CORFU: A Shared Log Design for Flash Clusters

Mahesh Balakrishnan[§], Dahlia Malkhi[§], Vijayan Prabhakaran[§]
Ted Wobber[§], Michael Wei[‡], John D. Davis[§]

[§] Microsoft Research Silicon Valley [‡] University of California, San Diego

Abstract

CORFU[1] organizes a cluster of flash devices as a single, shared log that can be accessed concurrently by multiple clients over the network. The CORFU shared log makes it easy to build distributed applications that require strong consistency at high speeds, such as databases, transactional key-value stores, replicated state machines, and metadata services. CORFU can be viewed as a distributed SSD, providing advantages over conventional SSDs such as distributed wear-leveling, network locality, fault tolerance, incremental scalability and geo-distribution. A single CORFU instance can support up to 200K appends/sec, while reads scale linearly with cluster size. Importantly, CORFU is designed to work directly over network-attached flash devices, slashing cost, power consumption and latency by eliminating storage servers.

1 Introduction

Traditionally, system designers have been forced to choose between performance and safety when building large-scale storage systems. Flash storage has the potential to dramatically alter this trade-off, providing persistence as well as high throughput and low latency. The advent of commodity flash drives creates new opportunities in the data center, enabling new designs that are impractical on disk or RAM infrastructure.

In this paper, we posit that flash storage opens the door to shared log designs within data centers, where hundreds of client machines append to the tail of a single log and read from its body concurrently. A shared log is a powerful and versatile primitive for ensuring strong consistency in the presence of failures and asynchrony. It can play many roles in a distributed system: a consensus engine for consistent replication; a transaction arbitrator [13, 22, 25] for isolation and atomicity; an execution history for replica creation, consistent snapshots, and geo-distribution [16]; and even a primary data store that leverages fast appends on underlying media. Flash is an ideal medium for implementing a scalable shared log, supporting fast, contention-free random reads to the body of the log and fast sequential writes to its tail.

One simple option for implementing a flash-based shared log is to outfit a high-end server with an expensive PCI-e SSD (e.g., Fusion-io [2]), replicating it to handle failures and scale read throughput. However, the resulting log is limited in append throughput by the bandwidth of a single server. In addition, interposing bulky, general-purpose servers between the network and flash can create performance bottlenecks, leaving bandwidth underutilized, and can also offset the power benefits of flash. In contrast, clusters of small flash units have been shown to be balanced, power-efficient and incrementally scalable [5]. Required is a distributed implementation of a shared log that can operate over such clusters.

Accordingly, we present CORFU, a shared log abstraction implemented over a cluster of flash units. In CORFU, each position in the shared log is mapped to a set of flash pages on different flash units. This map is maintained – consistently and compactly – at the clients. To read a particular position in the shared log, a client uses its local copy of this map to determine a corresponding physical flash page, and then directly issues a read to the flash unit storing that page. To append data, a client first determines the next available position in the shared log – using a sequencer node as an optimization for avoiding contention with other appending clients – and then writes data directly to the set of physical flash pages mapped to that position.

CORFU's client-centric design has two objectives. First, it ensures that the append throughput of the log is not a function of the bandwidth of any single flash unit. Instead, clients can append data to the log as fast as the sequencer can assign them 64-bit tokens, i.e., new positions in the log. Our current user-space sequencer runs at 200K tokens/s; assuming 4KB entries and two-way replication, this is sufficient to saturate the write bandwidth of a cluster of 50 Intel X25-V [1] drives, which in turn can support a million random 4KB reads/sec. Es-

[1]CORFU stands for Clusters of Raw Flash Units, and also for an island near Paxos in Greece.

sentially, CORFU's design decouples ordering from I/O, extracting parallelism from the cluster for appends while providing single-copy semantics for the shared log.

Second, placing functionality at the clients reduces the complexity, cost, latency and power consumption of the flash units. In fact, CORFU can operate over SSDs that are attached directly to the network, eliminating general-purpose storage servers from the critical path. In a parallel effort outside the scope of this paper, we have prototyped a network-attached flash unit on an FPGA platform; when used with the CORFU stack, this custom hardware provides the same throughput as a server-based flash unit while using an order of magnitude less power and providing 33% lower latency on reads. Over a cluster of such flash units, CORFU's logging design acts as a distributed SSD, implementing functionality found inside conventional SSDs – such as wear-leveling – at cluster scale.

To realize these benefits of a client-centric design, CORFU needs to handle failures efficiently. When flash units fail, clients must move consistently to a new map from log positions to flash pages. CORFU achieves this via a reconfiguration mechanism (patterned after Vertical Paxos [17]) capable of restoring availability within tens of milliseconds on drive failures. A challenging failure mode peculiar to a client-centric design involves 'holes' in the log; a client can obtain a log position from the sequencer for an append and then crash without completing the write to that position. To handle such situations, CORFU provides a fast hole-filling primitive that allows other clients to complete an unfinished append (or mark the log position as junk) within a millisecond.

CORFU's target applications are infrastructure layers that use the shared log to implement high-level interfaces. In this paper, we present two such applications. CORFU-Store is a key-value store that supports atomic multi-key puts and gets, fast consistent checkpointing and low-latency geo-distribution; these are properties that are difficult to achieve on conventional partitioned key-value stores. CORFU-SMR is a state machine replication library where replicas propose commands by appending to the log and execute commands by playing the log. In addition to these, we are currently building a database, a virtual disk layer, and a reliable multicast mechanism over CORFU.

We evaluate a CORFU implementation on a cluster of 32 Intel X25-V drives attached to servers, showing that it saturates the aggregate storage bandwidth of the cluster at speeds of 400K 4KB reads per second and nearly 200K 4KB appends per second over the network. We also evaluate CORFU running over an FPGA-based network-attached flash unit, showing that it performs end-to-end reads under 0.5 ms and cross-rack mirrored appends under 1 ms. We show that CORFU is capable of recovering from holes within a millisecond, and from crashed drives within 30 ms. Finally, we show that CORFU-Store provides atomic operations at the speed of the log (40K 10-key multi-gets/s and 20K 10-key multi-puts/s, with 4KB values), and that CORFU-SMR runs at 70K 512-byte commands/s with 10 state machine replicas.

To summarize the contributions of this paper: we propose the first complete design and implementation of a shared log abstraction over a flash cluster. This is also the first distributed, shared log design where maximum append throughput is not a function of any single node's I/O bandwidth. We describe low-latency fault-tolerance mechanisms for recovering from flash unit crashes and holes in the log. We present designs for a strongly consistent key-value store and a state machine replication library that use CORFU. Finally, we evaluate CORFU throughput, latency, fault-tolerance and application performance on a 32-drive cluster of server-attached SSDs as well as an FPGA-based network-attached SSD.

2 Motivation

As stated earlier, the key insight in this paper is that flash storage is an ideal medium for shared log designs. The primary argument is that flash provides fast, contention-free random reads, which enables designs where hundreds of clients can concurrently access a shared log. However, the CORFU design of a shared, distributed log makes sense for other reasons, as well. We first offer a quick primer on the properties of flash storage, and then expand on the rationale for a shared, distributed log.

Flash is read and written in increments of pages (typically of size 4KB). Before a page can be overwritten, it must be erased; erasures can only occur at the granularity of multi-page blocks (of size 256KB). Significantly, flash wears out or ages; as a page is erased and overwritten, it becomes less reliable and is eventually unusable. From a performance standpoint, overwriting a randomly selected flash page requires the surrounding block to undergo an erase operation in the critical path, resulting in poor random write speeds.

At the level of a single drive, these problems are masked from applications by the Flash Translation Layer (FTL) within an SSD. SSDs implement a logical address space over raw flash, mapping logical addresses to physical flash pages. Since flash chips within an SSD cannot be easily replaced, the primary goal of the FTL is wear-leveling: ensuring that all flash chips in the drive age, and expire, in unison. FTLs also speed up random writes by maintaining a free pool of extra, pre-erased blocks; however, sequential writes are still significantly faster. Additionally, the OS can *trim* logical addresses on an SSD, allowing the FTL to reclaim and reuse physical pages.

As a result of these properties, the best data structure for a single flash device is a log; it is always best to write sequentially to flash. One reason is performance; random writes are slower than sequential writes both on raw flash and SSDs (as explained above). Depending on the design of the FTL, random writes can also cause significantly greater wear-out than sequential writes. Accordingly, almost all single-machine filesystems or databases designed for flash storage implement a log-structured design within each device; for example, FAWN [5] organizes each of its individual drives as a log.

CORFU extends this theme by organizing an entire cluster of flash drives as a single log. This log is *distributed* across multiple drives and *shared* by multiple clients. We now explain the rationale for these two design decisions.

The case for a shared log: We stated earlier that a shared log is a powerful building block for distributed applications that require strong consistency. We now expand on this point by describing the different ways in which applications can use a fast, flash-based shared log. By 'shared', we mean that multiple clients can read and append to the log concurrently over the network, regardless of how the log is implemented.

Historically, shared log designs have appeared in a diverse array of systems. QuickSilver [13, 22] and Camelot [25] used shared logs for failure atomicity and node recovery. LBRM [14] uses shared logs for recovery from multicast packet loss. Shared logs are also used in distributed storage systems for consistent remote mirroring [16]. In such systems, CORFU can replace disk-based shared logs, providing higher throughput and lower latency.

However, a flash-based shared log also enables new applications that are infeasible on disk-based infrastructure. For instance, Hyder [7] is a recently proposed high-performance database designed around a flash-based shared log, where servers speculatively execute transactions by appending them to the shared log and then using log order to decide commit/abort status. In fact, Hyder was the original motivating application for CORFU and is currently being implemented over our code base. While the original Hyder paper included a brief design for a flash-based shared log, this was never implemented; later in this paper, we describe how – and why – CORFU departs significantly from the Hyder proposal.

Interestingly, a shared log can also be used as a consensus engine, providing functionality identical to consensus protocols such as Paxos (a fact hinted at by the name of our system). Used in this manner, CORFU provides a fast, fault-tolerant service for imposing and durably storing a total order on events in a distributed system. From this perspective, CORFU can be used as a drop-in replacement for disk-based Paxos implementations, leveraging flash storage to provide better performance.

The case for a distributed log: Existing flash-based storage systems scale capacity and throughput by partitioning data across multiple, independent logs, each of which resides on a single flash drive. In a partitioned system, a total order no longer exists on all updates, making it difficult to support operations such as consistent snapshots and atomic updates across partitions. Strongly consistent operations are usually limited in size and scope to a single partition. The high throughput/size ratio of flash results in smaller drives, exacerbating this problem. Even if all the data involved in an atomic update miraculously resides on the same fine-grained partition, the throughput of updates on that data is limited by the I/O capacity of the primary server of the partition.

Specific to flash storage, other problems arise when a system is partitioned into individual per-SSD logs. In particular, the age distribution of the drives in the cluster is tightly coupled to the observed workload; skewed workloads can age drives at different rates, resulting in unpredictable reliability and performance. For example, a range of key-value pairs can become slower and less reliable than the rest of the key-value store if one of them frequently overwritten. Additionally, striping data across SSDs of different ages can bottleneck performance at the oldest SSD in the stripe. Further, administrative policies may require all drives to be replaced together, in which case even wear-out is preferred. Conversely, specific patterns of uneven wear-out may be preferred, to allow the oldest subset of drives to be replaced periodically.

A distributed log solves these problems by spreading writes across the cluster in controlled fashion, decoupling the age distribution of drives from the observed workload; in effect, it implements distributed wear-leveling. In addition, the entire cluster still functions as a single log; as we show later, this makes it possible to implement strongly consistent operations at cluster scale.

Partitioning is ultimately necessary for achieving scale; we do not contend this point. However, modern systems choose to create very fine-grained partitions, at the level of a single drive or a single array attached to a server. CORFU instead allows an entire cluster to act as a single, coarse-grained partition.

3 Design and Implementation

The setting for CORFU is a data center with a large number of application servers (which we call clients) and a cluster of flash units (see Figure 1). Our goal is to provide applications running on the clients with a shared log abstraction implemented over the flash cluster.

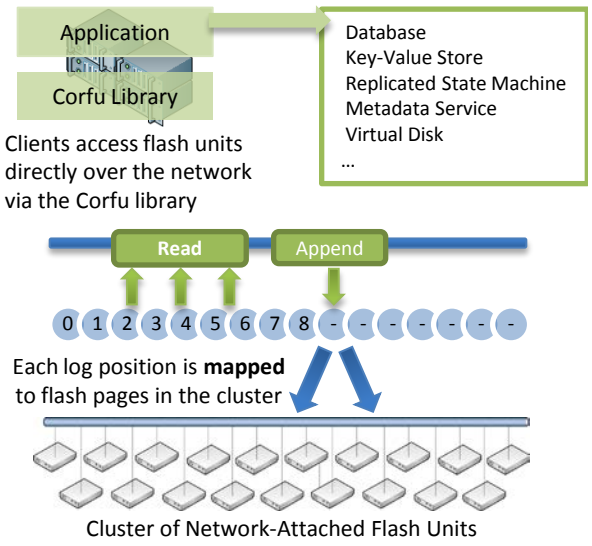

Figure 1: CORFU presents applications running on clients with the abstraction of a shared log, implemented over a cluster of flash units by a client-side library.

Our design for this shared log abstraction is driven by a single imperative: to keep flash units as simple, inexpensive and power-efficient as possible. We achieve this goal by placing all CORFU functionality at the clients and treating flash units as passive storage devices. CORFU clients read and write directly to the address space of each flash unit, coordinating with each other to ensure single-copy semantics for the shared log. Individual flash units do not initiate communication, are unaware of other flash units, and do not participate actively in replication protocols. CORFU does require specific functionality from flash units, which we discuss shortly.

Accordingly, CORFU is implemented as a client-side library that exposes a simple API to applications, shown in Figure 2. The *append* interface adds an entry to the log and returns its position. The *read* interface accepts a position in the log and returns the entry at that position. If no entry exists at that position, an error code is returned. The application can perform garbage collection using *trim*, which indicates to CORFU that no valid data exists at a specific log position. Lastly, the application can *fill* a position with junk, ensuring that it cannot be updated in future with a valid value.

CORFU's task of implementing a shared log abstraction with this API over a cluster of flash units – each of which exposes a separate address space – involves three functions:

- A **mapping function** from logical positions in the log to flash pages on the cluster of flash units.
- A **tail-finding mechanism** for finding the next available logical position on the log for new data.
- A **replication protocol** to write a log entry consistently on multiple flash pages.

$append(b)$	Append an entry b and return the log position ℓ it occupies
$read(\ell)$	Return entry at log position ℓ
$trim(\ell)$	Indicate that no valid data exists at log position ℓ
$fill(\ell)$	Fill log position ℓ with junk

Figure 2: API exposed by CORFU to applications

These three functions – combined with the ability of clients to read and write directly to the address space of each flash unit – are sufficient to support a shared log abstraction. To read data at a specific log position, the client-side library uses the mapping function to find the appropriate flash page, and then directly issues a read to the device where the flash page is located. To append data, a client finds the tail position of the log, maps it to a set of flash pages, and then initiates the replication protocol that issues writes to the appropriate devices.

Accordingly, the primary challenges in CORFU revolve around implementing these three functions in an efficient and fault-tolerant manner. Crucially, these functions have to provide single-copy semantics for the shared log even when flash units fail and clients crash.

In this section, we first describe the assumptions made by CORFU about each flash unit. We then describe CORFU's implementation of the three functions described above.

3.1 Flash Unit Requirements

The most basic requirement of a flash unit is that it support reads and writes on an address space of fixed-size pages. We use the term 'flash page' to refer to a page on this address space; however, the flash unit is free to expose a logical address space where logical pages are mapped internally to physical flash pages, as a conventional SSD does. The flash unit is expected to detect and re-map bad blocks in this address space.

To provide single-copy semantics for the shared log, CORFU requires 'write-once' semantics on the flash unit's address space. Reads on pages that have not yet been written should return an error code (*error_unwritten*). Writes on pages that have already been written should also return an error code (*error_overwritten*). In addition to reads and writes, flash units are also required to expose a trim command, allowing clients to indicate that the flash page is not in use anymore.

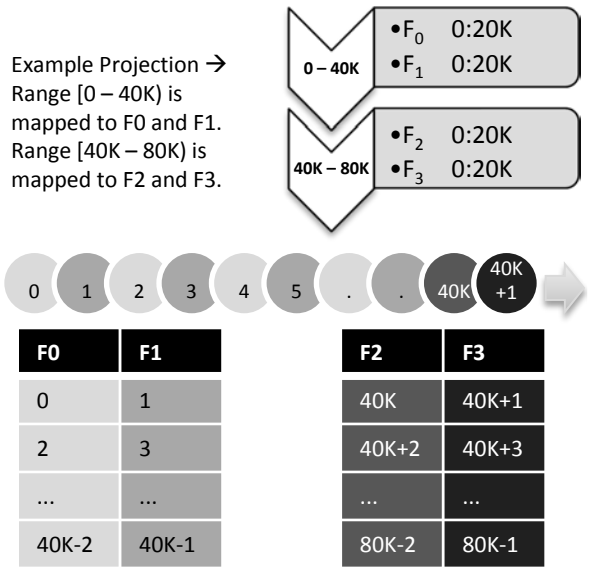

Figure 3: Example projection that maps different ranges of the shared log onto flash unit extents.

In addition, flash units are required to support a 'seal' command. Each incoming message to a flash unit is tagged with an epoch number. When a particular epoch number is sealed at a flash unit, it must reject all subsequent messages sent with an epoch equal or lower to the sealed epoch. In addition, the flash unit is expected to send back an acknowledgment for the seal command to the sealing entity, including the highest page offset that has been written on its address space thus far.

These requirements – write-once semantics and sealing – are sufficient to ensure CORFU correctness. They are also enough to ensure efficient appends and reads. However, for efficient garbage collection on general-purpose workloads (in terms of network/storage bandwidth and flash erase cycles), CORFU requires that the flash unit expose an infinite address space. We explain this last requirement in detail when we discuss garbage collection and the implementation of flash units.

3.2 Mapping in CORFU

Each CORFU client maintains a local, read-only replica of a data structure called a *projection* that carves the log into disjoint ranges. Each such range is mapped to a list of extents within the address spaces of individual flash units. Figure 3 shows an example projection, where range $[0, 40K)$ is mapped to extents on units F_0 and F_1, while $[40K, 80K)$ is mapped to extents on F_2 and F_3.

Within each range in the log, positions are mapped to flash pages in the corresponding list of extents via a simple, deterministic function. The default function used is round-robin: in the example in Figure 3, log position 0 is mapped to $F_0 : 0$, position 1 is mapped to $F_1 : 0$, position 2 back to $F_0 : 1$, and so on. Any function can be used as long as it is deterministic given a list of extents and a log position. The example above maps each log position to a single flash page; for replication, each extent is associated with a replica set of flash units rather than just one unit. For example, for two-way replication the extent $F_0 : 0 : 20K$ would be replaced by $F_0/F_0' : 0 : 20K$ and the extent $F_1 : 0 : 20K$ would be replaced by $F_1/F_1 : 0 : 20K$.

Accordingly, to map a log position to a set of flash pages, the client first consults its projection to determine the right list of extents for that position; in Figure 3, position $45K$ in the log maps to extents on units F_2 to F_3. It then computes the log position relative to the start of the range; in the example, this is $5K$. Using this relative log position, it applies the deterministic function on the list of extents to determine the flash pages to use. With the round-robin function and the example projection above, the resulting page would be $F_2 : 2500$.

By mapping log positions to flash pages, a projection essentially provides a logical address space implemented over a cluster of flash units. Clients can read or write to positions in this address space by using the projection to determine the flash pages to access. Since CORFU organizes this address space as a log (using a tail-finding mechanism which we describe shortly), clients end up writing only to the last range of positions in the projection ($[40K, 80K)$ in the example); we call this the *active range* in the projection.

3.2.1 Changing the mapping

In a sense, projections are similar to classical views. All operations on flash units – reads, writes, trims – are issued by clients within the context of a single projection. When some event occurs that necessitates a change in the mapping – for example, when a flash unit fails, or when the tail of the log moves past the current active range – a new projection has to be installed on all clients in the system. In effect, each client observes a totally ordered sequence of projections as the position-to-page mapping evolves over time; we call a projection's position in this sequence its epoch. When an operation executes in the context of a projection, all the messages it generates are tagged with the projection's epoch.

As with conventional view change protocols, all participants – in this case, the CORFU clients – must move consistently to a new projection when a change occurs. The new projection should correctly reflect all activity that was successfully completed in any previous projection; i.e., reads must reflect writes and trims that completed in older projections. Further, any activity in-flight

during the view change must be aborted and retried in the new projection.

To achieve these properties, CORFU uses a simple, auxiliary-driven *reconfiguration* protocol. The auxiliary is a durably stored sequence of projections in the system, where the position of the projection in the sequence is equivalent to its epoch. Clients can read the ith entry in the auxiliary (getting an error if no such entry exists), or write the ith entry (getting an error if an entry already exists at that position). The auxiliary can be implemented in multiple ways: on a conventional disk volume, as a Paxos state machine, or even as a CORFU instance with a static, never-changing projection.

Auxiliary-driven reconfiguration involves two distinct steps:

1. Sealing the current projection: When a client C_r decides to reconfigure the system from the current projection P_i to a new projection P_{i+1}, it first seals P_i; this involves sending a seal command to a subset of the flash units in P_i. A flash unit in P_i has to be sealed only if a log position mapped to one of its flash pages by P_i is no longer mapped to the same page by P_{i+1}. In practice, this means that only a small subset of flash units have to be sealed, depending on the reason for reconfiguration. Sealing ensures that flash units will reject in-flight messages – writes as well as reads – sent to them in the context of the sealed projection. When clients receive these rejections, they realize that the current projection has been sealed, and wait for a new projection to be installed; if this does not happen, they time out and initiate reconfiguration on their own. The reconfiguring client C_r receives back acknowledgements to the seal command from the flash units, which include the highest offsets written on those flash units thus far. Using this information, it can determine the highest log position reached in the sealed projection; this is useful in certain reconfiguration scenarios.

2. Writing the new projection at the auxiliary: Once the reconfiguring client C_r has successfully sealed the current projection P_i, it attempts to write the new projection P_{i+1} at the $(i+1)th$ position in the auxiliary. If some other client has already written to that position, client C_r aborts its own reconfiguration, reads the existing projection at position $(i+1)$ in the auxiliary, and uses it as its new current projection. As a result, multiple clients can initiate reconfiguration simultaneously, but only one of them succeeds in proposing the new projection.

Projections – and the ability to move consistently between them – offer a versatile mechanism for CORFU to deal with dynamism. Figure 4 shows an example sequence of projections. In Figure 4(A), range $[0, 40K)$ in the log is mapped to the two flash unit mirrored pairs F_0/F_1 and F_2/F_3, while range $[40K, 80K)$ is mapped

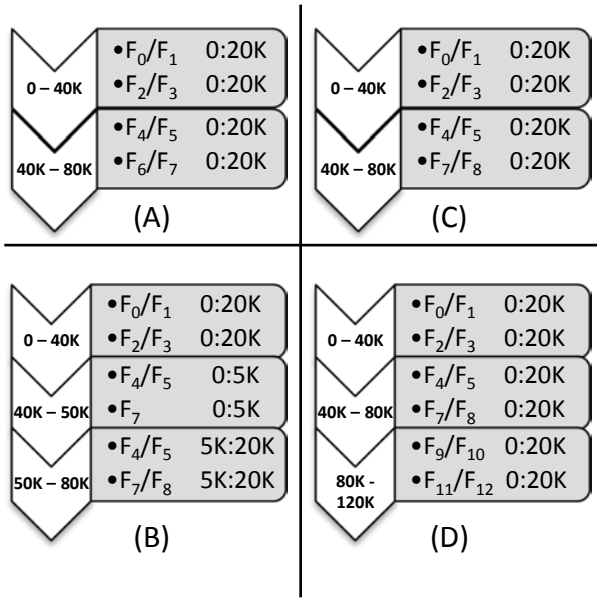

Figure 4: Sequence of projections: When F_6 fails in (A) with the log tail at 50K, clients move to (B) in order to replace F_6 with F_8 for new appends. Once old data on F_6 is rebuilt, F_8 is used in (C) for reads on old data as well. When the log tail goes past $80K$, clients add capacity by moving to (D).

to F_4/F_5 and F_6/F_7. When F_6 fails with the log tail at position $50K$, CORFU moves to projection (B) immediately, replacing F_6 with F_8 for new appends beyond the current tail of the log, while servicing reads in the log range $[40K, 50K)$ with the remaining mirror F_7.

Once F_6 is completely rebuilt on F_8 (by copying entries from F_7), the system moves to projection (C), where F_8 is now used to service all reads in the range $[40K, 80K)$. Eventually, the log tail moves past $80K$, and the system again reconfigures, adding a new range in projection (D) to service reads and writes past $80K$.

3.3 Finding the tail in CORFU

Thus far, we described the machinery used by CORFU to map log positions to sets of flash pages. This allows clients to read or write any position in a logical address space. To treat this address space as an appendable log, clients must be able to find the tail of the log and write to it.

One possible solution is to allow clients to contend for positions. In this case, when a CORFU instance is started, every client that wishes to append data will try to concurrently write to position 0. One client will win, while the rest fail; these clients then try again on position 1, and so on. This approach provides log semantics if only one write is allowed to 'win' on each position;

i.e., complete successfully with the guarantee that any subsequent read on the position returns the value written, until the position is trimmed. We call this property safety-under-contention.

In the absence of replication, this property is satisfied trivially by the flash unit's write-once semantics. When each position is replicated on multiple flash pages, it is still possible to provide this property; in fact, the replication protocol used in CORFU (that we describe next) does so. However, it is clear that such an approach will result in poor performance when there are hundreds of clients concurrently attempting appends to the log.

To eliminate such contention at the tail of the log, CORFU uses a dedicated sequencer that assigns clients 'tokens', corresponding to empty log positions. The sequencer can be thought of as a simple networked counter. To append data, a client first goes to the sequencer, which returns its current value and increments itself. The client has now reserved a position in the log and can write to it without contention from other clients.

Importantly, the sequencer does not represent a single point of failure; it is merely an optimization to reduce contention in the system and is not required for either safety or progress. For fast recovery from sequencer failure, we store the identity of the current sequencer in the projection and use reconfiguration to change sequencers. The counter of the new sequencer is determined using the highest page written on each flash unit, which is returned by the flash unit in response to the seal command during reconfiguration.

However, CORFU's sequencer-based approach does introduce a new failure mode, since 'holes' can appear in the log when clients obtain tokens and then fail to use them immediately due to crashes or slowdowns. Holes can cripple applications that consume the log in strict order, such as state machine replication or transaction processing, since no progress can be made until the status of the hole is resolved. Given that a large system is likely to have a few malfunctioning clients at any given time, holes can severely disrupt application performance. A simple solution is to have other clients fill holes aggressively with a reserved 'junk' value. To prevent aggressive hole-filling from burning up flash cycles and network bandwidth, flash units can be junk-aware, simply updating internal meta-data to mark an address as filled with junk instead of writing an actual value to the flash.

Note that filling holes reintroduces contention for log positions: if a client is merely late in using a reserved token and has not really crashed, it could end up competing with another client trying to fill the position with junk, which is equivalent to two clients concurrently writing to the same position. In other words, the sequencer is an optimization that removes contention for log positions in the common case, but does not eliminate it entirely.

3.4 Replication in CORFU

Once a client reserves a new position in the log via the sequencer, it maps this position to a replica set of flash pages in the cluster using the current projection. At this point, it has to write data at these flash pages over the network. The protocol used to write to the set of flash pages has to provide two properties. First, it has to provide the safety-under-contention property described earlier: when multiple clients write to the replica set for a log position, reading clients should observe a single value. Second, it has to provide durability: written data must be visible to reads only after it has sufficient fault tolerance (i.e., reached $f + 1$ replicas). Given the relatively high cost of flash, we require a solution that tolerates f failures with just $f + 1$ replicas; as a result, data must be visible to reads only after it reaches all replicas.

One approach is to have the client write in parallel to the flash units in the set, and wait for all of them to respond before acknowledging the completion of the append to the application. Unfortunately, when appending clients contend for a log position, different values can be written on different replicas, making it difficult to satisfy the safety-under-contention property. Also, satisfying the durability property with parallel writes requires the reading client to access all replicas to determine if a write has completed or not.

Instead, CORFU uses a simple chaining protocol (essentially, a client-driven variant of Chain Replication [28]) to achieve the safety-under-contention and durability properties. When a client wants to write to a replica set of flash pages, it updates them in a deterministic order, waiting for each flash unit to respond before moving to the next one. The write is successfully completed when the last flash unit in the chain is updated. As a result, if two clients attempt to concurrently update the same replica set of flash pages, one of them will arrive second at the first unit of the chain and receive an *error_overwrite*. This ensures safety-under-contention.

To read from the replica set, clients have two options. If they are unaware of whether the log position was successfully written to or not (for example, when replaying the log after a power failure), they are required to go to the last unit of the chain. If the last unit has not yet been updated, it will return an *error_unwritten*. This ensures the durability property. Alternatively, if the reading client knows already that the log position has been successfully written to (for example, via an out-of-band notification by the writing client), it can go to any replica in the chain for better read performance.

By efficiently handling contention, chained appends allow a client to rapidly fill holes left in the log by other clients that crashed midway through an append. To fill holes, the client starts by checking the first unit of the

chain to determine if a valid value exists in the prefix of the chain. If such a value exists, the client walks down the chain to find the first unwritten replica, and then 'completes' the append by copying over the value to the remaining unwritten replicas in chain order. Alternatively, if the first unit of the chain is unwritten, the client writes the junk value to all the replicas in chain order. CORFU exposes this fast hole filling functionality to applications via a *fill* interface. Applications can use this primitive as aggressively as required, depending on their sensitivity to holes in the shared log.

Reconfiguration and Replication: How does this replication protocol interact with the reconfiguration mechanism described in Section 3.2.1? We first define a *chain property*: the replica chain for a log position in a projection has a written prefix storing a single value and an unwritten suffix. A completed write corresponds to a chain with a full-length prefix and a zero-length suffix.

When the system reconfigures from one projection to another, the replica chain for a position can change from one ordered set of flash pages to another. Trivially, the new replica chain is required to satisfy the chain property. Further, we impose two conditions on the transition: if the old chain had a prefix of non-zero length, the new chain must have one as well with the same value. If the old chain had a zero-length suffix, the new chain must have one too. Lastly, to prevent split-brain scenarios, we require that at least one flash unit in the old replica set be sealed in the old projection's epoch.

To meet these requirements, our current implementation follows the protocol described earlier in Figure 4 for flash unit failures. The system first reconfigures to a new chain without the failed replica. It then prepares a new replica by copying over the completed value on each position, filling any holes it encounters in the process. Once the new replica is ready, the system reconfigures again to add it to the end of the replica chain.

Relationship to Paxos: Each chain of flash units in CORFU can be viewed as a single consensus engine, providing (as Paxos does) an ordered sequence of state machine replication (SMR) commands. In conventional SMR implemented using Paxos, the throughput of the system is typically scaled by partitioning the system across multiple instances, effectively splitting the single stream of totally ordered commands into many unrelated streams. Conversely, CORFU scales SMR throughput by **partitioning over time, not space**; the consensus decision on each successive log entry is handled by a different replica chain. Stitching these multiple command streams together is the job of the projection, which is determined by a separate, auxiliary-driven consensus engine, as described earlier. In a separate report, we examine the foundational differences between Paxos and CORFU in more detail [19].

3.5 Garbage Collection

CORFU provides the abstraction of an infinitely growing log to applications. The application does not have to move data around in the address space of the log to free up space. All it is required to do is use the *trim* interface to inform CORFU when individual log positions are no longer in use. As a result, CORFU makes it easy for developers to build applications over the shared log without worrying about garbage collection strategies. An implication of this approach is that as the application appends to the log and trims positions selectively, the address space of the log can become increasingly sparse.

Accordingly, CORFU has to efficiently support a sparse address space for the shared log. The solution is a two-level mapping. As described before, CORFU uses projections to map from a single infinite address space to individual extents on each flash unit. Each flash-unit then maps a sparse 64-bit address space, broken into extents, onto the physical set of pages. The flash unit has to maintain a hash-map from 64-bit addresses to the physical address space of the flash. In fact, this is a relatively minor departure from the functionality implemented by modern SSDs, which often employ page-level maps from a fixed-size logical address space to a slightly larger physical address space.

Crucially, a projection is a range-to-range mapping, and hence it is quite concise. Despite this, it is possible that an adversarial workload can result in bloated projections; for instance, if each range in the projection has a single valid entry that is never trimmed, the mapping for that range has to be retained in the projection for perpetuity.

Even for such adversarial workloads, it is easy to bound the size of the projection tightly by introducing a small amount of proactive data movement across flash units in order to merge consecutive ranges in the projection. For instance, we estimate that adding 0.1% writes to the system can keep the projection under 25 MB on a 1 TB cluster for an adversarial workload. In practice, we do not expect projections to exceed 10s of KBs for conventional workloads; this is borne out by our experience building applications over CORFU. In any case, handling multi-MB projections is not difficult, since they are static data structures that can be indexed efficiently. Additionally, new projections can be written as deltas to the auxiliary.

3.6 Flash Unit Implementations

Flash units can be viewed as conventional SSDs with network interfaces, supporting reads, writes and trims on an address space. Recall that flash units have to meet three major requirements: write-once semantics, a seal capa-

bility, and an infinite address space.

Of these, the seal capability is simple: the flash unit maintains an epoch number *cur_sealed_epoch* and rejects all messages tagged with equal or lower epochs, sending back an *error_badepoch* error code. When a seal command arrives with a new epoch to seal, the flash unit first flushes all ongoing operations and then updates its *cur_sealed_epoch*. It then responds to the seal command with *cur_highest_offset*, the highest address written in the address space it exposes; this is required for reconfiguration, as explained previously.

The infinite address space is essentially just a hash-map from 64-bit virtual addresses to the physical address space of the flash. With respect to write-once semantics, an address is considered unwritten if it does not exist in the hash-map. When an address is written to the first time, an entry is created in the hash-map pointing the virtual address to a valid physical address. Each hash-map entry also has a bit indicating whether the address has been trimmed or not, which is set by the *trim* command.

Accordingly, a read succeeds if the address exists in the hash-map and does not have the trim bit set. It returns *error_trimmed* if the trim bit is set, and *error_unwritten* if the address does not exist in the hash-map. A write to an address that exists in the hash-map returns *error_trimmed* if the trim bit is set and *error_overwritten* if it is not. If the address is not in the hash-map, the write succeeds.

To eventually remove trimmed addresses from the hash-map, the flash unit also maintains a watermark before which no unwritten addresses exist, and removes trimmed addresses from the hash-map that are lower than the watermark. Reads and writes to addresses before the watermark that are not in the hash-map return *error-trimmed* immediately. If the address is in the hash-map, reads succeed while writes return *error_overwritten*.

The flash unit also efficiently supports fill operations that write junk by treating such writes differently from first-class writes. Junk writes are initially treated as conventional writes, either succeeding or returning *error_overwritten* or *error_trimmed* as appropriate. However, instead of writing to the flash or SSD, the flash unit points the hash-map entry to a special address reserved for junk; this ensures that flash cycles and capacity is not wasted. Also, once the hash-map is updated, the entry is immediately trimmed in the scope of the same operation. This removes the need for clients to explicitly track and trim junk in the log.

Currently, we have built two flash unit instantiations: *Server+SSD*: this consists of conventional SATA SSDs attached to servers with network interfaces. The server accepts CORFU commands over the network from clients and implements the functionality described above, issuing reads and writes to the fixed-size address space of the SSD as required.

FPGA+SSD: an FPGA with a SATA SSD attached to it, prototyped using the Beehive [27] architecture on the BEE3 hardware platform [10]. The FPGA includes a network interface to talk to clients and SATA ports to connect to the SSD. A variant under development runs over raw flash chips, implementing FTL functionality on the FPGA itself.

4 CORFU Applications

We are currently prototyping several applications over CORFU. Two such applications that we have implemented are CORFU-Store, a key-value store, and CORFU-SMR, an implementation of State Machine Replication [23].

CORFU-Store: This is a key-value store that supports a number of properties that are difficult to achieve on partitioned stores, including atomic multi-key puts and gets, distributed snapshots, geo-distribution and distributed rollback/replay. In CORFU-Store, a map-service (which can be replicated) maintains a mapping from keys to shared log offsets. To atomically put multiple keys, clients must first append the key-value pairs to the CORFU log, append a commit record and then send the commit record to the map-service. To atomically get multiple keys, clients must query the map-service for the latest key-offset mappings, and then perform CORFU reads on the returned offsets. This protocol ensures linearizability for single-key puts and gets, as well as atomicity for multi-key puts and gets.

CORFU-Store's shared log design makes it easy to take consistent point-in-time snapshots across the entire key space, simply by playing the shared log up to some position. The key-value store can be geo-distributed asynchronously by copying the log, ensuring that the mirror is always at some prior snapshot of the system, with bounded lag. Additionally, CORFU-Store ensures even wear-out across the cluster despite highly skewed write workloads.

CORFU-SMR: Earlier, we noted that CORFU provides the same functionality as Paxos. Accordingly, CORFU is ideal for implementing replicated state machines. Each SMR server simply plays the log forward to receive the next command to execute. It proposes new commands into the state machine by appending them to the log. The ability to do fast queries on the body of the log means that the log entries can also include the data being executed over, obviating the need for each SMR server to store state persistently.

In our current implementation, each SMR server plays the log forward by issuing reads to the log. This approach can stress the shared log: with N SMR servers running at T commands/sec, the CORFU log will see

$N * T$ reads/sec. However, we retained this design since we did not bottleneck at the shared log in our experiments; instead, we were limited by the ability of each SMR server to receive and process commands. In the future, we expect to explore different approaches to playing the log forward, perhaps by having dedicated machines multicast the contents of the log out to all SMR servers.

Other Applications: As mentioned previously, we are implementing the Hyder database [7] over CORFU. One application in the works is a shared block device that exposes a conventional fixed-size address space; this can then be used to expose multiple, independent virtual disks to client VMs [20]. Another application is reliable multicast, where senders append to a single, channel-specific log before multicasting. A receiver can detect lost packets by observing gaps in the sequence of log positions, and retrieve them from the log.

5 Evaluation

We evaluate CORFU on a cluster of 32 Intel X25V drives. Our experiment setup consists of two racks; each rack contains 8 servers (with 2 drives attached to each server) and 11 clients. Each machine has a 1 Gbps link. Together, the two drives on a server provide around 40,000 4KB read IOPS; accessed over the network, each server bottlenecks on the Gigabit link and gives us around 30,000 4KB read IOPS. Each server runs two processes, one per SSD, which act as individual flash units in the distributed system. Currently, the top-of-rack switches of the two racks are connected to a central 10 Gbps switch; our experiments do not generate more than 8 Gbps of inter-rack traffic. We run two client processes on each of the client machines, for a total of 44 client processes.

In all our experiments, we run CORFU with two-way replication, where appends are mirrored on drives in either rack. Reads go from the client to the replica in the local rack. Accordingly, the total read throughput possible on our hardware is equal to 2 GB/sec (16 servers X 1 Gbps each) or 500K/sec 4KB reads. Append throughput is half that number, since appends are mirrored.

Unless otherwise mentioned, our throughput numbers are obtained by running all 44 client processes against the entire cluster of 32 drives. We measure throughput at the clients over a 60-second period during each run.

In addition to this primary deployment of server-attached SSDs, we also provide some results over the prototype FPGA+SSD flash unit. In this case, the FPGA has two SSDs attached to it and emulates a pair of flash units, and a single CORFU client accesses it over the network. The FPGA runs at 1 Gbps or roughly 30K 4KB reads/sec. When running at full speed, it consumes around 15W; in contrast, one of our servers consumes 250W. We also experimented with low-power Atom-based servers, but found them incapable of serving SSD reads over the network at 1 Gbps.

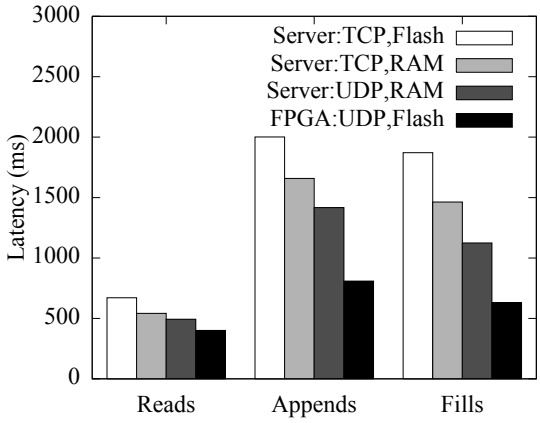

Figure 5: Latency for CORFU operations on different flash unit configurations.

5.1 End-to-end Latency

We first summarize the end-to-end latency characteristics of CORFU in Figure 5. We show the latency for *read*, *append* and *fill* operations issued by clients for four CORFU configurations. The left-most bar for each operation type (Server:TCP,Flash) shows the latency of the server-attached flash unit where clients access the flash unit over TCP/IP when data is durably stored on the SSD; this represents the configuration of our 32-drive deployment. To illustrate the impact of flash latencies on this number, we then show (Server:TCP,RAM), in which the flash unit reads and writes to RAM instead of the SSD. Third, (Server:UDP,RAM) presents the impact of the network stack by replacing TCP with UDP between clients and the flash unit. Lastly, (FPGA:UDP,Flash) shows end-to-end latency for the FPGA+SSD flash unit, with the clients communicating with the unit over UDP.

Against these four configurations we evaluate the latency of three operation types. Reads from the client involve a simple request over the network to the flash unit. Appends involve a token acquisition from the sequencer, and then a chained append over two flash unit replicas. Fills involve an initial read on the head of the chain to check for incomplete appends, and then a chained append to two flash unit replicas.

In this context, Figure 5 makes a number of important points. First, the latency of the FPGA unit is very low for all three operations, providing sub-millisecond appends and fills while satisfying reads within half a millisecond. This justifies our emphasis on a client-centric

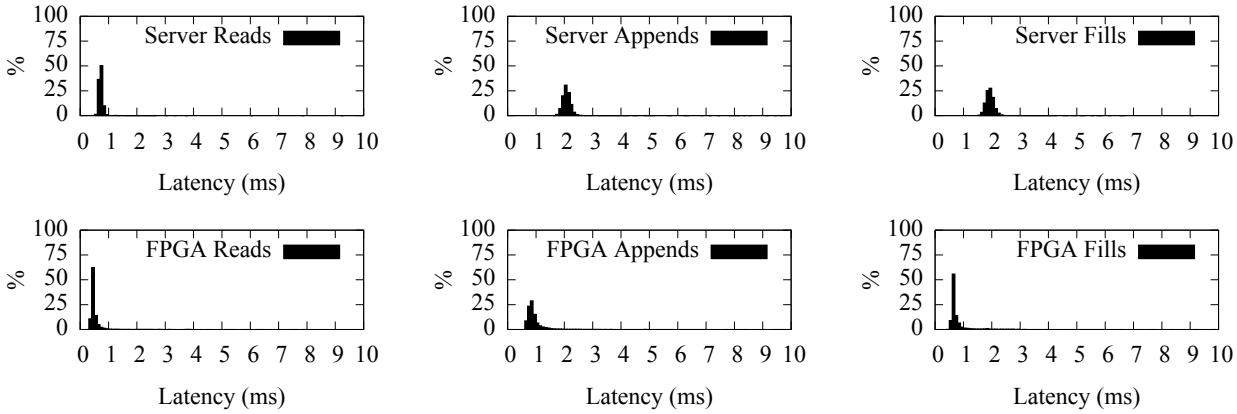

Figure 6: Latency distributions for CORFU operations on 4KB entries.

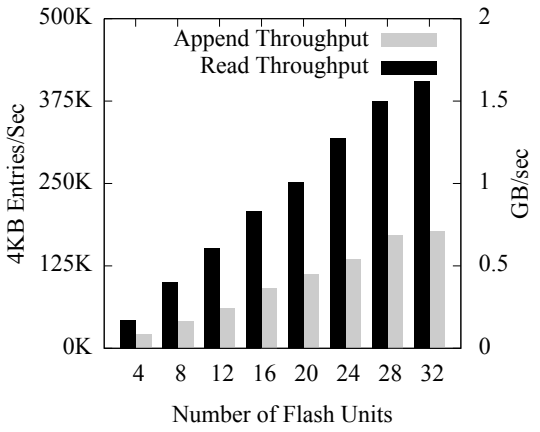

Figure 7: Throughput for random reads and appends.

design; eliminating the server from the critical path appears to have a large impact on latency. Second, the latency to fill a hole in the log is very low; on the FPGA unit, fills complete within 650 microseconds. CORFU's ability to fill holes rapidly is key to realizing the benefits of a client-centric design, since hole-inducing client crashes can be very frequent in large-scale systems. In addition, the chained replication scheme that allows fast fills in CORFU does not impact append latency drastically; on the FPGA unit, appends complete within 750 microseconds.

5.2 Throughput

We now focus on throughput scalability; these experiments are run on our 32-drive cluster of server-attached SSDs. To avoid burning out the SSDs in the throughput experiments, we emulate writes to the SSDs; however, all reads are served from the SSDs, which have 'burnt-in' address spaces that have been written to completely. Emulating SSD writes also allows us to test the CORFU design at speeds exceeding the write bandwidth of our commodity SSDs.

Figure 7 shows how log throughput for 4KB appends and reads scales with the number of drives in the system. As we add drives to the system, both append and read throughput scale up proportionally. Ultimately, append throughput hits a bottleneck at around 180K appends/sec; this is the maximum speed of our sequencer implementation, which is a user-space application running over TCP/IP. Since we were near the append limit of our hardware, we did not further optimize our sequencer implementation.

At such high append throughputs, CORFU can wear out 1 TB of MLC flash in around 4 months. We believe that replacing a $3K cluster of flash drives every four months is acceptable in settings that require strong consistency at high speeds, especially since we see CORFU as a critical component of larger systems (as a consensus engine or a metadata service, for example).

5.3 Reconfiguration

Reconfiguration is used extensively in CORFU, to replace failed drives, to add capacity to the system, and to add, remove, or relocate replicas. This makes reconfiguration latency a crucial metric for our system.

Recall that reconfiguration latency has two components: sealing the current configuration, which contacts a subset of the cluster, and writing the new configuration to the auxiliary. In our experiments, we conservatively seal all drives, to provide an upper bound on reconfiguration time; in practice, only a subset needs to be sealed. Our auxiliary is implemented as a networked file share.

Figure 8 (Left) shows throughput behavior at an appending and reading client when a flash unit fails. When

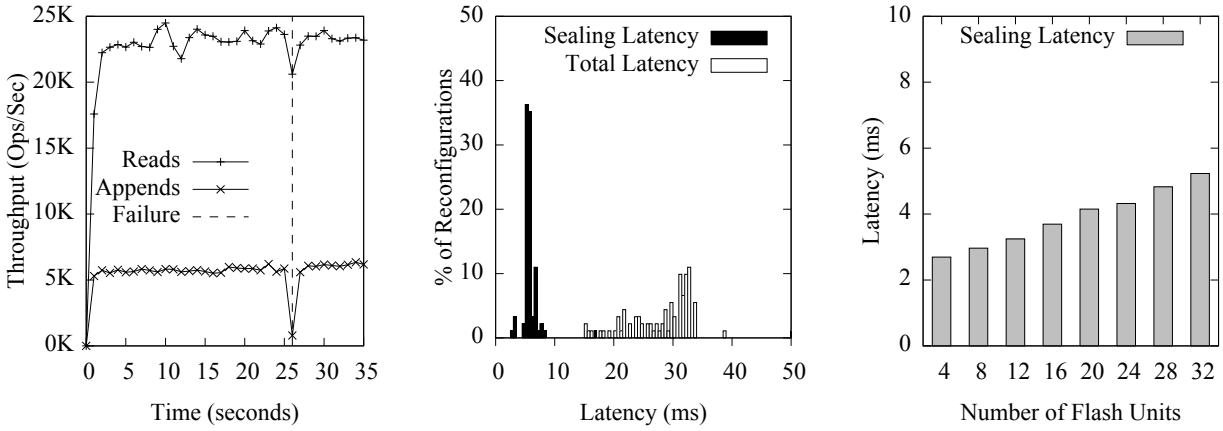

Figure 8: Reconfiguration performance on 32-drive cluster. Left: Appending client waits on failed drive for 1 second before reconfiguring, while reading client continues to read from alive replica. Middle: Distribution of sealing and total reconfiguration latency for 32 drives. Right: Scalability of sealing with number of drives.

the error occurs 25 seconds into the experiment, the appending client's throughput flat-lines and it waits for a 1-second timeout period before reconfiguring to a projection that does not have the failed unit. The reading client, on the other hand, continues reading from the other replica; when reconfiguration occurs, it receives an error from the replica indicating that it has a stale, sealed projection; it then experiences a minor blip in throughput as it retrieves the latest projection from the auxiliary. The graph for sequencer failure looks identical to this graph.

Figure 8 (Middle) shows the distribution of the latency between the start of reconfiguration and its successful completion on a 32-drive cluster. The median latency for the sealing step is around 5 ms, while writing to the auxiliary takes 25 ms more. The slow auxiliary write is due to overheads in serializing the projection as human-readable XML and writing it to the network share; implementing the auxiliary as a static CORFU instance and writing binary data would give us sub-millisecond auxiliary writes (but make the system less easy to administer).

Figure 8 (Right) shows how the median sealing latency scales with the number of drives in the system. Sealing involves the client contacting all flash units in parallel and waiting for responses. The auxiliary write step in reconfiguration is insensitive to system size.

5.4 Applications

We now demonstrate the performance of CORFU-Store for atomic multi-key operations. Figure 9 (Left) shows the performance of multi-put operations in CORFU-Store. On the x-axis, we vary the number of keys updated atomically in each multi-put operation. The bars in the graph plot the number of multi-puts executed per second. The line plots the total number of CORFU log appends resulting as a result of the multi-put operations; a multi-put involving k keys generates $k+1$ log appends, one for each updated key and a final append for the commit record. For small multi-puts involving one or two keys, we are bottlenecked by the ability of the CORFU-Store map-service to handle and process incoming commit records; our current implementation bottlenecks at around 50K single-key commit records per second. As we add more keys to each multi-put, the number of log appends generated increases and the CORFU log bottlenecks at around 180K appends/sec. Beyond 4 keys per multi-put, the log bottleneck determines the number of multi-puts we push through; for example, we obtain $\frac{180K}{6}$=30K multi-puts involving 5 keys.

Figure 9 (Right) shows performance for atomic multi-get operations. As we increase the number of keys accessed by each multi-get, the overall log throughput stays constant while multi-get throughput decreases. For 4 keys per multi-get, we get a little over 100K multi-gets per second, resulting in a load of over 400K reads/s on the CORFU log.

Figure 10 shows the throughput of CORFU-SMR, the state machine replication library implemented over CORFU. In this application, each client acts as a state machine replica, proposing new commands by appending them to the log and executing commands by playing the log. Each command consists of a 512-byte payload and 64 bytes of metadata; accordingly, 7 commands can fit into a single CORFU log entry with batching. In the experiment, each client generates 7K commands/sec; as we add clients on the x-axis, the total rate of commands injected into the system increases by 7K. On the y-axis, we plot the average rate (with error bars signifying the

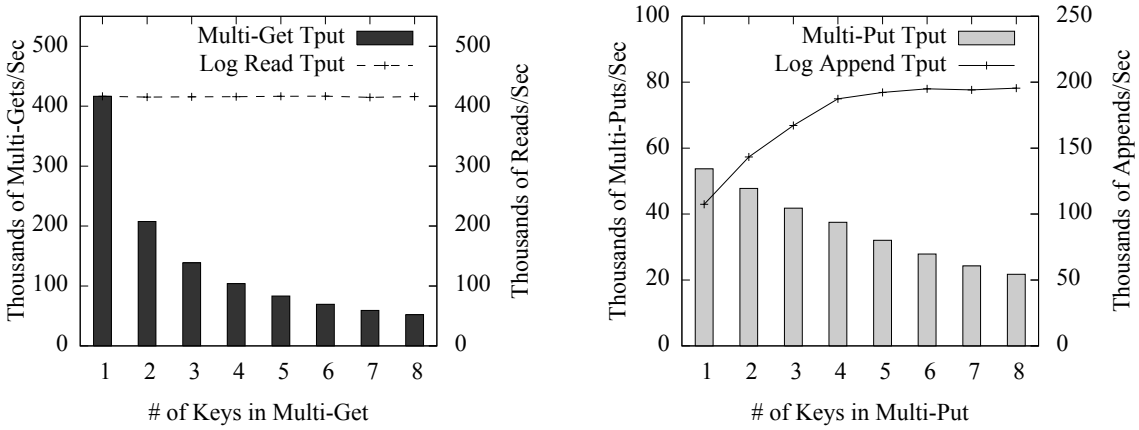

Figure 9: Example CORFU Application: CORFU-Store supports atomic multi-gets and multi-puts at cluster scale.

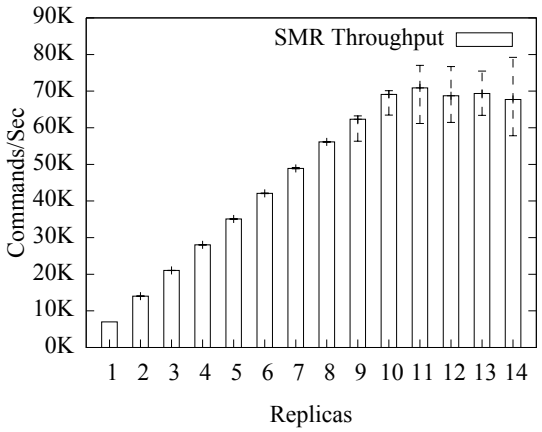

Figure 10: Example CORFU Application: CORFU-SMR supports high-speed state machine replication.

min and max across replicas) at which commands are executed by each replica. While the system has 10 or less CORFU-SMR replicas, the rate at which commands are executed in the system matches the rate at which they are injected. Beyond 10 replicas, we overload the ability of a client to execute incoming commands, and hence throughput flat-lines; this means that clients are now lagging behind while playing the state machine.

6 Related Work

Flash in the Data Center: CORFU's architecture borrows heavily from the FAWN system [5], which first argued for clusters of low-power flash units. CORFU's network-attached flash units are a natural extension of the FAWN argument that low-speed CPUs are more power-efficient; our FPGA implementation consisted of a ring of 100 MHz cores. While FAWN showed that such clusters could efficiently support parallelizable workloads, our contribution lies in extending the benefits of such an architecture to strongly consistent applications.

In contrast to the FAWN and CORFU architecture are PCI-e drives such as Fusion-io [2], which offer up to 10 Gbps of random I/O. Accessing such drives from remote clients requires a high-end, expensive storage server to act as a conduit between the network and the drive. The server is a single point of failure, acts as a bottleneck for reads and writes, is a power hog, costs as much as the drive, and is not incrementally scalable. Storage appliances based on PCI-e drives [3, 4] can solve the first two issues via replication but not the others. CORFU offers similar performance for an order of magnitude less power consumption and less than half the cost (our 32-drive X25-V cluster cost $3K), while offering all the benefits of a distributed solution.

Replicated Data Stores: A traditional design partitions a virtual block drive or a file system onto a collection of independent replica sets. Internally, each object (a file or a block) is replicated using primary-backup mirroring (e.g., [18]) or quorum replication (e.g., [11]). Mapping each object onto a replica set is done by a centralized configuration manager (usually implemented as a replicated state machine). As we explained earlier, CORFU partitions data across time rather than space. In addition, CORFU can act as the configuration manager in such systems, adding a fast source of consistency that other applications can use for resource and lock management. This vital role is is currently played by systems like Chubby [9] and ZooKeeper [15]. Another type of effort related to SMR concentrates on high throughput by allowing reads to proceed from any replica [8, 26]. CORFU faces a similar challenge of learning which appends have committed in order to read from any replica.

Log-structured Filesystems: All contemporary designs for flash storage borrow heavily from a long line of

log-structured filesystems starting with LFS [21]. Modern SSDs are testament to the fact that flash is ideally suited for logging designs, eliminating the traditional problem with such systems of garbage collection interfering with foreground activity [24]. Zebra [12] and xFS [6] extended LFS to a distributed setting, striping logs across a collection of storage servers. Like these systems, CORFU stripes updates across drives in a log-structured manner. Unlike them, it implements a single shared log that can be concurrently accessed by multiple clients while providing single-copy semantics.

Comparison with Hyder: As mentioned, Hyder was our original motivating application, and is currently being implemented over CORFU. The Hyder paper [7] included a design outline of a flash-based shared log (we call this Hyder-Log) that was simulated but not implemented. In fact, the design requirements for CORFU emerged from discussions with the Hyder authors on adding fault-tolerance and scalability to their proposal.

CORFU differs from the Hyder-Log design in multiple ways. First, CORFU's use of an off-path sequencer ensures that no single flash unit has to process all append I/O in the system; in contrast, Hyder-Log funnels all appends through a primary unit, and accordingly append throughput is bounded by the speed of a single unit. Second, Hyder-Log parallelizes appends by striping individual entries across units; this forces applications to use large entry sizes that are multiples of 4KB (typical flash page granularity). To achieve parallelism without imposing large entry sizes, CORFU uses projections to map log positions to units in round-robin fashion.

Finally, our experience with CORFU identified holes in the log as the most common mode of failure and source of delays; accordingly, we focused on providing a sub-millisecond hole-filling primitive that ensures high, stable log throughput despite client crashes. Hyder-Log suffers from holes as well; it resorts to a heavy recovery protocol that involves sealing the system and forcing clients to move to a new view every time a hole has to be filled. As a result of these differences, CORFU departs extensively in its design from the Hyder-Log proposal.

7 Conclusion

New storage designs are required to unlock the full potential of flash storage. In this paper, we presented the CORFU system, which organizes a cluster of flash drives as a single, shared log. CORFU offers single-copy semantics at cluster-scale speeds, providing a scalable source of atomicity and durability for distributed systems. CORFU's novel client-centric design eliminates storage servers in favor of simple, efficient and inexpensive flash chips that attach directly to the network.

References

[1] Intel x25-v datasheet. `http://www.intel.com/design/flash/nand/value/technicaldocuments.htm`.

[2] Fusion-io. `http://www.fusionio.com/`, 2011.

[3] Texas memory systems. http://www.ramsan.com/, 2011.

[4] Violin memory. http://www.violin-memory.com/, 2011.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP 2009*.

[6] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 109–126. ACM, 1995.

[7] P. Bernstein, C. Reid, and S. Das. Hyder – A Transactional Record Manager for Shared Flash. In *CIDR 2011*, pages 9–20, 2011.

[8] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos Replicated State Machines as the Basis of a High-performance Data Store. In *NSDI 2011*.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI 2006*.

[10] J. Davis, C. P. Thacker, and C. Chang. BEE3: Revitalizing computer architecture research. In *MSR-TR-2009-45*.

[11] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Enterprise Storage Systems on a Shoestring. In *HotOS 2003*.

[12] J. Hartman and J. Ousterhout. The zebra striped network file system. *ACM TOCS*, 13(3):274–310, 1995.

[13] R. Haskin, Y. Malachi, and G. Chan. Recovery management in QuickSilver. *ACM TOCS*, 6(1):82–108, 1988.

[14] H. Holbrook, S. Singhal, and D. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *ACM SIGCOMM CCR*, 25(4):328–341, 1995.

[15] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, pages 11–11. USENIX Association, 2010.

[16] M. Ji, A. Veitch, J. Wilkes, et al. Seneca: remote mirroring done write. In *USENIX 2003 Annual Technical Conference*, 2003.

[17] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *PODC 2009*, pages 312–313. ACM, 2009.

[18] E. Lee and C. Thekkath. Petal: Distributed virtual disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, 1996.

[19] D. Malkhi, M. Balakrishnan, J. D. Davis, V. Prabhakaran, and T. Wobber. From Paxos to CORFU: A Flash-Speed Shared Log. *ACM SIGOPS Operating Systems Review*, 46(1):47–51, 2012.

[20] D. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. Feeley, N. Hutchinson, and A. Warfield. Parallax: Virtual Disks for Virtual Machines. In *Eurosys 2008*.

[21] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM TOCS*, 10(1), Feb. 1992.

[22] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *ACM SIGOPS OSR*, volume 25. ACM, 1991.

[23] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[24] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX ATC 1995*.

[25] A. Spector, R. Pausch, and G. Bruell. Camelot: A flexible, distributed transaction processing system. In *Compcon Spring'88*.

[26] J. Terrace and M. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Usenix ATC 2009*.

[27] C. P. Thacker. Beehive: A many-core computer for FPGAs. Unpublished Manuscript.

[28] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI 2004*.