To appear at ATC'20

Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores

Shin-Yeh Tsai, Yizhou Shan, Yiying Zhang
Purdue University and University of California, San Diego

Abstract

Many datacenters and clouds manage storage systems separately from computing services for better manageability and resource utilization. These existing disaggregated storage systems use hard disks or SSDs as storage media. Recently, the technology of persistent memory (PM) has matured and seen initial adoption in several datacenters. Disaggregating PM could enjoy the same benefits of traditional disaggregated storage systems, but it requires new designs because of its memory-like performance and byte addressability.

In this paper, we explore the design of disaggregating PM and managing them remotely from compute servers, a model we call *passive disaggregated persistent memory*, or *pDPM*. Compared to the alternative of managing PM at storage servers, pDPM significantly lowers monetary and energy costs and avoids scalability bottlenecks at storage servers.

We built three key-value store systems using the pDPM model. The first one lets all compute nodes directly access and manage storage nodes. The second uses a central coordinator to orchestrate the communication between compute and storage nodes. These two systems have various performance and scalability limitations. To solve these problems, we built Clover, a pDPM system that separates the location, communication mechanism, and management strategy of the data plane and the metadata/control plane. Compute nodes access storage nodes directly for data operations, while one or few global metadata servers handle all metadata/control operations. From our extensive evaluation of the three pDPM systems, we found Clover to be the best-performing pDPM system. Its performance under common datacenter workloads is similar to non-pDPM remote in-memory key-value store, while reducing CapEx and OpEx by 1.4× and 3.9×.

1 Introduction

Separating (or "disaggregating") storage and compute has become a common practice in many datacenters [11, 18] and clouds [3, 4]. Disaggregation makes it easy to manage and scale both the storage and the compute pools. By allowing the storage pool to be shared across applications and users, disaggregation consolidates storage resources and reduces their cost. As a recent success story, Alibaba listed their RDMA-based disaggregated storage system as one of the five reasons that enabled them to serve the peak load of 544,000 orders per second on the 2019 Single's Day [62].

Existing disaggregated storage systems are all SSD- or HDD-based. Today, a new storage media, non-volatile memory (or persistent memory, *PM*) has arrived [27, 29] and has already seen adoption in several datacenters [21, 25, 46]. Existing distributed PM systems [42, 56, 69] have mainly taken a non-disaggregated approach, where each server in a cluster hosts PM for applications running both on the local server and remote servers (Figure 1(a)).

Disaggregating PM could enjoy the same management and resource-utilization benefits as traditional disaggregated storage systems. However, building a PM-based disaggregated system is very different from traditional disaggregated storage systems as PM is byte addressable and orders of magnitude faster than SSDs and HDDs. It is also different from disaggregated memory systems [41, 55], since when treated as storage systems, disaggregated PM systems need to sustain power failure and be crash consistent.

There are two possible design directions in building disaggregated PM systems, and they differ in where management software runs. The first type, and the type that has been adopted in traditional disaggregated storage systems, runs management software at the storage nodes, *i.e.*, actively managing data at where the data is. When applying this model to PM, we call the resulting system *active disaggregated PM*, or *aDPM* (Figure 1(b)). By co-locating data and their management, aDPM could offer low-latency performance to applications. However, aDPM requires significant processing power at storage nodes to sustain high-bandwidth networks and to fully deliver PM's superior performance.

In this paper, we explore an alternative approach of building disaggregated PM by treating storage nodes as *passive* parties that do not perform any data processing or data management tasks, a model we call *pDPM*. pDPM offers several practical benefits and research value. First, pDPM lowers owning and energy cost. Without any processing need, a PM node (we call it a *data node* or *DN*) can either be a regular server that dedicates its entire CPU to other applications or a hardware device that directly attaches a NIC to PM. Second, pDPM avoids DN's processing power being the performance scalability bottleneck. Finally, pDPM is an approach in the design space of disaggregated storage systems that has largely been overlooked in the past. Exploring pDPM systems would reveal various performance, scalability, and cost tradeoffs that could help future researchers and systems builders make better design decisions.

pDPM presents several new challenges, the biggest of which is the need to avoid processing all together from where data is hosted. Existing in-memory data stores heavily rely

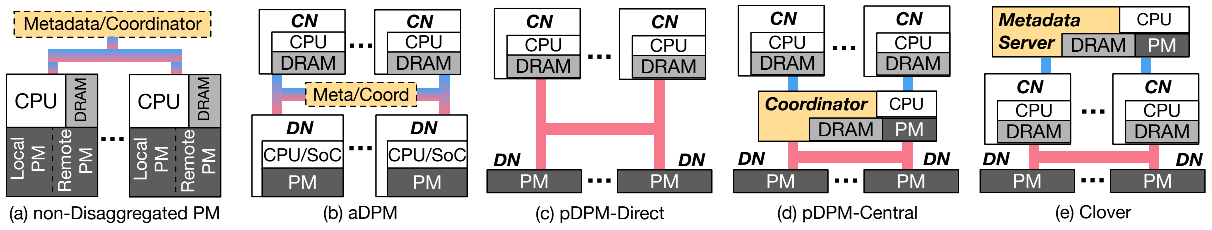


Figure 1: **PM Organization Comparison.** *Blue bars indicate two-way communication and pink ones indicate one-way communication. Bars with both blue and pink mean support for both. Dashed boxes mean some but not all existing solutions adopt centralized metadata server (or a coordinator).*

on local processing power for both the data path and the control path. Without any processing power, accesses to DNs have to come all from the network, which makes data operations like concurrent writes especially hard. Moreover, DNs cannot perform any management tasks or metadata operations locally, and each DN can fail independently.

A key question in designing pDPM systems is where to perform data and metadata operations when we cannot perform them at DNs. Our first approach is to let client/compute nodes (*CN*s) perform all the tasks by directly accessing DNs with *one-sided* network communication, a model we call *pDPM-Direct* (Figure 1(c)). After building and evaluating a real pDPM-Direct key-value store system, we found that since CNs cannot be efficiently coordinated, pDPM-Direct performs and scales poorly when there are concurrent reads/writes to the same data. Our second approach is *pDPM-Central* (Figure 1(d)), where we use a central server (the *coordinator*) to manage DNs and to orchestrate all accesses from CNs to DNs. Although pDPM-Central provides a way to coordinate CNs, it adds more hops between CNs and DNs, and the coordinator is a new scalability bottleneck.

To solve the issues of the above two pDPM systems, we build Clover, a key-value store system with a new architecture of pDPM (Figure 1(e)). Clover's main ideas are to separate the location of data and metadata, to use different communication mechanisms to access them, and to adopt different management strategies for them. Data is stored at DNs. Metadata is stored at one or few global metadata servers (*MS*s). CNs directly access DNs for all data operations using *one-sided* network communication. They use *two-sided* communication to talk to MS(s). MS(s) perform all metadata and control operations.

Clover achieves low-latency, high-throughput performance while delivering the consistency and reliability guarantees that are commonly used in traditional distributed storage systems. We designed a set of novel techniques at the data and the metadata plane to achieve these goals. Our data plane design is inspired by log-structured writes and skip lists. This design achieves 1-/2-RTT read/write performance when there is no high write contention, while ensuring proper synchronization and crash consistency of concurrent writes with satisfactory performance. We move all metadata and control operations off performance critical path. We *completely* eliminate the need for the MS to communicate with DNs; it performs space management and other control tasks without accessing DNs. In addition, Clover supports replicated writes for high availability and reliability.

We evaluate Clover, pDPM-Direct, and pDPM-Central using a cluster of servers connected with RDMA network (some acting as CNs and MSs, some acting as emulated DNs). We compare these pDPM systems with two non-disaggregated PM systems [42, 56] and an aDPM key-value store system [30] running on CPU-based servers and on ARM-SoC-based RDMA SmartNIC [44]. We perform an extensive set of experiments to study the latency, throughput, scalability, CPU utilization, and owning cost of these systems using microbenchmarks and YCSB workloads [13, 71]. Our evaluation results demonstrate that Clover is the best-performing pDPM system, and it significantly outperforms traditional distributed PM systems. Clover achieves similar or better performance as aDPM systems under common datacenter workloads, while reducing CapEx and OpEx by 1.4× and 3.9×. However, we also discovered a fundamental limitation of pDPM-based storage systems: no processing at where data sits could hurt write performance, especially under high contention of concurrent accesses to the same data entry. Fortunately, most datacenter workloads are read-most [7]. Thus, we believe pDPM and Clover to be good choices future systems builders can consider, given their overall benefits in cost, performance, and scalability.

Overall, this paper makes the following contributions:

- Thorough exploration of the passive disaggregated persistent-memory architecture, revealing its benefits, tradeoffs, and pitfalls.
- Implementation of Clover and two alternative pDPM key-value stores, all guaranteeing proper synchronization, crash consistency, and high availability.
- A detailed design of how to separate the data plane and the metadata plane under the pDPM model.
- Comprehensive evaluation results that can guide future DPM research.

All our pDPM systems are publicly available at https://github.com/WukLab/pDPM.

2 Background and Related Work

This section includes background and related work on in-memory data stores, RDMA, and PM in datacenter settings.

2.1 PM and Distributed PM Storage

Non-volatile memory (or PM) technologies such as 3D-XPoint [28], PCM, STTM, and the memristor provide byte addressability, persistence, and latency that is within an order of magnitude of DRAM [59, 70]. PM has attracted extensive research efforts in the past decade, most of which focus on single-node environments. The first commercial PM product, Intel Optane DC, has finally come to market [27]. It is pressing to seek solutions to incorporate PM in datacenters.

Existing distributed PM systems [42, 56, 69] have mainly adopted a *symmetric* architecture where each node in a cluster hosts some PM that can be accessed both locally and by other nodes (Figure 1(a)). Some of these systems expose a file system interface [42, 69], and others expose a memory interface [56, 73]. Among them, Orion [69] uses a global server for metadata, and the rest co-locate metadata with data. These systems have fast local-data accesses but lack flexibility in managing compute and storage resources, and they cannot scale these resources independently.

2.2 RDMA and RDMA-Based Data Stores

Remote Direct Memory Access, or *RDMA*, is a network technology that offers low-latency and low-CPU-utilization accesses to memory at remote machines. RDMA supports two communication patterns: *one-sided* and *two-sided*. One-sided RDMA operations allow one node to directly access the memory at another node without involving the latter's CPU. Two-sided RDMA involves both sender's and receiver's CPUs, similar to traditional network messaging.

Because of its performance and cost benefits [22, 36, 49], RDMA has been deployed in major datacenters like Microsoft [63] and Alibaba [2]. Several recent distributed systems such as in-memory key-value stores [15, 16, 38, 47, 48, 58] and in-memory databases/transactional systems [8, 10, 66, 72] use RDMA to perform their network communication. Most of them use a combination of one- and two-sided RDMA or pure two-sided RDMA. For example, FaSST [32] is an RDMA-based RPC system built entirely with two-sided RDMA. FaRM [15, 16], an RDMA-based distributed memory platform, uses one-sided communication for reads and performs both one- and two-sided operations for replicated writes. Pilaf [47] is a key-value store system that uses one-sided RDMA *read* for *get* and two-sided RDMA for *put*. HERD [30, 31] is another RDMA-based key-value store system. For each *get* and *put*, HERD uses two RDMA operations: client sending a one-sided RDMA *write* request to server and server sending an RDMA *send* response to client.

To achieve low-latency performance, most existing systems use busy-polling threads to receive incoming two-sided RDMA requests. They also perform management tasks such as memory allocation and garbage collection at CPUs in data-hosting nodes [15, 72]. Consequently, even when one-sided RDMA operations help reduce CPU utilization, practical RDMA-based data stores still require a CPU and significant amount of energy at each data-hosting server. For example, although FaRM [15] tries to use as much one-sided communication as possible, it still requires processing power at data nodes to perform metadata operations and certain steps in its write replication protocol.

HyperLoop [35] is a recent system that provides a mechanism to extend default one-sided RDMA operations to support more functionalities. These additional functionalities are performed at RDMA NICs without involving host CPU. HyperLoop's computation offloading technique could be applied to pDPM systems to offload certain data operations to DNs, which could potentially improve pDPM's performance. However, it is difficult to offload the more complex metadata operations to RDMA NICs, and HyperLoop still performs them at CPUs. Clover demonstrates how to efficiently separate the metadata plane and run it at a global metadata server.

2.3 Resource Disaggregation

Resource disaggregation is a notion to separate different types of resources into pools (*e.g.*, a compute pool and a storage pool), each of which can be independently managed, configured, and scaled [6, 26, 55]. Because of its efficiency in resource utilization and management, many datacenters and clouds have taken this approach when building storage systems [3, 4, 11, 18, 65].

Disaggregation could take two forms: disaggregating resources and managing them at where they are (active), and disaggregating resources but managing them at the compute pool (passive). Existing storage and memory systems have mainly taken the active approach, with most of them building disaggregated resource pools using regular CPU-based servers [51, 65]. To sustain high-bandwidth networks and fast PM, these systems will require many CPU cores to just poll and process requests. Another way to build active disaggregated systems is to offload computation at storage nodes to hardware [9, 12, 17, 33, 35, 38, 39, 54, 57, 57]. These solutions either require significant hardware implementation efforts (*e.g.*, FPGA-based) or incur performance scalability issues (*e.g.*, ARM-SoC-based).

Compared to active disaggregation, the passive approach of disaggregation largely reduces the owning, energy, and development costs of storage nodes by avoiding busy polling at storage nodes and shifting the rest of the computation to compute nodes. Unfortunately, the passive approach has largely been overlooked in the community. HPE's "The Machine" (Memory-Driven Computing) project [19, 23, 24, 37, 64] is one of the few existing proposals [40, 41] that adopt the passive model. So far, HPE has (separately) built a hardware prototype and a software layer. The hardware prototype [20] connects a set of SoCs to a set of DRAM/PM chips in a rack over a proprietary photonic network. To use this hardware prototype, application developers need to build software layers to manage and access data in DRAM/PM. HPE has also been building a software memory-store solution on top of a Superdome NUMA machine [37, 64]. This solution assumes certain features from future interconnect technologies, does not support data redundancy, and is work done in parallel with us. Although being a significant initial step in passive disaggregation research, the Machine project only explores one design choice and relies heavily on special network to access and manage disaggregated memory. Moreover, its design details are not open to the public.

3 pDPM Overview

This section gives an overview of pDPM, its unique challenges, the interface and guarantees our proposed pDPM systems have, and the network layer they employ. Table 1 summarizes the comparison of our proposed pDPM systems and traditional distributed PM and remote memory systems.

3.1 Passive Disaggregated Persistent Memory

Our definition of the pDPM architecture consists of two concepts: separating PM from compute servers into a PM-based storage pool and eliminating processing needs at these separated PM nodes (DNs).

The first concept is in the same spirit of current disaggregated storage systems and shares many of their benefits: it is flexible to manage and customize the PM storage pool; it offers high resource utilization, since data can be allocated at any DNs; datacenters can scale DNs independently from other servers; and it is easy to add, remove, and upgrade DNs without the need to change existing (compute) servers.

The second concept follows the more aggressive disaggregation approach of forming resource pools with just hardware (PM in our case). Such PM pools can be a set of regular servers equipped with PM or a set of network-attached devices with just network functionality and some PM. The former frees entire server CPUs to perform other tasks, while the latter eliminates the need for a processor and its hardware/server packaging all together, reducing not only the energy cost but also the building cost of DNs. Moreover, by removing processing from DNs, pDPM also avoids DN-side processor being a performance scalability bottleneck.

3.2 pDPM Challenges

pDPM offers many cost and manageability benefits and is now feasible to build with fast, “one-sided” network communication technologies like RDMA. However, it is only attractive when there is no or minimal performance loss compared to other more expensive solutions. Building a pDPM storage system that can lower the cost but maintain the performance of non-pDPM systems is hard. Different from traditional distributed storage and memory systems, DNs can only be accessed *and* managed remotely. A major technical hurdle is in providing good performance with concurrent data accesses. The lack of processing power at DNs makes it impossible to orchestrate (e.g., serialize) concurrent accesses there. Managing distributed PM resources without any pDPM-local processing is also hard and when performed improperly, can largely hurt foreground performance. In addition, DNs can fail independently. Such failures should be handled properly to ensure data reliability and availability.

Different from traditional disaggregated storage that is based on SSDs or hard disks, PM is orders of magnitude faster [70]. Although today’s datacenter network speed has also improved significantly [43], pDPM storage systems should still try to minimize network RTTs.

Different from disaggregated memory systems [40, 41, 55], pDPM is a persistent storage system and should sustain power failures and node failures. Thus, we need to ensure the consistency of data and metadata during crash recovery and provide redundancy for high availability and reliability.

3.3 System Interface and Guarantees

Clover and our two alternative pDPM systems provide the same interface and guarantees to applications. They are key-value stores supporting variable-sized entries, where users can create, read (get), write (put), and delete a key-value entry. Different CNs can have shared access to the same data. All our pDPM systems ensure the atomicity of an entry across concurrent readers and writers. A successful write indicates that the data entry is committed (atomically). Reads only see committed value. We choose to build key-value stores on the pDPM architectures because key-value stores are widely used in many datacenters. Similarly, we choose single-entry atomic write and read committed because these consistency and isolation levels are widely used in many data store systems [30, 47] and can be extended to other levels.

Our pDPM systems are intended for storing data persistently. They provide crash consistency, data reliability, and high availability. After recovering from crashes at arbitrary points, each data entry is guaranteed to contain either only new data values or only old ones. In addition, all our three systems support replicated writes across DNs.

3.4 Network Layer

We choose RDMA as the network technology to connect all servers and DNs. We use RDMA’s RC (Reliable Connection) mode which supports one-sided RDMA operations and ensures lossless and ordered packet delivery. Similar to prior solutions [15, 30, 31, 67], we solve RDMA’s scalability issues by registering memory regions using huge memory pages with RDMA NICs. Note that we use regular RDMA writes as persistent write for our evaluation, since the RDMA durable write commit in the IETF standard takes one network round trip [60], same as non-durable RDMA write.

4 Alternative pDPM Systems

Before Clover, we built two other pDPM systems during our exploration of the pDPM architectures. They follow the same interface and deliver the same consistency and relia-

System	CapEx ($)	R-RTT	W-RTT	Energy	Scalability	Metadata	Performance
Distributed PM	46736	0-N	0-N	High	Neither	Large	Good only when accessing data on local node
aDPM w/ CPU	79888	1	1	High	w/ both⋆	Small	Good overall
aDPM w/ BlueField	80080	1	1	Low	Neither	Small	Good under light load
pDPM-Direct	53096	1	4(4)	Low	w/ DN†	Large	Best for small-sized read
pDPM-Central	58096	2	2(2)	High	Neither	Small	Not good for read-intensive traffic
Clover	58096	1	2(3)	Low	w/ both	Medium	Good overall (unless high write contention)

Table 1: **Comparison of Distributed PM Architectures.** *The CapEx column represents dollar cost to build eight CNs and eight PMs. Section 6 discusses the details of CapEx and energy (CPU utilization) calculation. The R-RTT and W-RTT columns show the number of RTTs required to perform a read and a write (with replication). All RTT values are measured when there is no contention. RTTs in distributed PM's read/write, N, depends on protocols and whether data is local. The Scalability column shows if a system is scalable with the number of CNs, the number of DNs, both, or neither. ⋆ only when there are enough CPU cores. † only scalable when there is no contention. The metadata columns show the space needed to store the metadata of a data entry.*

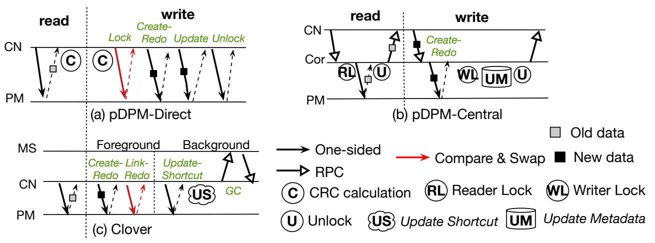


Figure 2: **Read/Write Protocols of pDPM Systems.**

bility guarantees as Clover. Even though the main system we present in this paper is Clover, we have spent significant amount of efforts on optimizing the performance of these alternative systems and on adding replication and crash recovery support to them. They can be used as stand-alone systems apart from being comparison points of Clover. Because of space constraint, we only briefly present their basic data structures and read/write protocols. We omit the discussion of their replication and crash recovery protocols.

4.1 Direct Connection

Our first alternative pDPM system, *pDPM-Direct*, connects CNs directly to DNs (Figure 1(c)). CNs perform un-orchestrated, direct accesses to DNs using one-sided RDMA operations. The main challenge in designing pDPM-Direct is the difficulty in coordinating CNs for various data and metadata operations.

To avoid frequent space allocation (which requires coordination across CNs), we pre-assign two spaces for each data entry, one to store committed data where reads go to (the *committed space*) and one to store in-flight, new data (the *un-committed space*). CNs allocate these spaces at data-entry creation time with the help of a distributed consensus protocol. Afterwards, their locations do not change until data-entry free time. CNs locally store all the metadata (*e.g.*, the locations of committed and uncommitted spaces) to avoid reading and writing metadata to DNs and the cost of ensuring metadata consistency under concurrent accesses.

To support synchronized concurrent data accesses and to avoid reading intermediate data during concurrent writes, a straightforward method and our strawman version is to always lock a data entry when reading or writing it using a distributed lock. Doing so incurs two additional network RTTs for each data access (one for lock and one for unlock).

For better performance, we adopt a lock-free, checksum-based read mechanism, which allows reads to take only one RTT. Specifically, we associate a CRC (error detection code) checksum with each key-value entry at DNs. To read a data entry, a CN uses its stored metadata to find the location of the data entry's committed space. It then reads both the data and its CRC from the DN with an RDMA read. Afterwards, the CN calculates the CRC of the fetched data and compares this calculated CRC with the fetched CRC. If they do not match, then the read is incomplete (an intermediate state during an ongoing write), and the CN retries the read request. Although calculating CRCs adds some performance overhead, it is much lower than the alternative of locking. Figure 2(a) illustrates pDPM-Direct's read and write protocols.

pDPM-Direct still requires locking for writes. We designed an efficient, RDMA-based implementation of write lock. We associate an 8-byte value at the beginning of each data entry as its lock value. To acquire the lock, a CN performs an RDMA `c&s` (compare-and-swap) operation to the value. The `c&s` operation compares whether the value is 0. If so, it sets it to 1. Otherwise, the CN retries the `c&s` operation. To release the lock, the CN performs an RDMA write and sets the value to 0. Our lock implementation leverages the unique feature of the pDPM model that all memory accesses to DNs come from the network (i.e., the NIC). Without any yprocessor's accesses to memory, the DMA protocol guarantees that network atomic operations like `c&s` are atomic across the entire DN [14, 61].

To write a data entry, a CN first calculates and attaches a CRC to the new data entry. Afterwards, the CN locates the entry with its local metadata and locks the entry (one RTT). The CN then writes the new data (with the CRC) to the un-committed space (one RTT), which serves as the *redo* copy used during recovery if a crash happens. Afterwards, the CN writes the new data to the committed space with an RDMA write (one RTT). At the end, the CN releases the lock (one RTT). The total write latency is four RTTs plus the CRC calculation time (when no contention), and two of these RTTs contain data.

4.2 Connecting Through Coordinator

Our second alternative pDPM system, *pDPM-Central* (Figure 1(c)), uses a central *coordinator* to orchestrate all data accesses and to perform metadata and management operations. All CNs send RPC requests to the coordinator, which handles them by performing one-sided RDMA operations to DNs. We implement our RPC using HERD's RPC design [30]; other RPC designs can easily be integrated too. To achieve high throughput, we use multiple RPC handling threads at the coordinator. Figure 2(b) illustrates pDPM-Central's read and write protocols.

Since all requests go through the coordinator, it can serve as the serialization point for concurrent accesses to a data entry. We use a local read/write lock for each data entry at the coordinator to synchronize across multiple coordinator threads. In addition to orchestrating data accesses, the coordinator performs all space allocation and de-allocation of data entries. The coordinator uses its local PM to persistently store all the metadata of a data entry.

To perform a read, a CN sends an RPC read request to the coordinator. The coordinator finds the location of the entry's committed data using its local metadata, acquires its local lock of the entry, reads the data from the DN using a one-sided RDMA read, releases its local lock, and finally replies to the CN's RPC request. The end-to-end read latency a CN observes (when there is no contention) is two RTTs, and both RTTs involve sending data.

To perform a write, the coordinator allocates a new space at a DN for the new data and then writes the data there. We do not need to lock (either at coordinator or at the DN) during this write, since it is an out-of-place write to a location that is not exposed to any other coordinator RPC handlers. After the write, the coordinator updates its local metadata with the new data's location and flushes this new location to its local PM for crash resistance. The total write latency without contention is two RTTs, both containing data.

4.3 pDPM-Direct/-Central Drawbacks

pDPM-Direct delivers great read performance when read size is small, since it only requires one lock-free RTT and it is fast to calculate small CRC. Its write performance is much worse because of high RTTs and lock contention. Its scalability is also limited because of lock contention during concurrent writes. Moreover, pDPM-Direct requires large space for both data and metadata. For each data entry, it doubles the space because of the need to store two copies of data. The metadata overhead is also high, since all CNs have to store all the metadata.

pDPM-Central largely reduces write RTTs over pDPM-Direct and thus has good write performance when the scale of the cluster is small. Unlike pDPM-Direct, CNs in pDPM-Central do not need to store any metadata. However, from our experiments, the coordinator soon becomes the performance bottleneck when either the number of CNs or the number of DNs increases. pDPM-Central's read performance is also worse than pDPM-Direct with the extra hop between a CN and the coordinator. In addition, the coordinator's CPU utilization is high, since it needs many RPC handler threads to sustain parallel requests from CNs.

5 Clover

To solve the problems of the first two pDPM systems we built, we propose Clover (Figure 1(e)). The main idea of Clover is to separate the location, the communication method, and the management strategy of the data plane and the control plane. It lets CNs directly access DNs for all data operations and uses one or few metadata servers (*MS*s) for all control plane operations.

To avoid MS being the scalability bottleneck, we support multiple MSs, each serving a shard of data entries. Each MS stores the metadata of the data entries it is in charge of in its local PM. We keep the amount of metadata small. The storage overhead of metadata is below 2% for 1 KB data entries. CNs cache the metadata of hot data entries. Under memory pressure, CNs evict metadata with a replacement policy (we currently support FIFO and LRU).

Clover aims to deliver scalable, low-latency, high-throughput performance at the data plane and to avoid the MS being the bottleneck at the control plane. Our overall approaches to achieve these design goals include: 1) moving all metadata operations off performance critical path, 2) using lock-free data structures to increase scalability, 3) employing optimization mechanisms to reduce network round trips for data accesses, and 4) leveraging the unique atomic data access guarantees of pDPM. Figure 2(c) shows the read and write protocol of Clover. Figure 3 illustrates the data structures used in Clover.

5.1 Data Plane

To achieve our data plane design goals, we propose a new mechanism to perform lock-free, fast, and scalable reads and writes. Specifically, we allow multiple committed versions of a data entry in DNs and link them into a *chain*. Each committed write to a data entry will move its latest version to a new location. To avoid the need of updating CNs with the new location, we use a self-identifying data structure for CNs to find the latest version.

We include a *header* with each version of a data entry. The header contains a *pointer* and some metadata bits used for garbage collection. The pointers chain all versions of a data entry together in the order that they are written. A `NULL` pointer indicates that the version is the latest.

A CN acquires the header of the chain head from the MS at the first access to a data entry. It then caches the header locally to avoid the overhead of contacting MS on every data access. We call a CN-cached header a *cursor*.

Read. Clover reads are lock-free. To read a data entry, a CN performs a *chain walk*. The chain walk begins with fetching

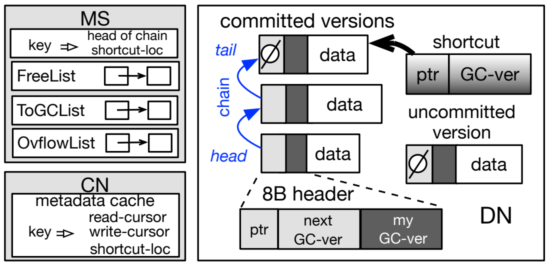


Figure 3: **Clover System Design.**

the data buffer version that the CN's cursor points to. It then uses the pointer in this fetched buffer to read the next version. The CN repeats this step until reading a `NULL` pointer, which indicates that it has read the latest version. All steps in the chain walk use one-sided RDMA reads. After a chain walk, the CN updates its cursor to point to the latest version.

A chain walk can be slow with long chains when a cursor is not up to date [68]. Inspired by skip lists [53], we solve this issue by using a *shortcut* to directly point to the latest version or a recent version of each data entry. Shortcuts are *best effort* in that they are intended but not enforced to always point to the latest version of an entry. The shortcut of a data entry is stored at its DN. The location of a shortcut never changes during the lifetime of the entry. MS stores the locations of all shortcuts. When a CN first accesses a data entry, it retrieves the location of its shortcut from MS and caches it locally.

The CN issues a chain walk read and a shortcut read in parallel. It returns the user request when the faster one finishes and discards the other result. We do not replace chain walks completely with shortcut reads, since shortcuts are updated asynchronously in the background and may not be updated as fast as the cursor. When the CN's cursor points to the latest version of a data entry, a read only takes one RTT.

Write. Clover never overwrites existing data entries and performs a lock-free, out-of-place write before linking the new data to an entry's chain. To write a data entry, a CN first selects a free DN space assigned to it by MS in advance (see §5.2). It performs a one-sided RDMA write to write the new data to this buffer. Afterwards, the CN performs an RDMA `c&s` operation to link this new data to the tail of the entry's version chain. Specifically, the `c&s` operation is on the header that the CN's cursor points to. If the pointer in the header is `NULL`, the `c&s` operation swaps the pointer to point to the new data, and we treat this new data as a *committed* version. Otherwise, it means that the cursor does not point to the tail of the chain and the CN performs a chain walk to reach the tail and then issues another `c&s`.

Afterwards, the CN uses a one-sided RDMA write to update the shortcut of the entry to point to the new data version. This step is off the performance critical path. The CN also updates its cursor to the newly written version. We do not invalidate or update other CNs' cursors at this time to improve the scalability and performance of Clover.

Clover' chained structure and write mechanism ensure that writers do not block readers and readers do not block writers. They also ensure that readers can only view committed data. Without high write contention to the same data entry, one write takes only two RTTs.

Retire. After committing a write, a CN can *retire* older versions of the data entry, indicating that the buffer spaces can be reclaimed. To improve performance and minimize the need to communicate with MS, CNs lazily send asynchronous, batched retirement requests to MS in the background. We further avoid the need for MS to invalidate CN-cached metadata using a combination of timeout and epoch-based garbage collection (see §5.2).

5.2 Control Plane

CNs communicate with MS using two-sided operations for all metadata operations. MS performs all types of management of DNs. We carefully designed the MS functionalities for best performance and scalability.

Space allocation. With Clover's out-of-place write model, Clover has high demand for DN space allocation. We use an efficient space allocation mechanism where MS packages free spaces of all DNs into *chunks*. Each chunk hosts data buffers of the same size. Different chunks can have different data sizes. Instead of asking for a new free space before every write, each CN requests multiple spaces at a time from MS in the background. This approach moves space allocation off the performance critical path and is important to deliver good write performance.

Garbage collection. Clover' append-only chained data structure makes its writes very fast. But like all other append-only and log-structured storage systems, Clover needs to garbage collect (GC) old data. We design a new efficient GC mechanism that does not involve *any* data movement or communication to DN. It also minimizes the communication between MS and CNs.

The basic flow of GC (a strawman implementation) is simple: MS processes incoming retire requests from CNs by putting reclaimed spaces to a free list (*FreeList*). It gets free spaces from the FreeList when a CN requests more free buffers. A free space can be used by any CN for any new writes, as long as the size fits.

Although the above strawman implementation is simple, making GC work correctly, efficiently, and scale well is challenging. First, to achieve good GC performance, we should avoid the invalidations of CN cached cursors after reclaiming buffers to minimize the network traffic between MS and CNs. However, with the strawman GC implementation, CNs' outdated cursors can cause failed chain walks. We solve this problem using two techniques: 1) MS does not clear the header (or the content) of a data buffer after reclaiming it, and 2) we assign a *GC version* to each data buffer. MS increases the GC version after reclaiming a data buffer. It gives this new GC version together with the location of the buffer when assigning the buffer as a free space to a CN, CN_k. Before CN_k uses the space for a new write, the content of this space at the DN contains old data and old GC version. After CN_k uses the space for a write, it contains new data and new GC version. Other CNs that have cached cursors to this buffer need to differentiate these two cases. A CN tells if a buffer contains its intended data by comparing the GC version in its cached cursor to the one it reads from the DN. If they do not match, the CN will discard the read data and invalidate its cached cursor. Our GC-version approach not only avoids the need for MS to invalidate cursor caches on CNs, but also eliminates the need for MS to access DNs during GC.

The next challenge is related to our goal of read-isolation and atomicity guarantees (*i.e.*, readers always read the data that is consistent to its metadata header). An inconsistent read can happen if the read to a data buffer takes long, and during the reading time, this buffer has been retired by another CN, reclaimed by MS, assigned to a CN as a newly allocated buffer, and used to perform a write. We use a read timeout scheme similar to FaRM [15] to prevent this inconsistent case. Specifically, we abort a read operation after two RTTs, since the above steps in the problematic case take at least (and usually a lot more than) two RTTs (one for a CN to submit the retirement request to MS and one for MS to assign the space to a new CN).

The final challenge is the overflow of GC versions. We can only use limited number of bits for GC version in the header of a data buffer (currently 8 bits), since the header needs to be smaller than the size of an atomic RDMA operation. When the GC version of a buffer increases beyond the maximum value, we have to restart it from zero. With just our GC mechanism so far, CNs will have no way to tell if a buffer matches its cached cursor version or has advanced by $2^8 = 256$ versions. To solve this rare issue without invalidation traffic to CNs, we use an epoch-based timeout mechanism. When MS finds the GC version of a data buffer overflows, it puts the reclaimed buffer into an *OvflowList* and waits for T_e (a configurable time value) before moving it to the *FreeList*. All CNs invalidate their own cursors after an inactive period of T_e (if during this time, the CN access the buffer, it would have advanced the cursor already). To synchronize epoch time, MS sends a message to CNs after T_e. Epoch messages are the only communication from MS to CNs during GC.

5.3 Discussion

The Clover design offers four benefits. First, Clover yields the best performance among all pDPM systems; it outperforms pDPM-Direct and pDPM-Central for both reads and writes, and both with and without contention. Achieving this low latency while guaranteeing atomic write and read committed is not easy. Four approaches enable us to reach this goal: 1) ensuring that the data path does not involve MS, 2) reducing metadata communication to MS and moving it off performance critical paths, 3) ensuring no memory copy in the whole data path, and 4) leveraging the unique advantages of pDPM to perform RDMA atomic operations.

Second, Clover scales well with the number of CNs and DNs, since its reads and writes are both lock free. Readers do not block writers or other readers and writers do not block readers. Concurrent writers to the same entry only contend for the short period of an RDMA `c&s` operation. Clover also minimizes the network traffic to MS and the processing load on MS, which enables MS to scale well with the number of CNs and with the amount of data operations.

Third, we avoid data movement and communication between MS and DNs entirely during GC. To scale and support many CNs with few MSs, we also avoid CN invalidation messages. MS does not need to proactively send any other messages to CNs either. Essentially, MS never *pushes* any messages to CNs. Rather, CNs *pull* information from MS. Furthermore, MS adopts a thread model that adaptively lets working threads sleep to reduce MS's CPU utilization.

Finally, the Clover data structure is flexible and can support load balancing very well. Different versions of a data entry do not need to be on the same DN. As we will see in §5.4 and §5.5, this flexible placement is the key to Clover's load balancing and data replication needs.

However, Clover also has its limitation. Each write in Clover requires two RTTs and under heavy contention, its write performance degrades. As we will see in §6, two-sided aDPM systems outperform Clover with write-intensive workloads, since they can complete writes in one RTT. Fortunately, most datacenter workloads are read-most [7], and under common cases, Clover delivers great performance.

5.4 Failure Handling

DNs can fail independently from CNs. Clover needs to handle both transient and permanent failures of a DN. For the former, Clover guarantees that a DN can recover all its committed data after reboot (*i.e.*, crash consistent). For the latter, we add the support of data replication across multiple DNs to Clover. In addition, Clover also handles the failure of MS.

Recovery from transient failures. Clover's recovery mechanism of a single DN's transient failure is simple. If a DN fails before a CN successfully links the new data it writes to the chain (indicating an un-committed write), the CN simply discards the new write by treating the space as unused.

Data redundancy. With the user-specified degree of replication being N, Clover guarantees that data is still accessible after $N-1$ DNs have failed. We propose a new atomic replication mechanism designed for the Clover data structure.

The basic idea is to link each version of a data entry D_N to all the replicas of the next version (*e.g.*, D^a_{N+1}, D^b_{N+1}, D^c_{N+1} for three replicas) by placing pointers to all these

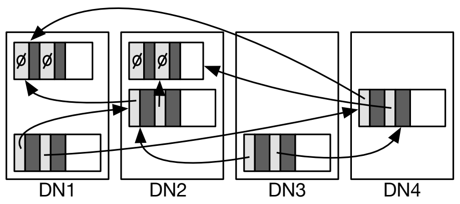


Figure 4: **Replicated Data Entry.** *A replicated data entry on four DNs. The replication factor is two.*

replicas in the header of D_N. Figure 4 shows an example of a replicated data entry (with the degree of replication being 2). With this all-way chaining method, Clover can always construct a valid chain as long as one copy of each version in a data entry survives.

Each data entry version has a primary copy and one or more secondary copies. To write a new version, D_{N+1}, to a data entry whose current tail is D_N with R replicas, a CN first writes the new data to R DNs. In parallel, the CN performs a one-sided c&s to a bit, B_w, in the header of the primary copy of D_N to test if the entry is already in the middle of a replicated write. If not, the bit will be set, indicating that the entry is now under replicated write. All the writes and the c&s operation are sent out in parallel to minimize latency. After the CN receives the RDMA acknowledgment of all the operations, it constructs a header that contains R pointers to the copies of D_{N+1} and writes it to all the copies of D_N. Once the new header is written to all copies of D_N, the system can recover D_{N+1} from crashes (up to $R-1$ concurrent DN failure).

MS redundancy. MSs manage several types of metadata. Among them, the only type of metadata that cannot be reconstructed is keys (of key-value entries) and the mapping from a key to the location of its data entries in DNs. To avoid MS being the single point of failure, we implement a mechanism to include one or more *backup* MS. When creating (deleting) a new key-value data entry, the primary MS synchronously replicates (removes) the key and the head of the value chain to all the backup MSs. These metadata are the only metadata that cannot be reconstructed. MSs reconstruct all other metadata by reading value chains in DNs.

5.5 Load Balancing

A pDPM system has a pool of DNs. It is important to balance the load to each of them. We use a novel two-level approach to balance loads in Clover: globally at MS and locally at each CN. Our global management leverages two features in Clover: 1) MS is the party that assigns all new spaces to CNs, and 2) data versions of the same entry in Clover can be placed on different DNs. To reduce the load of a DN, MS assigns more free spaces from other DNs to CNs at allocation time. Each CN internally balances the load to different DNs at runtime. Each CN keeps one bucket per DN to store free spaces. It chooses free spaces from different buckets for new writes according to its own load balancing needs.

6 Evaluation Results

This section presents the evaluation results of Clover. We compare it with pDPM-Direct, pDPM-Central, two distributed PM-based systems, Octopus [42] and Hotpot [56], and a two-sided RDMA-based key-value store, HERD [30]. All our experiments were carried out in a cluster of 14 machines, connected with a 100 Gbps Mellanox InfiniBand Switch. Each machine is equipped with two Intel Xeon E5-2620 2.40GHz CPUs, 128 GB DRAM, and one 100 Gbps Mellanox ConnectX-4 NIC.

In order to compare the pDPM architecture with a low-cost aDPM architecture, we use Mellanox BlueField, a SmartNIC that includes an ARM-based SoC and a 100 Gbps Mellanox ConnectX-5 NIC [44]. We port HERD to BlueField by migrating it from x86 to ARM (we call the ported HERD running on BlueField HERD-BF).

Unfortunately, at the time of writing, we cannot get hold of real Intel Optane DC, and we use DRAM as PM. Our experiments run on machines with PCIe 3.0 $\times 8$ (7.87 GB/s), and the bandwidth from RDMA-NIC to DRAM is capped by it, making the effective bandwidth at most 7.87 GB/s. Intel Optane DC's read bandwidth is 6.6 GB/s [70], which is close to PCIe 3.0 $\times 8$. Thus, we envision read results to be similar with real Optane. Optane's write bandwidth is 2.3 GB/s, and there may be some difference in our write results with real Optane. But since our target is read-most workloads, we believe that the conclusion we make from our evaluation will still be valid with real PM.

6.1 Micro-benchmark Performance

We first evaluate the basic read/write latency of Clover and the systems in comparison using a simple micro-benchmark where a CN synchronously reads/writes a key-value data entry on a DN. For this and all the rest of our experiments, we use HERD's default configuration of 12 busy polling receiving side's threads for both HERD and HERD-BF.

Figure 5 plots the read latency with different request sizes. We use native RDMA one-sided read as the baseline. Overall, Clover's performance is the best among all systems and is only slightly worse than native RDMA. pDPM-Direct has great read performance when the request size is small. However, when request size increases, the overhead of CRC calculation dominates, largely hurting pDPM-Direct's read performance. As expected, pDPM-Central's read performance is not good because of its 2-RTT read protocol. HERD performs worse than Clover because it requires some extra CPU processing time for each read. HERD-BF has a constant overhead over HERD mainly because its processing is performed in BlueField's low-power ARM cores.

Figure 6 plots the average write latency comparison. We use native RDMA one-sided write as a baseline here. Among pDPM systems, Clover and pDPM-Central achieve the best write latency. pDPM-Direct's write performance is the worst

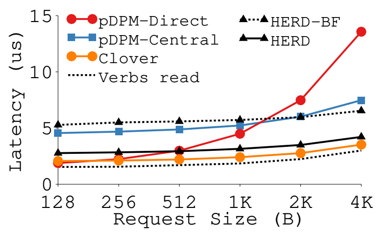


Figure 5: **Read Latency.**

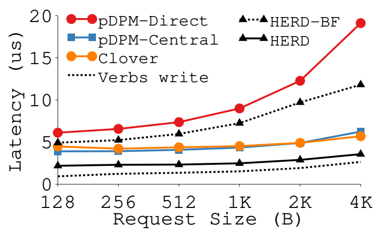


Figure 6: **Write Latency.**

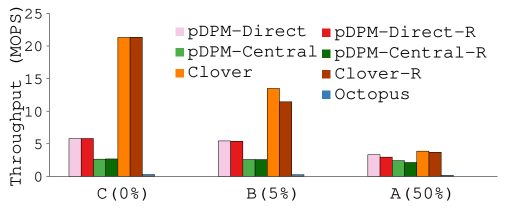


Figure 7: **Throughput Comparison with YCSB.** *Running YCSB on four CNs and four DNs.*

because of its 4-RTT write protocol. Its write performance also gets worse with larger request size because of the increased overhead of CRC calculation. HERD outperforms Clover and other pDPM systems because two-sided communication allows it to complete a write within one RTT. HERD-BF is still a lot worse than HERD because of BlueField's low processing power.

6.2 YCSB Performance and Scalability

We now present our evaluation results using the YCSB benchmark [13, 71]. We use a total of 100K key-value entries and 1M operations per test. The accesses to keys follow the Zipf distribution. We use three workloads with different read-write ratios: read only (workload C), 5% write (workload B), and 50% write (workload A). These three workloads follow common application patterns in datacenters [7] and are the set that previous PM and in-memory store systems used for evaluation [30, 38, 47, 69].

Basic performance. We first evaluate the performance of Clover, pDPM-Direct, pDPM-Central, Octopus, and Hotpot under the same configuration: 4 CNs and 4 DNs, each CN running 8 application threads. Neither Octopus nor Hotpot directly support key-value interface. In order to run the YCSB key-value store workloads, we run MongoDB [50], a key-value store database, on top of Octopus and Hotpot. Note that HERD only supports one DN and we cannot compare with HERD or HERD-BF in this experiment. We also evaluate replication with our three pDPM systems here (with degree of replication 2). Figure 7 shows the overall performance of these systems. Hotpot yields similar performance as Octopus and we omit its results in the figure.

Clover performs the best among all systems for all workloads. We further look into the Clover results and find that the average number of hops during chain walks is only 0.2 to 0.3 for reads and 3.7 to 3.9 for writes. pDPM-Direct performs better with read-most workloads than write-most workloads. This is because without the need to perform any locking, its read performance is not affected by contention. pDPM-Central's performance is the worst among pDPM systems, because under contention (Zipf distribution), the coordinator becomes the bottleneck.

The overall performance of Octopus and Hotpot is more than an order of magnitude worse than all pDPM systems. There are two main reasons. First, these systems do not directly support key-value interface, and running MongoDB on top of them adds overhead. Unfortunately, there is no existing distributed PM systems that directly support key-value interface as far as we know. Second, each read and write operation in these systems involves a complex protocol that requires RPCs across multiple nodes.

Workload	Median	Average	99%
C	1	1	1
B	1	1.26	5
A	1	1.33	6

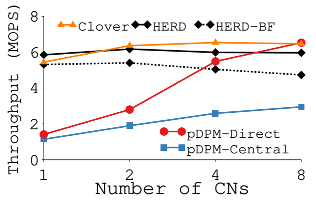
Table 2: **Clover RTTs.**

To further understand Clover's performance, we measure the number of RTTs incurred when running YCSB on Clover. Table 2 shows the median, average, and 99% RTTs of Clover. Clover requires only one RTT for read-most workloads. Even for 50% write (workload A), Clover only incurs six RTTs at 99% and one RTT at median.

Replication overhead. As expected, adding redundancy lowers the throughput of write operations for all pDPM systems. Even though these systems issue the replication requests in parallel, they only use one thread to perform asynchronous RDMA read/write operations, and doing so still has an overhead. However, the overhead is small.

Scalability. Next, we evaluate the scalability of different systems with respect to the number of CNs and the number of DNs. Figure 8 shows the scalability of pDPM systems, HERD, and HERD-BF when varying the number of CNs with a single DN. Clover and HERD have the best (and similar) performance with workload C. Both systems saturate network bandwidth, and neither have any scalability bottlenecks. With workload B, the performance of Clover is slightly worse than HERD because of increased write contention. HERD-BF performs worse and scales worse than Clover and HERD for both workloads mainly because of its limited processing power. pDPM-Central performs the worst and does not scale well with more CNs. pDPM-Direct also performs poorly with fewer CNs. Apart from the limitation of these system's designs, their inefficient thread models also contribute to their worse performance.

Figure 9 shows the scalability of pDPM data stores w.r.t. the number of DNs (HERD only supports single memory node and we cannot include it in this experiment). Clover scales well with DNs because CNs access DNs directly for data accesses, having no scalability bottleneck. pDPM-


(a) Workload C (0%)

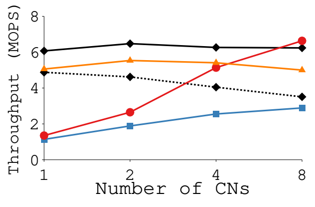
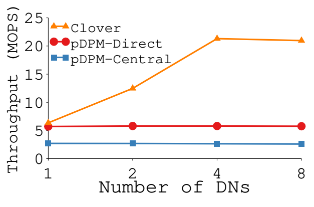

(b) Workload B (5%)

Figure 8: **Scalability w.r.t. CNs.** *Running 1 DNs.*


(a) Workload C (0%)

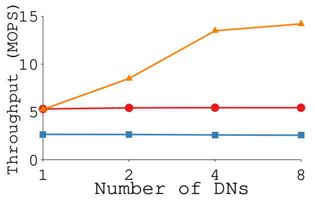

(b) Workload B (5%)

Figure 9: **Scalability w.r.t. DNs.** *Running 4 CNs.*

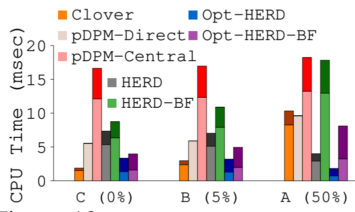


Figure 10: **CPU Utilization.** *Lighter colors and darker colors represent the CPU time used by the client side and the server side. Opt-HERD and Opt-HERD-BF are hypothetical optimal values.*

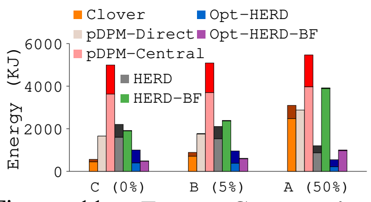


Figure 11: **Energy Consumption.** *Lighter colors and darker colors represent the CPU time used by the client side and the server side. Opt-HERD and Opt-HERD-BF are hypothetical optimal values.*

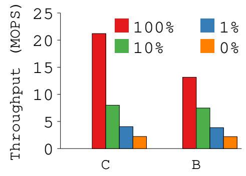


Figure 12: **Effect of Metadata Cache in Clover.**

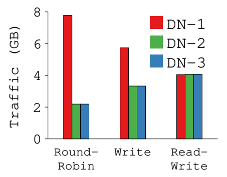


Figure 13: **Load Balancing in Clover.**

Central has poor scalability because of the coordinator being the bottleneck that all requests have to go through. Surprisingly, pDPM-Direct's scalability is also poor. This is because when the number of DNs increases, network bandwidth has not become a performance bottleneck, but CNs need to do more CRC calculation to read/write to more DNs. This computation overhead becomes the performance bottleneck.

6.3 CPU Utilization and Cost

CPU utilization and energy cost. We evaluate the CPU utilization of different systems by calculating the total CPU time to complete ten million requests in YCSB's workloads A, B, and C, as illustrated in Figure 10. We further separate the CPU time used at client side (CNs) and at server side (DNs, the coordinator, MS). The three pDPM systems run four CNs and four DNs. HERD and HERD-BF run four CNs and one DN. Since HERD only supports one DN, to estimate the CPU utilization and energy of a scale-out version of HERD, we hypothetically assume that HERD can achieve perfect scaling (*i.e.*, we reduce HERD's total run time by a factor of four to model it running on four CNs and four DNs). This hypothetical calculation is the optimal performance HERD could have achieved.

We further calculate the total energy cost using the power consumption of our CPU core [34] and the ARM core of BlueField [52]. Figure 11 plots this result. We do not include the energy cost of PM, since it is the same for all systems.

For read-most workload, pDPM-Direct and Clover use less CPU time than pDPM-Central and HERD because they perform one-sided RDMA directly from CNs to DNs. HERD's total CPU time is much longer than Clover even with optimal scale-out calculation, because it uses many busy-polling threads at its server side to achieve good performance (12 threads by default). Surprisingly, HERD-BF's energy is higher than HERD even when the power consumption of an ARM core is more than an order of magnitude lower than our CPU core. HERD-BF's worse performance makes each request to run longer and consumes more power. pDPM-Central has high CPU utilization because the coordinator's CPU spends time on every request, and the total time to finish the workloads with pDPM-Central is long. HERD's write performance and energy are both better than Clover. pDPM systems consume more energy for write-heavy workloads because of their degraded write performance.

CapEx. Table 1 summarizes the cost to build different data stores with 8 CNs and 8 DNs (for distributed PM, we use eight machines in total). The CapEx is calculated with the market price of 128 GB Intel Optane PM ($842 [5]), Mellanox ConnectX-4 NIC ($795 [45]), Mellanox BlueField NIC ($4168 [1]), and a DELL R740 server with the same configuration as what we use in our experiments ($5000). For servers with PM, we adjust the price difference between PM and DRAM to the whole server price ($4144). Distributed PM has the lowest CapEx because it can share PM and only needs eight machines in total. aDPM with CPU requires 16 machines in total (8 for CNs and 8 for DNs). The three pDPM systems and aDPM with BlueField do not require full machines for DNs and we only include PM and NIC costs for them. Surprisingly, the cost of BlueField is similar to a full machine. We suspect that this is because BlueField is in a new and small market, and we expect its price to drop in the future (but still not as low as pDPM).

6.4 Metadata Caching

Each data entry in Clover requires a constant of 8 B metadata (which is much smaller than typical key-value sizes of 100 B - 1000 B [7]). To evaluate the effect of different sizes of metadata cache at CNs in Clover, we ran the same YCSB workloads and configuration as Figure 7 and plot the results in Figure 12. Here, we use the FIFO eviction policy (we also tested LRU and found it to be similar or worse than FIFO). With smaller metadata cache, all workloads' performance drop because a CN has to get the metadata from the MS before accessing a data entry that is not in the local metadata cache. With no metadata cache (0%), CNs need to get metadata from the MS before every request. However, under Zipf distribution, with just 10% metadata cache, Clover can already achieve satisfying performance.

6.5 Load Balancing

To evaluate the effect of Clover's load balancing mechanism, we use a synthetic workload with six data entries, a, b, and $c1$ to $c4$. The workload initially creates a (no replication) and b (with 3 replicas) and reads these two entries continuously. At a later time, it creates $c1$ to $c4$ (no replication) and keeps updating them. One CN runs this synthetic workload on three DNs. Figure 13 shows the total traffic to the three DNs with different allocation and load-balancing policies. With a naive policy of assigning DNs to new write requests in a round-robin fashion and reading from the first replica, write traffic spreads evenly across all DNs but reads all go to *DN-1*. With write load balancing, MS allocates free entries for new writes from the least accessed DN. Doing so spreads write traffic more towards the lighter-loaded *DN-2* and *DN-3*. With read load balancing, Clover spreads read traffic across different replicas depending on the load of DNs. As a result, the total loads across the three DNs are completely balanced.

7 Conclusion and Discussion

This paper explores a passive disaggregated persistent memory architecture where remote PM data nodes do not need any processing. We present Clover, a low-cost, fast, and scalable pDPM key-value store which separates data and control planes. We compare Clover with two alternative pDPM systems we built, existing distributed PM systems, and disaggregated systems that include processing at data nodes. We performed extensive evaluation of these systems and learned both the benefits and the limitations of them. We end this paper by discussing our overall findings and our suggestions to future systems builders.

Cost Implication. pDPM's CapEx cost saving compared to aDPM is apparent: pDPM reduces the cost of a processor and hardware (server) packaging to host the processor. pDPM's OpEx cost saving mainly comes from avoiding polling at storage nodes. aDPM needs to busy poll network requests to achieve the low latency that can match PM's performance. Meanwhile, to sustain high-bandwidth network (*e.g.*, 100 Gbps and above), aDPM requires many CPU cores or a parallel-hardware unit like FPGA to poll and process requests in parallel, adding to more runtime cost.

Performance Implication. The major tradeoff of removing computation from storage nodes in pDPM is the potential increase in network RTTs to access storage nodes remotely. While it is generally true that moving computation towards data could achieve good performance, our pDPM key-value store systems demonstrate that with careful design, RTTs in pDPM systems could be minimized and in many cases be the same as aDPM systems. In other cases (*e.g.*, key-value put with high-contention), aDPM has unavoidable performance loss because of extra RTTs. On the other hand, not having enough processing power in aDPM (*e.g.*, when only using ARM-based SoC) could lead to significant performance loss.

Application Implication. Building applications with the pDPM model requires careful design. As we demonstrated with the three different pDPM key-value store systems, different application design choices could directly affect how well pDPM performs and scales. The best design would minimize RTTs while avoiding scalability bottlenecks, as with Clover. This paper focuses on key-value store systems, as they are widely used in many datacenter applications. Other systems such as remote file systems, databases, object stores, and data sharing can also build on the pDPM model. As systems complexity increases, it will be more difficult to optimize pDPM's RTTs with just RDMA read, write, and atomic operation interfaces. We believe that extended RDMA interfaces such as HyperLoop [35] could help in these cases.

Recommendation. This paper explores the extreme of completely removing computation power at storage nodes, which helps set baselines in designing disaggregated PM systems. Going forward, we believe that future disaggregated PM systems would benefit from a hybrid approach. Computation that fundamentally involves multiple data accesses can be moved to storage nodes, while the rest should be kept at compute nodes. Among the former, those that have require high performance can be placed on FPGA or ASIC to avoid high CPU cost, while those that can tolerate degraded performance can be placed at low-power cores.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Anirudh Badam for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper.

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

References

[1] Private conversation with mellanox sales department, March 2019.

[2] Alibaba Cloud. Super computing cluster. https://www.alibabacloud.com/product/scc, 2018.

[3] Amazon. Amazon elastic block store. https://aws.amazon.com/ebs/?nc1=h_ls, 2019.

[4] Amazon. Amazon s3. https://aws.amazon.com/s3/, 2019.

[5] Anton Shilov. Pricing of intels optane dc persistent memory modules leaks: From $6.57 per gb. https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks, 2019. visited on 01/15/20.

[6] Krste Asanovi. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers, February 2014. Keynote talk at the 12th USENIX Conference on File and Storage Technologies (FAST '14).

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, UK, June 2012.

[8] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[9] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10gbps line-rate key-value stores with fpgas. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '13)*, San Jose, CA, USA, June 2013.

[10] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems (EUROSYS '16)*, London, UK, April 2016.

[11] Brian Cho and Ergin Seyfe. Taking advantage of a disaggregated storage and compute architecture. In *Spark+AI Summit 2019 (SAIS '19)*, San Francisco, CA, USA, April 2019.

[12] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-Defined Network-Attached Key-Value Drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, Providence, RI, April 2019.

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

[14] Alexandras Daglis, Dmitrii Ustiugov, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. Sabres: Atomic object reads for in-memory rack-scale computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*, Taipei, Taiwan, October 2016.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, April 2014.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, October 2015.

[17] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (ATC '19)*, Renton, WA, July 2019.

[18] Facebook. Introducing bryce canyon: Our next-generation storage platform. https://code.fb.com/data-center-engineering/introducing-bryce-canyon-our-next-generation-storage-platform/, 2017.

[19] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*, Kartause Ittingen, Switzerland, May 2015.

[20] Gen-Z Consortium, 2018. https://genzconsortium.org.

[21] Google. Available first on Google Cloud: Intel Optane DC Persistent Memory. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory.

[22] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '16)*, Florianopolis, Brazil, August 2016.

[23] Hewlett Packard. The Machine: A New Kind of Computer. http://www.hpl.hp.com/research/systems-research/themachine/, 2005.

[24] Hewlett Packard Labs. Memory-driven computing. https://www.labs.hpe.com/memory-driven-computing, 2019.

[25] Huawei. Huawei Launches New-Gen Servers Running

on 2nd-Generation Intel Xeon Scalable Processors. https://www.huawei.com/en/press-events/news/2019/4/huawei-new-gen-servers-xeon-scalable-processors.

[26] Intel Corporation. Intel Rack Scale Architecture: Faster Service Delivery and Lower TCO. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-rack-scale-architecture.html.

[27] Intel Corporation. Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html, 2019.

[28] Intel Corporation - Product and Performance Information. Intel Non-Volatile Memory 3D XPoint. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html?wapkw=3d+xpoint, 2018.

[29] Intel Corporation - Product and Performance Information. Reimagining the data center memory and storage hierarchy. https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy/, 2019.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '14)*, Chicago, IL, USA, August 2014.

[31] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16)*, Denver, CO, USA, June 2016.

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

[33] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, Atlanta, Georgia, USA, April 2016.

[34] Patrick Kennedy. Dual intel xeon e5-2620 (v1, v2 and v3) compared. https://www.servethehome.com/dual-intel-xeon-e5-2620-v1-v2-v3-compared/, 2015.

[35] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*, Budapest, Hungary, August 2018.

[36] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, USA, 2019 2019.

[37] Mijung Kim, Jun Li, Haris Volos, Manish Marwah, Alexander Ulanov, Kimberly Keeton, Joseph Tucek, Lucy Cherkasova, Le Xu, and Pradeep Fernando. Sparkle: Optimizing spark for large memory machines and analytics. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, Santa Clara, CA, USA, September 2017.

[38] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[39] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, Lausanne, Switzerland, March 2020.

[40] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Austin, Texas, 2009.

[41] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*, New Orleans, LA, USA, February 2012.

[42] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (ATC '17)*, Santa Clara, CA, USA, July 2017.

[43] Mellanox. ConnectX-6 Single/Dual-Port Adapter supporting 200Gb/s with VPI. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card.

[44] Mellanox. Bluefield smartnic. http://

www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, 2018.

[45] Mellanox. Mellanox connectx-4 adapter product brief. https://www.mellanox.com/files/doc-2020/pb-connectx-4-vpi-card.pdf, 2020. visited on 06/01/20.

[46] Microsoft. Introducing new product innovations for SAP HANA, Expanded AI collaboration with SAP and more. https://azure.microsoft.com/en-us/blog/introducing-new-product-innovations-for-sap-hana-expanded-ai-collaboration-with-sap-and-more/.

[47] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC '13)*, San Jose, CA, USA, June 2013.

[48] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing cpu and network in the cell distributed b-tree store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC '16)*, Denver, CO, USA, June 2016.

[49] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*, Budapest, Hungary, August 2018.

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

[51] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, March 2017.

[52] Peng Peng, You Mingyu, and Xu Weisheng. Running 8-bit dynamic fixed-point convolutional neural network on low-cost arm platforms. In *2017 Chinese Automation Congress (CAC)*, Jinan, China, Oct 2017.

[53] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communication of the ACM*, 33(6):668–676, June 1990.

[54] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[55] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[56] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 8th Annual Symposium on Cloud Computing (SoCC '17)*, Santa Clara, CA, USA, September 2017.

[57] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, Heraklion, Greece, April 2020.

[58] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Bulletin of the Technical Committee on Data Engineering*, 40:40–52, March 2017. Special Issue on Distributed Data Management with RDMA.

[59] Kosuke Suzuki and Steven Swanson. The non-volatile memory technology database (nvmdb). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015.

[60] Tom Talpey and Jim Pinkerton. Rdma durable write commit. https://tools.ietf.org/html/draft-talpey-rdma-commit-00, 2016.

[61] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA '10)*, Bangalore, India, Jan 2010.

[62] TECHPP. Alibaba singles' day 2019 had a record peak order rate of 544,000 per second. https://techpp.com/2019/11/19/alibaba-singles-day-2019-record/, 2019.

[63] Tejas Karmarkar. Availability of linux rdma on microsoft azure. https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available, 2015.

[64] Haris Volos, Kimberly Keeton, Yupu Zhang, Milind Chabbi, Se Kwon Lee, Mark Lillibridge, Yuvraj Patel, and Wei Zhang. Memory-oriented distributed computing at rack scale. In *Proceedings of the ACM Symposium on Cloud Computing, (SoCC '18)*, Carlsbad, CA, USA, October 2018.

[65] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.

[66] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[67] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP*

'15), Monterey, CA, USA, October 2015.

[68] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, March 2017.

[69] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, USA, February 2019.

[70] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, February 2020.

[71] YCSB-C, 2015. https://github.com/basicthinker/YCSB-C.

[72] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment*, 10(6):685–696, 2017.

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