HyperLoop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems

Daehyeok Kim[1*], Amirsaman Memaripour[2*], Anirudh Badam[3], Yibo Zhu[3], Hongqiang Harry Liu[3†], Jitu Padhye[3], Shachar Raindel[3], Steven Swanson[2], Vyas Sekar[1], Srinivasan Seshan[1]

[1]Carnegie Mellon University, [2]UC San Diego, [3]Microsoft

ABSTRACT

Storage systems in data centers are an important component of large-scale online services. They typically perform replicated transactional operations for high data availability and integrity. Today, however, such operations suffer from high tail latency even with recent kernel bypass and storage optimizations, and thus affect the predictability of end-to-end performance of these services. We observe that the root cause of the problem is the involvement of the CPU, a precious commodity in multi-tenant settings, in the critical path of replicated transactions. In this paper, we present HyperLoop, a new framework that removes CPU from the critical path of replicated transactions in storage systems by offloading them to commodity RDMA NICs, with non-volatile memory as the storage medium. To achieve this, we develop new and general NIC offloading primitives that can perform memory operations on all nodes in a replication group while guaranteeing ACID properties without CPU involvement. We demonstrate that popular storage applications can be easily optimized using our primitives. Our evaluation results with microbenchmarks and application benchmarks show that HyperLoop can reduce 99^{th} percentile latency $\approx 800\times$ with close to 0% CPU consumption on replicas.

CCS CONCEPTS

• **Networks → Data center networks**; • **Information systems → Remote replication**; • **Computer systems organization → Cloud computing**;

KEYWORDS

Distributed storage systems; Replicated transactions; RDMA; NIC-offloading

ACM Reference Format:

Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, Srinivasan Seshan. 2018. HyperLoop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *SIGCOMM '18: ACM SIGCOMM 2018 Conference, August 20–25, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3230543.3230572

1 INTRODUCTION

Distributed storage systems are an important building block for modern online services. To guarantee data availability and integrity, these systems keep multiple replicas of each data object on different servers [3, 4, 8, 9, 17, 18] and rely on *replicated transactional operations* to ensure that updates are consistently and atomically performed on all replicas.

Such replicated transactions can incur large and unpredictable latencies, and thus impact the overall performance of storage-intensive applications [52, 57, 58, 75, 86, 92]. Recognizing this problem, both networking and storage communities have proposed a number of solutions to reduce average and tail latencies of such systems.

Networking proposals include kernel bypass techniques, such as RDMA (Remote Direct Memory Access) [64], and userspace networking technologies [26, 90]. Similarly, there have been efforts to integrate non-volatile main memory (NVM) [6, 11], and userspace solid state disks (SSDs) [7, 29, 98] to bypass the OS storage stack to reduce latency.

While optimizations such as kernel bypass do improve the performance for standalone storage services and appliance-like systems where there is only a single service running in

*The first two authors contributed equally to this work.
†The author is now in Alibaba Group.

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230572

the entire cluster [60, 100, 101], they are unable to provide low and predictable latency for multi-tenant storage systems.

The problem is two fold. First, many of the proposed techniques for reducing average and tail latencies rely on using CPU for I/O polling [7, 26]. In a multi-tenant cloud setting, however, providers cannot afford to burn cores in this manner to be economically viable. Second, even without polling, the CPU is involved in too many steps in a replicated storage transaction. Take a `write` operation as an example: i) It needs CPU for logging, processing of log records, and truncation of the log to ensure that all the modifications listed in a transaction happen ***a**tomically*; ii) CPU also runs a ***c**onsistency* protocol to ensure all the replicas reach identical states before sending an ACK to the client; iii) During transactions, CPU must be involved to lock all replicas for the ***i**solation* between different transactions for correctness; iv) Finally, CPU ensures that the data from the network stack reaches a ***d**urable* storage medium before sending an ACK.

To guarantee these ACID properties, the whole transaction has to stop and wait for a CPU to finish its tasks at each of the four steps. Unfortunately, in multi-tenant storage systems, which co-locate 100s of database instances on a single server [67, 92] to improve utilization, the CPU is likely to incur frequent context switches and other scheduling issues.

Thus, we explore an alternative approach for predictable replicated transaction performance in a multi-tenant environment by *completely removing* the CPU from the critical path. In this paper, we present HyperLoop, a design that achieves this goal. Tasks that previously needed to run on CPU are entirely offloaded to commodity RDMA NICs, with Non-volatile Memory (NVM) as the storage medium, without the need for CPU polling. Thus, HyperLoop achieves predictable performance (up to 800× reduction of 99^{th} percentile latency in microbenchmarks!) with nearly 0% CPU usage. Our insight is driven by the observation that replicated transactional operations are essentially a set of memory operations and thus are viable candidates for offloading to RDMA NICs, which can directly access or modify contents in NVM.

In designing HyperLoop, we introduce new and general *group-based* NIC offloading primitives for NVM access in contrast to conventional RDMA operations that only offload point-to-point communications via volatile memory. HyperLoop has a necessary mechanism for accelerating replicated transactions, which helps perform logically identical and semantically powerful memory operations on a group of servers' durable data without remote CPUs' involvement.

These group-based primitives are sufficient to offload operations that are conventionally performed by CPUs in state-of-the-art NVM and RDMA based replication systems. Such operations include consistent and atomic processing and truncating of log updates across all replicas, acquiring the same logical lock across all replicas, and durably flushing the volatile data across all replicas. HyperLoop can help offload these to the NIC.[1]

Realizing these primitives, however, is not straightforward with existing systems. Our design entails two key technical innovations to this end. First, we repurpose a less-studied yet widely supported RDMA operation that lets a NIC wait for certain events before executing RDMA operations in a special queue. This enables us to pre-post RDMA operations which are triggered only when the transactional requirements are met. Second, we develop a remote RDMA operation posting scheme that allows a NIC to enqueue RDMA operations on other NICs in the network. This is realized by modifying the NIC driver and registering the driver metadata region itself to be RDMA-accessible (with safety checks) from other NICs. By combining these two techniques, an RDMA-capable NIC can precisely program a group of other NICs to perform replicated transactions.

Our approach is quite general as it only uses commodity RDMA NICs and as such applications can adopt our primitives with ease. For example, we modified RocksDB (an open source alternative to Google LevelDB) and MongoDB (an open source alternative to Azure CosmosDB and Amazon DynamoDB) to use HyperLoop with under 1000 lines of code. We evaluate the performance of these systems using microbenchmarks as well as the Yahoo Cloud Storage Benchmark (YCSB) workload. The results show that running MongoDB with HyperLoop decreases average latency of insert/update operations by 79% and reduces the gap between average and 99^{th} percentile by 81%, while CPU usage on backup nodes goes down from nearly 100% to almost 0%. Further, microbenchmarks show that HyperLoop-based group memory operations can be performed more than 50× and 800× faster than conventional RDMA-based operations in average and tail cases, respectively.

2 BACKGROUND & MOTIVATION

In this section, we briefly define the key operations of replicated storage systems. We also present benchmarks that highlight the high tail latency incurred by these systems in a multi-tenant setting, even with state-of-the-art optimizations such as kernel bypass.

2.1 Storage Systems in Data Centers

Replicated storage systems maintain multiple copies of data to deal with failures. For instance, large-scale block stores [20, 22, 36, 53, 63], key-value stores [23, 43], and databases [19, 21, 24, 41] replicate data to increase availability and avoid data loss in the presence of failures. Such systems use comprehensive protocols [30, 33, 34] to ensure that every data update

[1] The idea of HyperLoop, *i.e.,* offloading replicated transactions to NICs in addition to point-to-point read/write operations, can be extended to other hardware combinations as long as the NIC can directly access the storage medium, such as FPGA based-Ethernet NICs that can access SSDs [55].

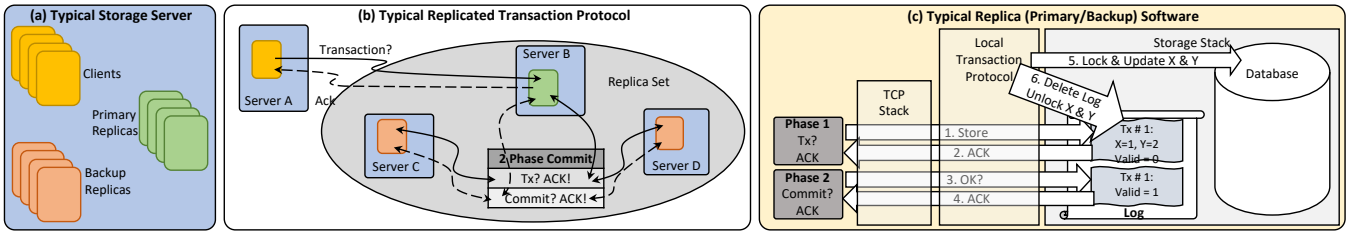


Figure 1: Servers in data center storage systems typically co-locate 100s of replicas of many tenants to improve utilization. This increases latency due to frequent context switches when replica processes, network and storage stacks continually vie for the CPU. Unfortunately, it is not economically viable for 100s of replica processes to be pinned on dedicated cores and poll.

is applied to enough (to sustain availability and durability) copies before acknowledging the changes to the client. At the heart of these protocols are mechanisms to make identical changes to multiple replicas before deeming the change durable and making it available for readers.

Sub-operations for transactions: Changes to storage contents are typically structured as an atomic transaction, consisting of a set of reads and writes. For instance, a transaction might modify objects X and Y as shown in Figure 1(c). The entire set of changes made by each transaction must be atomic, that is X and Y should both change to the values 1 and 2 simultaneously. Storage replicas perform a number of sub-operations per client change and a slowdown in any of these sub-operations can slow down the entire transaction.

Storage systems use *logging* (undo/redo/write-ahead) to achieve atomicity. The new values are first written to a log and later the objects are modified one by one. If the modifications are paused for any reason, then simply re-applying them from the log will ensure atomicity. Further, while processing the log, the replica needs to block other transactions from the objects involved. This is typically implemented using *locks*. For instance, in the above example, the storage system would lock objects X and Y while processing the log.

The storage system must further ensure that all or a sufficient number of replicas execute a transaction before it is considered committed. Typically, a *consensus* protocol called two-phase commit [30, 33, 34, 44] over a primary-backup setting is used to replicate transactions since it enables strong consistency and simplifies application development. As Figure 1(b) shows, the replicas first respond whether they are ready to commit the transaction, then actually commit it after all or enough replicas respond that they are ready.

Chain replication: Chain replication is a widely used primary-backup replication known for its simplicity [1, 10, 35, 46–48, 53, 62, 63, 81, 85, 89, 93–95]. In chain replication, the replicas are laid out in a linear chain. Writes begin at the head of the chain and propagate down the chain in the first phase. The head of the chain begins executing a transaction and readies the transaction to commit by creating a local log entry and grabbing the necessary locks, and only then forwards the transaction to the next replica in the chain.

Each replica prepares the transaction for commit and forwards it down the chain similarly. When the tail of the chain receives the request the second phase starts. The tail knows that everyone upstream is ready to commit. It then sends an ACK that propagates back to the head. Every replica gets the ACK, knows everyone is ready, and commits the transaction. Finally, the head gets the ACK and sends the transaction ACK to the application. In this way, chain replication provides high-throughput, high-availability, and linearizability in a simple manner.

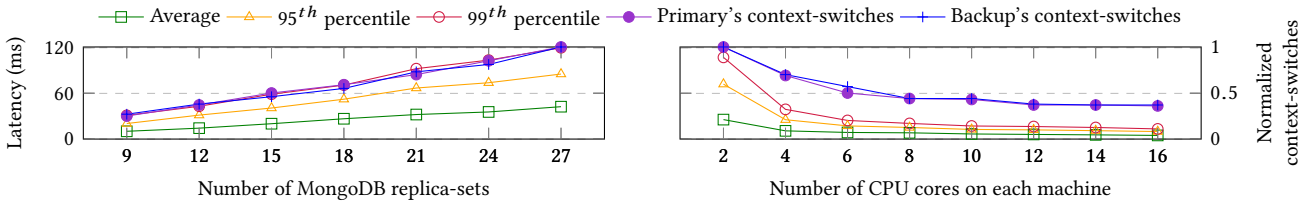
Given the wide popularity of chain replication, our immediate goal in this work is to develop NIC offload primitives for this mode of replication. However, we note that our primitives are generic enough to be used by other replication protocols, non-ACID systems, and a variety of consistency models (see §7).

2.2 Multi-tenancy Causes High Latency

Busy CPU causes high latency: Busy CPU is a major reason why replicas in such protocols may not be responsive – chain or otherwise. A replica's thread has to be scheduled on a CPU for it to receive the *log* via the network stack, and subsequently store the log via the storage stack. The thread needs to participate in the two-phase commit protocol and in the chain (or other) replication scheme. Finally, the thread must process the log and update the actual data objects. Locks are needed for many of these steps to ensure correctness and consistency.

In multi-tenant systems, CPUs are shared across multiple tenants each containing one or more processes. This can lead to heavy CPU load, and unpredictable scheduling latency. The delay for being scheduled to run on CPU causes inflated latencies for writes that are waiting for ACKs or locks.

The problem is further exacerbated by data partitioning. To increase server resource utilization, large-scale storage systems divide the data into smaller partitions such that each server stores partitions of multiple tenants. For instance, an online database partition ranges between 5–50GB of space while typical servers have 4–8 TB of storage space [66]. Thus, each server hosts 100s of tenants translating to 100s of replica processes since each tenant should be isolated in at least one process. Such a large number of processes easily saturates CPU, causing high latency.



(a) Varying number of replica-sets **(b) Varying number of CPU cores per machine**

Figure 2: Analyzing the impact of CPU on MongoDB's latency distribution using YCSB. Normalized context-switches is the number of context-switches for each configuration divided by the maximum for all configurations in each chart.

To demonstrate this problem, we measured the latency and CPU consumption of MongoDB [32] when running a variable number of instances. We ran Yahoo Cloud Storage Benchmark (YCSB) [56] against MongoDB. We used 6 machines (3 MongoDB servers and 3 YCSB clients) each with two 8-core Xeon E5-2650v2 CPUs, 64 GB of memory, and a Mellanox ConnectX-3 56 Gbps NIC. To avoid interference effects from storage medium, we use DRAM as the storage medium. Note that this setup is also representative for modern storage systems that use NVM as the storage medium [60, 100, 101].

For the most part, CPU hits 100% utilization since all MongoDB processes are fully active. The delay caused by CPU can be observed by CPU context-switches. Figure 2(a) shows how they impact the end-to-end MongoDB performance as we increase the number of partitions per server. Each partition is served by a replica-set, which consists of one primary replica process and two backup replica processes running on the three different MongoDB servers. As the number of partitions grow, there are more processes on each server, thus more CPU context switches and higher latencies.

Next, we verify that this inflated latency is mainly due to the heavy load and context switches on CPU, not network congestion or contention on memory bus. We stick to 18 replica-sets but change the number of available cores on each machine by disabling cores. Figure 2(b) shows that, even though the network throughput remains the same, the transaction latency and number of context switches decreases with more cores.

Existing solutions offer limited help: There are multiple proposals that partly tackle this problem. For example, user-level TCP such as mTCP [69] and RDMA-based designs [60, 71] offload (part of) the network stack to NICs, thus reducing the CPU overhead and context switching. However, the storage replication and transaction logic, which are heavier operations relative to the network stack, remain on CPUs.

Furthermore, these solutions rely on CPU core-pinning to avoid being swapped out due to other busy processes to achieve predictable latency. Some systems even employ busy polling [60, 71], which wastes CPU further.

Unfortunately, core-pinning and busy polling is not a viable option for multi-tenant systems. As explained above, the number of partitions is typically an order or two magnitudes higher than the number of cores in data centers. These partitions have to be isolated in different processes. Hence it is not feasible to pin each partition to a dedicated core, nor is busy polling per process feasible. Furthermore, it is not feasible to get good latency even when only a few dedicated threads at the user or kernel space poll on behalf of all the tenants since the tenant processes still need to get a hold of the CPU and wait for events from the polling threads.

In this paper, we aim to fundamentally address these issues by offloading all CPU tasks, including the network stack and storage replication and transaction logic to commodity NICs.

3 OVERVIEW

In this section, we present the design goals and architecture of new primitives that can allow NICs to execute operations needed for replicated NVM transactions at line rate without CPU involvement in the critical path.

3.1 Design Goals

Motivated by our observations about storage system requirements (§2.1) and problems (§2.2), we design HyperLoop with the following goals in mind.

No replica CPU involvement on the critical path: Since the root cause of the latency problem is that replica server CPUs are on the critical path, our solution is to offload those software components in storage systems to commodity NICs. The replica server CPUs should only spend very few cycles that initialize the HyperLoop groups, and stay away from the critical path after that.

This means that NICs by themselves should perform the tasks that previously ran on CPUs, *e.g.,* to modify data in NVM. RDMA NICs (RNICs) provide a promising way to do this while avoiding the CPU. However, it is still challenging to use RNICs to perform some of these tasks such as log processing *without* CPU. We will discuss this in §4.

Provide ACID operations for replicated transactional storage systems: HyperLoop's interfaces should be general enough that most replicated transactional storage systems can easily adopt HyperLoop without much modification to applications. This means that HyperLoop aims to provide a

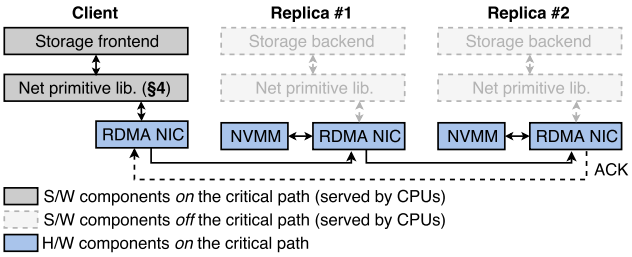


Figure 3: Storage system architecture with HyperLoop.

new set of RDMA primitives instead of end-to-end RDMA- and NVM-based storage system [59, 71].

These primitives should offload the operations most commonly required to ensure ACID properties. For example, many storage systems, such as MongoDB [32], perform the following steps for a transaction 1) replicate operation logs to all replicas and make sure every replica is ready to commit, 2) acquire a lock on every replica, 3) execute the transactions in operation logs, 4) flush all caches (if applicable) to make the transactions durable, and 5) release the lock.

To support this representative process, HyperLoop provides new primitives that handle the steps separately. In §4.2, we will explain the design of these primitives.

End-host only implementation based on commodity hardware: HyperLoop is designed to be widely adoptable by commodity data centers with little effort. Thus, HyperLoop should not rely on any additional special hardware or switch configurations. In addition, being implemented only on end-hosts, HyperLoop can avoid putting state on switches, therefore is easy to manage and can scale well.

3.2 HyperLoop Architecture

HyperLoop offloads all the tasks on the replicas to their NICs. To achieve this, we must turn those tasks into a form that can be processed by just the NICs, without the help of CPUs. Our insight is that replication, log processing, and global locking are essentially a set of memory operations and network commands, and therefore are viable candidates for offload from CPUs to NICs that can access memory (*e.g.,* RDMA). Since NVM offers the convenience of durable memory, it is a natural fit. However, as we show later, care must be taken when bridging between volatile caches of RNICs and the durable memory region.

Figure 3 illustrates the architecture of HyperLoop. A typical replication group in storage systems consists of multiple replicas with only the first one interfacing with clients. HyperLoop can support any number of replicas depending on the application's requirement. There are two software layers: HyperLoop network primitive library and storage applications (*e.g.,* MongoDB) that adopt the primitive library. Note that HyperLoop itself is not a full end-to-end replicated storage system. Its main goal is to provide key building blocks to build replicated transaction systems or make existing systems more efficient.

The network primitive library implements four group-based primitives, gFLUSH, gWRITE, gCAS, and gMEMCPY. They address the missing memory operations needed by general storage systems as explained in §3.1. Application developers can leverage these primitives to implement or modify their replication and transaction processing modules. As described in §5, we have adapted a few applications to use the HyperLoop library. Only small modifications were needed based on their original code.

HyperLoop works in a chain-based manner. When the storage application running on the client (also known as transaction coordinator) writes (*i.e.,* add or update) data, it updates a write-ahead log and then performs the transaction on the group. This is done by calling corresponding primitive exposed by HyperLoop primitive library. RNICs on replicas receive the operation from the client or previous replica, execute the operation and forward the operation to the next replica in the chain. Note that there is no CPU involved in the critical path of executing this operation on replicas. The NIC on the last replica sends back an ACK to the head of the chain – the client. To keep the design simple HyperLoop group failures are detected and repaired in an application specific manner much like traditional network connections in storage systems (see §5).

4 HYPERLOOP DESIGN

In this section, we describe the design of HyperLoop group-based network primitives. We start by explaining the key techniques that make HyperLoop possible.

4.1 Key Ideas

The challenge is how to control NICs without CPUs: RDMA and NVM can turn storage operations into memory operations that bypass CPU. However, in traditional RDMA programming, *when* and *what* memory operations to be performed is controlled by CPU. This is because the basic RDMA operations by themselves are only designed for point-to-point communication. Therefore, CPUs on *every* end-host must be involved on the critical path.

We elaborate this using a simple example, where a storage system needs to replicate data from the client to two or more replica nodes using chain replication [46, 47, 53, 81, 94]. Even though the client can write data to a memory region of the first (primary) replica without involving the replica's CPU using RDMA WRITE, the first replica still has to forward the data to the second replica.

Thus, on the first replica, a traditional RDMA implementation will let CPU pre-post a *receive request* and keep polling the completion queue (on the critical path). The client must use SEND or WRITE_WITH_IMM operations, which triggers the *receive completion* event on the first replica. Then the replica performs the operations in the request meant for it and posts a *send request* of the operations meant for the rest of the chain to the next replica, right after the receive

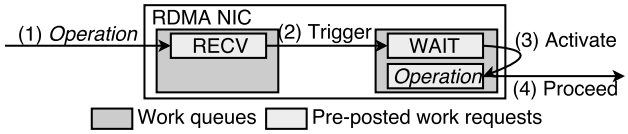


Figure 4: Forwarding an operation to another node in a group with RDMA WAIT.

completion event (on the critical path again). The CPU is needed not only for executing the memory operations in these requests but also for posting it to the next replica.

This means that the NIC on the replica does not know *when* and *what* it should forward to the next node until its CPU gets the receive completion event and generates a corresponding send request. This causes the high tail latency and the high CPU utilization problems shown in §2.2.

The general approach of HyperLoop is to design a set of group-based memory operation primitives that completely remove replica's CPUs from the critical path. This means that NICs on replicas should detect the receive events by themselves, process them and automatically trigger sending them to the next replica. In addition, the forwarded SEND operation must be based on the memory address and length of data received from upstream.

Leveraging WAIT for auto-triggering RDMA operations: We find that commodity RNICs support an operation which enables event-based work request triggering, called RDMA WAIT, between two communication channels (or queue pair (QP)) within a host.[2] In a nutshell, WAIT enables RNICs to wait for a completion of one or more *work requests* (WRs) posted on one WR queue, and trigger other WRs that are pre-posted on another WR queue, without CPU involvement. Although this feature has not been widely studied or used for general applications, we find it promising for determining *when* a remote NIC can proceed.

We leverage this feature to enable *forwarding* operations in HyperLoop. Figure 4 illustrates the data path for forwarding operations. The basic idea is that *every* replica pre-posts RECV WR on the WR queue of the QP connected to the previous node, and also WAIT and a two-sided *operation* (*e.g.,* SEND or WRITE_WITH_IMM) WR to the WR queue of the QP connected to the next node in the group. Upon completion of a RECV WR (Step 1 and 2), the WAIT WR is triggered and it activates *operation* WR, which was posted right next to it (Step 3 and 4). Thus, NICs on replicas forward the memory operations received from the previous node to the next node in the group, without involving the CPU.

Remote work request manipulation for replicating arbitrary data: While WAIT addresses the "when" problem, NICs also need to know *what* to do, whenever WAIT triggers. In common RDMA programs, when posting an RDMA WRITE or SEND work request, one must specify a memory

[2]Different manufacturers have different terms for the method, *e.g.,* "CORE-Direct" [25] by Mellanox.

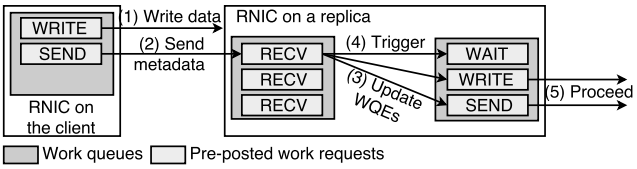


Figure 5: Manipulating pre-posted remote work requests from the client.

descriptor that contains a local source address, size of data to be written, and a remote destination address (in case of WRITE). Since WAIT can only trigger work request posted *in advance*, NICs can only forward a fixed size buffer of data at a pre-defined memory location, which we call *fixed replication*. This is clearly insufficient for general storage systems that require flexibility in memory management.

To address this limitation, we propose a solution called *remote work request manipulation*, which enables the client to manipulate the work requests pre-posted on the replicas' work queues. Our key insight is that we can register replicas' work queues as RDMA-writable memory regions and allow the client to modify memory descriptors (stored in a work queue entry (WQE) data structure) of pre-posted work requests on replicas. Along with the operation forwarding feature described above, the client can perform arbitrary memory operations targeted to RDMA-registered memory on a group of replicas without involving replicas' CPUs.

Figure 5 illustrates the workflow. Since the work requests are at a known memory location on replicas' memory region, NICs on replicas can post RECV requests that point the received metadata to update the memory descriptor in existing work requests. The metadata contains memory descriptors for every replica and is pre-calculated by the client and replicated to all replicas using *fixed replication*. Metadata can also contain additional information depending on different types of memory operation. Note that separate metadata memory regions are allocated for each primitive and each region takes ⟨size of memory descriptor⟩ × ⟨group size⟩ × ⟨number of metadata entries⟩. The sizes of memory descriptor are different by primitives and in our implementation, the maximum size is 32 bytes, which is the case for gCAS.

The client first builds metadata for each group memory operation. Then the client posts a WRITE and a SEND work request, which are used to replicate data to the replicas' memory region and send the metadata, respectively. Since the SEND request is a two-sided operation, it consumes a RECV work request posted by the replica. The RECV request will trigger the WAIT request and updates metadata region, the memory descriptors of WRITE and SEND work request, and activate them (*i.e.,* grant the ownership to the NIC). The activated WRITE and SEND work request will do the same procedure for the next node. While this mechanism uses additional work requests for sending metadata of operations and waiting a completion of a RECV work request, it does

not incur much network overhead since the size of metadata is small (at most 32 bytes × ⟨group size⟩), and a WAIT does not produce any network traffic.

Note that with current RDMA NIC driver implementations, when an application posts a work request, a NIC will immediately get the ownership of that request and because of this, the memory descriptor cannot be modified later. However, in HyperLoop, the client should be able to manipulate the memory descriptor of pre-posted work requests. To enable this, we modify the userspace NIC driver library (*e.g.,* `libmlx4` for Mellanox NICs) to make it not yield ownership to the NIC when the work request is posted. Instead, with our modified driver, the client will grant the ownership of the work request to NICs after it updates the memory descriptor of pre-posted work requests with our primitive.

Integration with other RDMA operations to support ACID: To support the demanding features by storage systems, like durable WRITE, transaction execution and group locking, we further build on the two ideas above.

Specifically, we further leverage the WAIT mechanism to let the last replica to send an ACK to the client as a group operation ACK. We use a 0-byte READ immediately following each WRITE to flush the data in NIC cache into memory or NVM, so that each ACK means the operation finishes and becomes durable. For two-phase transaction execution, after logs are replicated, we let RNICs perform "local RDMA" to move data from the log region to the actual data region. For locking, we leverage the RDMA compare-and-swap (CAS) atomic operation.

In traditional systems, all these tasks can only be done on a single host with the help of CPU. However, in HyperLoop, we turn all these into group-based primitives and removes replicas' CPUs from the critical path. Whenever the client of a replication group performs memory operations (*e.g.,* replication, transaction execution, or locking) in its own memory region, RNICs (not CPUs!) on replicas also perform the same operations against their memory regions. The result is significantly better tail latency and lower CPU consumption than traditional RDMA implementations.

Based on these techniques, we develop four group-based primitives summarized in Table 1. We describe each of them in detail next.

4.2 Detailed Primitives Design

Group write (gWRITE): In a replicated storage, when the client updates the transaction log, it needs to replicate the updated log to all replicas in a replication group. We abstract this process into the group memory write (gWRITE) primitive, which enables remote NICs to write any data to the specified memory location of their host memory region.

gWRITE allows the caller (*e.g.,* client) to write a data to memory regions of a group of remote nodes without involving their CPUs. The primitive takes a group ID, a memory offset of data which will be written to remote nodes' memory regions, and a size of data as input arguments. For the given offset and size, it builds metadata and initializes the command and follows the data path shown in Figure 5. gWRITE is used to implement replicated transaction log management described in Section 5.

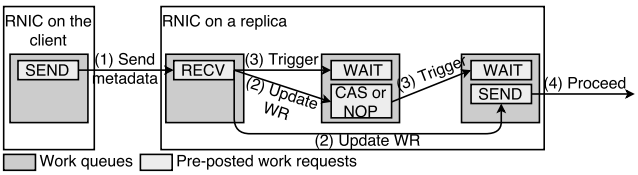


Figure 6: Datapath of gCAS primitive.

Group compare-and-swap (gCAS): To ensure data integrity during concurrent read-write, replicated storage systems often use some locking mechanism, which also involves the replicas' CPUs. We provide the group compare-and-swap (gCAS) primitive to offload such lock management schemes. This enables remote NICs to perform compare-and-swap against a specified memory location on their host memory region and update the value of the location based on the result of comparison. The client can acquire a logical group lock via this primitive without involving the replicas' CPUs.

gCAS extends the original RDMA single host compare-and-swap (CAS) operation to a group of nodes. It takes a group ID, an offset of memory location whose value to be compared, an old and new value, an execute map, and a result map as input arguments. The execute and result map are additional parameters different from the original CAS. It has a field for each node in the group. The client can specify whether each remote node has to execute the CAS operation on the execute bitmap (*i.e.,* marking the corresponding field). This capability of selective execution is necessary especially for a client which needs to undo a gCAS operation when the operation failed to be executed on some remote nodes due to mismatch between the expected and actual value of the target memory location. To undo the operation, the client issues another gCAS operation by marking a set of execute fields corresponding to the nodes on which the previous CAS operation was successfully executed and swapping the old and new values. Each replica updates the result of locally performed CAS to the result map and the client will receive the final result map as an ACK of performing gCAS.

Figure 6 shows how gCAS works. On each replica, HyperLoop creates an additional QP for performing CAS operation locally. When a RECV work request is consumed by the SEND work request initiated by the previous node, it updates the memory descriptor of CAS and SEND work requests.

It is important to note that depending on a value in the execute map, each replica has to determine whether it performs CAS *without involving its CPU*. We found that when granting an ownership of work request to the NIC, we can change the type of pre-posted work request. Using this observation, the

Table 1: Group-based network primitives for transactions supported by HyperLoop

Primitives	Semantics
`gFLUSH(data_addr, dest_addr, size)`	Writing data at *data_addr* to *dest_addr* by flushing a volatile cache on the NIC to a durable medium.
`gWRITE(gid, offset, size)`	Replicating the caller's data located at *offset* to remote nodes' memory region at *offset*.
`gCAS(gid, offset, old_val, new_val, execute_map, result_map)`	On each node, if the corresponding bit in the execute map is set, (1) compare the value of data at *offset* with *old_value*. (2) if they are the same, replace the value with *new_value*. (3) update the corresponding result field with the original value of *offset*.
`gMEMCPY(gid, src_offset, dest_offset, size)`	Copying the data size of *size* from *src_offset* to *dest_offset* for all nodes.

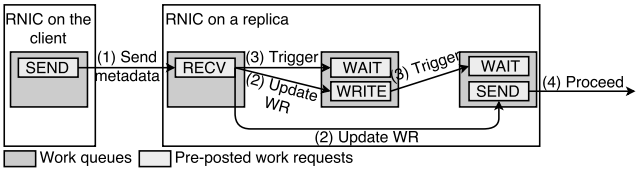


Figure 7: Datapath of gMEMCPY primitive.

caller makes the CAS be NOP operation if the corresponding remote node is chosen not to perform the CAS operation.

If the replica performs CAS, it will update the corresponding field in the result map. The completion of CAS work request triggers the WAIT and SEND requests for the next node. If the replica is the last one in a group, it will forward the result map to the client as an ACK using WRITE_WITH_IMM. gCAS is used to implement a group locking scheme in storage systems, as described in §5.

Group memory copy (gMEMCPY): In many replicated storage systems, replica CPUs execute (commit) a transaction by copying the data corresponding to the transaction from a log region to a persistent data region. We abstract this process into the remote memory copy (gMEMCPY) primitive, which lets remote NICs perform a memory copy on their host memory for given parameters, *i.e.,* data source address, destination address, and size.

When the client executes transactions via this primitive, on all replicas, the NICs will copy the data from the log region to the persistent data region without involving the CPUs. gMEMCPY takes a group ID, a memory offset of source region and destination region, and a size of data being copied. When this primitive is called, the NICs on replicas perform memory copy for given source and destination offset against their host memory without involving their CPUs.

Figure 7 shows how gMEMCPY works. Similar to the datapath of gCAS primitive, HyperLoop creates an additional QP for performing memory copy operation locally on each replica. Upon receiving the command from the previous node, the receive work request updates the memory descriptors of write and send work requests and triggers the wait request. Then the NIC performs local memory copy with the write work request. When the local memory copy is successfully completed, the NIC will trigger the wait and forwards the operation to the next node using SEND. If the next node is the client, it will send an ACK using WRITE_WITH_IMM. gMEMCPY is used to implement remote log processing described in §5.

Group RDMA flush (gFLUSH): To support the durability of transactions, data written with RDMA writes should be durable even in the case of system failure (*e.g.,* power outage). However, the current RDMA protocol implemented on NICs does not guarantee the durability. The destination NIC sends an ACK in response to RDMA WRITE as soon as the data is stored in the NIC's *volatile* cache. This means that the data can be lost on power outage before the data is flushed into NVM. Thus, we need a new RDMA FLUSH primitive that supports the durability at the "NIC-level".

To address this, we design Non-volatile RDMA FLUSH (gFLUSH) primitive which enables durable RDMA WRITE based on the existing RDMA operations. We leverage a feature supported by the NIC firmware that flushes a memory region in the cache when it becomes dirty. In our implementation, when an RDMA FLUSH is issued, the NIC immediately (without for waiting for an ACK) issues a 0-byte RDMA READ to the same address. Then the destination's NIC flushes the cache for the READ and the source's NIC gives the application ACK after the 0-byte READ is acknowledged. This will ensure that each RDMA FLUSH will flush volatile caches to host memory hierarchy which is non-volatile with battery support [73]. Similar to gWRITE, gFLUSH operations are also propagated down the chain for durability across the replicas.

There is an important distinction between gFLUSH and other primitives. gFLUSH can either be issued by itself or as an interleaved operation with gWRITE, gCAS or gMEMCPY. For instance, an interleaved gWRITE and gFLUSH on a replica would first flush the cache and only then forward the operations down the chain. This helps ensure that durable updates are propagated in the order needed by the chain or between the primary and backups in other protocols.

To the best of our knowledge, HyperLoop is the first system to describe the design and implementation of an RDMA

NIC that can guarantee durability with NVM. Existing systems [59, 72] assume this ability but do not provide details of the design and implementation.

Summary: We implement the primitives with 3,417 lines of C library. We also modify 58 lines of code in `libmlx4` and `libibverbs`[3] to implement gFLUSH and remote work request manipulation described in §3. In the next section, we will describe a couple of case study storage systems whose performance can be improved by using HyperLoop.

5 HYPERLOOP CASE STUDIES

HyperLoop helps NVM- and RDMA-based replicated databases to offload transactions to NICs. Based on HyperLoop primitives, different consistency models required by applications can be implemented as we will show in this section and discuss in §7. To demonstrate the benefits of HyperLoop, we modify and optimize two widely used databases to an NVM- and RDMA-based chain replication using state-of-the-art CPU-polled kernel bypass approach. We then offload the work done by the CPUs to the RDMA NICs by using HyperLoop. We choose RocksDB (open source alternative to Google's LevelDB), and MongoDB (open source alternative to Azure DocumentDB and AWS DynamoDB) because these databases have in-memory (RAMCloud [85] like) implementations for volatile DRAM. Such implementations makes it easier to adapt them to NVM.

For such systems, requirements for maintaining a database typically translate to initializing NVM to contain a database and a write-ahead log, appending transactions to the log, processing the log and applying the transactions to the database, and obtaining locks to facilitate concurrent accesses to the database.

Database Initialization: The `Initialize` function uses values specified in a configuration object to set up HyperLoop and create required connections between the replica and it's upstream and downstream replicas in the chain. Additionally, it creates/opens a handle to the NVM for the replica. This NVM area contains space enough to hold a write-ahead log as well as the database as defined in the configuration object.

Log Replication: Each log record is a redo-log and structured as a list of modifications to the database [83]. Each entry in the list contains a 3-tuple of `(data, len, offset)` representing that `data` of length `len` is to be copied at `offset` in the database. Every log record sent by a client is appended to the replicas' write-ahead logs in the chain by calling `Append(log record)`, which is implemented using gWRITE and gFLUSH operations.

A client in our case study is a single multi-threaded process that waits for requests from applications and issues them into the chain concurrently. Multiple clients can be supported in the future using shared receive queues on the first replica in the chain.

Log Processing: The remote log processing interface enables the client to copy data from the write-ahead log of replicas to their database regions without involving replica CPUs. For each log record and starting from the head of the write-ahead log, `ExecuteAndAdvance` processes its entries one by one. To process each entry, it issues a gMEMCPY to copy *len* bytes of data from *data* to *offset* on all replicas followed by a gFLUSH to ensure durability. Once all operations for a log record are processed, `ExecuteAndAdvance` updates the head of the write-ahead log using a gWRITE and gFLUSH.

Locking and Isolation: `ExecuteAndAdvance` allows updating the application data on replicas without involving replica CPUs. However, if clients are allowed to read from all replica nodes, they might observe inconsistent updates since `ExecuteAndAdvance` does not offer isolation. To address this issue, we offer group locking which is a single writer multiple reader locking mechanism. The client calls `wrLock` and `wrUnlock` to acquire and release exclusive write locks so that other transactions working on the same data are blocked.

HyperLoop allows lock-free one-sided reads from exactly one replica (head or tail of the chain) similar to existing systems that use integrity checks to detect incorrect or inconsistent values. However, we also implement read locks for systems that need them and note that HyperLoop design does not hinder systems from performing lock-free one-sided reads when it is possible to detect inconsistent reads and reject their values (*e.g.,* FaRM [60]). However, such systems can have low read throughput since only replica can serve the reads. In HyperLoop, we additionally provide read locks that work concurrently with the write locks to help all replicas simultaneously serve consistent reads for higher read throughput. Unlike write locks, read locks are not group based and only the replica being read from needs to participate in the read lock.

We use these APIs to modify two existing popular transactional storage systems (RocksDB [39] and MongoDB [32]) to show the efficacy of HyperLoop.

5.1 RocksDB

In this case study, we show how all the critical path operations in a write transaction can be offloaded to the NICs of replicas. This helps the system require the CPU only for coarse-grained off-the-critical path operations. Note that even these operations can be offloaded to the NIC, which we show in the next section.

RocksDB is an persistent key-value store library that can be integrated into application logic. It serves all requests using an in-memory data structure and uses a durable write-ahead log to provide persistence. It periodically dumps the

[3] `libibverbs` and `libmlx4` are userspace libraries/drivers that allow userspace processes to use InfiniBand/RDMA Verbs on Mellanox hardware.

in-memory data to persistent storage and truncates the write-ahead log. We replace the persistent storage in RocksDB with NVM. Further, we modify the interface between RocksDB's write-ahead log and NVM to use HyperLoop APIs instead of native NVM APIs. We modify/add only 120 lines of code.

Our version of RocksDB uses Append to replicate log records to replicas' NVM instead of using the native unreplicated append implementation. Replicas need to wake up periodically off the critical path to bring the in-memory snapshot in sync with NVM. Only one replica in the chain updates its in-memory copy of the data structure in the critical path. Therefore, reads from other replicas in our RocksDB implementation are eventually consistent [94]. Thus, HyperLoop helps us convert an unreplicated system into a replicated one with accelerated transactions with only a few modifications. Section 6.2 shows the performance of our replicated RocksDB.

RocksDB Recovery: Our recovery protocol for RocksDB is fairly straightforward. A new member in the chain copies the log and the database from either downstream or an upstream node; writes are paused for a short duration of catch-up phase. Only then does it formally join the chain. While this reduces write availability, we note that the focus of this paper is not to optimize the control path but rather to facilitate developers to build faster data paths. Complementary efforts in chain replication research have outlined various ways to speed up recovery.

Our primitives are low-level enough not to interfere with the recovery protocols that deal with temporary and permanent replica failures. Such a design is crucial for ensuring that the control path of replication protocols remains unchanged while only the data path is accelerated with the new primitives. Therefore, in the absence of failures, HyperLoop accelerates the transactions and as soon as a failure is detected, the recovery protocol takes over to bring back a stable data path as soon as possible. A configurable number of consecutive missing heartbeats is considered a data path failure [45].

5.2 MongoDB

MongoDB is a NoSQL database server (*i.e.,* document store), which offers both in-memory and persistent storage. It also provides replication by copying operation logs to replicas and asynchronously executing them against the replica version of the application data [38]. Here, we use MongoDB's memory storage engine, which accesses persistent data through issuing loads/stores against memory-mapped files [31] – a model conducive for NVM and RDMA. We split the MongoDB code base into a front end and a back end. The front end is integrated with the client while the backend is HyperLoop-based replicas with NVM.

We achieve this with modifying only 866 lines of code. We use Append to replicate MongoDB's write-ahead log (i.e., journal) entries to replicas. Then, we execute transactions on replicas using ExecuteAndAdvance and the replicated write-ahead log data. Additionally, to allow clients to read from replicas, we surround each ExecuteAndAdvance on the primary with wrLock and wrUnlock. Additional replicas only wake up to serve read requests when the chain is overloaded, when they must acquire and release a shared lock using rdLock and rdUnlock. Such an implementation completely offloads both critical and off-the-critical path operations for write transactions to the NIC while providing strong consistency across the replicas. We note that there are techniques (*e.g.,* FaRM [60]) to perform lock free one-sided reads that can be performed in HyperLoop as well if reads are restricted only to one replica. The advantage of HyperLoop is that it reduces the cost of keeping replicas strongly consistent and therefore, reads can be served from more than one replica to meet demand.

MongoDB Recovery: The goal of our recovery protocol here was to bring the chain to a state where vanilla MongoDB recovery can take over. To achieve this, whenever the membership changes (heartbeat is lost), the entire chain flushes the log of all valid entries, rejects invalid entries, block reads temporarily and hand-off control to MongoDB recovery protocol which helps the new empty replica catch up and join a newly established HyperLoop data path. Rather than focusing on control path optimizations that are out of the scope of this paper, we focus on correctness and feasibility of adopting the accelerated data path of HyperLoop.

6 EVALUATION

In this section, we evaluate HyperLoop by answering the following questions:

1. How do HyperLoop primitives help improve the performance of group memory access (§6.1)?
2. How does HyperLoop help improve the performance of real-world storage systems (§6.2)?

Testbed setup: Our experimental testbed consists of 20 machines each equipped with two 8-core Xeon E5-2650v2 CPUs, 64 GB DRAM, and a Mellanox ConnectX-3 56 Gbps RDMA-capable NIC. The operating system running on the machines is Ubuntu Linux 14.04 with kernel version 3.13.0-137-generic. In our evaluation, we assume storage nodes are equipped with battery-backed DRAM [60] which is a form of NVM available today. We emulate battery-backed DRAM by mounting a tmpfs file system to DRAM and run applications on it. We expect to observe similar benefits, *i.e.,* no CPU overhead on replicas while exhausting the bandwidth of the network and/or NVM, when replacing battery-backed DRAM with emerging technologies such as 3D XPoint [6].

Baseline RDMA implementation: To evaluate the performance benefit of offloading the primitives to the NIC, we develop a naïve version of primitives using RDMA operations as a comparison point for HyperLoop. It can perform the

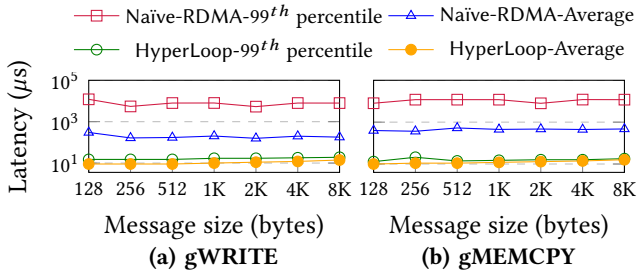


Figure 8: Latency of gWRITE and gMEMCPY compared to Naïve-RDMA.

same set of operations (*i.e.,* gWRITE, gMEMCPY, gCAS) as HyperLoop and provides the same APIs, but involves backup CPUs to handle receiving, parsing, and forwarding RDMA messages in contrast to HyperLoop. In the rest of this section, we call this implementation Naïve-RDMA.

6.1 Microbenchmark

We evaluate the performance of HyperLoop primitives in terms of latency, throughput and CPU consumption compared to Naïve-RDMA. We also evaluate the scalability of HyperLoop under different replication group sizes.

Benchmark tools: We build custom latency and throughput measurement tools for the microbenchmark. Our latency benchmark generates 10,000 operations for each primitive with customized message sizes and measures the completion time of each operation. The throughput benchmark for gWRITE writes 1GB of data in total with customized message sizes to backup nodes and we measure the total transmission time to calculate the throughput. Also, we emulate background workloads by introducing CPU-intensive tasks using stress-ng [42] in the testbed for HyperLoop and observe the impacts on latency and throughput. The Naïve-RDMA however uses a pinned core for best case performance.

HyperLoop significantly reduces both average and tail latency compared to Naïve-RDMA: Figure 8 shows the average and tail latency of gWRITE and gMEMCPY primitives with different message sizes, fixing replication group size (number of member nodes) to 3. For both of gWRITE and gMEMCPY, Naïve-RDMA shows much higher 99th percentile latency than HyperLoop. Particularly, for gWRITE, we can see that with HyperLoop, 99th percentile latency can be reduced by up to 801.8× with HyperLoop. gMEMCPY shows a similar result; HyperLoop reduces the 99th percentile latency by up to 848× compared to Naïve-RDMA. Table 2 shows the latency statistics of gCAS, in which HyperLoop shortens the average latency by 53.9× and 95th and 99th latencies by 302.2× and 849×, respectively.

HyperLoop achieves high throughput with almost zero CPU usage: Figure 9 presents the operation throughput and CPU utilization under different message sizes, fixing replication group size to 3. While HyperLoop provides a similar throughput compared to Naïve-RDMA, almost no CPUs are consumed in the critical path of operations in contrast to Naïve-RDMA which utilizes a whole CPU core. This is because CPUs are involved in polling NICs, receiving, parsing and forwarding messages in Naïve-RDMA while HyperLoop offloads all these processes to the NICs.

Table 2: Latency of gCAS compared to Naïve-RDMA.

	Average	95th percentile	99th percentile
Naïve-RDMA	539us	3928us	11886us
HyperLoop	10us	13us	14us

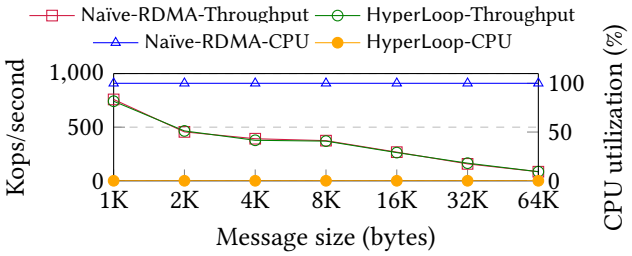


Figure 9: Throughput and critical path CPU consumption of gWRITE compared to Naïve-RDMA.

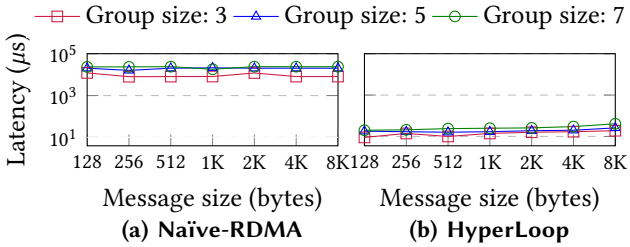


Figure 10: 99th percentile latency of gWRITE with different group sizes compared to Naïve-RDMA.

HyperLoop is scalable with increasing group size: We evaluate the latency with different replication group sizes. The latency is measured from a client that sends a ping into the chain. Figure 10 illustrates the 99th percentile latency of gWRITE when group size is 3, 5, and 7. As shown previously, Naïve-RDMA incurs much higher tail latency in all group sizes. With HyperLoop, there is no significant performance degradation as the group size increases (Fig. 10(a)), while with Naïve-RDMA, 99th percentile latency increases by up to 2.97× (Fig. 10(b)). We also observed that HyperLoop shows a smaller variance of average latency between the group sizes compared to Naïve-RDMA. This means that by offloading operations to NICs, HyperLoop can significantly reduce the latency and make it predictable regardless of the group size.

6.2 Performance in Real Applications

We measure the latency and CPU utilization of two real applications (RocksDB and MongoDB), with YCSB [56], an industry standard storage benchmark. In these experiments, we use the 3 physical machines and set the replication group size to 3. Table 3 shows properties of each workload within YCSB. For MongoDB and RocksDB experiments, we initialize

Table 3: The percentage of read, update, insert, modify (read and update) and scan operations in each YCSB workload.

Workload	Read	Update	Insert	Modify	Scan
YCSB-A	50	50	-	-	-
YCSB-B	95	5	-	-	-
YCSB-D	95	-	5	-	-
YCSB-E	-	-	5	-	95
YCSB-F	50	-	-	50	-

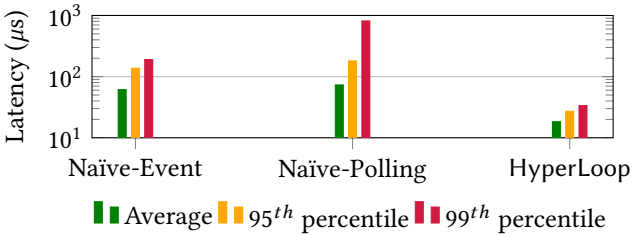


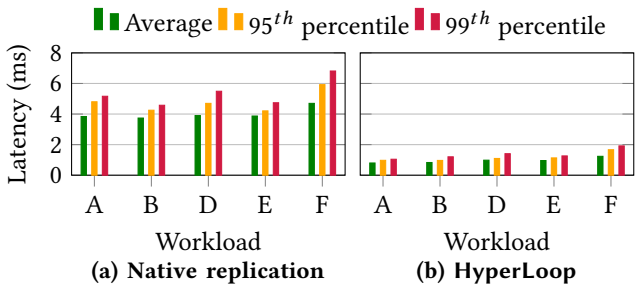
Figure 11: Latency distribution of replicated RocksDB using Naïve-RDMA (event-based and polling) and HyperLoop.

a data-store with 100 K and 1 M key-value pairs (32-byte keys and 1024-byte values) prior to running 100 K and 16 M operations against the data-store, respectively.

RocksDB: We compare three versions of replicated RocksDB (three replicas): (i) *Naïve-Event* version which uses event-based Naïve-RDMA for replication; (ii) *Naïve-Polling* which uses polling-based Naïve-RDMA and uses CPU polling in backup nodes to reduce latency; (iii) HyperLoop-based version. To perform experiments in settings representative of multi-tenant data centers, we co-located replicated RocksDB processes with multiple instances of I/O intensive background tasks (our own MongoDB instances described next) on the same CPU socket, each serving a YCSB client running on the remote socket of the same server. The number of application threads on each socket is 10× the number of its CPU cores.

We use traces from YCSB (workload A) to compare the three versions in average and tail latencies of update operations. As shown in Figure 11, HyperLoop offers significantly lower tail latency in contrast to *Naïve-Event* (5.7× lower) and *Naïve-Polling* (24.2× lower). Furthermore, HyperLoop reduces the CPU overhead on backup nodes since it does not involve backup CPUs in the critical path of replication. In contrast, *Naïve-Polling* version burns two cores on backup nodes to poll incoming messages. Interestingly, however, *Naïve-Event* has lower average and tail latency compared to *Naïve-Polling* as multiple tenants polling simultaneously increases the contention enough that context switches start increasing the average and tail latencies.

MongoDB: We compare two versions of MongoDB with polling-based and HyperLoop-enabled replication. We run YCSB workloads against a chain with three replicas. We co-locate multiple instances (10:1 processes to cores ratio) to emulate multi-tenant behavior from multi-tenant data centers and then measure the latency. Figure 12 shows the



(a) Native replication **(b) HyperLoop**

Figure 12: Latency distribution of MongoDB with native and HyperLoop-enabled replication.

performance comparison of the two versions. HyperLoop reduces replication latency by up to 79% by reducing the reliance on overburdened CPUs, and the remainder of the latency is mostly due to the high overhead inherent to MongoDB's software stack in the client that requires query parsing and other checks and translations before a query is executed. For MongoDB, HyperLoop also reduces the gap between average and 99^{th} percentile latency by up to 81%. Furthermore, HyperLoop completely obviates involvement of backup CPUs in the process of replication by offloading transaction replication and execution to the RDMA NIC. In contrast, MongoDB's native replicas saturate the cores while running YCSB workloads.

7 DISCUSSION

Supporting other storage systems: In designing our primitives and APIs, we were motivated by state-of-the-art transactional programming frameworks for NVM [12, 13], and the messages in the data path of replicated transactional systems [60]. Such a design allows us not only to move applications modified with existing APIs [14–16] easily to HyperLoop but also give higher confidence to developers to use these APIs for faster adoption of NVM-aware systems.

HyperLoop is also useful for non-ACID systems and/or weaker consistency systems. In designing low-level primitives for fully-ACID and strongly consistent systems, we enable other weaker models as well. For instance, by ignoring the durability primitive, systems can get acceleration for RAMCloud [85] like semantics. By not using the log processing primitive inside the transaction critical path, systems can get eventually consistent reads at higher throughput. Further, by not using the log processing and durability in the critical path, systems can get replicated Memcache [27] or Redis [37] like semantics.

Supporting other replication protocols: We explored chain replication for its simplicity, popularity and inherent load-balancing of network resources. It is well known that the scalability of the RDMA NICs decreases with the number of active write-QPs [71]. Chain replication has a good load balancing property where there is at most one active write-QP per active partition as opposed to several per partition such as in fan-out protocols.

While HyperLoop is optimized for chain replication protocol, HyperLoop's primitives can be used for any general replication protocol. For example, if a storage application has to rely on a fan-out replication (a single primary coordinates with multiple backups) such as in FaRM [60], HyperLoop can be used to help the client offload the coordination between the primary and backups from the primary's CPU to the primary's NIC.

For instance, in FaRM, the primaries poll for lock requests which it forwards to the backups that are also polling for meeting those requests. Similarly, primaries and backups poll for log record processing, making them durable, and log truncation. Techniques described in §4.1 can be used so that the client can offload these operations to the primary's NIC and manage the locks and logs in backups via the primary's NIC without the need for polling in the primary and the backups, and ensuring the required ordering of operations between between them.

Security analysis of HyperLoop: In designing HyperLoop, we assume that servers running storage front and backends are protected from attackers who can compromise the servers. We did not consider a comprehensive threat model since attackers can easily control the entire storage cluster once gaining access to a single server and even without exploiting a replication channel through HyperLoop. On the other hand, in HyperLoop, applications have access to RDMA queues on remote NICs. To ensure security and privacy, we use the same set of best practices when managing local queues to manage remote queues. The queues are registered to have access to only a restricted set of registered memory regions that belong to the same tenant, each remotely accessible queue is separately registered with a token known only to the tenant, and finally, managed languages are used to encapsulate the QPs from the application code with stricter safety checking (*e.g.,* type check) in the client. In the future, as RDMA virtualization improves, we wish to leverage even more hardware virtualization techniques to secure the remote queues.

8 RELATED WORK

Optimized storage and networking stacks: As storage and networking hardware technology have evolved, traditional OS storage and networking stacks became a performance bottleneck in applications, especially in distributed storage systems as shown in previous works [54, 68, 88]. MegaPipe [65], Affinity-Accept [87], TCP port reuse [2] and Fastsocket [77] optimize the Linux kernel TCP stack to support better scalability for multi-core systems and high-speed NICs. Likewise file system and other storage optimizations [51, 66, 76, 99] propose making existing OSes friendly to SSDs. However, even with the optimizations, there are still remaining overheads including kernel-user space context switching, data copying, bounds and access control checks can cause high and unpredictable latency in storage systems.

New storage and network hardware: To avoid the OS overheads, storage and networking techniques for kernel bypass have been proposed [7, 26, 80, 90] to poll for I/O completions from NIC or storage medium (*e.g.,* SSD). Additionally, based on those techniques, user-space networking stacks [28, 40, 49, 69, 79] and storage stacks [29, 54, 61, 84, 97, 98] have been developed. While these techniques help reduce the latency of standalone storage services and appliances [60, 71, 72, 78, 82, 91, 100], they do not work effectively in multi-tenant environments where polling is expensive and CPU still needs to coordinate between network and storage stacks. In contrast, since HyperLoop does not rely on polling and offloads the datapath of replicated NVM transactions to RDMA NICs, it does not require CPU usage.

Existing remote SSD access systems offer only point to point read/writes and often use CPU based polling for I/O completions [55, 74, 84]. In contrast, HyperLoop provides group-based primitives for replicated transactions with predictable performance without polling.

Multicast support in data centers: Reliable multicast [50] is a seemingly effective network primitive for replication-based distributed systems. Traditionally, since IP multicast is unreliable [5, 96], it has not been widely adopted. Recent developments in NIC technologies promote a revisiting of reliable multicast at the network level. Derecho [70] proposes an RDMA-based multicast protocol, which involves CPUs in the critical path of operations and relies on polling. Thus, it possibly incurs high tail latency especially in multi-tenant settings.

9 CONCLUSIONS

Predictable low latency for both average and tail cases is critically lacking in modern replicated storage systems. Based on the observation that CPU involvement is the root cause of this behavior in multi-tenant deployments, we designed HyperLoop, a new framework that completely eliminates CPUs from the critical path of replicated transactions in multi-tenant storage systems. HyperLoop entirely offloads replicated transactions to commodity RDMA NICs, with NVM as a storage medium. We realize this by designing new group-based networking primitives that support ACID transactions and demonstrate that existing storage systems can be easily extended and optimized using HyperLoop. Our evaluation with two popular storage systems shows that HyperLoop can produce substantial reductions in both average and tail latency and CPU consumption on replicas. Looking forward, even though our specific focus in this paper was on storage systems, we believe that the design and insights underlying HyperLoop can be more broadly applicable to other data center workloads.

ACKNOWLEDGMENTS

We would like to thank the anonymous SIGCOMM reviewers and our shepherd, Marco Canini for their helpful comments. This work was supported in part by NSF awards CNS-1565343 and CNS-1513764.

REFERENCES

[1] 2008. MongoDB Managed Chain Replication. https://docs.mongodb.com/manual/tutorial/manage-chained-replication/. Accessed on 2018-01-25.
[2] 2013. The SO REUSEPORT socket option. https://lwn.net/Articles/542629/. Accessed on 2018-01-25.
[3] 2013. Transactions for AWS Dynamo DB. https://aws.amazon.com/blogs/aws/dynamodb-transaction-library/. Accessed on 2018-01-25.
[4] 2014. Replication in AWS Dynamo DB. https://aws.amazon.com/dynamodb/faqs/#scale_anchor. Accessed on 2018-01-25.
[5] 2015. InfiniBand Architecture Volume 1, Release 1.3. http://www.infinibandta.org/content/pages.php?pg=technology_public_specification. Accessed on 2018-01-25.
[6] 2015. Intel/Micron 3D-Xpoint Non-Volatile Main Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html. Accessed on 2018-01-25.
[7] 2016. Intel Storage Performance Development Kit. https://software.intel.com/en-us/articles/introduction-to-the-storage-performance-development-kit-spdk. Accessed on 2018-01-25.
[8] 2016. Replication in Google Cloud Datastore. https://cloud.google.com/datastore/docs/concepts/overview. Accessed on 2018-01-25.
[9] 2016. Transactions in Google Cloud Datastore. https://cloud.google.com/appengine/docs/standard/java/datastore/transactions. Accessed on 2018-01-25.
[10] 2017. Chain Replication in SAP HANA. https://www.sap.com/documents/2013/10/26c02b58-5a7c-0010-82c7-eda71af511fa.html. Accessed on 2018-01-25.
[11] 2017. HP Enterprise Non-Volatile DRAM. https://www.hpe.com/us/en/servers/persistent-memory.html. Accessed on 2018-01-25.
[12] 2017. Intel Persistent Memory Development Kit. http://pmem.io/pmdk/. Accessed on 2018-01-25.
[13] 2017. Oracle NVM Programming APIs. https://github.com/oracle/nvm-direct. Accessed on 2018-01-25.
[14] 2017. Persistent Memory KV Java Embedding. https://github.com/pmem/pmemkv-java. Accessed on 2018-01-25.
[15] 2017. Persistent Memory Optimizations for MySQL. http://pmem.io/2015/06/02/obj-mysql.html. Accessed on 2018-01-25.
[16] 2017. Persistent Memory Optimizations for Redis. https://libraries.io/github/pmem/redis. Accessed on 2018-01-25.
[17] 2017. Replication in Azure Cosmod DB. https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_0/. Accessed on 2018-01-25.
[18] 2017. Transactions in Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/programming#database-program-transactions. Accessed on 2018-01-25.
[19] 2018. Amazon Relational Database Service (RDS) - AWS. https://aws.amazon.com/rds/. Accessed on 2018-01-25.
[20] 2018. Amazon Simple Storage Service (S3) - Cloud Storage - AWS. https://aws.amazon.com/s3/. Accessed on 2018-01-25.
[21] 2018. Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db. Accessed on 2018-01-25.
[22] 2018. Azure Storage - Secure cloud storage | Microsoft Azure. https://azure.microsoft.com/en-us/services/storage/. Accessed on 2018-01-25.
[23] 2018. Bigtable - Scalable NoSQL Database Service | Google Cloud Platform. https://cloud.google.com/bigtable/. Accessed on 2018-01-25.
[24] 2018. Cloud SQL - Google Cloud Platform. https://cloud.google.com/sql/. Accessed on 2018-01-25.
[25] 2018. CORE-Direct The Most Advanced Technology for MPI/SHMEM Collectives Offloads. http://www.mellanox.com/related-docs/whitepapers/TB_CORE-Direct.pdf. Accessed on 2018-01-25.
[26] 2018. Intel Data Plane Development Kit. http://dpdk.org/. Accessed on 2018-01-25.
[27] 2018. Memcache: A Distributed Memory Object Caching System. https://memcached.org/. Accessed on 2018-01-2015.
[28] 2018. Messaging Accelerator (VMA). http://www.mellanox.com/page/software_vma?mtag=vma. Accessed on 2018-01-25.
[29] 2018. Micron Userspace NVMe/SSD Library. https://github.com/MicronSSD/unvme. Accessed on 2018-01-25.
[30] 2018. Microsoft SQL Server Two-Phase Commit. https://msdn.microsoft.com/en-us/library/aa754091(v=bts.10).aspx. Accessed on 2018-01-25.
[31] 2018. MMAPv1 Storage Engine - MongoDB Manual. https://docs.mongodb.com/manual/core/mmapv1/. Accessed on 2018-01-12.
[32] 2018. MongoDB. https://www.mongodb.com/. Accessed on 2018-01-25.
[33] 2018. MongoDB Two-Phase Commits. https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/. Accessed on 2018-01-25.
[34] 2018. Oracle Database Two-Phase Commit Mechanism. https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_txns003.htm#ADMIN12222. Accessed on 2018-01-25.
[35] 2018. Oracle MySQL/MariaDB Chain Replication Option. https://dev.mysql.com/doc/refman/5.7/en/replication-options-slave.html. Accessed on 2018-01-25.
[36] 2018. Persistent Disk - Persistent Local Storage | Google Cloud Platform. https://cloud.google.com/persistent-disk/. Accessed on 2018-01-25.
[37] 2018. redis. https://redis.io/. Accessed on 2018-01-25.
[38] 2018. Replication - MongoDB Manual. https://docs.mongodb.com/manual/replication/. Accessed on 2018-01-12.
[39] 2018. RocksDB. http://rocksdb.org/. Accessed on 2018-01-25.
[40] 2018. Seastar. http://www.seastar-project.org/. Accessed on 2018-01-25.
[41] 2018. SQL Database - Cloud Database as a Service | Microsoft Azure. https://azure.microsoft.com/en-us/services/sql-database/. Accessed on 2018-01-25.
[42] 2018. Stress-ng. http://kernel.ubuntu.com/~cking/stress-ng/. Accessed on 2018-01-25.
[43] 2018. Table storage. https://azure.microsoft.com/en-us/services/storage/tables/. Accessed on 2018-01-25.
[44] 2018. Two-Phase Commit Protocol. https://en.wikipedia.org/wiki/Two-phase_commit_protocol. Accessed on 2018-01-25.
[45] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 1997. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *International Workshop on Distributed Algorithms*. Springer, 126–140.
[46] Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *ACM EuroSys* (2013).
[47] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *ACM SOSP* (2009).
[48] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *USENIX NSDI* (2012).

[49] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI* (2014).

[50] K. Birman and T. Joseph. 1987. Exploiting Virtual Synchrony in Distributed Systems. *ACM SIGOPS Oper. Syst. Rev.* 21, 5 (1987).

[51] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *USENIX FAST* (2017).

[52] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX ATC* (2013).

[53] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *ACM SOSP* (2011).

[54] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing Safe, User Space Access to Fast, Solid State Disks. In *ACM ASPLOS* (2012).

[55] Adrian M. Caulfield and Steven Swanson. 2013. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *ACM/IEEE ISCA* (2013).

[56] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC* (2010).

[57] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013).

[58] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *ACM SOSP* (2007).

[59] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *USENIX NSDI* (2014).

[60] 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 *ACM SOSP* (2015).

[61] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *ACM EuroSys* (2014).

[62] Robert Escriva, Bernard Wong, and Emin GÃijn Sirer. 2012. HyperDex: A Distributed, Searchable Key-value Store. In *ACM SIGCOMM* (2012).

[63] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *ACM SOSP* (2003).

[64] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *ACM SIGCOMM* (2016).

[65] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX OSDI* (2012).

[66] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *USENIX FAST* (2017).

[67] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *USENIX FAST* (2017).

[68] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified Address Translation for Memory-mapped SSDs with FlashMap. In *ACM/IEEE ISCA* (2015).

[69] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX NSDI* (2014).

[70] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Sydney Zink, Ken Birman, and Robbert Van Renesse. 2017. Building Smart Memories and Cloud Services with Derecho. In *Technical Report* (2017).

[71] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *ACM SIGCOMM* (2014).

[72] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *USENIX OSDI* (2016).

[73] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. 2017. Viyojit: Decoupling Battery and DRAM Capacities for Battery-Backed DRAM. In *ACM/IEEE ISCA* (2017).

[74] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *ACM ASPLOS* (2017).

[75] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010).

[76] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *USENIX FAST* (2015).

[77] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *ACM ASPLOS* (2016).

[78] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *USENIX ATC* (2017).

[79] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *ACM SIGCOMM* (2014).

[80] Mellanox. 2018. RDMA Aware Networks Programming User Manual. http://www.mellanox.com/. Accessed on 2018-01-25.

[81] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Upyears for Non-volatile Main Memories with Kamino-Tx. In *ACM EuroSys* (2017).

[82] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC* (2013).

[83] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992).

[84] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *USENIX NSDI* (2017).

[85] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *ACM SOSP* (2011).

[86] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *USENIX OSDI* (2010).

[87] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In *ACM EuroSys* (2012).

[88] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *USENIX OSDI* (2014).

[89] Amar Phanishayee, David G. Andersen, Himabindu Pucha, Anna Povzner, and Wendy Belluomini. 2012. Flex-KV: Enabling High-performance and Flexible KV Systems. In *ACM MBDS* (2012).

[90] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC* (2012).

[91] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *ACM SoCC* (2017).

[92] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, Renato Ferreira, Mohamed Nassar, Michael Koltachev, Ji Huang, Sudipta Sengupta, Justin Levandoski, and David Lomet. 2015. Schema-agnostic Indexing with Azure DocumentDB. *Proc. VLDB Endow.* 8, 12 (Aug. 2015).

[93] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. 2016. Replex: A Scalable, Highly Available Multi-index Data Store. In *USENIX ATC* (2016).

[94] Jeff Terrace and Michael J. Freedman. 2009. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *USENIX ATC* (2009).

[95] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *USENIX OSDI* (2004).

[96] Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Robert Burgess, Gregory Chockler, Haoyuan Li, and Yoav Tock. 2010. Dr. Multicast: Rx for Data Center Communication Scalability. In *ACM EuroSys* (2010).

[97] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *USENIX FAST* (2016).

[98] Jisoo Yang, Dave B. Minturn, and Frank Hady. 2012. When Poll is Better Than Interrupt. In *USENIX FAST* (2012).

[99] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for Flash-based SSDs with Nameless Writes. In *USENIX FAST* (2012).

[100] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *ACM ASPLOS* (2015).

[101] Yanqi Zhou, Ramnatthan Alagappan, Amirsaman Memaripour, Anirudh Badam, and David Wentzlaff. 2017. HNVM: Hybrid NVM Enabled Datacenter Design and Optimization. In *Microsoft Research TR* (2017).