Oct 1993.

[78] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the Ninth International Conference on Data Engineering*, Washington, DC, USA, 1993.

[79] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, Cambridge, Massachusetts, USA, 1996.

[80] M. Shapiro and P. Ferreira. Larchant-rdoss: A distributed shared persistent memory and its garbage collector. Technical report, Ithaca, NY, USA, 1994.

[81] M. Shapiro, S. Kloosterman, F. Riccardi, and T. Perdis. Perdis - a persistent distributed store for cooperative applications. In *In Proc. 3rd Cabernet Plenary W*, 1997.

[82] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, May 1990.

[83] M. Stumm and S. Zhou. Fault Tolerant Distributed Shared Memory Algorithms. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing (IPDPS '90)*, Dallas, Texas, December 1990.

[84] M. Stumm and S. Zhou. Fault tolerant distributed shared memory algorithms. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing 1990*, Dec 1990.

[85] K. Suzuki and S. Swanson. The Non-Volatile Memory Technology Database (NVMDB). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015.

[86] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. 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.

[87] S.-Y. Tsai and Y. Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26nd ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, 2017.

[88] H. Volos, A. J. Tack, and M. 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.

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

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

[91] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI '12)*, San Jose, California, April 2012.

[92] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[93] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[94] Y. Zhang and S. Swanson. A Study of Application Performance with Non-Volatile Main Memory. In *Proceedings of the 31st IEEE Conference on Massive Data Storage (MSST '15)*, Santa Clara, California, June 2015.

[95] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, March 2015.

[96] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.

[97] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, Seattle, Washington, USA, 1996.