Cyclone: Warp Speed Replication for Key Value Stores

Paper **#224**

13 pages

Abstract

Persistent key value stores are rapidly becoming an important component of many distributed data serving solutions with innovations targeted at taking advantage of growing flash speeds. Unfortunately their performance is hampered by the need to maintain and replicate a write ahead log to guarantee availability in the face of machine and storage failures. Cyclone offers a solution to this problem by leveraging a small amount of directly attached non-volatile memory (NVM) such as NVDIMMs to maintain a two level log: the upper level in a small amount of NVM draining into a larger lower level log maintained on flash. Replicating the smaller log kept in NVMs transforms the replication problem into a packet switching problem as the leader sends the exact same packet to all follower replicas. We solve this problem by using the RAFT protocol implemented as a software multicast module on the Data Plane Development Kit (DPDK). In addition, we leverage the observation that key value store interfaces are commutative for operations to different keys to scale our two level log horizontally into a number of physical two level logs. A hash of the key is used to choose the physical log. Each log is replicated by an independent instance of RAFT, thereby leveraging the scalability afforded by classic software packet switching when using multiple CPU cores. Cyclone is able to replicate millions of small updates a second using only commodity 10 gigabit ethernet adapters, improving the performance and availability of RocksDB, a popular persistent key value store by an order of magnitude when compared to its own write ahead log without replication.

1. Introduction

Persistent key value stores are an increasingly important component of datacenter scale storage services. Key value stores such as Rocksdb [5], LevelDB [6] and FloDB [10] represent large amounts of effort on both the engineering and research fronts. These key value stores include sophisticated in-memory data structures built around Log Structured Merge trees [23] and are heavily tuned to extract maximum performance from flash-based solid state drives (SSDs).

These key value stores however tend to ignore an important component: the write ahead log. A machine or storage failure leading to total loss or temporary unavailability of data is unacceptable in many internet scale services where high availabilty and revenue are often interconnected. Key value stores therefore usually incorporate support for a write ahead log that if replicated and kept durable for every appended update provides the necessary high availability. Unfortunately the write ahead is a performance achilles heel for these systems - eclipsing much of the work on improving the performance of the LSM component. To illustrate the impact of the write ahead log, consider Figure 1. The line marked 'Rocksdb' shows the performance of Rocksdb without the write ahead log. The performance when persisting the write ahead log without replicating it is shown as the line marked 'rocksdb/WAL'. The line marked 'rocksdb/3 way rep.' is for simply replicating the log three ways without persisting it, using RAFT [24] running over TCP/IP. Either persisting every update to the log or replicating it using TCP/IP causes performance to drop by an order of magnitude (note the log scale on the x-axis). It is therefore no suprise that deployments of Rocksdb often turn off the write ahead log [7]. On the other side of the spectrum, key value store research such as FloDB [10] turns off the write ahead log to be able to showcase benefits of sophisticated extensions to LSM datastructures.

Cyclone is a high speed strongly consistent replicated write ahead logging service specialized for key value stores such as Rocksdb. Cyclone provides the order of magnitude better performance required to close the performance gap in Figure 1 both on the storage and network side. Cyclone requires *a small amount of* directly attached Non Volatile Memory (NVM) on the host - such as in the form of NVDIMMs [14] or directly attached persistent memory [15].

Cyclone makes use of a limited amount of attached NVM using two ideas. First, it leverages the observation that with persistence being a first class property of a subset of directly addressable DRAM, replicating the same log entry from the

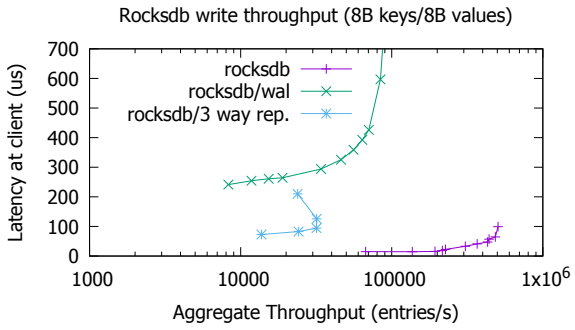

Figure 1. Rocksdb write ahead logging impact

leader to follower replicas is no different from multicast using software packet switching techniques. It directly leverages this observation to implement a version of RAFT that replicates the NVM log using classic packet switching techniques such as separation of control and data planes. Second, Cyclone uses a two level log structure where it drains entries from the NVM log in IO friendly batches to a log placed on a flash drive. This allows Cyclone to use only a limited amount of NVM following the reasoning that current forms of NVM such as NVDIMMs are more expensive than DRAM. Our experiments use an NVM footprint of a mere 64 MB to achieve performance comparable to placing the entire log in NVM. This allows Cyclone to be used in deployments where the key value store is maintained entirely in main memory rather than as a multi-level LSM tree structure, with the log serving as the source for recovery in the event of failure.

We discuss the two level structure focusing on the packet switching components in Section 2. We then describe how we can horizontally scale this basic idea across multiple cores by leveraging the commutativity of key value store APIs across different keys in Section 3. We then demonstrate in the evaluation in Section 4 that Cyclone can replicate millions of updates a second using only commodity 10 Gigabit ethernet adapters thereby closing the performance gap between key value store performance and write ahead logging performance using only a small amount of directly attached NVM. A discussion of related work follows before we conclude.

2. Two Level Log

A fundamental challenge in building an efficient replicated log is dealing with network overheads. Cyclone's construction is predicated on the observation that in the common case, protocols such as RAFT simply multicast the same entry received from a client, from the leader replica to all follower replicas. This suggests that an efficient way to implement a replicated log is to treat it as a software packet switching problem. However peristing logs on flash represents an

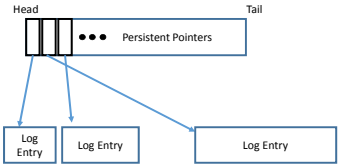

Figure 2. NVM log structure

impediment in that block storage devices are not good candidates to hold packets being software switched. Our solution is to split the log into two levels. The upper level is held in directly attached non-volatile memory (NVM) such as NVDIMMs. The lower level is held on a flash SSD - we term the lower level log as the flashlog. We periodically drain entries in batches from the NVM log to the backing flashlog. An important consequence of this two level approach is that we need only a limited amounts of NVM with the bulk of the log residing on flash - an arrangement that maps well to the current relative costs of the two types of memory. We begin by describing the NVM log and how we implement the RAFT consensus protocol in the packet switching paradigm around it. We then describe the flashlog and how we drain entries in batches from the NVM level to the flash level.

2.1 NVM Log

The NVM log is maintained as a circular log of fixed sized pointers (a pointer log) to memory buffers - each of which holds a client request. This arrangement is shown in Figure 2 that also illustrates the advantage of such an arragement. Adding a level of indirection in the NVM log allows us to separate the FIFO ordered circular log being manipulated by the RAFT from variable sized client being managed by the memory allocator of the packet switching library we use. Another advantage is that it makes recovery from NVM easy - appends to the circular pointer log are atomic and we can use the pointer log to recover allocator state i.e. what pieces of NVM are currently in use by the log.

We now describe how we implement the RAFT consensus protocol to replicate the NVM log across a set of replicas. The software packet switching idea we apply is to separate the control plane from the data plane using the Data Plane Development Kit (DPDK [1]) library for direct userspace access to the network interface.

2.1.1 Dataplane Separation

We use RAFT [24] as the underlying protocol for log replication. RAFT replicates a log across a set of replicas. Its operations are leader-based and RAFT incorporates a leader election algorithm that elects a leader with the most up to date log. The leader receives log entries from clients, appends them to the persistent log (NVM log in this case) and multicasts the new entry to follower replicas. On a response

from enough follower replicas (constituting a majority quorum) the leader replica can then execute the command in the log entry. In our case this involves accessing the key value store and sending the response back to the client.

We implement the common case flow in RAFT as a software packet switch using DPDK. The dataplane now encompasses the Network Interface Card (NIC) and the CPU caches with the NIC placing packet data directly in the CPU cache via DDIO [2] - a standard feature of most NICs today. The dataplane for the RAFT protocol is therefore separated from the control plane. The control plane runs in a thread pinned to a dedicated CPU core and takes care of the protocol state machine: checking for correct term, log index and that the replica receiving the packet from the client is indeed the leader. On the follower side it is responsible for checking that the leader is in the correct term before accepting the packet by appending it to the NVM log and initiating a response. In addition, the control plane is responsible for uncommon case activities such as dealing with timeouts and leader elections.

The pseudocode in Figure 3 describes part of the control plane in Cyclone organized as event handlers triggered on receiving a packet at the leader. We focus on one event for illustration (and brevity): the event where a request is received from a client. A key concern when we built Cyclone was adhering as far as possible to the software packet switching principle of separating the control plane from the dataplane. This means that the CPU should avoid examining every byte of the packet as far as possible.

The sequence of operations when a packet is received from the client needs careful design for efficient software packet switching. To illustrate how this is achieved, Figure 4 shows how Cyclone manipulates packet layouts across the two steps of prepending a RAFT header and transmitting to follower replicas. DPDK describes packets using an "mbuf" data structure. Roughly speaking, an "mbuf" consists of a flat array of bytes actually containing the packet and a fixed size piece of external metadata that describes various aspects of the packet, most crucially a pointer to the start and end of the packet in the byte array. DPDK's userspace drivers receive packets from the NIC such that they are offset in the byte array by a configurable amount referred to as "headroom". We strip off the existing network headers in the packet and prepend RAFT related information specific to each log entry in the headroom by shifting the start pointer appropriately. These operations are standard enough for software packet switches that DPDK provides convenient library calls for it. For the final step, we need to prepare the packet for transmission to the various follower replicas. To do this we prepare a different packet containing an ethernet header for *each* targeted replica and "chain" the data packet to each of these headers. Each header is then separately handed off

```
event_receive_client_req()
{
  if(!check_is_leader()) {
    drop_msg
    return
  }
  Prepend raft log term and index
  Persist to local log
  Transmit to follower replicas
}
```

Figure 3. Cyclone control plane fragment

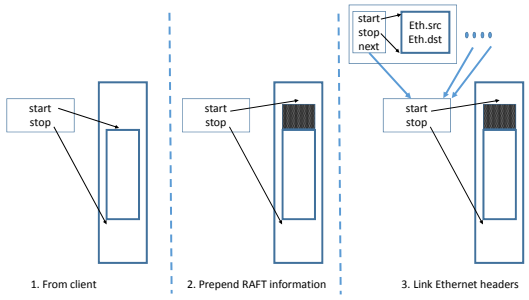

Figure 4. Cyclone network packet layout

to the driver for transmission via the NIC, carrying the data packet with it by association.

We note that by discarding stream oriented protocols such as TCP, the network layer is no longer reliable or ordered. This is not a problem for RAFT that is designed to tolerate network failures resulting in drops or reordering of network packets. However, this rules out consensus protocols such as Zookeeper Atomic Broadcast (ZAB [3]) that depend on a reliable stream-oriented network layer to function correctly.

Finally, we turn our attention to the persistence step in Figure 3. RAFT requires that the log entry be persisted before it is multicast out to follower replicas. Since the NVM is directly attached we do this by executing a cacheline flush (`clflush`) instruction for every cacheline in the packet and the pointer in the pointer buffer to persist these via the memory bus. This is the one exception to our otherwise clean separation between the control and data plane - necessitated by the fact that at the moment NICs do not support persistent memory. This is not too onerous a burden because we can use the newly introduced `clflush-opt` [4] instruction specifically intended to efficiently flush to persistent memory without the overhead of the serialization normally introduced by `clflush`. This allows us to hit full memory bandwidth on present generation platforms, a quantity in excess of 200 Gb/s per core, well above the near term speeds of network interface cards.

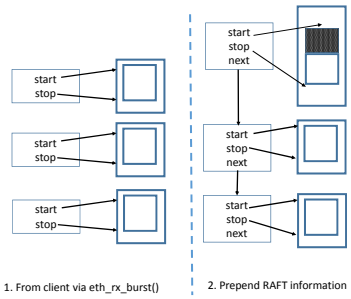

Figure 5. Batching

2.1.2 Batching

Although the control plane code only runs operations related to the RAFT state machine it is still slow enough relative to the dataplane to add significant overhead for each packet. We tackle this problem by applying another classic software packet switching technique: batching - illustrated in Figure 5. We use a burst receive call available in the DPDK userspace driver to receive a burst of client packets at a time. We then chain these packets together and treat them as a single log entry from the perspective of RAFT, amortizing the control plane overheads over the packets (at most 32 at a time due to current driver limitations).

We note that batching in Cyclone does not involve a latency-throughput tradeoff like in many other systems [11]. The batch receive call we use in DPDK returns immediately with whatever number of packets is available, including zero. We always flush the transmit buffer after every call to DPDK to transmit packets to replicas. Therefore, we never tradeoff latency for throughput when batching.

2.2 Flash Log

Cyclone is designed to use only limited amounts of NVM and drains log entries down into a log placed on a flash SSD - that we term the flashlog. The flashlog is written out in segments of configurable size (we use 128KB segments). A segment buffer is prepared in memory (volatile DRAM) using the layout shown in Figure 6. We do not allow objects in the flashlog to cross a 4KB boundary - linking multiple objects together with a special flag encoded into the size if necessary. We flush log segments out to the log file using asynchronous direct IO, and therefore we fill one log segment buffer while keeping IO to another one outstanding to the flash drive. To avoid having to do a synchronous metadata flush, we preallocate (using the posix_fallocate call) a gigabyte worth of zero filled disk pages at the end of the log file whenever we hit its end.

In order to recover from a crash, we make two important assumptions about the underlying SSD. First, we assume that 4KB is the *minimum* atomic unit for updating pages on the SSD even under power failure i.e. there are no shorn

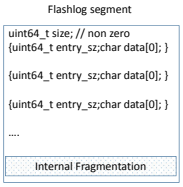

Figure 6. Flashlog segment

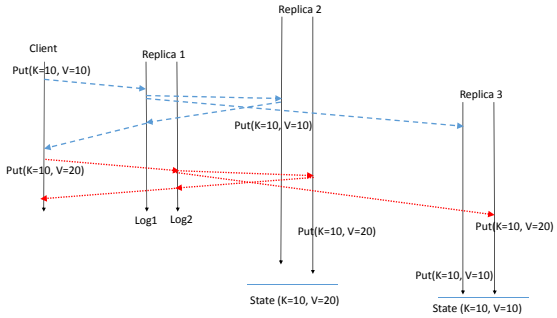

Figure 7. Race with multiple physical logs

writes (otherwise known as torn writes) on a 4KB page [28]. We also assume that the SSD has power loss data protection meaning that writes cached in the drive's volatile cache are written to the SSD using a backup capacitor in the event of power failure - a property of many data center class SSDs today, including the Intel DC P3600 SSD [8] we use in our evaluation. Together these two assumptions mean that we can recover a consistent prefix of the log on a power failure.

We move log entries from the head of the NVM log to the flashlog buffers in FIFO order. The NVM log entry is only actually removed when the IO for the corresponding flashlog page is complete. This means that during recovery we can have the same log entry both in the NVM log and the flashlog, a condition that can be detected by examining the RAFT related information that is embedded as part of the logged packet.

The fact that we can immedately absorb updates into an NVM log and only move them to the flashlog in suitable units for block IO sidesteps a common problem with logging to secondary storage - the need to do group commit. Logging systems often wait to collect a batch of entries to write to secondary storage trading latency for throughput. This is unnecessary in Cyclone as the use of directly attached NVM with a two level log removes the need to make this tradeoff.

3. Horizontal Scaling

The basic principle in Cyclone is to turn log replication into a software packet switching problem. A key ingredient in software packet switching is applying multiple CPU cores to parallelize the control plane, scaling message processing throughput. In this section, we describe how we can horizon-

tally scale Cyclone's single *logical* log across multiple *physical* logs. For each physical log, we run an independent version of RAFT with the corresponding control plane mapped to a dedicated CPU core. Further, we assume a concurrent key value store where a set of CPU cores independently run operations from the physical logs on the shared memory key-value store data structure, synchronizing among themselves as necessary.

We begin by showing how we divide CPU cores and NIC resources on the plaform across replication and key value store work, and show how a KV store request flows across them. We then describe how a key-value store API can be mapped to independent physical logs without divergence in replica state. Finally, we describe special handling in Cyclone for multi-key operations.

3.1 Core Allocation

A key-value store integrated with Cyclone runs two types of threads. The first type of thread does key-value store work. This includes servicing lookup and update requests and background threads that flush in-memory data structures to secondary storage and do maintainence work on secondary storage data structures such as compacting log-structured merge trees [23]. The second type of thread is a dedicated thread per-physical log to run the corresponding control plane including the associated RAFT state machine. We use static assignment of threads to cores as recommended for packet switching libraries such as DPDK for the best possible performance. Therefore we use the terms thread and core interchangeably in this paper. Client requests are always received by the control plane core for the physical log the request is directed to. A response for the core is always sent by the KV store core that finally handles the request.

An important resource for Cyclone is the NIC that must be shared among the control plane cores that receive and replicate requests and the KV store cores that send back responses. Most NICs provide independent transmit/receive queue pairs and we aim for synchronization free access to these to aid horizontal scalability with respect to the network resource. DPDK already provides fairly scalable access to NIC queues, with the queue number being a parameter to send and receive calls. All that remains for us is to map the queues to the control plane and KV store cores, keeping in mind that there may be multiple NICs on the system. We do this by iterating over all control plane cores followed by all KV store cores. We assign two queue pairs to each control plane core, one to receive requests from the client and the other to transmit and receive replication protocol related messages. We assign one queue pair to each KV store core to send back responses to clients. We are careful to spread the different queue pairs for a core across the available NICs as far as possible. This is important due to the asymmetry between the different types of traffic. Queue pairs carrying replication traffic will likely be more busy than those carrying client request responses due to the amplification in message count to multicast the same message to different replicas.

The flow for an update operation in Cyclone is as follows. The network packet is received by the control plane for the targeted physical log. The packet is then immediately sent out for replication as described in the previous section. When responses from a majority quorum is received, the request is passed to the targeted KV store core for execution. The KV store core processes the request and sends back the response to the client. We note that replication is pipelined and decoupled from execution and thus multiple updates are usually in flight on the network for replication at the same time.

The flow for a read operation is simpler. It is still targeted to a physical log but instead of replicating it, the control plane core simply passes it on straight away to the KV store core for execution and response.

The control plane cores communicate with the KV store cores through a one-way lock-free FIFO queue passing pointers to packets. We leverage the convenient ring library in DPDK for this.

Finally, an asynchronous source of work in the system is maintenance of the two-level log. We depart from a 1:1 mapping between the first level NVM log and the second level flashlog here. Each physical log consists of one NVM log managed by the corresponding control plane core and multiple flashlogs each of which is managed as an additional responsibility by a KV store core. We therefore enforce a many to one mapping between key value store cores and a front end control plane core - thereby partitioning KV store cores across control plane cores. A request picks a control plane core and any one of the associated KV store cores for execution. The request is logged in the NVM log of the control plane core and garbage collected once it is drained to the flashlog of the selected KV store core.

3.2 Leveraging API Commutativity

Update requests to a key-value store include a key and an associated operation - such as "put" that sets the value for the key or "merge" that calls a user-defined operator to merge a new value for the key into the existing value. A naive approach that maps a key to a random physical log and subsequently to a random core on the system for execution runs into the determinism problem - replicas could diverge with respect to the final value assigned to a key on different replicas. We illustrate this problem with an example in Figure 7 where two updates to the same key race with each other on different physical logs and execution cores to end up being

applied in different orders on different replicas, leading to a divergence of state.

The solution to this problem is observe that the key-value store is a concurrent data structure that ultimately admits a serialization of operations to it. Further, consecutive operations to different keys in the serial history can be interchanged, because operations to different keys are *commutative*. The correct way to use different physical logs is therefore to simply hash the key to select the physical log *and* the core on which the operation executes. This guarantees determinism as the serial history of two different replicas can be transformed into the same serial history by reordering operations to different keys. Operations to the same key are already in the same order in the serial histories by virtue of being replicated on the same physical log and being executed on the same KV store core.

Further relaxations to the mapping scheme are possible when operations to the same key are also commutative. A good example is when key value stores are used to maintain counters. Different increment operations to the same counter are commutative if one is prepared to tolerate non-determinism in the sequence of updates seen by a reader with the counter reaching the same eventual value. This specific use-case, in fact, lead to the addition of the merge operation in Rocksdb, a key value store that we use for evaluation. Merge operations therefore could be dispatched to a random physical log with no harm. For this paper however, we stick to the stricter case that operations to the same key are not commutative. However, Cyclone's client side libraries leave it to the user to specify the exact physical log and key-value store core to use in order to enable more relaxed mappings when necessary.

We note that commutativity as assumed by Cyclone is an *API level property*. There is no assumption as to how application cores synchronize to access the shared key-value state or the underlying data structures for holding key-value pairs. We emphasize that this means replicas ultimately have the same content but not necessarily the same memory state.

Finally, Cyclone provides seraializable reads but does not - at the moment - guarantee the ability to read one's own writes, repeatable reads or more generally, linearizable reads. The reason is that on a failover the new leader replica might still have update operations in its NVM logs that are yet to be applied to the key value store. However the original failed leader replica might have applied these updates and a client might have already observed them. Users who desire linearizable reads can issue a special API call to ensure that reads are also routed through the logs. Flushing the NVM logs on failover is a better solution that we are currently in the process of engineering into Cyclone.

3.3 Ganged Operations

Key value stores (such as Rocksdb) usually provide a batched write operation that atomically updates a set of key-value pairs. A batched write is often a key primitive for building more sophisticated operations such as transactions. Batched writes require special handling in Cyclone due to the requirement that we route a write request by hashing the key to select both the control plane core (physical log) as well as KV store core. A batched write has multiple keys and therefore we need to synchronize the progress of the operation across multiple physical logs (and associated replication quorums) as well as KV store cores. In effect, the serial history contains a grouped set of operation on multiple keys as a single atomic operation. All serial histories across replicas therefore continue to map to the same equivalent serial history modulo reordering of operations to different keys.

One way to do this is to use a distributed transaction, using two-phase commit to couple the otherwise independent replication quorums and KV store cores. However this method has a number of problems. First, it introduces an additional network round trip that is normally not necessary for a key value store with a single shared memory image holding all the keys. Second, handling failures with distributed transactions would have caused an explosion in complexity in Cyclone, something we wished to avoid in the interests of practical deployment to independent end users who need to understand and maintain Cyclone as part of their embedded KV store infrastructure.

Our solution instead is to use a technique we term ganged operations. It is based on the observation that in the failure free case, executing a batched write across multiple control plane cores and KV store cores does not look very much different from a synchronous shared memory barrier. The challenge is to effectively manage failures in replication without causing participating cores to stop making progress. The first step is to reconfigure RAFTs leader election protocol to ensure that the leaders for all the physical logs are co-located on the same host. Clients always dispatch batched writes to a fixed control plane core (called the injection core) which is then responsible for forwarding the request to the participating control plane cores (for corresponding physical logs). The injection core uses DPDK's packet cloning primitives to avoid dataplane work while forwarding copies of the same batched write packet to multiple control plane cores. The injection core also adds a "nonce" - a unique timestamp to the packet. In addition, the injection core adds a unique view number to the packet containing the term numbers of all the participating RAFT instances (read from shared memory of the co-located leaders). The flow for ganged operations is described in the pseudocode of Figure 8. We defer the discussion of how we generate the nonce to later in this section. We also assume that a unique barrier is allocated in shared

memory for each ganged operation. We discuss later how this is done without using dynamic allocation.

In the absence of failures Figure 8 is straightforward. Once replication is complete on all participating physical logs, the corresponding KV store cores execute `event_replicated_ganged_op`. They use a bitmask for the barrier setting the appropriate bit for themselves. Once all KV store cores have arrived at the barrier the operation is executed. A distinguished leader core (we use the minumum numbered participating KV store core) is responsible for sending back the response to the client. Two types of failure need to be considered. First, replication on the physical log corresponding to one of the non-leader cores could fail. The remaning cores would never progress past the barrier. To solve this the leader core monitors the remaning cores for their current view - which is the RAFT term of the last executed log entry. If the view has moved past the view in the packet, replication failed on that physical log. The leader then sets the bit for the core that will no longer participate in the barrier to aid progress (a technique we borrowed from lock-free data structures [17]). It also sets a failed flag to signal failure to the other participating cores. The second failure case is that the leader core might not receive the operation if it fails during replication from corresponding physical log. This case is detected in a similar manner by the remaining cores by monitoring the published view from the leader core in shared memory. We add that on a leadership change due to timeout we enqueue and execute a nop on the physical log and all associated KV store cores to ensure previous entries are committed - a corner case in the RAFT protocol.

The assumption in Figure 8 is that each ganged operation is mapped to a unique barrier. We achieve this by using a fixed piece of memory owned by the leader core to hold the barrier and write the nonce to it in order to indicate that the barrer is active for the corresponding batched write operation. Non-leader cores watch for the nonce to know when to execute the ganged operation barrier in Figure 8, while also monitoring the leader's published view to detect the case where the ganged operation fails to replicate on the leader's physical log.

Finally, we describe how we generate the nonce. The nonce is generated on the injector core by concatenating the ethernet MAC ID of the first NIC on the system with a 64 bit value that is the number of CPU timestamp counter cycles since epoch time (read from the real time clock at startup plus the number of cycles from the CPU `rdtsc` instruction). The nonce can only be repeated if the same machine manages to fail and come back up in less time than the real time clock drift (controlled with NTP), a possibility that we discount.

Ganged operations possibly consititute the most complex part of Cyclone but the code weighs in at well under a couple of hundred lines. We believe that this additional complexity is still small compared to distributed transactions.

4. Evaluation

We evaluate Cyclone on a 12 node x86 Xeon cluster connected via a 10 GigE switch. Three of the machines are equipped with 1.6TB Intel DC P3600 SSDs and 4*10 GigE ports. The remaining nine machines do not have SSDs and have only one 10 GigE port, serving as clients for most of the experiments. We turn on jumbo frame support in the 10 GigE switch to enable maximum use of batching in Cyclone. As with other work [20], we use DRAM on the machines to proxy for NVDIMMs where necessary - the persistent memory needed never exceeds 64 MB regardless of the size of the key value store or second level log on flash. We divide the evaluation into three parts. First, we evaluate Cyclone's performance with a single level log as a pure software packet switch. Next, we evaluate performance when adding a second level of log on flash. Finally, we evaluate performance when integrated with Rocksdb [5] as an alternative to Rocksdb's write ahead log. Unless otherwise mentioned, we use a 60 byte header followed by an optional payload for experiments. We log both the header and payload. In all cases the server echoes the received entry (header and payload) back to the client. We fix the number of KV Store threads at 32 using at most 8 physcial logs with associated control plane cores for running RAFT instances.

Cyclone is built on top of DPDK to apply software packet switching techniques to the log replication problem. We therefore begin by systematically evaluating optimizations applied in Cyclone to replicate the NVM log in Figure 9 - with no payload. The y-axis reports latency seen at the client (which means two network round trips with replication). Using TCP/IP to replicate a RAFT log tops out at around 30K entries/s. Switching to DPDK (the line marked +DPDK) improves the throughput by an order of magnitude to around 500K entries/s. Using batching (the line marked +batching) improves the performance further bringing us close to a million entries/s. Scaling horizontally to 8 physical logs (+8 phy logs) improves performance to close to 2M entries/s. Finally using all 4 ethernet ports on the machine to replicate entries improves performance considerably to 6M entries/s. In all, performance improves by 200X over the TCP/IP single log baseline. Cyclone also considerably improves the latency for replication, from close to 100us with TCP/IP to around 30us at peak throughput.

There are two factors that can have significant impact on Cyclone's performance. First, the number of replicas dictates the outgoing message rate from the leader replica and therefore increasing the replication factor can decrease Cy-

```
// Control plane core
event_recv_ganged_op(packet)
{
  Begin replication of packet
}

// KV store core
event_replicated_ganged_op (packet, barrier)
{
  if(leader KV core)
    atomic set bit me in barrier.mask
    do
     for each KV store core in packet
      if core_public_data[core].view > packet.view
       barrier.failed = true
       atomic set bit for core in barrier.mask
    while barrrer.mask != mask of participating KV cores
    if barrier.failed
     Ganged replication failed. Send retry to client. return.
    else
     execute operation
     send response
  else
    wait until
     core_public_data[leader core].view > packet.view OR
     barrier.mask == mask of participating KV cores
    if core_public_data[leader core].view > packet.view OR
       barrier.failed
     Ganged replication failed. Send retry to client. return.
    else
     execute operation
}
```

Figure 8. Ganged Operation

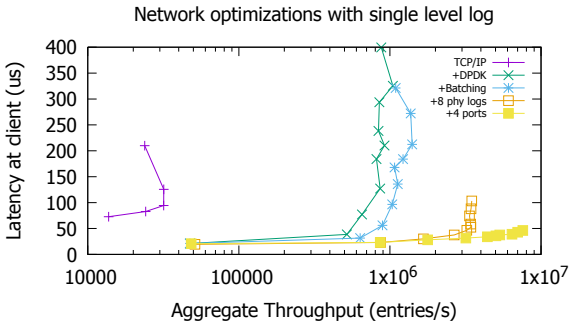

Figure 9. Network optimizations for top level log

clone's performance. Figure 10 shows the impact of varying replica count. Using only a single replica cuts out a network round trip and shows the best unloaded latency (10 us) and peak throughput (near 10M entries/s). Adding replicas decreases the peak throughput down to around 2M entries/s with 5 replicas. We note that a number of previous pieces of work [20, 14] use three replicas and therefore we focus on three replicas for the replicated cases we consider below. The second factor that dictates Cyclone's performance is the size of the log entry being replicated. Figure 11 shows the effect of increasing the payload size from zero to 512 bytes. Peak throughput drops from 6M entries/s to approximately 2M entries/s. At this replication rate, the leader replica needs to transmit data at approximately 30 Gbit/s. Coupled with the cost of network headers all four 10 GigE ports are now saturated and therefore Cyclone hits the network line rate bottleneck at this point.

We now turn our attention from the network component of Cyclone to the storage one by adding the second level flashlog. We evaluate the impact of adding block storage to our hitherto pure packet switching scenario in Figure 12. Batching entries from the top level NVDIMM log to the second level flashlog is clearly beneficial as adding flash storage at the second level has almost no impact on peak performance in Cyclone for small entries. The situation however

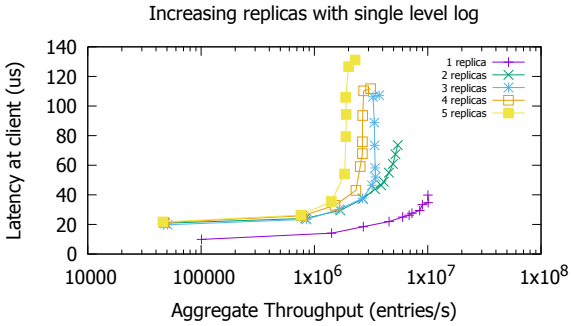

Figure 10. Impact of replica count

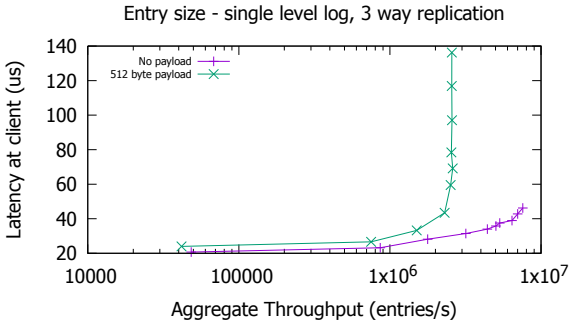

Figure 11. Impact of payload size

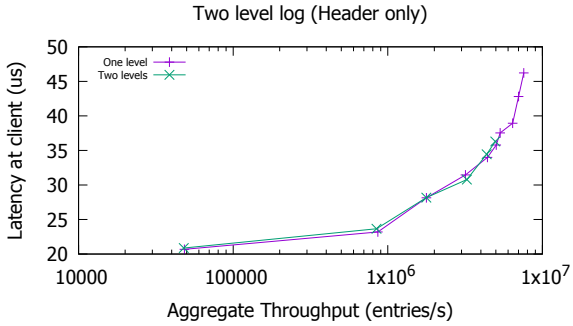

Figure 12. Impact of adding second level log

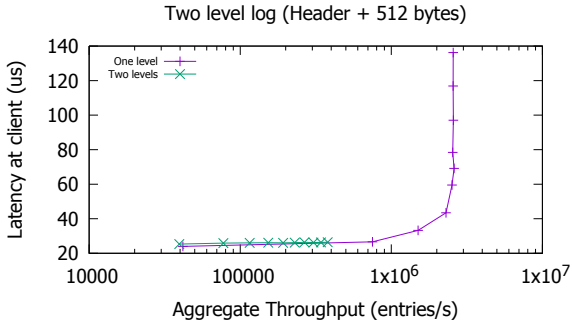

Figure 13. Second level log with 512 bytes payload

changes for larger entries. Figure 13 shows that using a 512 byte payload has a significant impact on peak throughput - it drops to approximately 350K ops/sec. This corresponds to around 50K 4KB IOPS to the SSD to write out the flashlog pages. The peak for the drive is 160K IOPS using a queue depth (concurrency) of 128. With 32 application threads we expect a lower peak throughput around 40K IOPS explaining our bottleneck at 50K IOPS. It is possible to tune our observed performance further by aligning the flush boundary to increase the number of outstanding requests - we do not do so in this paper, keeping Cyclone agnostic to the exact size of entry being replicated. A final point about Figure 13 is that once we are past the storage bottleneck the latency spike is dramatic and large enough to trigger Cyclone's failure detector and repeated retries from the clients. There are - therefore - no points on the "knee" of the curve as in the pure packet switched one-level log case.

The final dimension we evaluate is of ganged operations. The primary purpose of ganged operations is to avoid the need for distributed transactions to manipulate what is a single shared memory image and therefore we were most concerned about unloaded latency given the complexity of synchronizing different replication quorums as well executing our rendezvous protocol on multiple cores. We therefore setup an experiment where a single client - reflecting the unloaded case - made ganged requests to the replicas. We varied the number of cores participating in the request from one to the full complement of 32 application cores. Figure 14 shows the results both using a single level log as well as a two level log. The primary takeaway is that unloaded latency increases slowly as we increase the number of active cores - to around 40 us from the baseline of 20 us. There are two causes of this. First, there is a quicker rate of increase from 1 to 8 active cores, the reason being that this corresponds to an increase in the number of physical logs that must be synchronized on. There is some amount of synchronization involved in the userspace DPDK driver for access to common resources for accessing the NIC and this affects the ability to simultaneously issue replication messages on all quorums. The smaller contribution to increasing latency that continues past the 8 core case is due to cacheline pingponging when executing the rendevous of Figure 8. Both these sources of latency drift could be corrected using replication quorums mapped to dedicated NICs and more scalable rendezvous designs (such as with machine aware broadcast trees [19]). However we deemed the complexity of such optimizations unnecessary. The added latency for most cases is well under the extra round trip delay (including replication) for all phases of a 2PC.

We now evaluate Cyclone integrated with the Rocksdb persistent key value store. Rocksdb is a complex industrial strength persistent key value store and this means that it

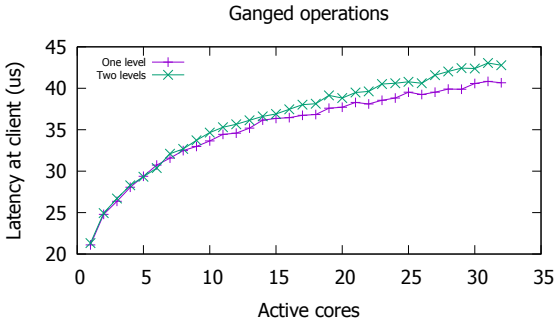

Figure 14. Ganged Operations

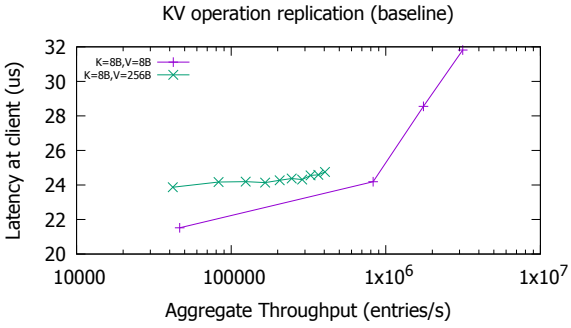

Figure 15. Baseline KV replication performance

is accompanied by a complex array of performance tuning knobs to get the best performance from flash. We were also aware that SSD performance is slated for dramatic increases in the coming years with the introduction of new memory technology such as 3DXPoint based Optane SSDs. To demonstrate that Cyclone is future proof we eliminated flash media performance from the picture as far as the key value store is concerned by placing all files for the key value store (SSTables) on a RAMdisk, which presumably represents the limit in performance for flash in the near future. All log files on secondary storage however - both Rocksdb's own write ahead log and the alternative of Cyclone's second level flashlog - are placed on the SSDs.

We evaluate two different request sizes: 8 byte keys with 8 byte values and 8 byte keys with 256 byte values. We run 8 physical logs on 8 cores and devote 32 cores to Rocksdb. Since we are interested in performance of the log, which is only used for update requests, our workload is 100% writes.

Before evaluating with Rocksdb, we measure the baseline performance of replicating the log with Cyclone for the given request sizes - Rocksdb performs a no-op. We note that in addition to the key and value, we are also logging RocksDB specific request data such as operation type and the request header. Figure 15 shows the baseline performance for the chosen request sizes. With the smaller request size, Cyclone can conservatively sustain close to a million requests a second at a latency of just under 25us. With the larger request size, Cyclone can sustain around 350K requests a second, again at a latency of just under 25 us. Armed with these baseline numbers we now examine how well Cyclone performs with Rocksdb.

The performance of Rocksdb with Cyclone for the small update workload is shown in Figure 16 - essentially presenting the solution to the problem description graph in Figure 1. We consider four different setups. The line labeled 'rocksdb' is the key value store running with no logging whatsover - a system crash would lead to data loss. The line labeled 'rocksdb/wal' is for Rocksdb running with its write ahead logging turned on. The large gap between these two is the

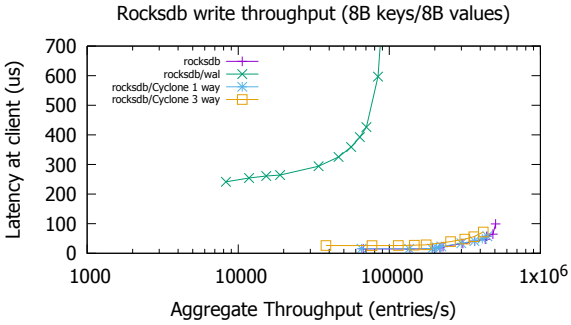

Figure 16. Rocksdb - small updates

overhead of the existing Rocksdb WAL solution. The line labeled 'rocksdb/Cyclone 1 way' is a two level Cyclone log but without any replication. The line almost exactly tracks the performance of Rockdb. As suggested by the baseline replication performance, Cyclone is able to provide a write ahead log with no overhead to Rocksdb. The line labeled 'rocksdb/Cyclone 3 way' is with 3-way replication turned on. Other than a 20us delta due to the extra network round trip, the line almost exactly tracks Rocksdb performance with no logging. Cyclone therefore provides high availability to Rocksdb at a fraction of the cost of its existing single machine write ahead log. We also repeat the experiment for the larger update size in Figure 17. The conclusions are identical: Cyclone solves Rocksdb's write ahead logging problem.

Next, we consider the problem of supporting Rocksdb's write-batch operation that atomically writes a set of key-value pairs into the KV store. In order to perform the operation with Cyclone managing the log, the client must issue a ganged operation across cores owning the keys in the write batch. This is in contrast to baseline Rocksdb where the entire write batch can be sent to any core. A key concern here was whether Cyclone would add any latency to the operation due to the extra synchronization needed across replication quorums and participating application cores. We examine the problem for the unloaded case and small updates in Fig-

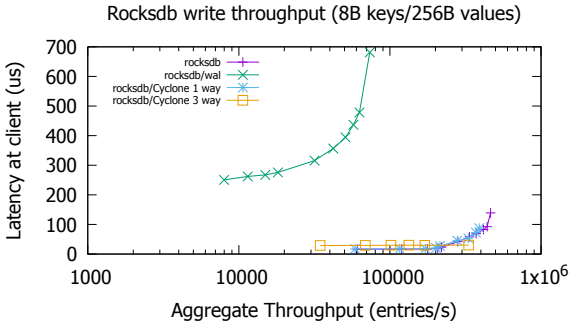

Figure 17. Rocksdb - large updates

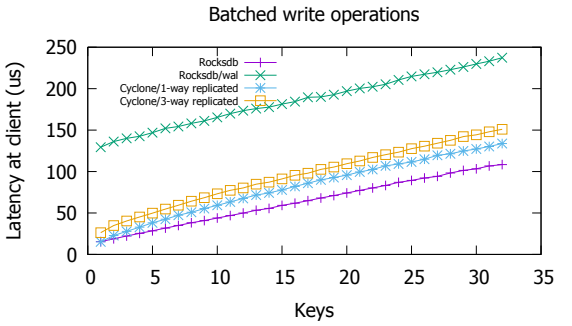

Figure 18. Rocksdb - batched writes

ure 18 for a single client with increasing number of keys in the batch - till 32 keys that covers all application cores. The line labeled Rocksdb is with no write ahead logging. We note an increasing latency for this baseline indicating Rocksdb takes longer with larger key batches. The existing option of Rocksdb/wal has considerably larger latency. Cyclone does an effective job of cutting down on this latency even as it needs to pay a price for synchronizing multiple quorums and application cores making it somewhat slower than running Rocksdb with no logging for batched writes. A distributed 2PC transaction still remains more expensive just in terms of network delay than using Cyclone.

Finally, we showcase the benefit of using Cyclone beyond pure performance as compared to the existing single machine Rocksdb write ahead log. Cyclone brings multi-machine availability with the ability to automatically failover. We demonstrate this in Figure 19 that shows the timeline of a run where we kill the server process on the leader replica. Cyclone is configured to with a 30ms failure detection timeout after which the client library tries replicas in turn to locate the leader - in this case it fails over in about 60ms.

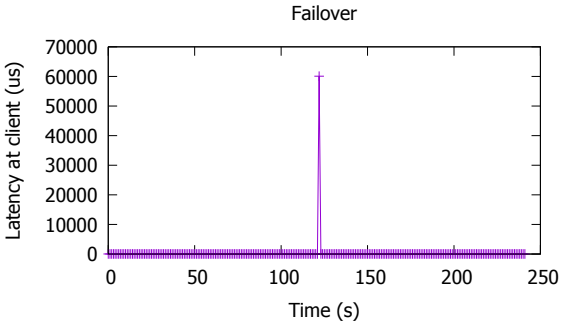

Figure 19. Rocksdb - failover

5. Related Work

Cyclone takes a software only approach to improving replication performance using commodity network hardware. At the other end of the spectrum, Consensus in a box [18] implements the Zookeeper atomic broadcast protocol together with a NIC on an FPGA. That work ignored durability focusing purely on replication performance. Latency numbers are impressive due to the FPGA that cuts out the path from CPU caches to the NIC and are reported to be as low as 3us for a round trip from leader to follower replica compared to the 7us we observe with DPDK. Notably however they reported 7us when using TCP/IP from the FPGA rather than their own connection oriented network protocol. On the other hand peak replication throughput appears to be 6M replications/sec on 66 Gbps of aggregate network bandwith, comparable to the mark obtained by Cyclone on an aggregate 40 Gbps of network bandwidth. We believe the general applicability afforded by our software only packet switching solution makes it a compelling alternative to an FPGA based solution even given the larger latency.

Network Ordered Paxos [22] reprograms top of rack switches to order client requests and multicast them to replicas - attacking replication latency by removing the need to direct requests through a leader replica. It is hard to compare performance due to the 1Gbps links used to leaf nodes in NoPaxos but we note that avoiding changes to switching infrastructure was a specific constraint in the design of Cyclone due to the difficulty we encountered in convincing public cloud customers to allow such changes to be made.

CRANE [13] is a system for replicating mutithreaded programs using Paxos. It uses deterministic multi-threading to ensure replicas converge to the same state as opposed to semantic equivalence in Cyclone using the commutativity of the key value store interface. Although CRANE is more generally applicable in that it makes no assumptions about the application API, deterministic multithreading comes with high overheads for applications that store large amounts of data - CRANE reports around a 2X slowdown for MySQL

likely rendering it unsuitable as a replacement for the write ahead log in key value stores.

A number of researchers have proposed using high performance networking hardware for improving the performance of various distributed system primitives such as distributed transactions [14] and replication [25, 27]. We note that the ideas in Cyclone are not specific to ethernet. The same ideas of packet switching and horizontal scaling can be equally applied to RDMA. However, we chose commodity ethernet due to short term deployment plans that are a priority design target.

The scalable commutativity rule [12] generalizes the idea of commutativity in key value store interfaces that we have used for horizontal scaling in Cyclone. Conveniently, it provides a tool for checking when API calls can commute in a history. It is a simple exercise to express a key value store API in COMMUTER and check that operations to different keys commute. This provides a theoretical basis for horizontal scaling, something we have omitted for brevity from this paper. The scalable commutativity rule also suggests that scalable key value store implementations would necessarily have commutative interfaces, something we confirm and exploit for key value stores.

There is an in progress transformation of durable memory in datacenters with the advents of 3DXPoint. Researchers have speculated on the impact of cheap and high volume directly attached 3DXPoint [16]. Memory vendors have also begun shipping software libraries such as NVML [9] for building durable in-memory data structures using undo logging. A durable key value store using 3DXPoint would be an idea target for Cyclone because each NVM log entry can be discarded as soon as it is applied to the persistent key value store removing the need for the flashlog. This would allow Cyclone to operate at pure packet switching speeds, a goal we hope to achieve in the future with a single level log for NVML data store applications.

Cyclone provides a client library that automatically times out and retries requests to deal with failure. Protection against repeated execution of the same request is currently left to the key value store and client, since we felt that exactly once semantics is optional when updates such as put operations are idempotent. We have experimented with exactly once semantics as a basis for distributed transactions across shards running Cyclone integrated key value stores, along the lines of similar work [21], using a fixed client space and maintaining information about last seen sequence numbers from clients in NVM. Distributed transactions on Cyclone shards however are outside the scope of this paper.

Finally, we note that scaling a log horizontally for concurrency is also an idea adopted by filesystems. NOVA [26] is an example that uses a per-inode log to improve concurrency when the filesystem is placed on NVM attached to the memory bus. Cyclone also uses a distinct NVM log per control plane core in a similar vein but does not maintain a namespace since the number of NVM logs is a small fixed number equal to the number of control plane cores.

6. Conclusion

Persistent key value stores struggle today to provide high availability using a write ahead log without a significant impact to performance. Cyclone solves this problem by leveraging a small amount of directly attached non-volatile memory that transforms the replication problem into a software packet switching problem. Cyclone and its Rocksdb bindings are slated for release to the open source community and we hope both the code and the ideas in the system itself will help key value store builders and researchers overcome their logging problem.

References

[1] `http://dpdk.org`

[2] `http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html`

[3] `http://www.tcs.hut.fi/Studies/T-79.5001/reports/2012-deSouzaMedeiros.pdf` 2012.

[4] 2015.

[5] `http://rocksdb.org/` 2017.

[6] `http://leveldb.org/` 2017.

[7] 2017.

[8] `http://www.intel.com/content/www/us/en/solid-state-ssd-dc-p3600-spec.html` 2017.

[9] `http://nvml.io` 2017.

[10] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 80–94. ACM, 2017.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 49–65. USENIX Association, 2014.

[12] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Symposium on Operating Systems Principles*, pages 1–17. ACM, 2013.

[13] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of the Symposium on Operating Systems Principles*, pages 105–120. ACM, 2015.

[14] 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 Symposium on Operating Systems Principles*, pages 54–70. ACM, 2015.

[15] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[16] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the European Conference on Computer Systems*, pages 15:1–15:16. ACM, 2016.

[17] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, 2004.

[18] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proceedings of the Conference on Networked Systems Design and Implementation*, pages 425–438. USENIX Association, 2016.

[19] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. Machine-aware atomic broadcast trees for multicores. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 33–48. USENIX Association, 2016.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 185–201. USENIX Association, 2016.

[21] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the Symposium on Operating Systems Principles*, pages 71–86. ACM, 2015.

[22] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 467–483. USENIX Association, 2016.

[23] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385.

[24] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the Annual Technical Conference*, pages 305–320. USENIX Association, 2014.

[25] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118. ACM, 2015.

[26] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the Conference on File and Storage Technologies*, pages 323–338. USENIX Association, 2016.

[27] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18. ACM, 2015.

[28] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of ssds under power fault. In *Proceedings of the Conference on File and Storage Technologies*, pages 271–284. USENIX Association, 2013.