TH-DPMS: Design and Implementation of an RDMA-enabled Distributed Persistent Memory Storage System

JIWU SHU, YOUMIN CHEN, QING WANG, BOHONG ZHU, JUNRU LI, and YOUYOU LU, Tsinghua University

The rapidly increasing data in recent years requires the datacenter infrastructure to store and process data with extremely high throughput and low latency. Fortunately, persistent memory (PM) and RDMA technologies bring new opportunities towards this goal. Both of them are capable of delivering more than 10 GB/s of bandwidth and sub-microsecond latency. However, our past experiences and recent studies show that it is non-trivial to build an efficient and distributed storage system with such new hardware. In this article, we design and implement TH-DPMS (TsingHua Distributed Persistent Memory System) based on persistent memory and RDMA, which unifies the memory, file system, and key-value interface in a single system. TH-DPMS is designed based on a unified distributed persistent memory abstract, pDSM. pDSM acts as a generic layer to connect the PMs of different storage nodes via high-speed RDMA network and organizes them into a global shared address space. It provides the fundamental functionalities, including global address management, space management, fault tolerance, and crash consistency guarantees. Applications are enabled to access pDSM with a group of flexible and easy-to-use APIs by using either raw read/write interfaces or the transactional ones with ACID guarantees. Based on pDSM, we implement a distributed file system and a key-value store named pDFS and pDKVS, respectively. Together, they uphold TH-DPMS with high-performance, low-latency, and fault-tolerant data storage. We evaluate TH-DPMS with both micro-benchmarks and real-world memory-intensive workloads. Experimental results show that TH-DPMS is capable of delivering an aggregated bandwidth of 120 GB/s with 6 nodes. When processing memory-intensive workloads such as YCSB and Graph500, TH-DPMS improves the performance by one order of magnitude compared to existing systems and keeps consistent high efficiency when the workload size grows to multiple terabytes.

CCS Concepts: • **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Additional Key Words and Phrases: Storage system, distributed system, remote direct memory access, persistent memory

ACM Reference format:
Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. 2020. TH-DPMS: Design and Implementation of an RDMA-enabled Distributed Persistent Memory Storage System. *ACM Trans. Storage* 16, 4, Article 24 (October 2020), 31 pages.
https://doi.org/10.1145/3412852

This work is supported by National Key Research and Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61772300, 61832011), and Huawei.

Authors' address: J. Shu, Y. Chen, Q. Wang, B. Zhu, J. Li, and Y. Lu (corresponding author), Department of Computer Science and Technology, Tsinghua University, 30 Shuangqing Road, 201 East Main Building, Beijing, 100084, China; emails: shujw@tsinghua.edu.cn, {chenym16, q-wang18, zhubh18, lijr19}@mails.tsinghua.edu.cn, luyouyou@tsinghua.edu.cn.

1553-3077/2020/10-ART24 $15.00
https://doi.org/10.1145/3412852

1 INTRODUCTION

From 2013 to 2020, the digital universe grew by a factor of 10, from 4.4T gigabytes to 44T, which more than doubles every two years [5]. Today, there are 30B connected devices generating unprecedented amounts of data, which requires the datacenter infrastructure to collect, process, store, and analyze data with extremely high throughput. Meanwhile, emerging datacenter applications, such as e-commerce, autopilot, and financial trading, are rapidly evolving from simple data-serving tasks to sophisticated analytics in response to real-time queries [65], which brings great challenges to the storage system design.

To minimize the response latency and maximize the processing throughput, a number of recent projects are proposed to keep the data directly in DRAM (e.g., Spark [90], SAP-HANA [31], and Redis [11]). Compared to external devices (e.g., SSD/HDD), DRAM delivers orders of magnitude lower latency (~60 ns) and much higher bandwidth (~100 GB/s). However, DRAM has its own limitations: (1) *Low capacity* – a single DRAM DIMM only has up to 128 GB of space; (2) *High power consumption* – DRAM consumes as much as half of the total system power in a computer [33]; (3) *Expensive* – 1 GB of DRAM costs \$7.6, which is 20× higher than that of NAND Flash and 200× higher than hard disks; (4) *Volatile* – DRAM requires power to maintain the stored information, so when the power is interrupted, the stored data are quickly lost; and (5) *Unstable performance* due to the background refreshing. What's more, today's datacenters are built on top of monotonic servers connected with commodity networking technology; node-to-node communication delay can be as high as 100 μs. Therefore, deploying distributed in-memory storage in datacenters is less attractive: moving the data from disk to DRAM yields a 100,000× reduction in latency, but distributing the memory through commodity network eliminates 1000× of such benefit [65]. These shortcomings have hindered its large-scale deployment in the industry.

Emerging non-volatile memory (NVM) technologies, such as PCM [49, 72, 94], ReRAM [14], and 3D XPoint [4], are capable of providing data persistency while achieving close performance and much higher density than DRAM (see Table 1 for details). By attaching directly to the memory bus, NVMs can be accessed via byte-addressable Load/Store instructions. Hence, such NVMs are also called as persistent memory (PM). Intel Optane DC Persistent Memory Modules (DCPMM), as the only available PM device in the market, began shipping in 2019. Moreover, remote direct memory access (RDMA) reduces latency by directly accessing the remote memory at user-level without the involvement of remote CPUs and offloading network processing to the adapter. Recent ConnectX-6 adapters are capable of delivering 200 Gbps of bandwidth and sub-microsecond latency [1]. These attractive features make them promising to build large-capacity, high-performance, and low-latency memory storage system.

Over the past decade, a large number of works designed file systems [19, 22, 23, 25, 27, 30, 44, 48, 67, 69, 77, 82, 84, 85, 91, 93], data-structures [18, 35, 50, 62, 70, 76, 83, 89, 95], and new programming models [9, 24, 36, 55, 57–59, 78] for NVMs, and tried to use RDMA in distributed systems to improve their performance [20, 28, 29, 40–42, 51, 52, 61, 65, 71, 71, 79, 80]. However, it is non-trivial to build large memory storage system using persistent memory and RDMA simultaneously:

- **Lack of abstraction.** Different persistent systems include similar functions. For instance, both key-value stores and file systems require functions such as memory allocation, crash consistency, and fault tolerance. However, these functions are independently designed in different systems, which are redundant. Meanwhile, different applications in a data center prefer different interfaces, including file systems, key-values, or even memory interface. Therefore, an abstraction is necessary for different systems to coexist efficiently.
- **Remote persistence.** To ensure data persistence and crash consistency, storage systems typically use hardware instructions (e.g., `clwb/clflushopt`) to flush the modified data from

Table 1. Comparison of Different Storage Technologies [13, 30, 34, 78]

	Optane DCPMM	DRAM	NVDIMM	Optane SSD†	NAND FLASH‡	HDD
Capacity*	Up to 512 GB	Up to 128 GB	100s of GBs	Up to 1 TB	Up to 4 TB	Up to 14 TB
Read Lat.	300 ns	10 - 20 ns	50 ns	9 μs	35 μs	10 ms
Write Lat.	150 ns	10 - 20 ns	150 ns	30 μs	68 μs	10 ms
Price	$4/GB	$7.6/GB	$3-13/GB	$1.30/GB	$0.38/GB	$0.03/GB
Addressability	Byte	Byte	Byte/Block	Block	Block	Block
Volatility	Non-volatile	Volatile	Non-volatile	Non-volatile	Non-volatile	Non-volatile

*Per module; †Intel Optane SSD 905P Series (960 GB) (AIC PCIe ×4 3D XPoint).
†Samsung 960 Pro 1 TB M.2 SSD with 48-layer 3D NAND (Source: Wikibon).

volatile CPU cache to PM. However, it is challenging to ensure such guarantee when PM space is exported to remote servers via RDMA, since local CPUs are unaware of such remote write events.

- **Microsecond-scale software design.** Developing storage systems with such fast hardware devices requires refined software design. First, CPU, which was not supposed to be the bottleneck, becomes overloaded when accessing to PM or transferring data via RDMA; thus, any waste of CPU resources, i.e., OS thread schedule, will block potential of fast hardware devices from unleashing. Second, since latency of RDMA and PM is microsecond-scale, any subtle operations such as cross-NUMA access can bring unacceptable latency. Hence, mechanisms such as user-level polling, coroutine-based task handling, and NUMA-aware policies could be applied in the system design to improve efficiency.

In this article, we introduce a generic layer to abstract the distributed persistent memory layer. On top of this layer, we build traditional storage systems with legacy APIs (e.g., file, key-value) to support existing applications; meanwhile, some emerging applications can interact with it directly by leveraging the memory interface. With this, we design and implement TH-DPMS. The generic layer named pDSM (persistent distributed shared memory) connects the persistent memory of different nodes via high-speed RDMA NIC. pDSM differs from traditional DSM systems [17, 47, 53] in that it is far more than just a global address manager. Instead, it is a full-fledged abstraction of distributed persistent memory with high efficiency, crash consistency, fault tolerance, and comprehensive interfaces. Specifically, pDSM consists of six major subsystems:

(1) iRDMA. It is an efficient and flexible RDMA-based network primitive that supports RDMA-based RPC, light-weight remote copying, and remote persistence.
(2) Global shared address management. Similar to traditional DSM systems, pDSM organizes the distributed PM into a global address space for unified accessing, but it adopts a 2-GB-grained paging mechanism to reduce the overhead of address management.
(3) Centralized monitor is responsible for managing a global view (e.g., global segment table management, handover when configuration changes).
(4) Space management. We use two approaches to manage the 2 GB segments, including a persistent allocator (i.e., PAllocator) for fine-grained space allocation (e.g., metadata) and an object store to store large data chunks.
(5) Replication system. The 2 GB segment is the minimal unit used in TH-DPMS for replication. It differs from the traditional approach in that it replicates small-sized metadata via an operation log [16], so the networking overhead can be amortized by batching.
(6) Distributed transactional system. As the main contribution, pDSM enables the applications to access distributed PM spaces through transactional interfaces. Our concurrency

control protocol is based on FaRM [29], despite that crash consistency is the main consideration in our implementation.

Based on the above subsystems, pDSM abstracts a bunch of flexible and easy-to-use APIs, enabling applications to access distributed PM either by directly using raw read/write interfaces or through transactional interfaces with ACID guarantees. As two examples, we implement a distributed persistent memory file system (pDFS) and a distributed key-value store (pDKVS) based on pDSM. The existence of pDSM layer significantly enhances productivity when we implement these two storage systems. Meanwhile, our evaluation also shows that both pDFS and pDKVS are capable of delivering bandwidth and latency approaching the hardware performance. When running large-scale and memory-intensive workloads such as YCSB and Graph500, TH-DPMS improves the performance by one order of magnitude. It also keeps consistent and high efficiency as the workload size grows to multiple terabytes.

2 BACKGROUND

2.1 Optane DCPMM

Intel Optane DC Persistent Memory Module (Optane DCPMM), the first PM product, was released in April 2019. It is attached to the memory bus and can be accessed via CPU Load/Store instructions. Optane DCPMM has asymmetric read/write performance: With six Optane DIMMs, the overall read bandwidth can reach 37.6 GB/s, while the write bandwidth peaks only at 13.2 GB/s. Besides, Optane DCPMM shows higher read latency (~300 ns), which is much higher than that of DRAM. Programming in PM is also quite different. CPU issues writes to PM in 8-byte failure-atomic unit, which is smaller than that of HDDs or SSDs (which are 512-B sectors or 4-KB pages). This requires extra techniques (e.g., redo/undo logs) to achieve atomicity. In PMs, these writes are first cached in the volatile CPU cache and are then written back to the PM in an arbitrary order. Guaranteeing the consistency of data requires us to order the writes. Such ordering is ensured from the following two aspects: (1) flushing the cache line via `clflush`, `clwb`, or `clflushopt` instructions, to ensure that the order of writes is consistent with the actual order when they reach the PM controller; (2) issuing a `mfence` after each write, to ensure that the current write becomes visible before any other following writes. The small-sized failure-atomic unit and the ordering constraints make it more challenging to design consistent system software.

2.2 Remote Direct Memory Access

RDMA, as its name implies, can directly access remote memories without the involvement of remote CPUs. It reduces end-to-end latency by enabling data transfers over InfiniBand and Converged Ethernet fabrics. RDMA can be configured into three types of connections, which are reliable connection (RC), unreliable connection (UC), and unreliable datagram (UD). Among them, RC and UC only support one-to-one communication paradigm, so they need to create separate queue pairs (QP) for different connections. Instead, UD supports one-to-many data transferring, but it has limited MTU size, which is only 4 KB. RDMA supports two types of verbs, which are one-sided (a.k.a., memory semantic) and two-sized (a.k.a., message semantic). One-sided verbs (e.g., read, write, atomic) can directly access remote memory, while two-sided verbs (e.g., send/recv) require both sides to get involved. `write-with-imm`, as a special verb, shares the characteristics of both one- and two-sided verbs: It can write remote memory directly like one-sided verb does, but requires the remote server CPU to post a `recv` verb beforehand, to receive the 32-bit immediate value. As shown in Table 2, different kinds of verbs are supported differently in each connection mode.

Table 2. RDMA Verbs and MTU Sizes in Different Connection Modes

Type	Send/Recv	Write[-imm]	Read/Atomic	MTU
RC	✔	✔	✔	2 GB
UC	✔	✔	✗	2 GB
UD	✔	✗	✗	4 KB

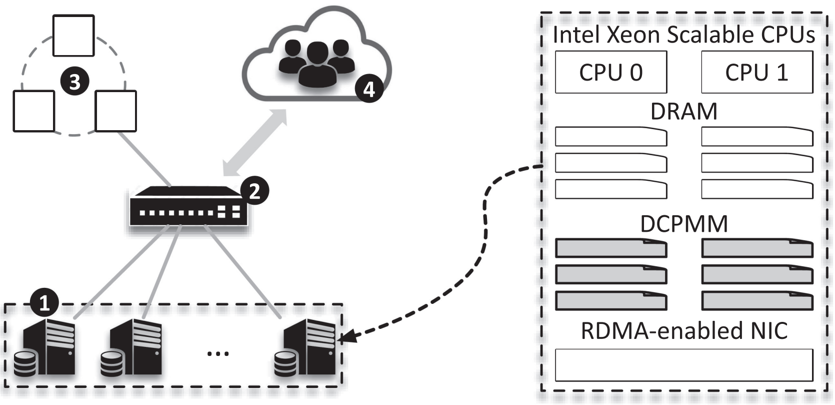

Fig. 1. Hardware configuration of TH-DPMS (❶: TH-DPMS PM servers; ❷: RDMA-enabled high-speed switch; ❸: Global monitor; ❹: Client nodes that run applications.)

3 THE DESIGN OF THE TH-DPMS

TH-DPMS is a high-performance, low-latency, and crash-consistent distributed memory storage system. It achieves these goals by leveraging the RDMA-capable networks and persistent memory with redesigned system software. It proposes a novel component named persistent Distributed Shared Memory (pDSM) with global consistent address space. pDSM abstracts the common features that a storage system must provide (e.g., space management, failure-atomic update protocol, replication) to reduce redundant data copy and simplify the software stack.

3.1 Hardware Configuration

Figure 1 illustrates the hardware configuration of TH-DPMS. TH-DPMS consists of four parts, which are ❶ the TH-DPMS PM servers, ❷ the RDMA-capable high-speed switches, ❸ the centralized monitor that does coarse-grained coordination out of the critical path, and ❹ the client nodes running applications.

The TH-DPMS PM servers, also the key parts of the TH-DPMS system, are depicted in detail at the right part of Figure 1. The PM servers are equipped with the actual persistent memory—Intel Optane DC Persistent Memory [4]—and second-generation Xeon Scalable processors. Optane DCPMMs are configured in 100% App Direct mode [38], so TH-DPMS has direct byte-addressable access to the PM. Recent work [88] points out that NUMA effects for Optane are much larger than they are for DRAM, so designers should work even harder to avoid cross-socket memory traffic. Regarding this, each PM server is equipped with two Mellanox 100 Gbps network adapters to reduce the overhead of cross-NUMA data transferring. Other parts (centralized monitor and client nodes) only have one adapter, since the network is not the bottleneck. Table 3 reports the relevant details of each PM server. Note that TH-DPMS can work perfectly with arbitrary number

Table 3. Configuration Details of the PM Server Node and the RDMA Switch

CPU	Type	2× Intel Xeon Gold 6240M
	# of physical cores	36 in total
	Frequency	3.3 GHz
	Caches	L1: 32 KB Icache, 32 KB Dcache L2: 1 MB, L3: 25 MB (shared)
MEM	PM Capacity	1.5 TB (256 GB/DIMM)
	PM Read Latency	302 ns (random 8-byte read)
	PM Write Bw	13.2 GB/s (4 KB sequential)
	DRAM Capacity	192 GB (32 GB/DIMM)
OS	Release Version	Ubuntu 18.04.3 LTS
	Kernel	Linux 4.15.0
NIC	2× Mellanox MCX555A-ECAT 100 Gbps, single port	
Switch	Mellanox MSB7790-ES2F 100 Gbps, 36 ports	

of network interfaces, we choose to deploy two network interfaces only to eliminate the cross-NUMA effects.

The centralized monitor is responsible for maintaining a global view of the cluster. The clients, which run the actual applications, access the TH-DPMS with the given APIs. The TH-DPMS PM servers, the monitor, and the clients are connected via an RDMA-capable 100 Gbps switch.

3.2 Overview of TH-DPMS

Figure 2 describes the software architecture of TH-DPMS. The key component, pDSM, is highlighted with gray color. It connects the PM devices of different PM servers via the RDMA network and organizes them into a globally consistent address space. In this way, the clients can transparently access remote persistent memory via pDSM. Based on the global address space, pDSM abstracts the common components that most storage systems rely on (e.g., naive heap, replication, transaction, and object), and provides a bunch of flexible APIs to fulfill the requirements of different storage systems (e.g., file system, key-value store, database).

From the bottom up, pDSM first incorporates an RDMA-based communication primitive named iRDMA. Since the storage systems transfer data in different ways and impose different performance requirements to the network, iRDMA is designed with the following considerations: (1) zero-copy data transferring with high bandwidth; (2) low latency message passing by implementing a customized RPC, which is required when sending light-weight command messages or accessing metadata with small payloads; (3) providing *remote persistence primitive* to enable persistence of updates to remote byte-addressable persistent memory, since the visibility of RDMA updates on the remote server is not the same as persistence of those updates [46]. With iRDMA, pDSM can organize the PM devices of different PM servers into a unified address space, just as the traditional DSM system does. Similar to existing virtual memory management in the operating system, TH-DPMS uses *segment* to organize the address space, but at much coarser granularity (i.e., 2 GB in our implementation). It also assigns each segment extra permission bits to enforce fine-grained access control.

The pDSM space is managed in two ways: One is the naive heap, which uses the global address from the bottom up; the other is the object space that uses the address in the opposite way. We use address size of 128-bit, so the two spaces are less likely to overlap with each other. The naive

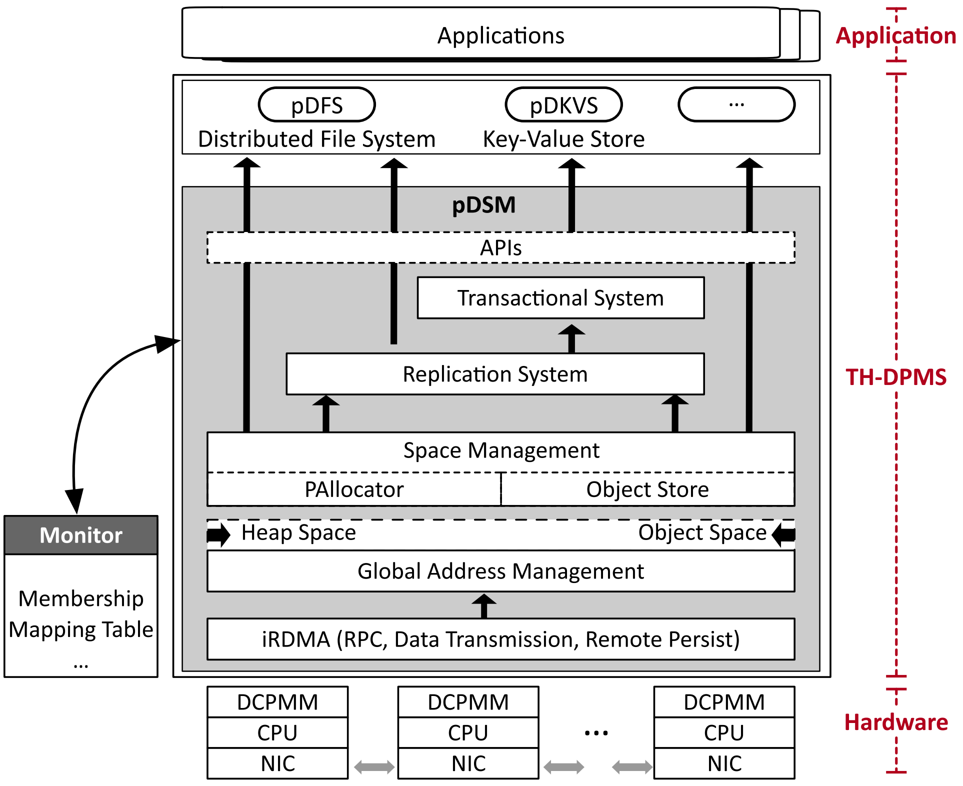

Fig. 2. Software architecture of TH-DPMS.

heap space is managed with a persistent and consistent allocator to serve those fine-grained space allocation requests (e.g., allocate space for metadata and other small-sized items); the object space, instead, stores coarse-grained data (e.g., file data). To improve the reliability and fault tolerance, both the naive heap and the object space are replicated via the replication system. It replicates the naive heap and objects in different ways: The small-sized items in the heap space are synchronized to the remote nodes via an operation log [16], so the networking overhead can be amortized by batching among multiple accesses; the objects, which have larger sizes, are mirrored to other replicas by the clients directly via a primary-backup replication protocol. Finally, pDSM incorporates the distributed transactional system, so applications can update data with ACID guarantees by using the transactional interfaces.

Note that the standalone monitor is introduced to maintain a global view of the cluster. Such global view includes the membership status, global address mapping table, mappings of replica, as well as load information, and so on. As the view change events happen infrequently and most of the actions can be done out of the critical path, such centralized design does not hurt the overall performance too much (see Section 4.1.3).

With pDSM, upper-layer storage systems are capable of selecting the most suitable interfaces (or functional components) to implement their internal mechanisms. For instance, a distributed file system can use the naive heap with the transactional interface to manage its metadata and put the file data in the object store and replicate them to remote servers via the replication system. Similarly, a key-value store typically consists of an index structure (e.g., hash table or a b-tree) and the actual key-value items. Therefore, the index items, as well as the small-sized key-value items, can be put in the naive heap directly, and large key-value items are stored separately by the object store.

The insight behind the design of TH-DPMS lies in the following aspects:

(1) **High performance.** With the abstraction of the pDSM layer, the clients are capable of directly pushing/pulling data to/from the storage system without redundant data copy overhead as in traditional network/storage stacks. It also adopts different mechanisms to process small and large data items, in terms of space management and replication.
(2) **Simplified software stack.** The common components, such as the space management, failure-atomic update protocol, fault tolerance, networking, and so on, are required by most storage systems. TH-DPMS incorporates the pDSM layer to manage the above functionalities, and upper layer software can directly use its APIs. Therefore, the software redundancy is eliminated.
(3) **Flexible APIs.** TH-DPMS provides a bunch of APIs. Applications can choose different interfaces based on their own requirements. An application can directly access the naive heap without any crash consistency guarantees. One can also use high-level APIs (e.g., transactional interfaces) to avoid the burden of implementing them from scratch.
(4) **Failure isolation.** External storage devices (e.g., SSD/HDD) are managed by the kernel, and any accesses to them are carefully authenticated. Persistent memory, instead, can be accessed arbitrarily by applications in user space arbitrarily. So, a buggy process may corrupt other applications by introducing stray writes to the PM. TH-DPMS avoids this by managing all the PM space from the cluster, and any accesses to PMs are authenticated by pDSM (by checking their permissions).

4 IMPLEMENTATION

For performance considerations, TH-DPMS is implemented from scratch without the help of previous tools like PMDK [9]. In this part, we describe how TH-DPMS is implemented and illustrate the key techniques it introduces.

4.1 pDSM: Persistent Distributed Shared Memory

pDSM, as the key component in TH-DPMS, mainly consists of six parts, which are the *networking*, *global address management*, *centralized monitor*, *space management*, *replication system*, and *distributed transaction*. In this section, we describe the implementation of pDSM by introducing the above six subsystems.

4.1.1 iRDMA: Flexible and Efficient RDMA-based Network Primitive. TH-DPMS uses Reliable Connection (RC) for inter-node communications, as RC supports reliable and variable-sized data transferring [20]. The connections (i.e., queue pairs) are established in the following way: (1) Each PM server assigns its cores with unique ID starting from zero. Each core creates QPs with the cores on remote servers that own the similar ID (with non-uniform machines in cluster, the connection is made by using mod operations for the core numbers for mapping). In TH-DPMS, communications between PM servers occur only between cores with the same ID. (2) A client creates a QP with each PM server by randomly selecting a server core based on a manifest pre-aquired from the monitor. (3) Each PM server core and each client create a QP with the monitor. Such coreID-to-coreID connection mapping is based on tradeoffs made between QP connection scalability and CPU races. Furthermore, for the RDMA RC QP scalability, we use ConnectX-5 IB NICs, which supports 60K connections with less than 10% performance drop. When using coreID-to-coreID connection mapping, it could support up to 2K servers for 36-core servers.

To support efficient communications in TH-DPMS, we implement the following three networking primitives:

- RDMA-based RPC. RPCs are used to transfer messages with small payloads (e.g., metadata requests). We choose RDMA write-with-imm verb to send both request and response messages. write-with-imm allows us to encapsulate a 32-bit immediate value in the message to notify the remote cores of the arrival of remote write access. We leverage the 32-bit immediate field to store the address (i.e., offset) of the written data. Hence, a remote server core does not need to scan the memory buffer repeatedly to discover new messages [56].
- Light-weight Remote Copy. RDMA supports zero-copy data transferring. However, we find it is hard to achieve zero-copy between the application's buffer and storage image: application's buffer is temporarily allocated without being registered to the NIC. Hence, we need to register them before each time issuing the remote copy. However, the registration function (ibv_reg_mr()) is time-consuming. Hence, we choose to build a buffer pool as an in-transit area, which is registered beforehand to serve remote copy. To write data remotely, for example, the data are first copied to the buffer pool and then transferred to remote node.
- Remote Persistence. Remote persistence is an important primitive in persistent memory storage system. During an RDMA transferring, there are many volatile buffers between the two hosts (NIC cache, LLC, memory controller, etc.). Hence, we need a way to ensure that the transmitted data have been persistently stored on remote servers. In TH-DPMS, to send data with persistence requirements, we directly use the RDMA write-with-imm verb. This verb supports remote direct write and is capable of notifying remote side with a 32-bit immediate value. Hence, the receiver can actively persist the newly written data once the data have been transferred, and the `remote persistence` primitive is achieved with only one network round-trip. However, the immediate field is too small to store the size and the address of the written data. Fortunately, we observe that it is common in storage systems for a client to post a RPC beforehand to find the metadata before actually accessing data. Hence, remote server can record such information locally when it receives such RPCs and use such information to find the written data when it is notified by the immediate value. When the above condition cannot be met, remote persistence is achieved by legacy RPCs.

4.1.2 Global Address Management. TH-DPMS organizes the PM spaces in the cluster via a two-layered paging mechanism. TH-DPMS first cuts the PM space of each PM server into 2 GB segments, which is the minimum granularity that global address manager (i.e., the monitor) manages. Each segment also works as the minimum unit for replication, isolation, and recovery. Within each 2 GB segment, TH-DPMS utilizes the PAllocator (Section 4.1.4) and the object store (Section 4.1.5) for finer space management.

As shown in Figure 3, the global address is formatted as $<\mathrm{Descriptor}, \mathrm{Segment\ No.}, \mathrm{In\text{-}segment\ Offset}>$ with 128 bits. Among them, the descriptor describes the internal attributes of a 2 GB segment, such as the permission bits, the number of replicas, and so on. Similar to existing memory management in the operating system, the global address is indexed via a page table-like structure, named mapping table in this article. The mapping table is persistently stored at the monitor. The accessed mapping items are also cached at each PM server to reduce the network round-trips when accessing the global address space. Each mapping item has the following format (three replicas, for example): $< \mathrm{S_d}, \mathrm{S_{no}} > \Rightarrow < (\mathrm{N_i}, \mathrm{off_i}), (\mathrm{N_j}, \mathrm{off_j}), (\mathrm{N_k}, \mathrm{off_k}) >$, which indicates that a 2 GB segment with segment number of $\mathrm{S_{no}}$ and segment descriptor of $\mathrm{s_d}$ are mapped to three PM servers, with node ID of $\mathrm{N_i}$, $\mathrm{N_j}$, and $\mathrm{N_k}$, respectively, at their corresponding offset of $\mathrm{off_i}$, $\mathrm{off_j}$, and $\mathrm{off_k}$. The first node in the mapping item always indicates the primary replica, while the other two represent the backup replicas.

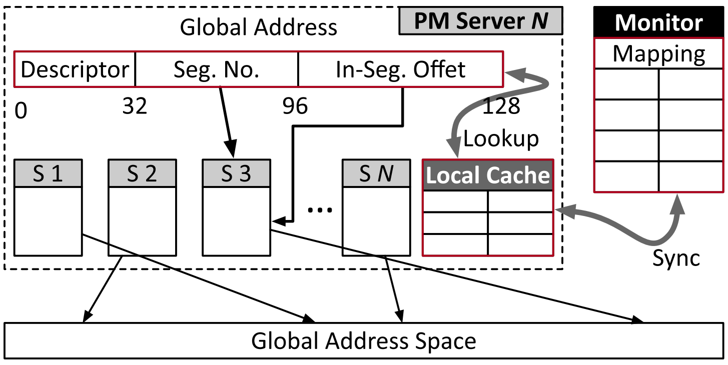

Fig. 3. Global Address Space management.

We choose 2 GB as the default segment size for two reasons: (1) it is big enough, so the mapping table will not grow extremely large. For example, the mapping table of a 1000s TB PM pool consumes merely several megabytes of space, which can perfectly fit into last level cache; (2) the 2 GB segment is also small enough to support fine-grained access control and efficient replication. Note that the size of a physical segment does not impact the overall performance directly, because allocations on critical path are actually accomplished by PAllocator and object store.

To enable zero-copy data transferring, TH-DPMS needs to register the PM space to the NIC to enable remote direct memory access. However, it is dangerous to register all the PM spaces and export them to the clients, since a malicious client can easily corrupt the data by writing some arbitrary data. In TH-DPMS, we cut the PM space into 2 GB segments and register each segment separately to the NIC to generate different keys for them. These keys are stored along with the mapping items, and the monitor is responsible for managing these keys. The monitor only assigns the keys to the clients that have access permissions. In this way, a client cannot read/write the segment that it does not have the access rights. Currently, we do not support revoking permission on clients. Using contiguous 2 GB segments for replication also greatly simplifies the software design: RDMA enables us to push the data to different replicas using one-sided verbs without the involvement of remote CPUs (see Section 4.1.7).

4.1.3 Centralized Monitor. TH-DPMS relies on the centralized monitor for cluster-wide coordination, which ensures that all the PM servers agree on a consistent global configuration. To prevent the monitor from impacting the overall storage performance, we choose three standalone servers to run the monitor instance, and they are synchronized using the Raft protocol [68]. One of the servers is elected to work as the leader and accepts the requests from the PM servers. The monitor is involved in the following procedures:

Mapping Table Management. Similar to existing virtual memory management, the global address space is never mapped to any physical PM space when TH-DPMS is initialized. We describe how such mapping is established by illustrating the case below: The client issues a global address request, e.g., space allocation, to one of a PM server (e.g., N_i), TH-DPMS follows the steps below:

(i) Upon receiving the alloc request, the PM server first checks the permissions of this client and finds the existing local 2 GB segments available for allocation. Since this is the first start, the PM server reserves a 2 GB segment locally from its mapped PM space. It then sends a request with node ID, offset, and registered key, to the monitor.
(ii) The monitor finds an available global address from the mapping table and assigns it to this PM segment.

(iii) The monitor also sends request to some of the PM nodes to reserve the backup segments. The backup nodes are randomly selected from those nodes other than the primary node.
(iv) The monitor inserts the mapping item to the mapping table and returns it to the PM server.
(v) The PM server caches the mapping item locally when it receives the response message. It then processes the request by allocating from the newly reserved segment and finally returns back the allocated global address.

Configuration Changes and Client Cache Management. Once the membership changes, e.g., a new PM server joins in, or one of the PM server crashes, the monitor is responsible to renew the configurations, so TH-DPMS can consistently move to a new state with minimal effects to the online services.

We deploy multiple monitor servers, which are synchronized by the Raft protocol, so they can tolerate at most f node failure. Hence, we assume that the monitor service is always reliable. The monitor uses heartbeat to check the liveness of each PM server. It maintains a global epochID, which is increased each time the membership is changed. For example, when a PM server crashes, the monitor increases the epochID first and generates a new version of mapping table, where all the segments residing in the crashed server are remapped to the new places. After this, the monitor broadcasts the new epochID to all the PM servers. The PM servers with new epochID then update their configurations (mapping, etc.) and fetch the corresponding PM segments that remapped to it from either primary or backup replicas. During the reconfiguration phase, all requests to the remapped segments are blocked until they have been replicated to new places. Client cache can also be managed by this global epochID; with such epochID embedded in cache, servers can determine whether a request from client side is made based on expired cached information and reject it. Note that there is a short window during which a node crashes but a new epochID is not generated by the monitor yet. If a client posts requests (either RPCs or one-sided verbs) to the crashed node during this window, these requests are aborted actively after a timeout.

Since the centralized monitor only performs coarse-grained management, it is not the bottleneck. For the mapping table management for 2 GB segments, only allocation or free requests to the 2 GB segments go to the monitor, so the requests are infrequent. For the configuration changes, the requests are also infrequent due to infrequent configuration changes.

4.1.4 PAllocator. The naive heap described in Section 3.2 uses the lower address space with increasing order. It is managed with a persistent allocator named PAllocator to serve those fine-grained allocation requests.

As shown in Figure 4, PAllocator adopts a Hoard [15]-like layout. It first cuts the 2 GB segments into 4 MB chunks. Further, the 4 MB chunks are cut into different classes of data blocks. Note that the data blocks in the same chunk have the same size, and such cutting size is persistently recorded at the head of each chunk when it is ready for allocation. A bitmap is also placed at the head of each chunk to track the unused data blocks. To boost performance of allocation, we introduce in-memory free-lists to track the unused data blocks of different classes. Hence, we can get a free block by referring to the head of the free list, instead of scanning the bitmap linearly. We also use a global allocation table to track the free lists of all the segments. Considering the scalability issue, these 4 MB NVM chunks are partitioned to different server cores, and each server core owns a global allocation table. After receiving an allocating request, PAllocator first chooses a proper class of NVM chunk from its private allocation table and then allocates a free data block by modifying the bitmap and the free list.

In each allocation, we do not need to flush the bitmap when it is modified, since they can always be recovered to a consistent one with our design: The starting address of each chunk is

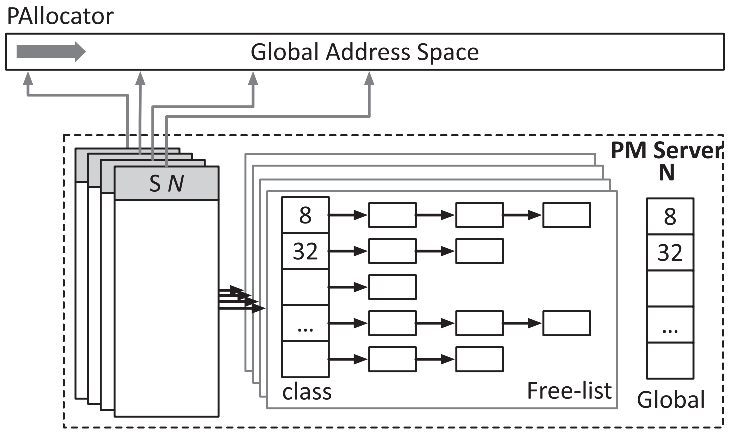

Fig. 4. Multi-class data format with PAllocator.

4-MB-aligned, and the allocation granularity of each chunk is specified at the head of it. Therefore, the offset of an allocated NVM block in the chunk can be calculated directly with the global address, enabling us to recover the bitmap even when they fail to be persisted before system crashes. Besides, the latency of RDMA ($\sim 1\mu$s) is still much higher than local PM accesses. Hence, TH-DPMS requires the PM server to always allocate spaces from local PM space, despite that we implement a global address space. Remote allocation occurs only after the local space has been used up. When a client allocates memory space from pDSM directly (e.g., the application running on the client node needs to use the shared memory space), it chooses a PM node randomly and then sends the allocation request to this node.

4.1.5 Object Store. The object store uses the higher address space with decreasing order, which is introduced to manage the large data chunks (e.g., file data). A unique 128-bit ObjectID is assigned to each object. In TH-DPMS, we use consistent hashing algorithm [45] to distribute the objects to the PM servers. The virtual node in consistent hashing is called table in TH-DPMS, which manages a group of objects and acts as the minimum unit for data migration. The objects in a table are stored in one or multiple 2 GB segments, and each segment can only store the objects belonging to the same table. We do not allow a table to span across multiple servers to avoid introducing distributed data structures. Once we need to migrate data from one PM server to another, all the PM segments related to this table are moved away.

Figure 5 shows how a table is formatted. The objects in the table are indexed via a hash table (for simplicity, we choose the chained hash policy). The hash table is persistently stored in PM, whose space (including both the table and the chained items) is allocated via the PAllocator. By referring to the hash table, we can find the corresponding object with the given objectID. In TH-DPMS, each object has a maximum size of 64 MB, consisting of a group of 4 KB pages. The object uses a skip list as the mapping to its physical data pages. We choose skip list as the index metadata of each object, because it has multiple layers of linked list-like data structure. Each higher layer acts as an "express lane" for the lower list layer. The list-based organization enables O(logN) of search complexity as well as atomic data update by performing pointer manipulations. Each node in the skip list stores an extent, pointing to contiguous 4 KB data pages. Overwriting to an object is processed in a copy-on-write way: We always redirect the modified data pages to new places and then atomically update the skip list to point to new pages. Since old data pages are reclaimed only after the modification is finished, the crash consistency is guaranteed.

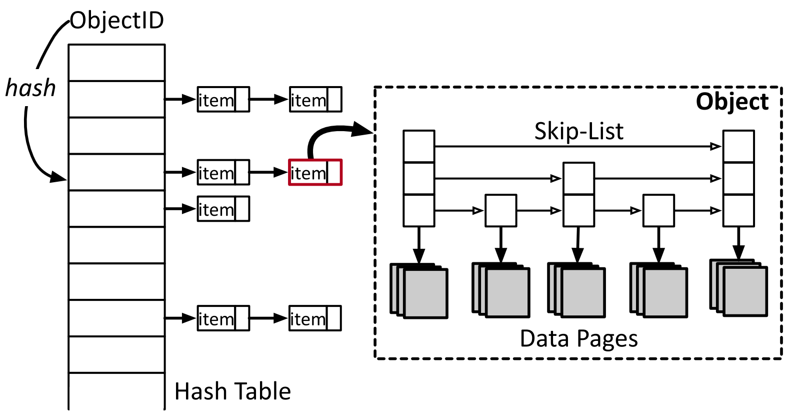

Fig. 5. Soft architecture of the object store.

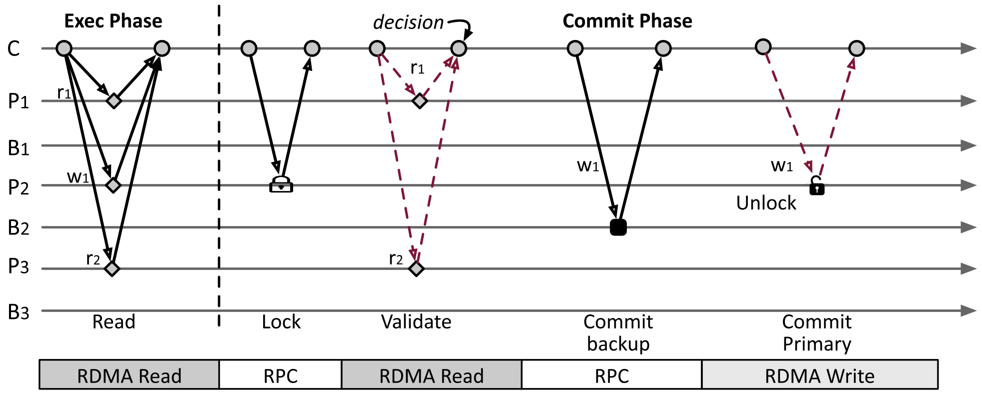

Fig. 6. The transaction commit protocol, including one coordinator (i.e., C), three participants (i.e., P_1, P_2, P_3), and three backup nodes (i.e., B_1, B_2, B_3). Different network primitives are used at different stages.

4.1.6 Distributed Transaction. We incorporate the FaRM [29] transaction protocol to provide ACID properties in TH-DPMS. FaRM integrates the transaction and replication protocols to reduce network messages, thus improving the performance. It exploits one-sided RDMA reads and writes for CPU efficiency and low latency and uses primary-backup replication for simplicity. To further reduce the network messages and simplify the transaction protocol, the coordinator in FaRM is unreplicated and stateless. FaRM enables this by preparing a log area at each PM server for each coordinator, and the coordinator writes all the related log states to remote participants. Hence, a transaction still can be efficiently recovered and decided whether to commit or not, based on the log state at each participant. FaRM uses optimistic concurrency control with read validation to coordinate the concurrent transactions. We slightly modify FaRM's protocol to support persistent memory.

Figure 6 shows how a transaction with one write item and two read items is processed. For simplicity, the figure only shows one backup (i.e., B_1, B_2, B_3) for each data item.

(1) *Exec phase.* During the execution phase, the coordinator uses one-sided RDMA read to fetch all the items referred to by this transaction.

(2) *Lock.* After the transaction has been executed locally, the coordinator sends a lock request to each participant of primary replica for any written items. The request encapsulates the modified data items as well, which are durably stored in the log by the participants. The participants then acquire the lock of these items and reply to the coordinator.
(3) *Validation.* When all the locks in the write set have been acquired, the coordinator then uses RDMA read to check the versions of the items read by this transaction. The validation fails if the version of any item has changed.
(4) *Commit backup.* Different from FaRM, committing the modified data to the backups is completed via RPCs, instead of one-sided RDMA writes. FaRM assumes that all the data are protected by backup batteries, so the data in the volatile cache can survive power failure events. However, this is not the case in TH-DPMS, so we need to persist the log synchronously before moving to the next step. Note that the proposed *remote persistence* primitive is not used here, since it is asynchronous, which still requires an extra RPC to confirm whether the data have been persisted or not.
(5) *Commit primary.* Finally, the coordinator sends the commit requests to the primary participants in the write set to commit the transaction.

The coordinator sends requests in the *Lock*, *Commit backup*, and *Commit primary* by directly writing them to remote log area, which is durably stored at each phase. Hence, the unfinished transactions still can be recovered after system/power failures. As the coordinator is aware of the state of each (un)committed transaction, it can truncate the remote logs without any coordination with remote servers.

4.1.7 Replication. The 2 GB PM segments are the minimal unit for replication in TH-DPMS. As described in Section 4.1.3, each primary segment can be configured to have zero to multiple backup segments stored in other PM nodes. We use different ways to replicate data between primary and backup segments. For fine-grained updates (e.g., update the heap area), we force the applications to use the transactional interface. Our transactional system is designed to be closely coupled with replication. The operations are synchronized to different replicas via the log information recorded during the transaction execution. Coarse-grained updates (e.g., write a new object) are transferred to backup segments directly via one-sided RDMA write verbs. Note that replicating data does not require a consensus protocol: (1) In TH-DPMS, all the data are out-of-place updated, so old version of data still keeps consistent when system crashes before a write finishes. (2) Only after the data have been replicated, the corresponding metadata are updated by executing a transaction.

4.1.8 Interfaces. Table 4 summarizes the APIs that TH-DPMS provides. These interfaces are classified according to the sub-components (i.e., global address space, PAllocator, object store, and transactional system) they belong to. Among them, TH-DPMS provides two APIs to directly access the global address space, which are `read` and `write`. These two raw interfaces deliver the hardware performance to applications, but without any guarantee on crash consistency and atomicity. *PAllocator* is responsible for managing the global address space for those fine-grained accesses. It provides three APIs to allocate and free memory buffer from/to the persistent memory space. Applications access the object store via the `put/get/del_obj` interfaces when they need to update, read, or delete objects. Finally, TH-DPMS enables transactional access to the pDSM. All the aforementioned APIs can be wrapped between the `tx_begin()` and `tx_end()` to deliver ACID properties.

4.2 pDFS: A Distributed File System

pDSM's memory-oriented APIs fundamentally change the way that applications access the persistent data. Applications are capable of directly manipulating the data in PM with memory format,

Table 4. Interface Set of pDSM

Components	Interfaces	Descriptionn
Global Address Space Accessing (§4.1.2)	`bool read(addr_t p, void *buf, size_t size)`	Read data from global address p to user buffer buf with size of size.
	`bool write(addr_t p, void *buf, size_t size)`	Write data from user buffer buf to global address p with size of size.
PAllocator (§4.1.4)	`addr_t pallocate(size_t size)`	Allocate a memory buffer with size of size from the global address space.
	`addr_t pallocate(size_t size, addr_t affi_addr)`	Allocate a memory buffer from a physical node that also maps to address `affi_addr`.
	`bool pfree(addr_t p)`	Free a memory buffer whose global address is p
Object Store (§4.1.5)	`bool put_obj(id_t id, val_t value)`	Insert an object with ID of id, create a new one if it does not exist.
	`val_t get_obj(id_t id)`	Read an object with ID of id.
	`bool del_obj(id_t id)`	Delete an object with ID of id.
Transactional System (§4.1.6)	`void tx_begin()`	Declare the start of a transaction.
	`void tx_end()`	Declare the end of a transaction.
	`void tx_commit()`	Commit a transaction.
	`void tx_abort()`	Actively abort a transaction.

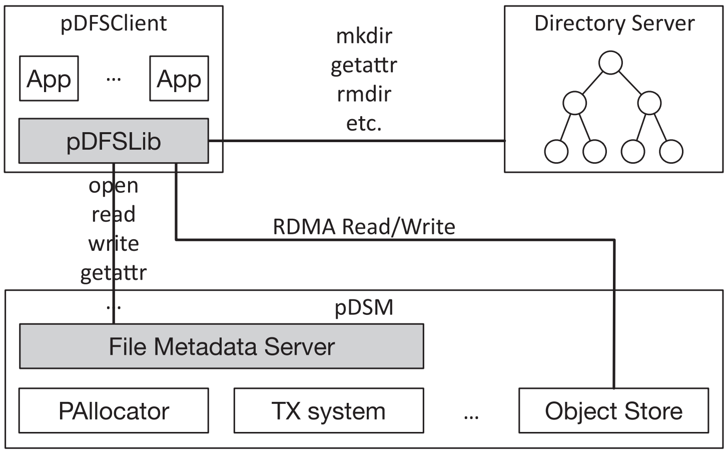

Fig. 7. Software architecture of pDFS.

avoiding the overhead of (de)serialization between external devices such as HDD/SSD. However, such APIs also require significant modification to most existing applications when deployed on pDSM. For example, many popular database systems (e.g., LevelDB [32] and MySQL) and big data processing systems (e.g., Hadoop and Spark) still organize data with files. To this end, TH-DPMS incorporates a distributed file system named pDFS to provide file-based access. pDFS is based on pDSM: it places the metadata and data in PAllocator and object store, respectively. We also borrow some key design principles from our past project (e.g., LocoFS [54] and Octopus [56]) when implementing pDFS.

As shown in Figure 7, pDFS consists of four key components: object store, file metadata server, directory metadata server, and pDFSLib. Each file consists of one or multiple objects, which are stored in the object store. The file and directory metadata server are responsible for managing the file system metadata and maintaining a hierarchical directory tree structure. The metadata are stored in pDSM via the PAllocator. pDFSLib runs at the client side and is linked to applications when they access the file system. pDFS is responsible for interacting with the object store and metadata server when processing the applications' requests.

pDFS mainly incorporates two mechanisms to improve the performance of a distributed file system. First, pDFS distributes the file system metadata according to the hash value of the pathname. For instance, the metadata of file `‘‘/home/sjw/file’’` is stored on node N, where $N = hash(\text{“}/home/sjw/file\text{”})$. In this way, pDFS delivers higher scalability for metadata performance. It also reduces metadata accessing latency, since each metadata access operation can be achieved with one network round-trip. However, hash-based distribution of metadata creates new challenges when renaming a directory. It requires that all the sub-files/directories inside the to-be-renamed directory be redirected to their new nodes, which causes significant overhead. To address this issue, pDFS decouples the file system metadata, the same as LocoFS does. Specifically, the file metadata are distributed among the file metadata servers (FMS) based on the hash value of the pathname, while the directory metadata are stored in a globally shared directory metadata server (DMS). The DMS maintains a directory tree and maps each directory to a global unique ID (i.e., GUID). GUID is never changed once a directory is created, even if it is renamed. The GUIDs of each directory in the pathname are used to compute the hash value of a file. In this way, renaming a directory does not need to move the metadata of all the sub-files, since the GUIDs of parent directories are still unchanged.

Second, pDFS incorporates the *Client-Active Data I/O* in Octopus for efficient data accessing. In traditional distributed file system, it is common to complete a data accessing request within one network round-trip. Take a file read operation for example: The client first sends a read request to the server, and then the server finds the corresponding data according to the file metadata, and finally sends the data back to the client. Similarly, a write request can also complete with one round-trip. We call such data-transferring paradigm as *server-active data I/O*. Such mode works well for slow Ethernet and external devices (e.g., SSD/HDD), but we find that the server CPU is always in high utilization and becomes a bottleneck when high-speed hardware is equipped.

Client-Active Data I/O is proposed to improve server CPU utilization by sacrificing the network round-trips. It includes three steps: First, the client sends a read or write request to the server; then, the server sends back the metadata information to the client, which describes where the accessed data are in the global address space. The above two steps are executed using the RPC primitive in iRDMA. Finally, the client reads or writes file data with the returned metadata information and directly accesses remote data using RDMA read and write verbs. Since RDMA read and write are one-sided operations, which access remote data without the involvement of remote CPUs, the server CPU has higher processing capacity. By doing so, a rebalance is made between the server and network overheads. With introduced extra network round-trips, server load is offloaded to clients, resulting in higher throughput for concurrent requests.

4.3 pDKVS: A Distributed Key-value Store

We implement a distributed key-value store named pDKVS on top of pDSM. For simplicity, pDKVS is based on consistent hashing (for load balancing) and Figure 8 shows the software architecture of pDKVS. With a given key, pDKS finds the corresponding key-value item with the following steps:

(1) Calculate the virtual node (i.e., V_{node} in the figure) according to the consistent hashing algorithm. Each virtual node corresponds to a hash table, which is locally stored on one of the physical PM nodes.
(2) Send a request to the monitor to find the physical PM node that stores the corresponding hash table. The monitor maintains a global mapping table to record the location of each

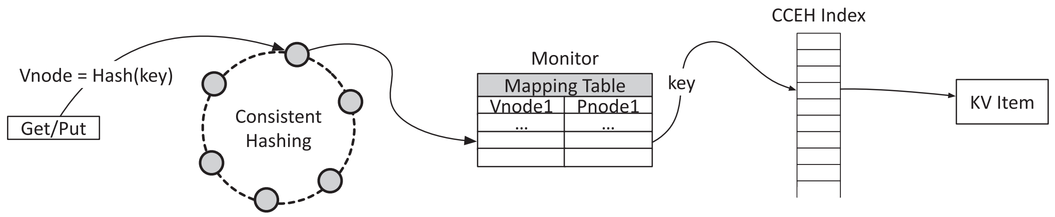

Fig. 8. Software architecture of pDKVS (CCEH [62] is a persistent memory hash index with high efficiency when resizing).

virtual node. To reduce the network round-trips, each client also keeps a local cache to record the recently accessed mappings.

(3) The client accesses a hash table index by posting a request via iRDMA to the physical node. Parameters such as the hash table ID and keys are encapsulated in the request as well. We directly use CCEH [62] as our persistent hash index, since it is efficient and causes low overhead when resizing. CCEH consists of two elements, including a global directory (i.e., an array), and one or multiple segments, each of which contains multiple buckets. We store CCEH in PM space allocated via PAllocator, CCEH itself is crash consistent, so we do not need to use extra logging mechanism.

(4) The item in the hash index only stores a pointer, which points to the actual key-value item. Key-value items are stored separately via PAllocator to support variable-length values. Note that we still need to use transactions when updating the CCEH index and key-value items to synchronize the modifications to backup segments.

5 EXPERIMENTS

In this section, we evaluate the overall performance of TH-DPMS with real-world applications and analyze the internal components with micro-benchmarks.

5.1 Experimental Setup

The hardware configuration details of PM servers have already been shown in Table 3. Our cluster consists of six such PM servers and 12 client nodes. Each client node has 128 GB of DRAM memory, two 2.2 GHz Intel Xeon E5-2650 v4 CPUs (24 cores in total), one MCX555A-ECAT ConnectX-5 EDR HCA (100 Gbps), and is installed with CentOS 7.4. All these servers are connected with a Mellanox MSB7790-ES2F switch. Our evaluation is conducted to answer the following two questions:

Q1 *Can TH-DPMS deliver comparable or even higher performance than those in-memory distributed storage systems whose data are completely placed in DRAM?*

Q2 *How does TH-DPMS behave with large workload that cannot be placed directly in DRAM?*

5.2 Overall Evaluation

Key-value Store. We emulate the ETC and SYS pools at Facebook [64] as the production workload to evaluate the behavior of TH-DPMS. ETC has 5% SETs and 95% GETs. The key size is fixed at 16 bytes and 90% of the values are evenly distributed between 16 and 512 bytes. Differently, SYS is SET-heavy, with 25% SET and 75% GET operations. 40% of the keys have length from 16 to 20 bytes, and the rest range from 20 to 45 bytes. Values of size between 320 and 500 bytes take up 80% of the entire data, 8% of them are smaller, and 12% sit between 500 and 10,000 bytes. Since no existing key-value store is designed with support of RDMA and persistent memory simultaneously, we directly compare with the RDMA-memcached [39], which is an open-sourced high-performance

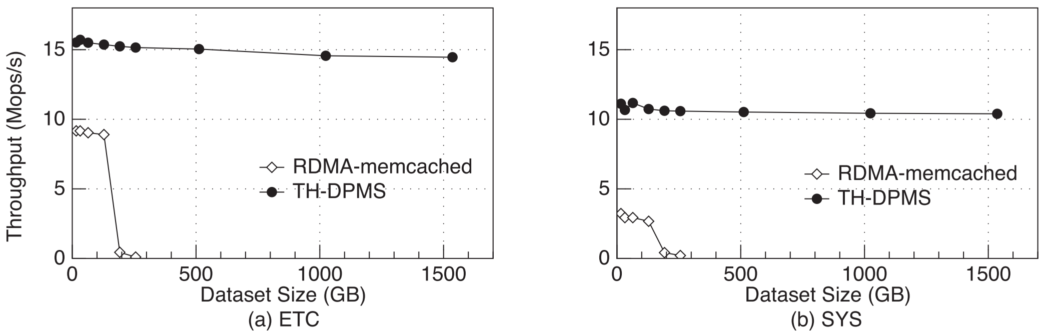

Fig. 9. The performance of pDKVS with varying workload size.

distributed memory object caching system designed for Infiniband network. The results are shown in Figure 9, and we make the following observations:

(1) For the workload with data size smaller than 128 GB, TH-DPMS achieves throughput that is 1.7× and 3.5× as much as that of `RDMA-memcached` in the `ETC` and `SYS`, respectively. `RDMA-memcached` places all its data in DRAM, but still underperforms TH-DPMS for two reasons: (i) A GET/PUT operation in `RDMA-memcached` needs multiple round-trips, while the clients of TH-DPMS access the key-value store through the well-optimized zero-copy RPC primitive in single round-trip. (ii) `RDMA-memcached` has scalability issues in terms of memory allocation and maintaining LRU linked-list.

(2) When the total amount of data further increases, `RDMA-memcached`'s performance dropped sharply to almost zero, while TH-DPMS's throughput is almost unchanged. Each PM server only has 192 GB of DRAM, which is not enough to accommodate the whole dataset. Under this circumstance, `RDMA-memcached` has to move the cold data to the SWAP device, and such data movements act as the performance killer.

Graph Processing. Graph processing is an increasingly popular yet challenging type of application for datacenters. We use graph processing as a demonstration how TH-DPMS can fit perfectly in memory-intensive and memory-consuming circumstances. To not dwell on any graph processing optimizing techniques, Graph500 is selected as benchmark for this experiment and BFS (Breadth First Search) the algorithm. Dataset includes two large-scale graphs generated using the graph generator of Graph500, the first one has a raw data size of 256 GB, contains 1B nodes and 16B edges, the other one is twice larger than the first one, hence double in node number and edge number. The experiment is conducted between two configurations, and the execution time of BFS algorithm is taken as measurement: In one configuration, clients are equipped with swap space backed by SSDs; in the other one, except for access to TH-DPMS, clients have no extra memory space but their own physical memory. To support large-scale graph processing given a limited memory capacity, small modifications have to be made to Graph500's code. Essentially, at any time, we only keep a fixed percentage of the in-memory graph data structure in DRAM and keep the rest of them in the TH-DPMS, for they account for most of the memory usage. "In-DRAM Ratio" in Table 5 shows the fixed percentage of the in-memory graph data structure stays in DRAM. All results are shown in Table 5.

Results show that graph processing can benefit from TH-DPMS. Even though parts of the in-memory graph data structure are keeping at the other side of the cable instead of locally in SSD, TH-DPMS still manages to achieve a speedup at almost 3.4×, and 9.7× for 512 GB sized graph,

Table 5. Total Runtime of BFS

System	Graph Size (GB)	# of Nodes	# of Threads	In-DRAM Ratio	Runtime (s)
TH-DPMS	256	1	16	40%	1,291
	512	2	32	40%	1,388
DRAM w/ SWAP	256	1	16	100%	4,500
	512	2	32	100%	13,496

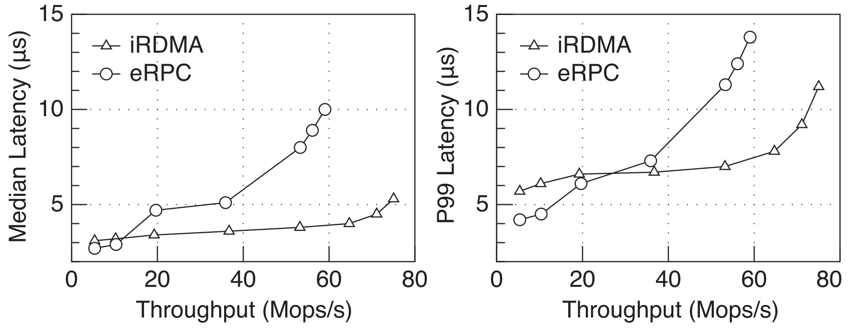

Fig. 10. Latency distribution of RDMA-based RPC.

giving credits to TH-DPMS's light-weight software stack and the ability to harness hardware bandwidth through its well-optimized RDMA-based RPC primitive.

5.3 Analysis of Individual Components

5.3.1 pDSM. We evaluate pDSM by analyzing the performance of RPC primitive, global address accessing, and distributed transactional system.

Latency Distribution of the RPC primitive. The RDMA-based RPC primitive is the most important component in TH-DPMS, whose performance directly determines the efficiency of many operations (e.g., metadata accessing, interaction with the monitors). We compare the RPC performance of TH-DPMS against eRPC [43], a state-of-the-art RPC framework tailored for modern network hardware. Figure 10 shows the throughput and the corresponding latencies (both median and 99th percentile latencies are shown) with one PM server and a varying number of clients, and we make the following observations:

(1) Our RPC primitive is capable of delivering a peak throughput of 76 Mops/s, while restricting its median latencies within 6 μs. Besides, its tail latencies are also low: For a target load running at 60 Mops/s, the 99th percentile latency is only 7 μs. This is as expected: Our RPC primitive is built directly on the `write-with-imm` verb, which directly reveals the hardware performance.

(2) Our RPC primitive has higher peak throughput (1.2×) as well as lower corresponding median latency (18%) and 99th percentile latency (47%) than eRPC. This is because eRPC uses datagram packet I/O and implements congestion control, flow control, and packet loss handling in software level, but for our RPC primitive, these functions are handled in RDMA NIC by using reliable connection.

Read/Write Bandwidth of pDSM. We measure the total read/write bandwidth of pDSM with an increasing number of PM nodes. All the 12 client nodes are launched for concurrent access; each client thread allocates a 64 KB global memory block and accesses the block by issuing read/write

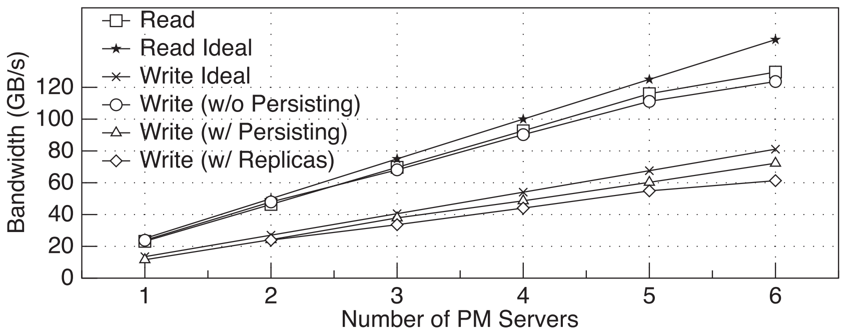

Fig. 11. Read/write bandwidth of pDSM.

operations repeatedly. When evaluating the write bandwidth, we choose three modes, which are (1) *Write w/o persisting*: the clients simply write data to remote PM servers with one-sided `write` verb and the server CPUs are unaware of such write events; (2) *Write w/ persisting*: the clients write data via `remote persistence` primitive; and (3) *Write w/ Replicas*: the clients write data to both the primary and the backup nodes (without persisting them). The results are shown in Figure 11, and we make the following observations:

(1) The bandwidths of *Read* and *Write w/o Persisting* operations are almost the same (both are 23.5 GB/s with one PM server), which are mainly restricted by the network bandwidth (200 Gbps with two NICs, *Read Ideal* in the figure). Notably, the bandwidth of *Write w/o Persisting* exceeds the raw write bandwidth of persistent memory (i.e., about 13 GB/s with one PM server), since the last level cache absorbs a large amount of write traffic by leveraging Intel's DDIO technology [7].
(2) Both read and write bandwidths scale linearly as the number of PM servers increases. By caching the frequently accessed address mapping table at client side, the clients do not need to interact with the monitor each time they access a global shared address. Hence, the centralized design of the monitor in TH-DPMS is not a bottleneck.
(3) *Write w/Persisting* and *Write w/Replicas* decreases the bandwidth by half. *Write w/Persisting* requires the involvement of remote CPUs to actively persist data via `clwb`, and its bandwidth is limited by the raw write bandwidth of persistent memory (i.e., *Read Ideal* in the figure); *Write w/Replicas* sends 2× as much data as that of *Write w/o Persisting*. These factors impact the overall bandwidth dramatically.

Transactional System. YCSB [26] is a widely used benchmark for key-value store evaluation. It is also commonly used in transactional database evaluation by accessing multiple records in a single transaction. In this part, we choose YCSB workloads with different read and write set sizes to evaluate our transactional system. As shown in Figure 12, both read-only and read-write transactions are evaluated. For the read-write transactions, we set the write ratio to 25% with both uniform (i.e., *25%-Write*) and skew (i.e., *25%-Write* with parameter of 0.99) access pattern. Among them, read-only transactions achieve the highest throughput (7.2 Mtxn/s with 6 PM servers). This is as expected, since there is no conflict between read-only transactions. Besides, we adopt OCC as the concurrency control protocol in our transactional system. Unlike 2PL, OCC does not make any modifications for read-only transactions and avoids all the false conflicts. The throughput of *25%-Write* also scales linearly. When accessing the records uniformly, it is less likely to happen for two transactions to conflict, so most of the transactions can be executed in parallel. Non-uniform accessing (i.e., zipfan in *25%-Write (0.99)*) exhibits the worst performance and is hard to scale,

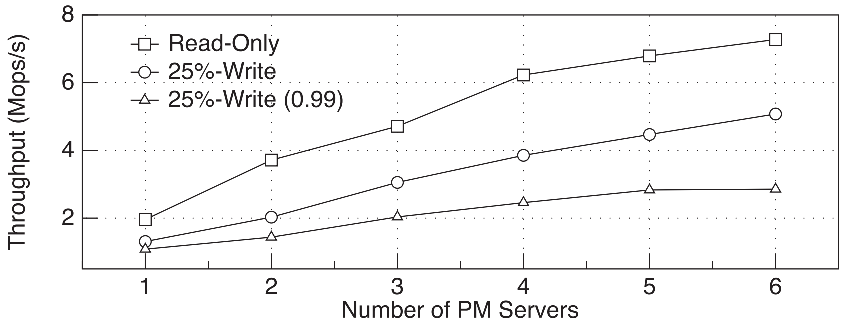

Fig. 12. Transaction throughput with YCSB workloads.

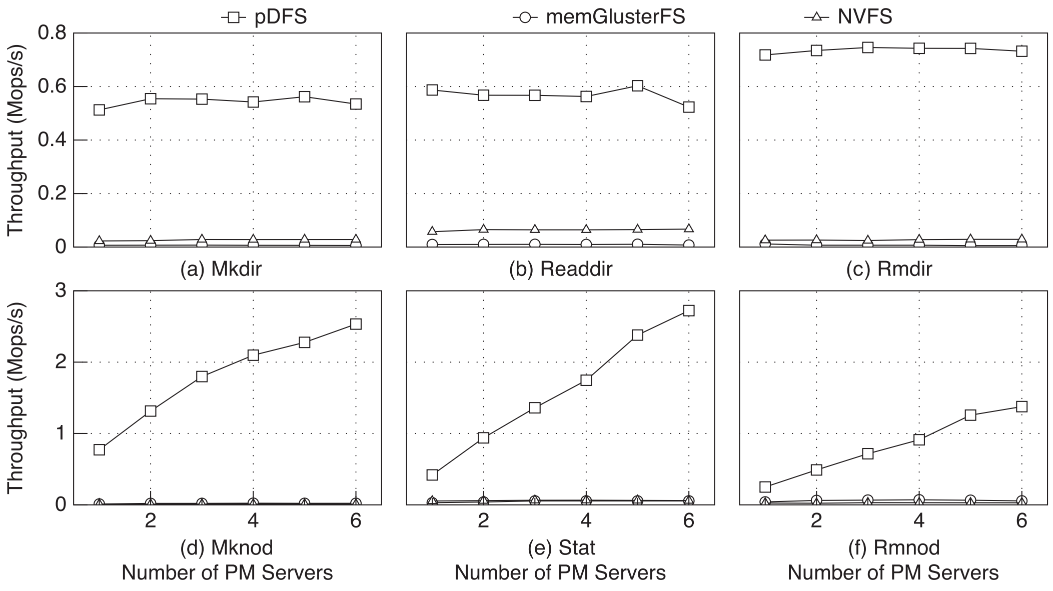

Fig. 13. Metadata performance in pDFS.

as the number of PM nodes varies. Since most of the transactions access a small portion of the records, transaction aborts happen frequently, which wastes most of the CPU cycles and reduces the performance.

5.3.2 Distributed File System. In this section, both metadata and data performance in pDFS are evaluated. We compare results with two distributed filesystems: memGlusterFS [3] and NVFS [37]. In memGlusterFS, all clients and servers in GlusterFS can communicate with each other using RDMA networks. NVFS is a version of HDFS that is specially optimized for NVM and RDMA networks. Both NVFS and memGlusterFS require running on top of a local filesystem; in this evaluation, we run both of them on *Ext4* with DAX feature enabled. In all file system evaluations, we only utilize one NUMA-side NVM out of two NVMs in one machine and one network interface out of two (NVFS can not support double RDMA network interface gracefully yet), which means only 768 GB size of NVM is utilized.

Metadata Operations. Figure 13 shows the file systems' performance in terms of metadata IOPS. We measure the performance of different metadata operations by varying the number of

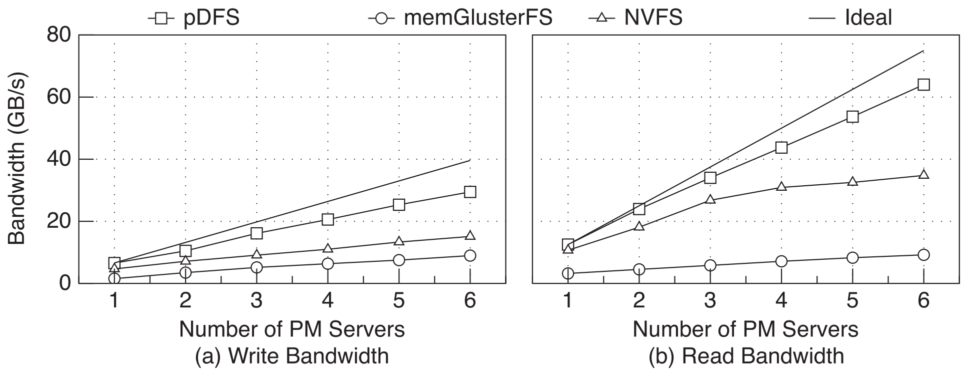

Fig. 14. Read/write bandwidth of pDFS.

PM servers. From the figure, we make two observations: (1) all the file metadata operations (e.g., `creat, unlink, stat`) show high scalability. In pDFS, we adopt the techniques in LocoFS [54] to organize the file system metadata. Specifically, we distribute the file metadata to different PM nodes according to the keyhash of the path name. Hence, multiple PM nodes can process these requests concurrently. However, NVFS and memGlusterFS both suffer from the inefficient architecture design and cannot deliver comparable performance. (2) The performance of directory operations (e.g., `mkdir, rmdir`) is almost unchanged, since we only introduce a single directory metadata server, even though pDFS still outperforms other filesystems a lot. Note that such performance can be further reasoned by: (1) In pDFS, a single directory metadata server is capable of achieving a throughput of more than 0.5M OPS, which is far higher than existing distributed file systems; (2) directory creation/deletion happens infrequently in real-world workloads [54]. For directory operations, NVFS is configured with one metadata server as well, but the inefficiency of the software layer hinders the performance it can deliver. We also notice that directory operations performance of memGlusterFS exhibits extremely low; we believe that, in addition to the stacked software architecture, the hash-based directory metadata management design is also to blame.

File Read/Write Bandwidth. Figure 14 shows the file system performance in terms of concurrent read/write bandwidth. We use multiple clients to access the file system with a varying number of PM servers. The results are shown in Figure 14, and we observe that both pDFS's read and write bandwidths are scalable and can increase in line with the number of PM servers. In particular, the read bandwidth is much higher than that of write operations, which is almost close to that of the Infiniband networks. Write bandwidth is much lower because of three reasons: (1) we only utilize one NUMA-side NVM on each machine—the maximum bandwidth is lower than that of networks; (2) pDFS has to update metadata after the file data are written, which causes extra network round-trips; (3) The written data have to be persisted via hardware instructions (we use the combination of `clwb/mfence` to flush data out of the CPU cache), which causes much higher overhead. MemGlusterFS shows poorly in terms of both read and write performance, it seems that data operations also fall victim of the inefficient software design. NVFS fails to scale further when number of servers keeps rising in read operation, because the single metadata server may have reached saturation.

Filebench Performance. Figure 15 shows the performance of the pDFS and other comparative filesystems using *filebench*. All clients are launched from same client node, and the cluster is configured with one data server. We choose four benchmarks from filebench, namely, *fileserver*, *webproxy*, *webserver,* and *varmail.* Originally, filebench is compliant with POSIX-like interface, both memGlusterFS and the pDFS can be seamlessly implemented, but the implementation for

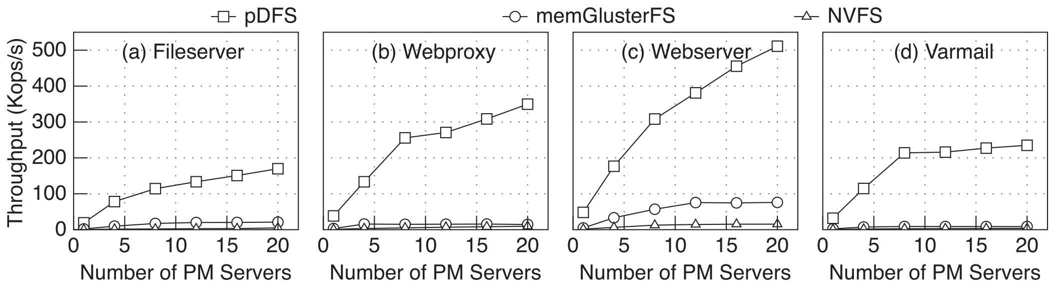

Fig. 15. Filebench performance of pDFS in comparison to memGlusterFS and NVFS.

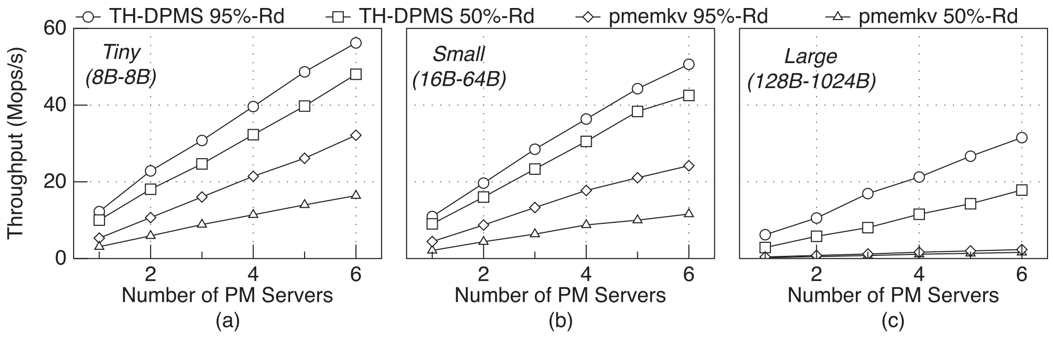

Fig. 16. Throughput of pDKVS in comparison with pmemkv for 95% and 50% read workloads.

NVFS needs extra efforts. We implement filebench for NVFS using hdfs library (i.e., libhdfs), however, as results show, such decision led to big software overheads. We believe such overhead comes from two aspects: (1) the overhead caused by communicating with hdfs through JNI (Java Native Interface); (2) the additional overhead caused when imitating POSIX-like behavior. Because of the inconsistency of the semantics, one operation could end up with invoking two hdfs interfaces. As shown in Figure 15, pDFS shows good performance under all four benchmarks and exceeds other file systems far beyond, attributing to metadata design and client-active data I/O.

5.3.3 Key-value Store. We finally evaluate the performance of pDKVS against pmemkv [10] by varying the item sizes, the number of PM servers, and the read-write ratio. Pmemkv is based on PMDK [9], and we use our RPC primitive for client-server interaction and its `cmap` (i.e., concurrent hash map) storage engine. The key-value items are accessed uniformly to measure the peak performance. Figure 16(a) plots the experiment results using tiny key-value items (8-byte keys and 8-byte values). Our key-value store achieves 48 Mops/s and 56 Mpos/s with 6 PM nodes in write-intensive (50% PUT) and read-intensive (95% GET) workloads, respectively, which are 2.9× and 1.7× higher than that of pmemkv, respectively. This is because pmemkv needs complex logging to guarantee crash consistency, but our key-value store is logless by using copy-on-write and atomic write. Small (16-byte keys and 64-byte values) and Large (128-byte keys and 1,024-byte values) key-value items show similar results in Figure 16(b) and (c).

5.4 Failure Recovery

To evaluate performance with failures, we ran the same benchmarks used in the pDKVS evaluation (i.e., 95%-Get, 64-byte values). We killed the process on one of the PM nodes to show a timeline with the throughput of the surviving machines aggregated at 1 ms intervals. The results

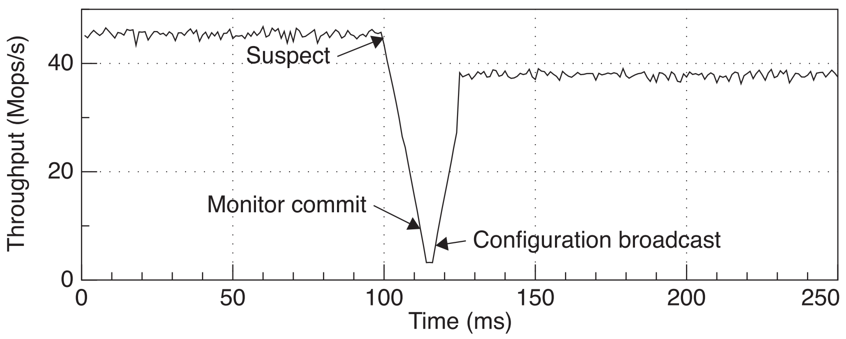

Fig. 17. pDKVS performance timeline with node failure.

are shown in Figure 17. It shows the time at which the failed machine's heartbeat expired on the monitor node ("Suspect"); the time at which the configuration file has been successfully updated and synchronized to backup monitors ("Monitor commit"); the time at which the new configuration was broadcast to all PM nodes ("Configuration broadcast"). We make two observations from the figure.

First, the system data becomes active in less than 30 ms. TH-DPMS uses `RDMA reads` to detect node failures; such one-sided verbs are issued by the centralized monitor at extremely high frequency (i.e., 1 ms in our implementation) without impacting the foreground throughput, since CPUs on the PM nodes are not involved when serving RDMA reads. Meanwhile, the configuration file only contains several megabytes of data, which can be quickly updated to backup monitors and other PM nodes.

Second, the system is back to peak throughput soon after the configuration has been broadcast to PM nodes. Similar to FaRM, we limit the recovery rate to a threshold to prevent the recovery load from impacting foreground operations. Specifically, each worker thread fetches 8 KB of data from new primary nodes every 2 ms, indicating that a 2 GB segment can be recovered within 17 s. Note that the peak throughput drops by almost 17% compared to that before the node failure occurs, since the active PM nodes reduce from 6 to 5.

6 RELATED WORKS

To process the unprecedented amounts of data, datacenter architects believe that a transformational change is required for both hardware and software designs.

Next-generation of hardware. The FireBox project [6] aims to develop a system architecture for next-generation Warehouse-Scale Computers (WSCs). A single Firebox can be scaled to contain up to 10K compute nodes and up to an Exabyte (2^{60} bytes) of non-volatile memory connected via a low-latency, high-bandwidth optical switch. Hewlett Packard Labs has a similar solution in The Machine [8], which adopts emerging SoCs and memristors connected via photonics. The Machine flattens complex data hierarchies and brings processing closer to the data.

Intel's development tools for Optane memory. The Persistent Memory Development Kit (PMDK) [9], formerly known as NVML, is a growing collection of libraries and tools that provides fine-grained persistent memory management and transaction support. However, TH-DPMS is developed from scratch, instead of using PMDK to manage the PM space in each node. PMDK has very complex transactional logic when maintaining its redo and undo log. As such, it often causes multiple writes to PM even when serving a simple allocation request. PMDK also provides the `librpmem` tool that provides low-level support for remote access to persistent memory utilizing RDMA-capable RNICs. However, it is still an experimental API. The Distributed Asynchronous Object Storage (DAOS) [2] is a scale-out object store that provides high bandwidth and low latency storage containers to HPC applications. It stores data on both storage-class memory (via PMDK) and NVMe storage and presents a native key-array-value storage interface.

Distributed shared memory. Hotpot [74] is a kernel-level DSM system that directly exposes a shared memory interface to applications. It improves data reliability and availability by introducing an integrated distributed memory caching and data replication protocol. FileMR [87] argues that the abstractions between NVM and RDMA are incompatible—an NVM-aware file system manages persistent memory as files, whereas RDMA uses memory regions to organize remotely accessible memory. As such, this article proposes the *file memory region* (FileMR), which combines memory regions and files: A client can directly access a file backed by NVM file system through RDMA, addressing its contents via file offsets. Remote region [12] is an abstraction for a process to export its DRAM space to remote hosts and to access the memory exported by others via RDMA network. Applications access remote regions through the usual file system operations (read, write, memory map, etc.). FaRM [28, 29, 73] is a new main memory distributed computing platform that exposes the memory of machines in the cluster as a shared address space and exploits RDMA to improve both latency and throughput. FaRM transaction, replication, and recovery protocols are designed from scratch leveraging the one-sided verbs of RDMA. FaRM is a main reference when we build TH-DPMS, despite that they use DRAM as the main storage space. Mojim [92] is a first attempt to build persistent memory storage system base on RDMA. It provides reliability and availability by using a two-tier architecture for data mirroring/replication. Highly optimized replication protocols, software, and networking stacks are used to minimize replication costs. Grappa [63] is a software-based DSM for in-memory data-intensive applications. It provides an easy-to-use programming model that enables users to program a cluster as if it were a single, large, non-uniform memory access (NUMA) machine. The core components in Grappa (i.e., tasking system and communication layer) enable it to work more efficiently. However, it does not use native RDMA support and places data directly in DRAM. Grappa works more like a computing framework, instead of a storage system.

Distributed file system. Octopus [56] is a distributed file system based on RDMA and persistent memory. It improves performance by abstracting a shared persistent memory pool, which enables the clients to directly access file data in the memory pool. Worth mentioning, it is our past experience of implementing Octopus that motivates us to further extend such design, and thus forms the generic abstraction of pDSM in TH-DPMS. Ziggurat [93] is a tiered file system that combines NVMM and slow disks to create a storage system with near-NVMM performance and large capacity. It steers incoming writes to NVMM, DRAM, or disk, depending on application access patterns, write size, and the likelihood that the application will stall until the write completes. Orion [86] also builds the distributed file system on RDMA and NVM. It is compatible with existing POSIX semantics, since it is implemented at the kernel level. It improves fault tolerance by adding Mojim-like mirroring for metadata and broadcast replication for file data. It also provides different consistency levels to meet the needs of different apps.

Remote data structure. AsymNVM [60] is a generic framework for asymmetric disaggregated non-volatile memories. It implements the fundamental primitives for building remote data structures, including space management, concurrency control, crash consistency, and replication. StoRM [66] is a fast transactional dataplane for remote data structures. It utilizes one-sided read and write-based RPC primitives to implement remote data structures and thereby addresses the challenges in RDMA (e.g., scalability, address translation, and pointer chasing). FlatStore [21] improves the performance of key-value stores by proposing a log-structured storage engine on top of Optane DCPMMs and RDMA. It leverages the mismatch between the persistence granularity of Optane itself and the access sizes of typical key-value items, and thus proposes the pipelined horizontal batching mechanism to amortize the persistence overhead among multiple requests.

Unified Abstraction. pDSM acts as the building block in TH-DPMS by providing the fundamental primitives with a shared global address space. This is similar to the Ceph storage system, at the core of which is the Reliable Autonomic Distributed Object Store (RADOS) service [81]. RADOS is highly scalable and provides self-healing, self-managing, replicated object storage with strong consistency. With RADOS service, Ceph provides three services: the RADOS Gateway (RGW), the RADOS Block Device (RBD), and CephFS, a distributed file system with POSIX semantics. NodeKernel [75] introduces a unified abstraction of file systems and key-values stores for temporary data, based on the observation that the software architectures of file systems and key-value stores look similar for temporary data. Different from NodeKernel, TH-DPMS is designed for persistent data storage and uses a unified abstraction of pDSM. The file system interface and key-value interface are designed on top of the pDSM abstraction.

7 CONCLUSION

High-speed NVM and RDMA hardware are promising technologies in the face of the increasing needs of data storage and transfer. However, these hardware push back the software evolution. In this article, we designed and implemented a distributed storage system based on RDMA and persistent memory named TH-DPMS. We abstract a generic layer named pDSM, which connects the PMs of different storage nodes via high-speed RDMA network, and organize them into a global shared address space. pDSM supports efficient space management, replication, and transactions. Based on pDSM, we further implement both a distributed file system and a key-value store. Evaluations show that TH-DPMS effectively explores hardware benefits.

ACKNOWLEDGMENTS

We thank the members of the Storage Research Lab at Tsinghua University, including Siyang Li, Jing Wang, Minhui Xie, Wenhao Lv, Pei Chen, and Yifan Qiao, for their efforts in the system prototyping.

REFERENCES

[1] Mellanox Technologies. 2019. ConnectX-6 VPI Card. Retrieved from https://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-6_VPI_Card.pdf.
[2] Intel Corporation. 2019. The Distributed Asynchronous Object Storage. Retrieved from https://daos-stack.github.io/.
[3] Red Hat. Inc. 2019. GlusterFS. Retrieved from https://www.gluster.org/.
[4] Intel Corporation. 2019. Intel Optane DC Persistent Memory. Retrieved from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.
[5] IDC. 2020. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. Retrieved from https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.
[6] Berkeley Architecture Research. 2020. The Firebox Project. Retrieved from https://bar.eecs.berkeley.edu/projects/firebox.html.
[7] Intel Corporation. 2020. Intel Data Direct I/O Technology. Retrieved from https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.
[8] HP Development Company. 2020. The Machine Project. Retrieved from https://www.hpl.hp.com/research/systems-research/themachine.
[9] Intel Corporation. 2020. PMDK: Persistent Memory Development Kit. Retrieved from https://github.com/pmem/pmdk.
[10] Intel Corporation. 2020. pmemkv. Retrieved from https://github.com/pmem/pmemkv/.
[11] Redis Labs. 2020. Redis. Retrieved from https://redis.io/.
[12] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: A simple abstraction for remote memory. In *Proceedings of the USENIX Annual Technical*

Conference (USENIX ATC'18). USENIX Association, 775–787. Retrieved from https://www.usenix.org/conference/atc18/presentation/aguilera.

[13] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* ACM, 707–722.

[14] I. G. Baek, M. S. Lee, S. Seo, M. J. Lee, D. H. Seo, D.-S. Suh, J. C. Park, S. O. Park, H. S. Kim, I. K. Yoo, et al. 2004. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Proceedings of the IEEE International Electron Devices Meeting.* IEEE, 587–590.

[15] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00).* Association for Computing Machinery, New York, NY, 117–128. DOI: https://doi.org/10.1145/378993.379232

[16] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2014. OpLog: A library for scaling update-heavy data structures. Retrieved from https://people.csail.mit.edu/nickolai/papers/boyd-wickizer-oplog-tr.pdf.

[17] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1995. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst.* 13, 3 (1995), 205–243.

[18] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. DOI: https://doi.org/10.14778/2752939.2752947

[19] Youmin Chen, Youyou Lu, Pei Chen, and Jiwu Shu. 2019. Efficient and consistent NVMM cache for SSD-based file system. *IEEE Trans. Comput.* 68, 8 (Aug. 2019), 1147–1158. DOI: https://doi.org/10.1109/TC.2018.2870137

[20] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys'19).* Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/3302424.3303968

[21] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM.

[22] Youmin Chen, Youyou Lu, Bohong Zhu, and Jiwu Shu. 2019. Kernel/User-level Collaborative Persistent Memory File System with Efficiency and Protection. arxiv:cs.OS/1908.10740

[23] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. 2018. HiNFS: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage* 14, 1 (Apr. 2018). DOI: https://doi.org/10.1145/3204454

[24] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. 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).* ACM, New York, NY, 105–118. DOI: https://doi.org/10.1145/1950365.1950380

[25] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09).* ACM, New York, NY, 133–146. DOI: https://doi.org/10.1145/1629575.1629589

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10).* Association for Computing Machinery, New York, NY, 143–154. DOI: https://doi.org/10.1145/1807128.1807152

[27] Mingkai Dong, Heng Bu, Jiefei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19).*

[28] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14).* 401–414.

[29] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles.* ACM, 54–70.

[30] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14).* ACM, New York, NY. DOI: https://doi.org/10.1145/2592798.2592814

[31] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: Data management for modern business applications. *ACM SIGMOD Rec.* 40, 4 (2012), 45–51.

[32] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB. Retrieved from https://github.com/google/leveldb.

[33] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, et al. 2018. What your DRAM power models are not telling you: Lessons from a detailed experimental study. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 3 (2018), 38.

[34] Morteza Hoseinzadeh. 2019. A survey on tiering and caching in high-performance storage systems. *arXiv preprint arXiv:1904.11560* (2019).

[35] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. 187.

[36] Taeho Hwang, Jaemin Jung, and Youjip Won. 2014. HEAPO: Heap-based persistent object store. *ACM Trans. Storage* 11, 1 (Dec. 2014). DOI: https://doi.org/10.1145/2629619

[37] Nusrat Sharmin Islam, Md. Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K. Panda. 2016. High performance design for HDFS with byte-addressability of NVM and RDMA. In *Proceedings of the International Conference on Supercomputing (ICS'16)*. Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/2925426.2926290

[38] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, et al. 2019. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[39] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. 2011. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. IEEE Computer Society, 743–752. DOI: https://doi.org/10.1109/ICPP.2011.37

[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2015. Using RDMA efficiently for key-value services. *ACM SIGCOMM Comput. Commun. Rev.* 44, 4 (2015), 295–306.

[41] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'16)*. 437–450.

[42] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 185–201.

[43] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association, 1–16.

[44] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a true direct-access file system with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*. 241.

[45] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC'97)*. Association for Computing Machinery, New York, NY, 654–663. DOI: https://doi.org/10.1145/258533.258660

[46] Sanidhya Kashyap, Dai Qin, Steve Byan, Virendra J. Marathe, and Sanketh Nalli. 2019. Correct, fast remote persistence. *arXiv preprint arXiv:1909.02092* (2019).

[47] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter Conference*, Vol. 1994. 23–36.

[48] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 460–477. DOI: https://doi.org/10.1145/3132747.3132770

[49] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 2–13.

[50] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 462–477.

[51] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. SocksDirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*. ACM, New York, NY, 90–103. DOI: https://doi.org/10.1145/3341302.3342071

[52] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 137–152.

[53] Kai Li. 1988. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing* 2 88 (1988), 94.

[54] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. LocoFS: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/3126908.3126928

[55] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 329–343. DOI: https://doi.org/10.1145/3037697.3037714

[56] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: An RDMA-enabled distributed persistent memory file system. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'17)*. USENIX Association, 773–785.

[57] Youyou Lu, Jiwu Shu, and Long Sun. 2015. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*. IEEE, 1–13.

[58] Youyou Lu, Jiwu Shu, and Long Sun. 2016. Blurred persistence: Efficient transactions in persistent memory. *ACM Trans. Storage* 12, 1 (Jan. 2016). DOI: https://doi.org/10.1145/2851504

[59] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-ordering consistency for persistent memory. In *Proceedings of the IEEE 32nd International Conference on Computer Design (ICCD'14)*. IEEE, 216–223.

[60] Teng Ma, Mingxing Zhang, Kang Chen, Xuehai Qian, Zhuo Song, and Yongwei Wu. 2020. AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

[61] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*. 103–114.

[62] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 31–44.

[63] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant software distributed shared memory. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX Association, 291–305. Retrieved from https://www.usenix.org/conference/atc15/technical-session/presentation/nelson.

[64] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, 385–398.

[65] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. *ACM SIGPLAN Not.* 49, 4 (2014), 3–18.

[66] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. 2019. StoRM: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*. Association for Computing Machinery, New York, NY, 97–108. DOI: https://doi.org/10.1145/3319647.3325827

[67] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambara. 2019. SplitFS: A file system that minimizes software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*.

[68] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, 305–320.

[69] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A high performance file system for non-volatile main memory. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2901318.2901324

[70] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*. ACM, New York, NY, 371–386. DOI: https://doi.org/10.1145/2882903.2915251

[71] Marius Poke and Torsten Hoefler. 2015. DARE: High-performance state machine replication on RDMA networks. In *Proceedings of the 24th International Symposium on High-performance Parallel and Distributed Computing*. ACM, 107–118.

[72] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 24–33.

[73] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojeviundefined, Dushyanth Narayanan, and Miguel Castro. 2019. Fast general distributed transactions with opacity. In *Proceedings of the International Conference on Management of Data (SIGMOD'19)*. Association for Computing Machinery, New York, NY, 433–448. DOI: https://doi.org/10.1145/3299869.3300069

[74] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *Proceedings of the Symposium on Cloud Computing (SoCC'17)*. Association for Computing Machinery, New York, NY, 323–337. DOI: https://doi.org/10.1145/3127479.3128610

[75] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. 2019. Unification of temporary storage in the NodeKernel architecture. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*. USENIX Association, 767–782. Retrieved from https://www.usenix.org/conference/atc19/presentation/studi.

[76] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*. USENIX Association, 5–5. Retrieved from http://dl.acm.org/citation.cfm?id=1960475.1960480.

[77] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2592798.2592810

[78] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, New York, NY, 91–104. DOI: https://doi.org/10.1145/1950365.1950379

[79] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 233–251.

[80] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 87–104.

[81] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. 2007. Rados: A scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing'07*. ACM, 35–44.

[82] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A file system for storage class memory. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2063384.2063436

[83] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'17)*. 349–362.

[84] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, 323–338. Retrieved from http://dl.acm.org/citation.cfm?id=2930583.2930608.

[85] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 478–496. DOI: https://doi.org/10.1145/3132747.3132761

[86] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, 221–234. Retrieved from https://www.usenix.org/conference/fast19/presentation/yang.

[87] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA networking for scalable persistent memory. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. USENIX Association, 111–125. Retrieved from https://www.usenix.org/conference/nsdi20/presentation/yang.

[88] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, 169–182. Retrieved from https://www.usenix.org/conference/fast20/presentation/yang.

[89] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, 167–181. Retrieved from http://dl.acm.org/citation.cfm?id=2750482.2750495.

[90] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. 95.

[91] Kaisheng Zeng, Youyou Lu, Hu Wan, and Jiwu Shu. 2017. Efficient storage management for aged file systems on persistent memory. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE'17)*. European Design and Automation Association, 1773–1778.

[92] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. 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). Association for Computing Machinery, New York, NY, 3–18. DOI:https://doi.org/10.1145/2694344.2694370

[93] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A tiered file system for non-volatile main memories and disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 207–219.

[94] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 14–23.

[95] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 461–476.

Received February 2020; revised May 2020; accepted July 2020