CORFU: A Distributed Shared Log

MAHESH BALAKRISHNAN, DAHLIA MALKHI, JOHN D. DAVIS,
and VIJAYAN PRABHAKARAN, Microsoft Research Silicon Valley
MICHAEL WEI, University of California, San Diego
TED WOBBER, Microsoft Research Silicon Valley

CORFU is a global log which clients can append-to and read-from over a network. Internally, CORFU is distributed over a cluster of machines in such a way that there is no single I/O bottleneck to either appends or reads. Data is fully replicated for fault tolerance, and a modest cluster of about 16–32 machines with SSD drives can sustain 1 million 4-KByte operations per second.

The CORFU log enabled the construction of a variety of distributed applications that require strong consistency at high speeds, such as databases, transactional key-value stores, replicated state machines, and metadata services.

Categories and Subject Descriptors: H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Design, Reliability, Algorithms

ACM Reference Format:
Balakrishnan, M., Malkhi, D., Davis, J. D., Prabhakaran, V., Wei, M., and Wobber, T. 2013. CORFU: A distributed shared log. ACM Trans. Comput. Syst. 31, 4, Article 10 (December 2013), 24 pages.
DOI: http://dx.doi.org/10.1145/2535930

1. INTRODUCTION

Traditionally, system designers have been forced to believe that the only way to scale up reliable data stores, whether on-premise or cloud-hosted, is to shard the database. In this manner, recent systems like Percolator [Peng and Dabek 2010]; Megastore [Baker et al. 2011]; WAS [Calder et al. 2011]; and Spanner [Corbett et al. 2012] are able to drive parallel IO across enormous fleets of storage machines. Unfortunately, these designs defer to costly mechanisms like two-phase locking or centralized concurrency managers in order to provide strong consistency across partitions.

At the same time, consensus protocols like Paxos [Lamport 1998] have been used as a building block for reliable distributed systems, typically deployed on a small number (between three to seven) servers, to provide a lever for a variety of consistent services: virtual block devices [Lee and Thekkath 1996]; replicated storage systems [Liskov et al. 1991; Thekkath et al. 1997]; lock services [Burrows 2006]; coordination facilities [Hunt et al. 2010]; configuration management utilities [MacCormick et al. 2004]; and transaction coordinators [Peng and Dabek 2010; Baker et al. 2011; Corbett et al. 2012]. As these successful designs have become pillars of today's data centers and cloud backends, there is a growing recognition of the need for these systems to scale in number of machines, storage-volume, and bandwidth.

Contact author's email address: Dahlia Malkhi: dalia@microsoft.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 0734-2071/2013/12-ART10 $15.00
DOI: http://dx.doi.org/10.1145/2535930

In this article we present CORFU, a shared global log design which alters this performance-safety tradeoff: On one hand, CORFU is designed to scale to hundreds of thousands of concurrent client operations per second. On the other hand, a shared log is a powerful and versatile primitive for ensuring strong consistency in the presence of failures and asynchrony.

Internally, CORFU is distributed over a cluster of machines and is fully replicated for fault-tolerance, without sharding data or sacrificing global consistency. CORFU allows hundreds of concurrent client machines in a large data-center to append to the tail of a shared log and to read from it over a network. Nevertheless, there is no single I/O bottleneck to either appends or reads, and the aggregated cluster throughput may be utilized by clients accessing the log.

Historically, shared log designs have appeared in a diverse array of systems. QuickSilver [Haskin et al. 1988; Schmuck and Wylie 1991] and Camelot [Spector et al. 1985] used shared logs for failure atomicity and node recovery. LBRM [Holbrook et al. 1995] uses shared logs for recovery from multicast packet loss. Shared logs are also used in distributed storage systems for consistent remote mirroring [Ji et al. 2003]. In such systems, CORFU fits the shared log role perfectly, pooling together the aggregate cluster resources for higher throughput and lower latency.

Moreover, a shared log is panacea for replicated transactional systems. For instance, Hyder [Bernstein et al. 2011] is a recently proposed high-performance database designed around a shared log, where servers speculatively execute transactions by appending them to the shared log and then use the log order to decide commit/abort status. In fact, Hyder was the original motivating application for CORFU and has been fully implemented over our code base.

Interestingly, a shared log can also be used as a consensus engine, providing functionality identical to consensus protocols such as Paxos (geographically speaking, Corfu and Paxos are neighboring Greek islands). 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 existing Paxos implementations, with far better performance than previous solutions.

Design. So far, we have argued the power, and hence the desirability, of a shared log. The key to its success is high performance, which we realize through a paradigm shift from existing cluster storage designs. In CORFU, each position in the shared log is projected onto a set of storage pages on different storage units. The projection 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 the map to determine a corresponding physical storage page, and then directly issues a read to the storage 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 storage pages mapped to that position.

In this way, the log in its entirety is managed without a leader, and CORFU circumvents the throughput cap of any single storage node. Instead, we can append data to the log at the aggregate bandwidth of the cluster, limited only by the speed at which the sequencer can assign them 64-bit tokens, that is, new positions in the log. The evaluation for this article was done with a user-space sequencer serving 200K tokens/s, whereas our more recent user-space sequencer is capable of 500K tokens/s. Moreover, we can support reads at the aggregate cluster bandwidth. Essentially, CORFU's design decouples ordering from I/O, extracting parallelism from the cluster for all IO while providing single-copy semantics for the shared log.

Naturally, the real throughput that clients obtain may depend on application workloads. However, we argue that CORFU provides excellent throughput in many realistic scenarios. CORFU's append protocol generates nearly perfect sequential write-load, which can be turned into a high throughput, purely sequential access pattern at the server units with very little buffering effort. Meanwhile, reads can be served from a memory cache [Rosenblum and Ousterhout 1991; Seltzer et al. 1995], or from from a cold standby as in Junqueira [2012]. This leaves nonsequential accesses, which may result from log compaction procedures. Because we design for large clusters, compaction need not be performed during normal operations, and may be fulfilled by switching between sets of active storage drives. Finally, as we originally argued in Balakrishnan et al. [2012] for the use of nonvolatile flash memory as CORFU storage units, which alleviates altogether any performance degradation due to random-accesses. Indeed, our present evaluation of CORFU is carried on servers equipped with SSD drives.

The last matter we need to address is efficient failure handling. When storage units fail, clients must move consistently to a new projection from log positions to storage pages. CORFU achieves this via a reconfiguration mechanism (patterned after Vertical Paxos [Lamport et al. 2009] and Virtually Synchronous Reconfiguration [Birman et al. 2010]) 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.

Applications. Applications can make use of the high throughput shared log design that CORFU offers in two fundamental ways. One is as a journal of commands which are applied in sequence to replicate a service, such as a replicated database. The other is as a log of data updates, to which clients dump revisions over the network, and CORFU durably stores data versions without overwriting in place. We exemplify both modes through two general-purpose service applications.

(1) *CORFU-Store.* This is a key-value store that supports atomic multikey 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.
(2) *CORFU-SMR.* This is a State Machine Replication (SMR) library where replicas propose commands by appending to the log and execute commands by playing the log.

The evaluation section reports on our experience with both of these services. In addition to these, we have also built a runtime called Tango that implements highly available, persistent data structures over CORFU, described in an accompanying paper [Balakrishnan et al. 2013].

Evaluation. 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 storage 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 multigets/s and 20K 10-key multiputs/s, with 4KB values), and that CORFU-SMR runs at 70K 512-byte commands/s with 10 state machine replicas.

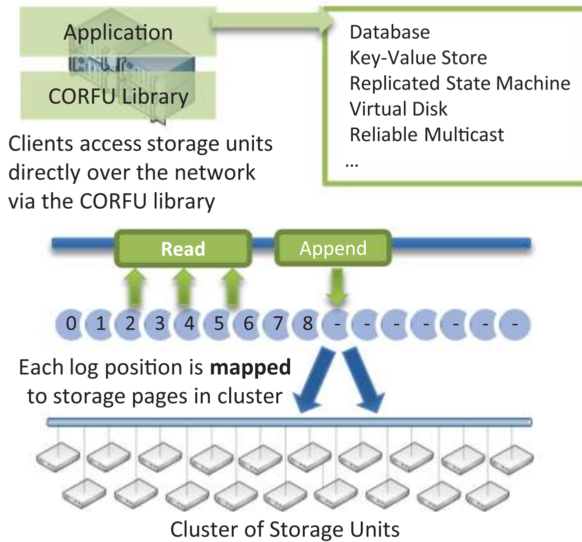

Fig. 1. CORFU presents applications running on clients with the abstraction of a shared log, implemented over a cluster of storage units by a client-side library.

Our design places most CORFU functionality at the clients, which reduces the complexity, cost, latency, and power consumption of the storage 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 [Wei et al. 2013], we have prototyped a network-attached storage unit on an FPGA platform; when used with the CORFU stack, this custom hardware provides the same throughput as a server-based storage unit while using an order of magnitude less power and providing 33% lower latency on reads. Over a cluster of such storage units, CORFU's logging design acts as a distributed SSD, implementing functionality found inside conventional SSDs—such as wear-leveling—at cluster scale.

Summary. To summarize the contributions of this article, we propose a complete design and implementation of a shared log abstraction over a storage cluster. This design is characterized by the property that 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 storage 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. 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 storage units (see Figure 1). Our goal is to provide applications running on the clients with a shared log abstraction implemented over the storage cluster.

Our design for this shared log abstraction aims to drive appends and random-reads to/from the log at the aggregate throughput that the storage cluster provides, while avoiding bottlenecks in the distributed software design. We strive to keep the server units' functionality as simple as possible, so that it can be realized by a standard host

append(b)	Append an entry b and return the log position ℓ it occupies
read(ℓ)	Return entry at log position ℓ
trim(ℓ)	Indicate that no valid data exists at log position ℓ
fill(ℓ)	Force append at log position ℓ to be completed

Fig. 2. API exposed by CORFU to applications.

equipped with a storage device, or even by a power-efficient FPGA controller (more on this below). We achieve these goals by devising a novel bottleneck-free distributed logging protocol, which places all CORFU functionality at the clients and lets them read and write directly to the address space of each storage unit. Storage nodes do not initiate communication, are unaware of other storage units, and do not participate actively in replication protocols.

The entire solution is given succinctly in Figure 4. On the left-hand side are the client's algorithms, and on the right-hand side, server functionalities (including the sequencer). In a nutshell, the solution consists of a client-side library which implements the API shown in Figure 2 to applications; and a server-side functionality which complements it to provide for reliable replication and efficient space management.

The client API includes the following operations. The *append* interface adds an entry to the log and returns its position. If the *append* operation encounters a problem, an error-code is returned, indicating that there is no certainty that the entry has been appended. 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 in the log, which may be needed to fill gaps in the log with valid content. The semantics provided for read/write operations is *linearizability* [Herlihy and Wing 1990], which for a log means that once a log entry is fully written by a client or has been read, any future read will see the same content (unless it is reclaimed via a *trim* operation).

CORFU's task of implementing a shared log abstraction with this API over a cluster of storage units—each of which exposes a separate address space—is described in the remainder of this section. We walk through the algorithms in detail below, starting with the storage-server, then moving to the way log entries are projected onto storage pages, the append functionality, and replication. We finish the description with mechanisms for reconfiguration and for garbage collection.

2.1. Storage Unit Functionality

The storage unit functionality is depicted in Figure 4 on the right-hand side. The most basic requirement of a storage unit is that it support READS and WRITES on an infinite, logical address space of fixed-size pages. We use the term "storage page" to refer to a page on this logical address space; logical pages are mapped internally to physical pages.

To support a specially-tailored replication protocol we have devised for the shared log, CORFU requires "write-once" semantics on the storage unit's address space. Reads on pages that have not yet been written should return an error code (`err_unwritten`). Writes on pages that have already been written should also return an error code

(err_written). In addition to reads and writes, storage units are also required to expose a DELETE command, allowing clients to indicate that the storage page is not in use anymore. Reading a previously deleted page should return an err_deleted error code.

Implementing the infinite address space entails keeping a hash-map from 64-bit virtual addresses to the physical address space of the storage. 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 deleted or not, which is set by the DELETE command.

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

To eventually remove deleted addresses from the hash-map, the storage unit also maintains a watermark before which no unwritten addresses exist, and removes deleted 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 *err_deleted* immediately.

The storage 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 *err_written* or *err_deleted* as appropriate. However, instead of writing to the block device, the storage unit points the hash-map entry to a special address reserved for junk; this ensures that storage pages are not wasted. Also, once the hash-map is updated, the entry is immediately deleted in the scope of the same operation. This removes the need for clients to explicitly track and trim junk in the log.

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

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

2.2. Projections

Each CORFU client holds a copy of a global *projection* that carves the log into disjoint ranges. Each such range is projected to a list of extents within the address spaces of individual storage units. Figure 3 shows an example projection, where range $[0, 40K)$ is projected onto extents on units F_0 and F_1, while $[40K, 80K)$ is projected onto extents on F_2 and F_3.

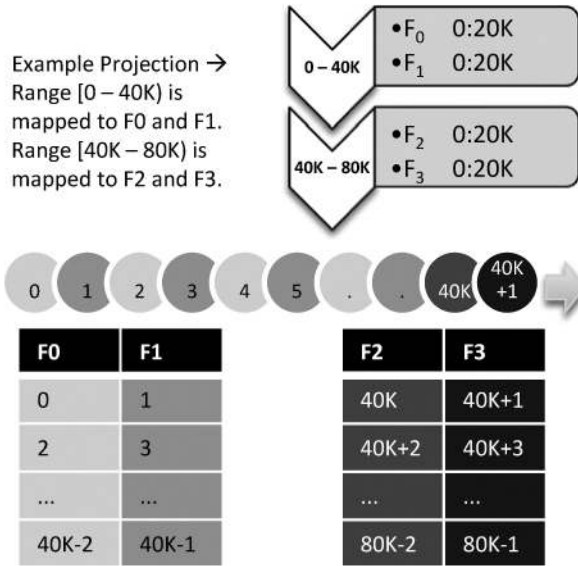

Fig. 3. Example projection that maps different ranges of the shared log onto storage unit extents.

Within each range in the log, positions are mapped to storage 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 storage page; for replication, each extent is associated with a replica set of storage 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 storage 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 is in the range $[40K, 80K]$ mapped 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 storage pages to use. With the round-robin function and the example projection above, the resulting page would be $F_2 : 2500$.

By projecting log positions onto storage pages, a projection allows clients to access the data pages associated with log entries directly at the storage units. Since CORFU organizes this address space as a log (using a tail-finding mechanism which we will 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.

2.3. Appending without Replication

Thus far, we have described the machinery used by CORFU to map log positions to sets of storage pages. This allows clients to directly access the page(s) associated with any position in a logical address space. Supporting reads is immediate from this. However,

```
Client:

local c_epoch, initially 0, and P, a projection

Operation READ(roffset):
  choose (serv, addr) in P(roffset)
    send ⟨read, c_epoch, addr⟩ to serv
  wait for reply from serv
    on ⟨err_sealed⟩: RECONFIG and redo READ
    on ⟨err_deleted⟩ or ⟨err_unwritten⟩:
      return the error code
    on ⟨page-content⟩: return the data

Operation APPEND(ctnt):
  send ⟨gettoken⟩ to tokenserver
  wait for reply woffset from tokenserver
  for each (serv, addr) in P(woffset):
    send ⟨write, c_epoch, addr, ctnt⟩ to serv
    wait for reply from serv
    on ⟨err_sealed⟩: RECONFIG and redo APPEND
    on ⟨err_deleted⟩ or ⟨err_written⟩:
      return it
  return(woffset)

Operation FILL(foffset):
  set f to err_junk
  for each (serv, addr) in P(woffset):
    send ⟨write, c_epoch, addr, f⟩ to serv
    wait for reply from serv
    on ⟨err_sealed⟩: RECONFIG and redo FILL
    on ⟨err_written⟩: set f to page data

Operation TRIM(toffset) at client:
  foreach (serv, addr) in P(woffset):
    send ⟨delete, addr⟩ to serv
    wait for reply from serv:

Operation RECONFIG(changes):
  send ⟨seal, c_epoch⟩ request to storage servers
  wait for replies
    from at least one replica in every extent
  compute new projection newP
    based on changes and ⟨sealed, ∗⟩ replies
  using any black-box consensus-engine:
    propose newP and receive decision
  set P to decision
  increment c_epoch
```

```
Storage server:

local s_epoch, initially 0, and status address-map

Upon ⟨read, epoch, addr⟩:
  if epoch ≠ s_epoch respond ⟨err_sealed⟩
  else respond according to addr status:
    if unwritten, ⟨err_unwritten⟩
    if deleted, ⟨err_deleted⟩
    if written, ⟨page-content⟩

Upon ⟨write, epoch, addr, c⟩:
  if epoch ≠ s_epoch respond ⟨err_sealed⟩
  else respond according to addr status:
    if deleted, ⟨err_deleted⟩
    if written, ⟨err_written, page-content⟩
    it available
      write c to addr in local store
      respond ⟨ack⟩

Upon ⟨delete, addr⟩ request:
  mark addr deleted
  respond ⟨ack⟩

Upon ⟨seal, epoch⟩:
  if epoch > s_epoch set s_epoch to epoch
  respond ⟨sealed, highaddr⟩ where
    highaddr is highest locally stored page address

Token-server:

local tkn, initially 0

Upon ⟨gettoken⟩:
  increment tkn and send it back
```

Fig. 4. CORFU protocols.

in order to treat this address space as an appendable log, clients must be able to find the tail of the log and exclusively write to it.

It is possible to allow clients to contend for positions in an unconstrained manner so long as only one write is allowed to "win" on each position; in the absence of replication, this property is satisfied trivially by the storage unit's write-once semantics. 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 linearizable log semantics, i.e., writes complete successfully with the guarantee that any subsequent read on the position returns the value written, until the position is trimmed.

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 starting counter of a new sequencer is determined using the highest page written on each storage unit, which is returned by the storage 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 storage pages and network bandwidth, storage units can be junk-aware, simply updating internal metadata to mark a logical address as filled with junk instead of writing an actual value to the storage.

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.

2.4. Appending with Replication

Naturally, the single-unit solution above does not provide fault tolerance. We achieve fault tolerance by projecting each position to a replica set of storage pages in the cluster using the current projection. The full append protocol is more intricate due to replication, as we now need to establish agreement on content among its replica set.

Although any replication protocol solves this correctly, several aspects of our settings guided us to devise a novel variant. First, since deploying a storage unit for data has nonmarginal cost, we require a consensus solution that tolerates f failures with just $f+1$ replicas. This means that we do not deploy any majority-based algorithm. Second, the CORFU design poses minimal functionality requisites on storage units, hence we

require replicas to not communicate with one another. Finally, we need to support fast reads that do not require a client to go to more than one replica to retrieve data.

To address these needs, CORFU uses a simple chaining protocol, which is in essence a client-driven variant of Chain Replication [van Renesse and Schneider 2004]. When a client wants to write to a replica set of storage pages (after having obtained a token for this position), it updates them in a deterministic order, waiting for each storage unit to respond before moving to the next one. The write is successfully completed when the last storage unit in the chain is updated. As a result, if two clients attempt to concurrently update the same replica set of storage pages, one of them will arrive second at the first unit of the chain and receive an *err_written*.

To read from the replica set, clients have two options. If the reading client already knows 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. Alternatively, a reader may probe the last unit of the chain to check if a write has completed. If indeed a write has completed on that log position, the last unit can return the content. If the last unit has not yet been updated, it will return an *err_unwritten*.

We need to address three kinds of failures. First, a client failure midway through an append can result in a replica chain partially or fully empty. Chained appends allow a client to rapidly fill such holes. 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.

Second, storage units may fail (or, they may require an upgrade, or become filled up and need replacement). We address configuration changes in the next section.

Third, due to reconfiguration, a client might fail to read a position even though a write on it has completed. This may happen as follows. Say that a projection is changed to remove a working unit, for example, due to a network disruption that causes the unit to temporarily detach. Some clients may remain uninformed about the reconfiguration, perhaps having detached from the main partition of the system due to the same network disruption. If this unit stored the last page in a replica set for some position in the previous projection, this client might continue to probe this unit for the status of this log position. Then the client may obtain an *err_unwritten* error code, even after the position has been completely written in the new configuration.

There are several ways to address this problem. Even with this somewhat rare failure scenario, the system upholds serializability of reads, which may suffice. Alternatively, we may disable fast reads altogether, and require reads to go to all replicas, unless they already know an entry has been filled. Another solution is to assign *leases* to storage units, which automatically expire unless the unit periodically renews them. Some entity in the system, for example, the sequencer, must manage these leases, which is not a heavy burden, as renewals are performed at fairly coarse intervals (typically in the order of seconds). However, a cost associated with this approach is that we need to wait for a lease expiration before we can remove a unit from the configuration.

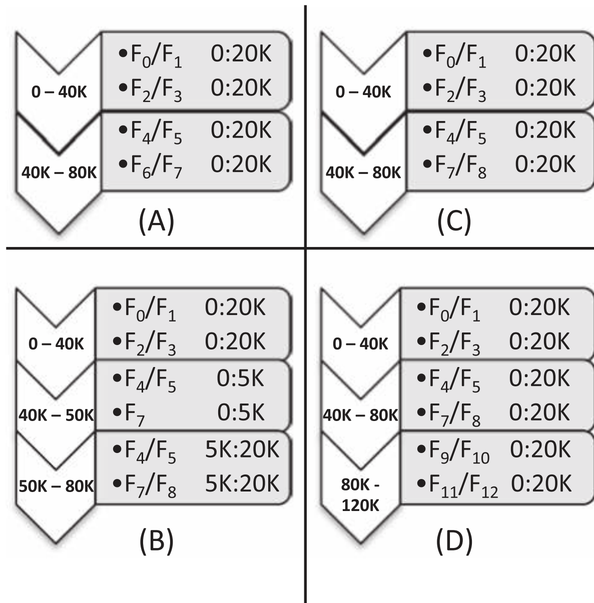

Fig. 5. Sequence of projections: When F_6 fails in (A) with the log tail at 50K, clients move to (B) in order to replace F_6 with F_8 for new appends. Once old data on F_6 is rebuilt, F_8 is used in (C) for reads on old data as well. When the log tail goes past $80K$, clients add capacity by moving to (D).

2.5. Changing Projections

When some event occurs that necessitates a change in a projection—for example, when a storage unit fails, or when the tail of the log moves past the current active range—we switch to a new projection. The succession of projections forms a sequence of *epochs*, and clients can read and write the storage units directly so long as their projection is up-to-date. When we change the projection, we invoke a seal request on all relevant storage units, so that clients with obsolete copies of a projection will be prevented from continuing to access them. All messages from clients to storage units are tagged with the epoch number, so messages from sealed epochs can be aborted.

In this sense, a projection serves as a *view* of the current configuration. However, a projection change does not necessarily ditch an old configuration and replace it with a new one. Rather, it may link a new range in a succession of ranges, keeping the old ones intact, and letting clients continue reading log entries which have already been filled. Likewise, it may affect the configuration of some past ranges and not others. So over time the log evolves in disjoint ranges, each one using its own projection onto a set of storage extents. Care must be taken to ensure that any clients operating in the context of previous epochs and clients in the new epoch read the log consistently.

Projections—and the ability to move consistently between them—offer a versatile mechanism for CORFU to deal with dynamism. Figure 5 shows an example sequence of projections. In Figure 5(A), range $[0, 40K)$ in the log is mapped to the two storage unit mirrored pairs F_0/F_1 and F_2/F_3, while range $[40K, 80K)$ is mapped 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.

Note that this upholds safety of fully written log entries: Any log offset which has already been mirrored to F_7 in the range $[40K, 50)$ can continue to be read in the next epoch.

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 $80K$.

The mechanics of forming a reconfiguration decision are tailored to the client-centric CORFU architecture, and in particular, involve no communication among the storage units. More specifically, our reconfiguration procedure involves two steps. The first is a "seal-and-snapshot" step, and its purpose is to ensure that any appended value in the current projection P_i survives reconfiguration. This would not be possible if clients continued writing directly to the active range of P_i indefinitely. We reject any client messages with obsolete epochs, writes as well as reads. When clients receive these rejections, they realize that the current projection has been sealed. We note that storage units not affected by the reconfiguration need not be sealed, and hence, often only a small subset of storage units have to be sealed, depending on the reason for reconfiguration.

The second step involves reaching an agreement decision on the next projection P_{i+1}. The decision must preserve the safety of any log entries, which have been fully written and might have been read by clients in the current active range. Therefore, we always "err on the conservative side," and include any entry that is filled in all surviving storage units. For example, consider the scenario in Figure 5. Moving from (A) to (B), we lost F_6 but retained the mirror F_7. The new projection (B) upholds the safety of any log entries that have been fully mirrored to F_7. In a slightly different scenario, we might lose F_7, and retain F_6. In this case, we won't know if entries $[40K, 50K)$ have been successfully mirrored or not, but we would have to assume that they might have been, and extend the projection accordingly.

We do not discuss the details of the actual consensus protocol here, as there is abundant literature on the topic. Our current implementation incorporates a Paxos-like consensus protocol using storage units in place of Paxos-acceptors. Note that multiple clients can initiate reconfiguration simultaneously, but only one of them succeeds in proposing the new projection.

Clients that read a storage unit and discover that their local projection is stale need to learn the new projection. We currently employ a shared network drive for storing the sequence of agreed-upon projections, but other methods may replicate this information more robustly.

2.6. 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 to 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 storage unit. Each storage-unit then maps a sparse 64-bit address space, broken into extents, onto the physical set of pages. The storage unit has to maintain a hash map from 64-bit addresses to the physical address space of the storage.

Another place where system information might grow large is the succession of projections. Each projection by itself is a range-to-range mapping, and hence it is quite concise. However, 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 storage 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 in 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 share.

3. CORFU APPLICATIONS

There are two fundamental modes in which distributed applications can cooperate over a shared log, one as a log structured store, and another as a totally ordered reliable broadcast channel. We explain both modes and exemplify their use through two general-purpose services, one for each mode, which we built over CORFU.

The first usage mode is a log-structured store. It holds a universe of data items which are updated over time by appending new versions to the log. This has several benefits: It automatically maintains a journal of versions and updates. It creates a sequential update workload instead of writing to random places. It can easily support multiitem operation atomicity. Last, it can easily support snapshot and mirroring operations by capturing and copying growing prefixes of the log.

We have prototyped a general-purpose service using this mode, named CORFU-Store, a key-value store. This is a key-value store that supports a number of properties that are difficult to achieve on partitioned stores, including atomic multikey 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 multikey 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.

The evaluation section reports more on our experience with CORFU-Store.

The second use case of CORFU is as a reliable, totally ordered broadcast channel. This is a quintessential tool for consistent replication, identical to the manner in which Paxos is used 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. Here, the log serves both as the authoritative source of ordering in the system and as a durable store; an SMR state can be soft, and may always be recovered by playing the log. In this capacity, CORFU is an enabler for a class of high throughput, in-memory replicated services. For example, we have implemented the Hyder database [Bernstein et al. 2011], which was our primary initial motivating use case, over CORFU.

We also built and evaluated an SMR library over CORFU, which deposits opaque commands in the log and uploads commands from the log for execution by the SMR servers. In our current implementation, each SMR server plays the log forward by issuing reads to the log. We present some performance numbers regarding this SMR library in the evaluation section.

Many of the research issues involved with CORFU applications are beyond the scope of this article, and will be reported separately. Briefly, when used as a log-structured store, the challenges concentrate around two dimensions. One is to maintain a mapping from data items to their most recent location; and to implement the distributed mapping, so that it doesn't become a bottleneck for users. Another is to compact and clean-up the log so that it does not grow sparse with old entries left at the tail of the log. When used for replication, one of the main challenges is to be able to keep up with the log by the replicated state-machines. We are investigating different approaches to playing the log forward, perhaps by having dedicated machines multicast the contents of the log out to all SMR servers. In both use modes, common challenges are to provide for checkpointing application state, so that recovering from a failure with the help of the log becomes fast, and to allow selective recovery from the log for a variety of purposes.

There are other potential applications. 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 [Meyer et al. 2008]. 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.

4. EVALUATION

Our current evaluation platform uses SATA SSDs for block devices. These are well-suited for our purposes for a number of reasons. Nonvolatile memories are fast, holding great potential for high performance storage in large data-centers. Additionally, the use of nonvolatile memory-based storage alleviates any potential performance hiccups due to nonsequential access for occasional reads from the middle of the log and for garbage collection. Finally, our log-structured access matches perfectly the best usage scenario for SSDs, because the Flash Translation Layer (FTL) of today's SSDs

internally organizes memory as a log store. Our storage units come in the following two forms.

(1) *Server-attached SATA SSDs.* This consists of a pair of conventional SATA SSDs attached to servers. The server accepts CORFU commands over the network from clients and implements the server functionality described above, issuing reads and writes to the fixed-size address space of the block device as required.
(2) *Network-attached flash.* This custom implementation consists of an FPGA, a network interface, and a SATA-connected SSD. The server functionality and network protocols are entirely implemented in hardware. This design eliminates the need for a server interposed between fast storage devices and clients sitting on a fast interconnect. We used the Beehive [Thacker] architecture on both the Xilinx XUPv5 development platform [Xilinx 2011] and the BEE3 hardware platform [Davis et al. 2009] for prototyping. When running at full speed, the XUPv5 prototype consumes around 15W; in contrast, a conventional storage server consumes on the order of 250W. With suitable Flash Translation Layer firmware, our network flash unit could run over raw flash rather than employing an SSD. Arguably, CORFU's inherent log structure allows a simpler FTL design as compared to a standard disk interface.

For throughput, 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 interrack 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.

4.1. End-to-End Latency

We first summarize the end-to-end latency characteristics of CORFU in Figure 6. We show the latency for *read*, *append*, and *fill* operations issued by clients for four CORFU configurations. The leftmost 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

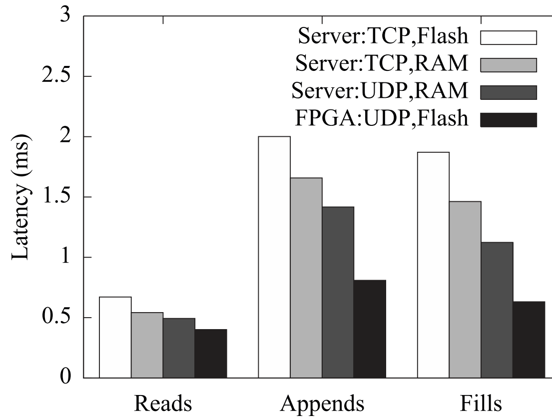

Fig. 6. Latency for CORFU operations on different flash unit configurations.

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 6 makes a number of important points. First, the latency of the FPGA unit is very low for all three operations, providing submillisecond appends and fills while satisfying reads within half a millisecond. This justifies our emphasis on a client-centric 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 drastically append latency; on the FPGA unit, appends complete within 750 microseconds.

4.2. 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 completely written to. 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.

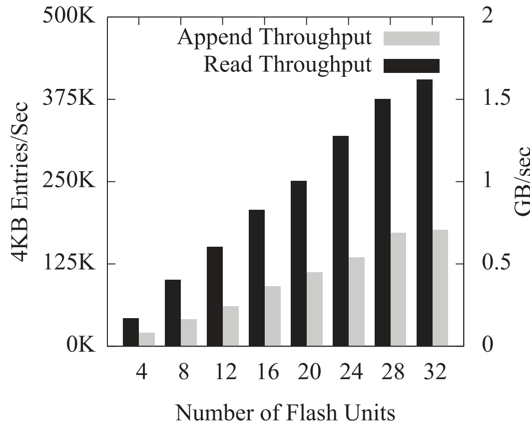

Fig. 7. Throughput for random reads and appends.

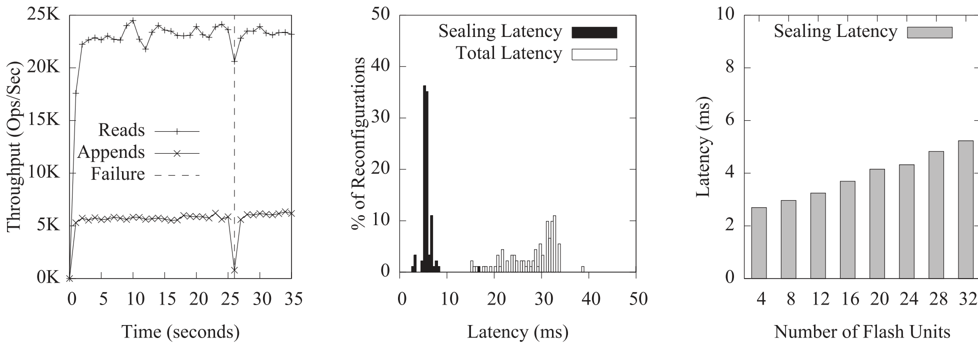

Fig. 8. Reconfiguration performance on 32-drive cluster. Left: Appending client waits on failed drive for 1 second before reconfiguring, while reading client continues to read from alive replica. Middle: Distribution of sealing and total reconfiguration latency for 32 drives. Right: Scalability of sealing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

At such high append throughputs, CORFU can wear out 1 TB of MLC flash in around four 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).

4.3. 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 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

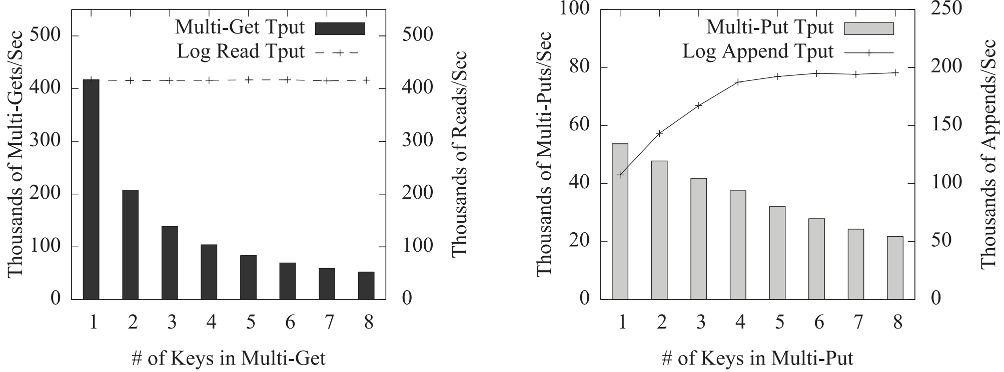

Fig. 9. Example CORFU application: CORFU-Store supports atomic multigets and multiputs at cluster scale.

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 submillisecond 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.

4.4. Applications

We now demonstrate the performance of CORFU-Store for atomic multikey operations. Figure 9 (left) shows the performance of multiput operations in CORFU-Store. On the x-axis, we vary the number of keys updated atomically in each multiput operation. The bars in the graph plot the number of multiputs executed per second. The line plots the total number of CORFU log appends as a result of the multiput operations; a multiput involving k keys generates $k+1$ log appends, one for each updated key and a final append for the commit record. For small multiputs 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 multiput, the number of log appends generated increases and the CORFU log bottlenecks at around 180K appends/sec. Beyond four keys per multiput, the log bottleneck determines the number of multiputs we push through; for example, we obtain $\frac{180K}{6} = 30$K multiputs involving five keys.

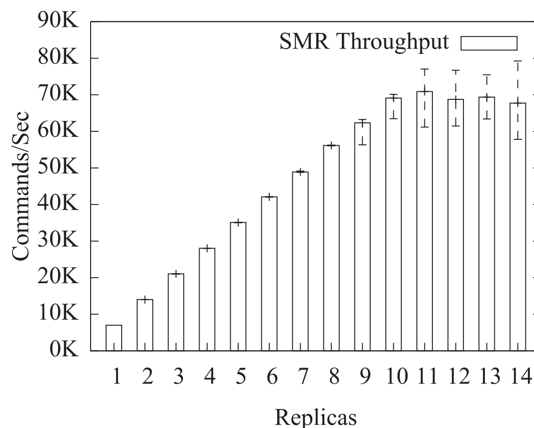

Fig. 10. Example CORFU application: CORFU-SMR supports high-speed state machine replication.

Figure 9 (right) shows performance for atomic multiget operations. As we increase the number of keys accessed by each multiget, the overall log throughput stays constant while multiget throughput decreases. For four keys per multiget, we get a little over 100K multigets 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, seven 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 min and max across replicas) at which commands are executed by each replica. While the system has ten or fewer CORFU-SMR replicas, the rate at which commands are executed in the system matches the rate at which they are injected. Beyond ten replicas, we overload the ability of a client to execute incoming commands, and hence throughput flatlines; this means that clients are now lagging behind while playing the state machine.

5. RELATED WORK

At its base, CORFU derives its success by scaling Paxos to data center settings and providing a durable engine for totally ordering client requests at tens of Gigabits per second throughput. The introduction already mentions a variety of real-life uses of Paxos for which CORFU provides a high-performance drop-in replacement, and we do not repeat them here.

But how far is CORFU from Paxos? A CORFU log implements a fault-tolerant total ordering service to which clients propose to append values (page-payloads) and receive them back in a unique sequence order. This service is at the core of the State-Machine-Replication (SMR [Schneider 1990]) approach, which forms a total ordering on client requests and delivers as output an ordered sequence of requests. Covering the multitude of existing SMR solutions is beyond our scope, and we refer the reader to an excellent survey by DeFago et al. [2003]. Our design borrows bits and pieces from various existing methods, yet we invented a new solution in order to scale SMR to large

cluster scales and deliver aggregate cluster throughput to hundreds of clients. This was made possible via two principles.

—First, we have many small replica-sets, typically each one with 2–5 servers, each responsible for one or more log entries. These entries form an abstract log, and we can write and read different entries in the log in parallel, thus utilizing the total bandwidth available in the cluster.
—Second, we make writing to the log fast by avoiding contention on any entry. Contention-free appends are managed using the CORFU-sequencer, which guides clients smoothly from one offset to the next. It is important to emphasize that the CORFU-sequencer does not have a precise analogy in traditional consensus protocols. The sequencer might look like a leader, but the big difference is that the sequencer does not process or store client requests, nor interact with data replicas; it only handles sequence-number "reservations". This crucial distinction lets the CORFU sequencer operate at extremely high rates, free of communication with data replicas.

Putting these two principles together, the replication protocol we devised for CORFU avoids any centralized IO bottleneck without partitioning data.

Stitching many small replica sets into a global order has also been the goal in a number of recent works, notably Mencius [Mao et al. 2008]; Kapritsos and Junqueira [2010]; and Calvin [Thomson et al. 2012]. Mencius rotates the tail of the log among the sets in a round-robin manner, requiring each set to declare a proposal or an empty announcement in its turn. In the latter two, replicas advance in lock-step from one global wave after another, essentially emulating Lamport's total ordering protocol [Lamport 1978].

The consensus protocol we use for replicating each log-entry is itself a somewhat novel consensus variant. The protocol works similarly to chain replication [van Renesse and Schneider 2004], with passive data-stores instead of servers at each link, and with the client copying the value from link to link instead of servers forwarding it to one another. Each replica enforces write-once semantics. Clients reach consensus by attempting to write to the head of the chain, and copying the head to the rest of the chain. Client-driven consensus protocols exist already [Gafni and Lamport 2000; Chockler and Malkhi 2005], but these previous approaches do no scale well with the number of clients.

Working with our architecture brought additional challenges, which were met with more deviations from standard Paxos. First, our design creates a new failure mode, a "hole," which happens when a client reserves a sequence offset and fails without completing a consensus decision on it. We fill such a hole by contending for the consensus decision with a competing "junk" mark. This leverages the power of consensus for lightweight fault-handling, without requiring system reconfiguration. Second, when reconfiguration is needed, we employ an auxiliary configuration manager, which lets us architect CORFU with only $F+1$ data replicas for each log-entry (rather than $2F+1$, as in Paxos). This design makes sense because the auxiliary is not a contention point, and participates only in low-volume operations, hence it can be shared throughout the cluster and itself be hardened using replication. The foundations for this approach have been formalized in Lamport et al. [2010]; Birman et al. [2010]; Lamport et al. [2009].

The broader context in which CORFU lies is back-end infrastructure systems for web applications that are characterized by the need to strike a balance between performance and consistency at unprecedented magnitudes. In this arena, the dominance

of high-end commercial relational database systems (RDBMS) like Oracle, SQL Server, MySQL, and DB2 was put to an almost abrupt stop when major Internet companies like Google and Amazon faced the challenge of storing huge quantities of data that need to be retrieved and displayed in real-time. Systems like Amazon Dynamo [DeCandia et al. 2007] and Google BigTable [Chang et al. 2008] arose, optimizing for simple and fast store/retrieve functionality. These systems and others like it, commonly known as *NoSQL* systems, give up on atomicity, consistency, isolation or durability (ACID) of traditional RDBMS in return for capacity and performance. The first generation of NoSQL systems includes several widely-deployed ones, among which are MongoDB [10gen 2011]; Cassandra [Lakshman and Malik 2010]; Voldemort [LinkedIn 2011]; CouchDB [Apache 2011] and, as already mentioned, Dynamo and BigTable. CORFU is not targeted to replace such systems, but it can serve as a viable building block for leveraging consistency. For example, CORFU-Store demonstrates the feasibility of constructing over CORFU a key-value store with multikey ACID functionality, which goes beyond the semantics of many NoSQL systems.

With the success of first-generation NoSQL systems came a realization that stronger semantics were needed after all. Percolator [Peng and Dabek 2010] and ReTSO [Junqueira et al. 2011] are two recent systems built on top of flat object stores, precisely to provide missing functionality across partitions. Unfortunately, both employ a centralized serialization authority to manage concurrency. More recently, systems designed for large-scale data services which provide support for transactions were designed from bare bones (e.g., Megastore [Baker et al. 2011]; WAS [Calder et al. 2011]; Walter [Sovran et al. 2011]; and Spanner [Corbett et al. 2012]). These systems employ costly concurrency control mechanisms like two-phase locking for cross-partition transaction support. A very recent distributed transactional system, Calvin, avoids two-phase locking using the idea of deterministic databases [Thomson et al. 2012], already mentioned above. Calvin leverages a common source of ordering for lock acquisition, which is implemented via a real-time distributed sequencer; CORFU can provide an asynchronous replacement for Calvin's sequencer.

It was argued in a design called Hyder [Bernstein et al. 2011] that one could build a high throughput in-memory lock-free transactional service over a shared log using optimistic concurrency control. Indeed, we have a fully working prototype of Hyder on top of CORFU. With CORFU operating an a shared log driving Hyder's optimistic concurrency control, we can sustain hundreds of thousands of transactions per second. In fact, Hyder was the motivating application that initially inspired the CORFU construction, although CORFU departs extensively from the Hyder-Log proposal in the mechanisms it employs, such as projections, sequencer, chain replication, the hole-filling protocol, and write-once units with infinite address spaces.

Finally, there is a strong connection between CORFU and log-structured file-systems (LFS) [Rosenblum and Ousterhout 1991]. Our rekindled interest in this approach is partly due to emerging solid-state memory technologies, although not limited to it. Modern SSD controllers manage storage internally as a log-store, making them ideally suited for log design, and for eliminating the traditional problem with such systems such as garbage collection interference with foreground activity [Seltzer et al. 1995]. Zebra [Hartman and Ousterhout 1995] and xFS [Anderson et al. 1995] 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.

6. CONCLUSION

Despite almost forty years of research into replicated storage schemes, the only approach so far to scale up capacity and throughput has been to shard data and trade consistency for performance. In this article, we presented the CORFU system, which breaks this seeming tradeoff by organizing a cluster of 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 any single I/O bottleneck between numerous clients and the cluster, allowing data to stream to and from the cluster in parallel.

REFERENCES

10Gen. 2011. MongoDB. http://www.10gen.com/white-papers.

Anderson, T., Dahlin, M., Neefe, J., Patterson, D., Roselli, D., and Wang, R. 1995. Serverless network file systems. *ACM SIGOPS Oper. Syst. Rev. 29*, 109–126.

Apache. 2011. CouchDB. http://couchdb.apache.org/.

Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., L'Eon, J., Li, Y., Lloyd, A., and Yushprakh, V. 2011. Megastore: providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 223–234.

Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobber, T., Wei, M., and Davis, J. 2012. Corfu: A shared log design for flash clusters. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association.

Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. 2013. Tango: Distributed data structures over a shared log. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York.

Bernstein, P., Reid, C., and Das, S. 2011. Hyder—A transactional record manager for shared flash. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*. 9–20.

Birman, K., Malkhi, D., and Van Renesse, R. 2010. Virtually synchronous methodology for dynamic service replication. Tech. rep. MSR-TR-2010-151, Microsoft Research.

Burrows, M. 2006. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, 335–350.

Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., et al. 2011. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York, 143–157.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. 2008. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2, 4.

Chockler, G. and Malkhi, D. 2005. Active disk Paxos with infinitely many processes. *Distrib. Comput. 18*, 1, 73–84.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, 251–264.

Davis, J., Thacker, C. P., and Chang, C. 2009. BEE3: Revitalizing computer architecture research. Tech. rep. MSR-TR-2009-45, Microsoft Research.

Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP'07)*.

Defago, X., Schiper, A., and Urban, P. 2003. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. 36*, 2004.

GAFNI, E. AND LAMPORT, L. 2000. Disk Paxos. In *Proceedings of the 14th International Conference on Distributed Computing (DISC'00)*. Springer, Berlin, 330–344.

HARTMAN, J. H. AND OUSTERHOUT, J. K. 1995. The zebra striped network file system. *ACM Trans. Comput. Syst. 13*, 3, 274–310.

HASKIN, R., MALACHI, Y., AND CHAN, G. 1988. Recovery management in quicksilver. *ACM Trans. Comput. Syst. 6*, 1, 82–108.

HERLIHY, M. P. AND WING, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3, 463–492.

HOLBROOK, H. W., SINGHAL, S. K., AND CHERITON, D. R. 1995. Log-based receiver-reliable multicast for distributed interactive simulation. *SIGCOMM Comput. Commun. Rev. 25*, 4, 328–341.

HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. 2010. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, 11–11.

JI, M., VEITCH, A., AND WILKES, J., ET AL. 2003. Seneca: Remote mirroring done write. In *Proceedings of the USENIX Annual Technical Conference*.

JUNQUEIRA, F. 2012. Durability with BookKeeper. In *Proceedings of LADIS'12*.

JUNQUEIRA, F., REED, B., AND YABANDEH, M. 2011. Lock-free transactional support for large-scale storage systems. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 176–181.

KAPRITSOS, M. AND JUNQUEIRA, F. P. 2010. Scalable agreement: Toward ordering as a service. In *Proceedings of the Sixth International Conference on Hot Topics In System Dependability (HotDep'10)*. USENIX Association, 1–8.

LAKSHMAN, A. AND MALIK, P. 2010. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 35–40.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM 21*, 7, 558–565.

LAMPORT, L. 1998. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 133–169.

LAMPORT, L., MALKHI, D., AND ZHOU, L. 2009. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC'09)*. ACM, New York, 312–313.

LAMPORT, L., MALKHI, D., AND ZHOU, L. 2010. Reconfiguring a state machine. *ACM SIGACT News 41*, 1, 63–73.

LEE, E. AND THEKKATH, C. 1996. Petal: Distributed virtual disks. *ACM SIGOPS Oper. Syst. Rev. 30*, 5, 84–92.

LINKEDIN. 2011. Voldemort. http://www.project-voldemort.com/voldemort/.

LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., AND SHRIRA, L. 1991. Replication in the harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*. ACM, New York, 226–238.

MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. 2004. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*. USENIX Association, Berkeley, CA, 105–120.

MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. 2008. Mencius: Building efficient replicated state machines for WANS. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 369–384.

MEYER, D. T., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. 2008. Parallax: virtual disks for virtual machines. *SIGOPS Oper. Syst. Rev. 42*, 4, 41–54.

PENG, D. AND DABEK, F. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, 1–15.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1991. The design and implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev. 25*, 5, 1–15.

SCHMUCK, F. AND WYLIE, J. 1991. Experience with transactions in quicksilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*. ACM, New York, 239–253.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4, 299–319.

SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. 1995. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX Technical Conference (TCON'95)*. USENIX Association, Berkeley, CA, 21–21.

Sovran, Y., Power, R., Aguilera, M. K., and Li, J. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, 385–400.

Spector, A. Z., Daniels, D., Duchamp, D., Eppinger, J. L., and Pausch, R. 1985. Distributed transactions for reliable systems. *SIGOPS Oper. Syst. Rev. 19*, 5, 127–146.

Thacker, C. P. Beehive: A many-core computer for FPGAs. Unpublished Manuscript.

Thekkath, C. A., Mann, T., and Lee, E. K. 1997. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, NY, 224–237.

Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. ACM, New York, 1–12.

Van Renesse, R. and Schneider, F. B. 2004. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI'04)*. USENIX Association, Berkeley, CA, 7–7.

Wei, M., Davis, J. D., Wobber, T., Balakrishnan, M., and Malkhi, D. 2013. Beyond block i/o: implementing a distributed shared log in hardware. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. ACM, New York, 21:1–21:11.

XILINX. 2011. Xilinx university program xupv5-lx110t development system. http://www.xilinx.com/univ/xupv5-lx110t.htm.

Received December 2012; accepted March 2013