outline!

Submission Type: Research

Abstract

To meet the needs of a diverse and growing set of cloud-based applications, modern distributed storage frameworks expose a variety of composable subsystems as building blocks. This approach gives infrastructure programmers significant flexibility in implementing application-specific semantics while reusing trusted components. Unfortunately, in current storage systems the composition of subsystems is a low-level task that couples (and hence obscures) a variety of orthogonal concerns, including functional correctness and performance. Building an application by wiring together a collection of components typically requires thousands of lines of carefully-written C++ code, an effort that must be repeated whenever device or subsystem characteristics change.

In this paper, we propose a declarative approach to subservice composition that allows programmers to focus on the high-level functional properties that are required by applications. Choosing an implementation that is consistent with the declarative functional specification then can be posed as a search problem over the space of parameters such as block sizes, storage interfaces (e.g. key/value or block storage) and concurrency control mechanisms. We present experimental evaluation of our prototype, (etc etc)

1 Introduction

Storage systems are increasingly providing features that take advantage of application-specific knowledge to achieve optimizations and provide unique services. However, this trend is leading to the creation of a large number of software extensions that will be difficult to maintain as system software and hardware continue to evolve.

The standardization of the POSIX file I/O interface has been a major success, allowing application developers to avoid vendor lock-in. However, large-scale storage systems have been dominated by proprietary products, preventing exploration of alternative interfaces and complicating future migration paths, eliminating the benefits of commodity systems. But the recent availability of high-performance open-source storage systems is changing this because these systems are modifiable, enabling interface change, and reducing the risks of lock-in. The widely deployed Ceph distributed storage system is an example of a storage system that supports application-specific extensions in the form of custom I/O interfaces to objects managed by the underlying RADOS object storage system [27, 28]. Organizations are increasingly reliant upon these extensions as is shown in Figure 1 by a marked increase in the number of object operations that are packaged as part of the Ceph distribution and widely used by internal Ceph subsystems and by applications such as OpenStack Swift and Cinder [2].

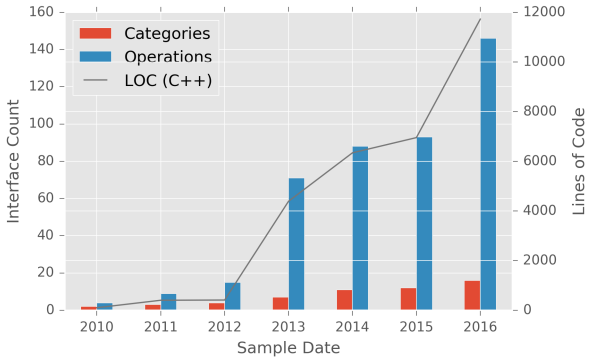
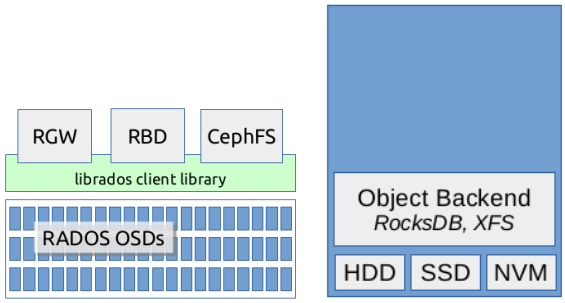


Figure 1: [source] Growth of officially supported, custom object interfaces in RADOS over 6 years. An *operation* is a function executed in the context of an object, and operations are grouped into different *categories* corresponding to applications or utilities, such as reference counting

In addition to the growth in the quantity of operations in use throughout Ceph installations, Figure 1 also depicts the amount of low-level C++ written to implement these operations. Unfortunately, this code is written assuming a performance profile defined by the combination of the hardware and software versions available at the time of development. While the bulk of these interfaces are created by core Ceph developers with a complete view of the performance model, this may be changing as the development community has been receptive to outside contributions with the recent inclusion by CERN developers of an extension for performing limited numeric operations on object data [1]. And while Ceph has not yet reached the point of directly exposing these features to non-administrative users, the recent inclusion of a mechanism for dynamically defining extensions using Lua [3]



(a) Ceph (b) OSD

Figure 2: Ceph and OSD... gotta cram more stuff in here...

suggests that aspects of this feature may soon appear. What is needed is support for creating storage interfaces using a method that allows transparent optimization as the system, application, and supporting environment evolve.

Previous work related to storage interface design has largely been in the context of standardization efforts and active storage. While the later has been specifically concerned with the creation of application-specific interfaces and demonstrating short-term benefits, the standardization seeks to develop a last set of interfaces. In neither case have we seen efforts focused on extensible interfaces with portability and longevity a primary concern.

We propose the introduction of a declarative language for building object based interfaces that allows a storage system to meet the needs of an application throughout the development process without requiring rewrites.

2 Background and Motivation

In this section we highlight the salient components of Ceph, especially its *object class* feature that offers users the ability to load and execute application-specific codes. We provide a description of our motivating example, a high-performance distributed shared-log built upon Ceph that makes extensive use of the object class facility.

2.1 Ceph and Storage Programmability

Figure 2a illustrates the collection of components commonly referred to as Ceph. At the bottom, a cluster of 10s–10,000s *object storage devices* compose the distributed object storage system called RADOS. Widely deployed applications such as the S3/Swift–compliant RADOS Gateway (RGW), RADOS Block Device (RBD), and the POSIX Ceph File System are built upon the *librados* client layer that presents a fault-tolerant always-on view of the RADOS cluster.

The object storage device (OSD), illustrated in Figure 2b, is the building block of the RADOS cluster and is responsible for managing and providing access to a set of named objects. The configuration of an OSD is flexible, and commonly contains a mix of commodity hardware such as HDD and SSD bulk storage, a multi-core CPU, GBs of RAM, and one or two 10 Gb Ethernet links. Clients access object data managed by an OSD by invoking native object operations exposed by the OSD such as reading or writing bytes, as well as more complex operations like taking snapshots or composing one or more native operations into compound procedures that execute in a transactional context.

The native object operations in RADOS roughly fall into two categories based on the type of data being accessed: key-value items, or bulk bytestream data. The key-value interface operates as a dedicated database associated with each object, and the bytestream interface supports random byte-level access similar to a file. At a low-level each of these abstract I/O interfaces map to hardware storage devices through a pluggable object backend storage service. For instance, LevelDB or RocksDB may be used to store key-value data, while the *FileStore* implementation maps the bytestream interface onto a local POSIX file system [16, 15]. Several backend implementations exist for storing data in targets such as the Kinetic Drive or in NVMe devices, among others.

Object Classes While Ceph provides a wide variety of native object operations, it also includes a facility referred to as *object classes* that allow developers to create application-specific object operations in the form of C++ shared libraries dynamically loaded into the OSD process at runtime. Object classes can be used to implement basic data management tasks such as indexing metadata, or used to perform complex operations such as data transformations or filtering. Table 1 summarizes the range of object classes maintained in the upstream Ceph project which support internal Ceph subsystems as well as applications and services that run on top of Ceph.

Category	Specialization	Methods
Locking	Shared Exclusive	6
Logging	Replica State Timestamped	3 4 4
Garbage Collection	Ref. Counting	4
Metadata	RBD RGW User Version	37 27 5 5

Table 1: A variety of RADOS object storage classes exist that expose reusable interfaces to applications.

A critical step in the development of application-specific object interfaces is deciding how to best make use of the native object interfaces. For instance if an application stores an image in an object, it may also extract and store EXIF metadata as key-value pairs in the object key-value database. However, depending on the application needs it may be sufficient or offer a performance advantage to store this metadata as a header within the bytestream. In the remainder of this section we will explore the challenges associated with these design questions.

2.2 Motivating Application: CORFU

The primary motivating example we will use in this paper is the CORFU distributed shared-log designed to provide high-performance serialization across a set of flash storage devices [6]. The shared-log is a powerful abstraction useful when building distributed systems and applications, but common implementations such as Paxos or Raft funnel I/O through a single node limiting total throughput [20]. The CORFU protocol addresses this limitation by de-coupling log entry storage from log metadata management, making use of a centralized, volatile, in-memory *sequencer service* that assigns positions to clients that are appending to the log. Since the sequencer is centralized serialization is trivial, and the use of non-durable state allows the sequencer service to operate at very high rates. The CORFU system has been used to demonstrate a number of interesting services such as transactional key-value and metadata services, replicated state machines, and an elastic cloud-based database management system [5, 8].

Two aspects of CORFU make its design attractive in the context of the Ceph storage system. First, CORFU assumes a cluster of flash devices because log-centric systems tend to have a larger percentage of random reads making it difficult to achieve high-performance with spinning disks. However, the speed of the underlying storage does not affect correctness. Thus, in a software-defined storage system such as Ceph a single implementation can transparently take advantage of any software or hardware upgrades, and make use of existing and future data management features such as tiering, allowing users to freely choose between media types such as SSD, spinning disks for archival storage, or emerging NVRAM technologies.

CORFU and Storage Programmability The second property of CORFU relevant in the context of Ceph is the dependency CORFU places on custom storage device interfaces used to guarantee serialization during failure and reconfiguration. Each flash device in a CORFU cluster exposes a 64-bit write-once address space consisting of the primary I/O interfaces *write(pos, data)* and *read(pos)* for accessing log entries, as well as *fill(pos)* and *trim(pos)* that invalidate and reclaim log entries, respectively. All I/O operations in CORFU initiated by clients are tagged with an *epoch* value, and flash devices are expected to reject client requests that contain an old epoch value. To facilitate recovery or handle system reconfiguration in CORFU, the storage devices are also required to support a *seal(epoch)* command that stores the latest epoch and returns the maximum position written to that device. The seal interface is used following the failure of a sequencer to calculate the tail of the log that the sequencer should use to repopulate its in-memory state.

While the authors of the CORFU paper describe prototype device interfaces implemented as both host-based and FPGA-based solutions, RADOS *directly* supports the creation of logical storage devices through its object class feature described previously in Section 2.1. Thus, by using software-based object interfaces offered by RADOS flash devices in CORFU can be replaced by software-defined storage offering significant flexibility and a simplified design.

The implementation of a custom object class that satisfies the needs of an application such as CORFU is often straightforward. However, as described in Section 2.1 there are a variety of native object I/O interfaces available, and it is not always immediately clear how best to utilize these interfaces.

3 Physical Design

As we have seen, Ceph provides a rich storage API and places few restrictions on the structure of applications. Thus a primary concern when implementing an object interface in Ceph is deciding how native interfaces are composed into compound operations in order to implement the semantics of the target interface. These types of decisions are commonly referred to as physical design, and can affect performance and application flexibility. As we will see the design space that developers must operate in is often large, and its dynamic nature can lead to design decisions that become obsolete and non-optimal.

To understand the developer process in the context of CORFU we have included the state-machine diagram in Figure 3 showing the composition of actions for each component of the CORFU interface which must be mapped onto Ceph object classes. For instance, all operations begin by applying an *epoch guard* that ensures the request is tagged with an up-to-date epoch value. The *read* (R) and *write* (W) operations both proceed by (1) examining metadata associated with the target position, (2) performing I/O to read or write the log entry, and in the case of a write, (3) updates metadata for the target log position.

As an example, one valid design option is to store each

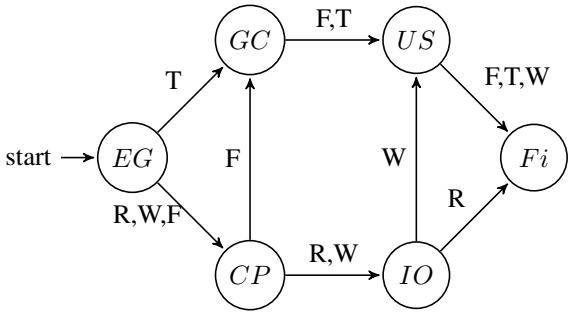


Figure 3: State transition diagram for read (R), write (W), fill (F), and trim (T) CORFU operations. The states epoch guard (EG), check position (CP), and update state (US) access metadata. The I/O performs a log entry read or write, and garbage collection (GC) marks entries for reclamation.

log position in an object with a name using a one-to-one mapping with the log entry position. This would simplify the design of the *write* interface because a small amount of metadata stored as a header in the object could describe the state of the log entry. However, as we will see this choice of a physical design can result in poor performance compared to other designs. In the remainder of this section we will define the entire design space and use a set of targetted benchmarks to arrive at a final design. Finally we will show that these design decisions can lead to non-optimal decisions and suggest an automated approach is desirable.

3.1 Challenges

The design space can be divided into three challenges: selecting a strategy for log entry addressing, choosing a native I/O interface for storing log entry content, and implementing efficient metadata management.

1. **Entry addressing.** We refer to the method by which a client locates a log entry in Ceph as entry addressing, and we consider two strategies. In a one-to-one (1:1) strategy each log entry is stored in a distinct object with a name derived from the associated log entry position. This is an attractive option because it is trivial for clients to locate a log entry given its position. In contrast, an N:1 strategy *stripes* log entires across a smaller set of objects, but this adds complexity to both the client and the object interface which must multiplex a set of entries.

2. **Log entry storage.** Clients read and write binary data associated with each log entry, and these entries can be stored in the bytestream or in the key-value database associated with an object. Retrieval of log entry payloads should perform well for both large (e.g. database checkpoint) and small log entries.

3. **Metadata management.** The CORFU protocol defines the storage interface semantics, such as enforcing up-to-date epoch values and a write-once address space. The object interface constructed in Ceph must implement these semantics in software by storing metadata (e.g. the current epoch) and validating requests against this metadata (e.g. has the target position been written?). A key-value store is a natural location for this type of data, but metadata management adds overhead to each request and must be carefully designed.

In the remainder of this section we will explore the full design space defined by the cross product of these design challenges to arrive at a final design.

3.2 Baseline Performance

We begin the process of exploring the design space by exploiting the fact that metadata management is a complexity and performance overhead that a full implementation must incur beyond the costs of storing log entry data. Therefore we first explore the design space by restricting the space to log entry addressing and log entry storage (the *I/O* action shown in Figure 3). These two dimensions are represented by the first two columns of Table 2 which describes the entire design space. In the I/O column *KV* corresponds to the key-value interface, and the bytestream interface is represented by *AP* for an append strategy, and *EX* for a strategy that writes to an explicit bytestream offset (both described shortly).

Map	I/O	Entry Size	Addressing	Metadata
1:1	KV	Flex	Ceph	KV/BS
	AP	Flex	Ceph/VFS	KV/BS
N:1	KV	Flex	KV/BS	
	EX	Fixed	VFS	KV/BS
	AP	Flex	KV/BS	KV/BS

Table 2: The high-level design space of mapping CORFU log entry storage onto the RADOS object storage system.

Figure **??** shows the expected performance of 1K log appends without metadata management overhead using each of the five strategies defined in Table 2. The first thing to notice in this figure is that both one-to-one strategies have relatively poor performance. The large period of reduced throughput for *11wr* corresponds to the OSD splitting file system directories in order to maintain a maximum directory size, and will occur in *11kv* although the threshold number of objects is not reached in this example due to reduced overall throughput of the *11kv* strategy.

The second lesson that we can learn from Figure **??** is that even when using an N:1 addressing strategy, the key-value interface imposes a large overhead. This is unfortunate because the key-value interface can provide a direct solution to addressing log entries within an object. Instead, what we find is that an N:1 addressing strategy that stores log entries in the bytestream using either object appends or writes to explicit offsets outperform all other strategies by a factor of over 2x.

The apparent performance tie between the strategies of appending to an object and writing to explicit offsets can be broken by considering the flexibility offered by each approach. The third column *entry size* in Table 2 shows if a particular strategy supports storage of entries with dynamic sizes, or if entries must be restricted to a fixed size. Notably the strategy that stores log entries at explicit object offsets is limited in this regard because each log entry is effectively pre-mapped into the storage system. This leaves the clear winner: an N:1 strategy that appends log entries to objects provides flexibility and the best write performance in this particular configuration. This result is corroborated by considering the read performance for each strategy as well. Figure **??** shows the expected random read performance from a log containing 1K entries in which an addressing strategy that stores log entries in the bytestream has the best performance.

3.3 Metadata Management

The previous results show that storing log entries in the bytestream has the potential to provide the best overall performance, but by design, those results do not contain the real-world overheads introduced by metadata management such as validating that requests are tagged with an up-to-date epoch value. Having only focused on entry I/O costs, in this section we consider the overhead of the remaining actions shown in Figure 3.

1. **Epoch guard.** Each client request must be validated against the current epoch. This singleton value in infrequently updated, but the cost of accessing the value must be incurred for every request.

2. **Per-entry metadata.** Unlike the singleton epoch value, metadata associated with each log position must be read, and optionally updated for every request. For instance the *fill* operation must ensure that it is not applied to a log position that has already been written. When an operation that changes a log position is successful (i.e. *write*, *fill*, *trim*) the per-entry metadata must also be updated to reflect the change. While the size of the per-entry metadata is small, it is accessed for every operation and a single object may manage a large number of entries.

The design space for managing metadata can be large depending on the application. In many cases this space is significantly reduced when the key-value interface is used because it handles a large portion of common data management challenges such as indexing. On the other hand, we have seen that the bytestream can be used to achieve much higher performance when used in lieu of the key-value database, but leaves the design space wide open.

We consider two high-level design prototypes in this paper based on both the key-value and bytestream interfaces. The summary of performance of these designs is shown in Figure **??** where we begin by highlighting two baseline throughputs labeled *librados* and *cls_null* that correspond to the baseline append throughput described in the previous section, and the append throughput achieved with *object class* overhead included. We will discuss the performance of our two prototype designs relative to the base cost of using *object class* facility.

The first design is based on the key-value interface. In this design the epoch value is stored under a fixed key, and the metadata associated with each log entry is stored under a key derived from the log position. The expected throughput of this design is shown in Figure **??** and labeled as *cls_full_omap*. This result shows us that even for very small values the cost of managing data in the key-value store introduces significant overhead. In contrast, the unstructured bytestream interface imposes little to no restriction on how it is used.

An obvious choice for encoding the singleton epoch value in the bytestream is to position it at a fixed offset such as in a header. A method for indexing entry metadata is far less obvious. Solutions span a wide design spectrum and include approaches such as encoding a search tree into the bytestream, or exploiting the regularity log addressing and storing a dense array that can be efficiently indexed. While there is clearly a wide variety of approaches, we have chosen to examine the overhead incurred by an hypothetical best-case design in which per-entry metadata can be accessed with a single I/O. The I/O overhead of the test includes (1) read epoch value from a fixed header location, (2) read the per-entry metadata, and in the case of an update (3) write the updated per-entry metadata. The expected throughput from such a design is shown in Figure **??** labeled as *cls_full_hdr* and performs with roughly a 6% overhead.

Application-level designs also contribute to an expanded design space. In the N:1 design of mapping CORFU onto Ceph the potential amount of I/O parallelism is controlled by the value N, however if this value becomes too large there may be no benefit or performance could suffer from reduced data locality. Other aspects of application-level design are even more flexible. A wide variety of data structures could be serialized into the byte

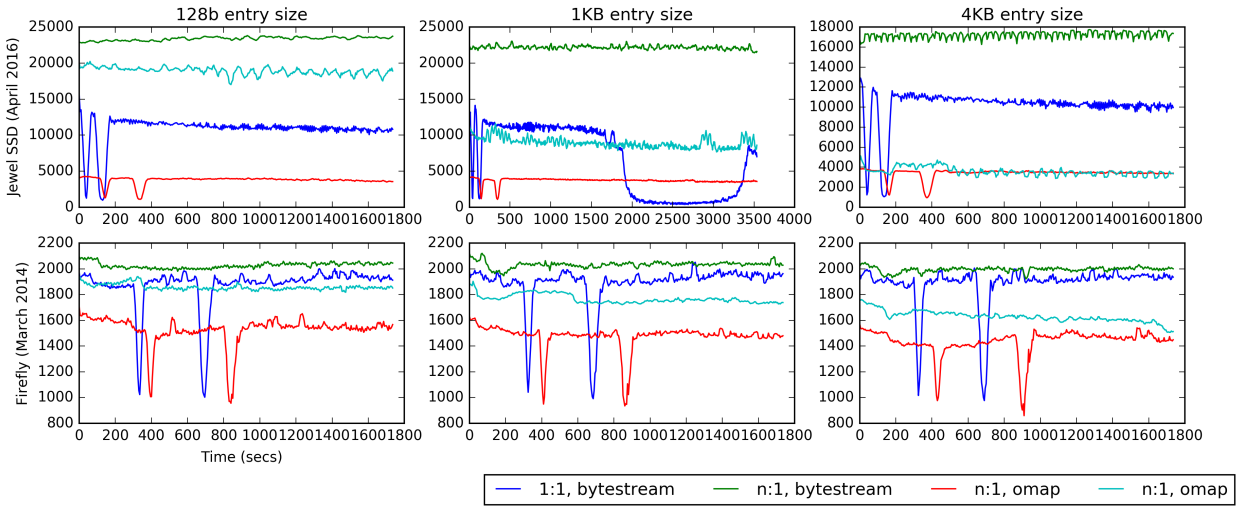


Figure 4: The cluster-specific optimizations depend heavily on the hardware and software of the storage system. This caption is quite terrible.

stream to handle metadata management tasks, but selecting a single external index that performs well across workloads and hardware may not be possible.

3.4 Cluster-Specific Optimizations

In this section we have demonstrated that a design based on an N:1 addressing scheme that stores both log entries and metadata in the bytestream can provide the best overall performance, and it does so by a large margin. However, this process of design we have outlined may not yield such clear cut results when applied in a different context. To illustrate the complexity and breadth of this design space, we describe the software, system tunables, and hardware parameters that affect the performance of ZLog.

3.4.1 Software Parameters

Ceph releases stable versions every year (Oct/Nov) and LTS versions every 3-4 months [10]. The head of the master branch moves quickly because there are over 400 contributors and an active mailing list. Over the calendar year, there were between 70 and 260 commits per week [9]. These releases contain performance fixes and new features that could directly ZLog.

Ceph Versions: to show the affect that this has on ZLog's design, we ran the same benchmark that measures expected append throughput shown in Figure **??** on identical hardware but instead use a long-term support version of Ceph released two years ago. The result of this benchmark is shown in Figure **??** in which the bytestream interface outperforms the key-value interface by 12%, compared to over 100% in the newer version of Ceph. Qualitatively, given the reduced overall throughput achieved in the older version, some developers may find that incurring the overhead of using the key-value interface is an easy decision given the reduced complexity of the design space for metadata management. If this choice had been made, then a future system upgrade could drop a significant number of iops on the floor.

Ceph Features: to show the affect of cutting-edge Ceph features, we now show results for BlueStore [26]. BlueStore is a replacement for the FileStore file system in the OSD (traditionally XFS). FileStore has performance problems with transactions and enumerations; namely the journal needed to assure atomicity incurs double writes and the file system metadata model makes object listings slow, respectively. BlueStore stores data directly on a block device and the metadata in RocksDB, which is provided by a minimalistic, non-POSIX C++ filesystem. This model adheres to the overall software defined storage strategy of Ceph because it gives the administrator the flexiblity to store the 3 componenets of BlueStore (e.g., data, RocksDB database, and RocksDB write-ahead log) on any partition on any device in the OSD.

3.4.2 System Tunables

The most recent version of Ceph (v10.2.0-1281-g1f03205) has 994 tunable parameters[1], where 195 of them pertain to the OSD itself and 95 of them focus on the OSD back end file system (i.e. its `filestore`). Ceph also has tunables for the subsystems it uses, like LevelDB (10 tunables), RocksDB (5 tunables), its own key-value

[1]This number comes from `src/common/config_opts.h` with debug options filtered out.

stores (5 tunables), its object cache (6 tunables), its journals (24 tunables), and its other optional filestores like BlueStore (49 tunables).

This many domain-specific tunables makes it almost impossible to come up with the best set of tunables, although auto-tuning like the work done in [7] could go a long way. Regardless of the technique that we use, it is clear the the number of tunables increases the physical design parameters to an unwieldy stae space size.

3.4.3 Hardware Parameters

Ceph is designed to run on a wide variety of commodity hardware as well as new NVMe devices, all of which their own set of characteristics and tunables (e.g., the IO operation scheduler type). In our experiments, we tested SSD, HDDs, NVMe devices, and

3.5 Discussion

This physical design process, of first thinking about the design trade-offs and then doing parameters sweeps, is insufficient when general-purpose systems. Had we built ZLog from the ground up (like CORFU) the story might be different but to deal wtih the large state space we need a way to automatically and autonomically decide which parameters to choose.

Next we present a declarative programming model that seeks to solve the issue of selecting an optimal mapping between application requirements and object class implementation.

4 Programming Model

Ceph is turning into a target for building distrubted services, bringing distributed programming into the hands of many new people that may have worked on the periphery of ceph.

Getting consistency and coordination correct is a challenge And as we have seen optimization at a low-level given a simple set of semantics can yield a large state space from which to draw designs Difficulty and challenges of writing distributed software. Programming distributed systems and applications.

We draw upon the recent work of Bloom a declarative lang thing

BLL (a.k.a. BRADOS) is a declarative language with relational (or tabular) semantics for RADOS objects, which corresponds to a subset of the BLOOM language (leaving out temporal logic). BLL borrows from BLOOM the table and interface collections, the instantaneous implication (¡=) operator, as well as all the collection operators.

BRADOS models the storage system state uniformly as a collection of relations. The composition of a collection of existing interfaces is then expressed as a collection of high-level *queries* that describe how a stream of requests (API calls) are filtered, transformed and combined with existing state to define streams of outputs (API call returns as well as updates to system state). Separating the relational semantics of such compositions from details of their physical implementations introduces a number of degrees of freedom, similar to the “data independence offered by database query languages. The choice of access methods (for example, whether to use a bytestream interface or a key/value store), storage decide classes(e.g., whether to use HDDs or SDDs), physical layout details (e.g. a 1:1 or 1:N mapping) and execution plans (e.g. operator ordering) can be postponed to execution time. The optimal choices for these physical design details are likely to change much more often than the logical specification, freeing the interface designer from the need to rewrite systems and device and interface characteristics change.

4.1 The CORFU Interface in Bloom

```
state do
  table :epoch, [:epoch]
  table :log, [:pos] => [:state, :data]

  interface input,  :op,
[:type, :pos, :epoch] => [:data]
  interface output, :ret,
[:type, :pos, :epoch] => [:retval]

  scratch :write_op, [:type, :pos, :epoch] => [:data]
  scratch :read_op,  [:type, :pos, :epoch] => [:data]
  scratch :trim_op,  [:type, :pos, :epoch] => [:data]
  scratch :fill_op,  [:type, :pos, :epoch] => [:data]
  scratch :seal_op,  [:type, :pos, :epoch] => [:data]

  # op did or did not pass the epoch guard
  scratch :valid_op,   [:type, :pos, :epoch] => [:data]
  scratch :invalid_op, [:type, :pos, :epoch] => [:data]

  # op's position was or was not found in the log
  scratch :found_op,
[:type, :pos, :epoch] => [:data]
  scratch :notfound_op, [:type, :pos, :epoch] => [:data]
end

bloom do
  # op's position found in log
  found_op <= (valid_op * log).lefts(pos => pos)
  notfound_op <= valid_op.notin(found_op)

  # demux on operation type
  write_op <= valid_op { |o| o if o.type == 'write' }
  read_op  <= valid_op { |o| o if o.type == 'read'  }
  fill_op  <= valid_op { |o| o if o.type == 'fill'  }
  trim_op  <= valid_op { |o| o if o.type == 'trim'  }
  seal_op  <= valid_op { |o| o if o.type == 'seal'  }
end
```

Listing 1: Epoch Guard

Listing 2 shows the epoch guard that is applied to all operations. The guard rejects requests that are tagged with old epoch values, ensuring that a client generating

a request has an up-to-date view of the system. First the `invalid_op` collection is defined to include the current operation if its epoch value is no larger than the stored epoch value. Next the `valid_op` collection is defined to be the inverse of `invalid_op` and is a helper used to refine other operations later in the dataflow. Finally we handle the case for all operations tagged with an out-of-date epoch by merging the `invalid_op` set into the output `ret` collection.

```
bloom :epoch_guard do
  invalid_op <= (op * epoch).pairs{|o,e| o.epoch <= e.epoch}
  valid_op <= op.notin(invalid_op)
  ret <= invalid_op{|o| [o.type, o.pos, o.epoch, 'stale']}
end
```

Listing 2: Epoch Guard

The process of sealing an object requires installing a new epoch value and returning the current maximum position written. Listing 3 implements the seal interface by first removing the current epoch value and replacing it with the epoch value contained in the operation. Next an aggregate is computed over the log to find the maximum position written, and this value is returned.

```
bloom :seal do
  epoch <- (seal_op * epoch).rights
  epoch <+ seal_op { |o| [o.epoch] }
  temp :maxpos <= log.group([], max(pos))
  ret <= (seal_op * maxpos).pairs do |o, m|
    [o.type, nil, o.epoch, m.content]
  end
end
```

Listing 3: Seal

Trimming a log entry always succeeds. In Listing 4 The <+− operator simulatenously removes the log entry with the given position and replaces it with an entry with its state set to *trimmed*.

```
bloom :trim do
  log <+- trim_op{|op| [op.pos, 'trimmed']}
  ret <=  trim_op{|op| [op.type, op.pos, op.epoch, 'ok']}
end
```

Listing 4: Trim

The write and fill interfaces are implemented similarily, and are show in Listing 5 and Listing 6, respectively. A valid write collection is created if the operation position is not found in the log. A valid write operation is then merged into log, otherwise a read only error is returend indicating that hte log position was already written to. The fill operation is identical except a the fill state is set on the log entry.

```
bloom :write do
  temp :valid_write <= write_op.notin(found_op)
  log <+ valid_write { |o| [o.pos, 'valid', o.data] }
  ret <= valid_write { |o| [o.type, o.pos, o.epoch, 'ok'] }
  ret <= write_op.notin(valid_write) { |o|
    [o.type, o.pos, o.epoch, 'read-only']
  }
end
```

Listing 5: Write

```
bloom :fill do
  temp :valid_fill <= fill_op.notin(found_op)
  log <+ valid_fill { |o| [o.pos, 'fill'] }
  ret <= valid_fill { |o| [o.type, o.pos, o.epoch, 'ok'] }
  ret <= fill_op.notin(valid_fill) { |o|
    [o.type, o.pos, o.epoch, 'read-only']
  }
end
```

Listing 6: Fill

The read interface is shown in Listing 7, and structured similar to the write interface. First we create a collection containing a valid read operation that is in the log and does not have the filled or trimmed state set. The data is returend in the case of a valid read op, otherwise an error is returend through the output interface.

```
bloom :read do
  temp :valid_read <= (read_op * log).pairs(pos => pos) { |o, l|
    [o.type, o.pos, o.epoch, l.data] unless
      ['filled', 'trimmed'].include?(l.state)
  }
  ret <= valid_read { |e| [e.type, e.pos, e.epoch, e.data] }
  ret <= read_op.notin(valid_read, type => type) do |o|
    [o.type, o.pos, o.epoch, 'invalid']
  end
end
```

Listing 7: Read

4.2 Optimizer

4.3 Other Interfaces

The subset of the Bloom language we are using is powerful enough to express the CORFU language, as well as many of the object class interfaces that are already included in Ceph. We briefly sketch the Bloom imlementation of an object class that provides a locking service. This is an interesting example because Bloom implementations use mechanisms for introducing points of coordination in execution, which in the context of Ceph, may be written in Bloom itself.

```
bloom :take_lock do
end

bloom :break_lock do
end

bloom :list_locks do
end

bloom :lock_info do
end
```

Listing 8: Hello

5 Evaluation

6 Related Work

Active storage. There is a wide variety of work related *active storage* that seeks to increase performance and reduce data movement by exploiting remote storage resources. This concept has been applied in the context of on-disk controllers and object-based storage (T10) [23, 14, 30]. Solutions to safety concerns have examined using managed lanuages, sandbox technologies, as well as restricting extension installation to vendors [19, 30, 24].

Techniques for remote storage processing have been applied in application-specific domains for database and file system acceleration, as well as in remote data filtering in cloud-based storage environments [25, 13, 21, 17] Others have collected and used statistics to optimize the location of computation in general workloads as well as database systems [12, 11, 22].

Most closely related to our work are efforts that consider specific programming models in the context of active storage such as a stream-based model [4], a model that assisted in optmiizing balancing computation [29], and optimization that took advantage of read-only data to reorder operations [18]. While our goals are similar to previous work, we start with a declarative language that will enable us to apply optimization techniques beyond the limited domain-specific optimizations found in previous work.

7 Conclusion and Future Work

Include performance management topic in future work.

References

[1] Merged pull request for cls_numops. https://github.com/ceph/ceph/pull/4869. Accessed: 2016-05-12.

[2] OpenStack Open Source Cloud Computing Software. http://www.openstack.org. Accessed: 2016-05-12.

[3] Pull request for cls_lua. https://github.com/ceph/ceph/pull/7338. Accessed: 2016-05-12.

[4] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM.

[5] Mahesh Balakrishnan et al. Tango: Distributed data structures over a shared log. In *SOSP '13*, Farmington, PA, November 3-6 2013.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. Corfu: A shared log design for flash clusters. In *NSDI'12*, San Jose, CA, April 2012.

[7] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra, Ruth Aydt, Quincey Koziol, Marc Snir, et al. Taming parallel i/o complexity with auto-tuning. In *Proceedings of the International Conference on High mance Computing, Networking, Storage and Analysis*, page 68. ACM, 2013.

[8] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder – a transactional record manager for shared flash. In *CIDR '11*, Asilomar, CA, January 9-12 2011.

[9] Ceph. Ceph github repository. https://github.com/ceph/ceph/graphs/commit-activity, May 2016.

[10] Ceph. Ceph releases. http://docs.ceph.com/docs/jewel/releases/, May 2016.

[11] Chao Chen and Yong Chen. Dynamic active storage for high performance i/o. In *ICPP '12*, 2012.

[12] Chao Chen, Yong Chen, and Philip C. Roth. Dosas: Mitigating the resource contention in active storage systems. In *CLUSTER '12*, 2012.

[13] Steve Chiu, Wei-keng Liao, and Alok Choudhary. *Computational Science — ICCS 2003: International Conference, Melbourne, Australia and St. Petersburg, Russia, June 2–4, 2003 Proceedings, Part IV*, chapter Design and Evaluation of Distributed Smart Disk Architecture for I/O-Intensive Workloads, pages 230–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[14] D. H. C. Du. Intelligent storage for information retrieval. In *International Conference on Next Generation Web Services Practices (NWeSP'05)*, pages 7 pp.–, Aug 2005.

[15] Facebook. RocksDB. http://rocksdb.org, 2013.

[16] Sanjay Ghemawat and Jeff Dean. LevelDB. http://code.google.com/p/leveldb, 2011.

[17] Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson, Dushyanth Narayanan, Florin Dinu, and Antony Rowstron. Rhea: Automatic filtering for unstructured cloud storage. In *NSDI'13*, Lombard, IL, April 2-5 2013.

[18] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, * Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *FAST '04*, 2004.

[19] T. M. John, A. T. Ramani, and J. A. Chandy. Active storage using object-based devices. In *2008 IEEE International Conference on Cluster Computing*, pages 472–478, Sept 2008.

[20] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[21] Hyeran Lim, Vikram Kapoor, Chirag Wighe, and David H.-C. Du. Active disk file system: A distributed, scalable file system. In *MSST '08*, 2008.

[22] L. Qiao, V. Raman, I. Narang, P. Pandey, D. Chambliss, G. Fuh, J. Ruddy, Y. L. Chen, K. H. Yang, and F. L. Lin. Integration of server, storage and database stack: Moving processing towards data. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1200–1208, April 2008.

[23] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *24th international Conference on Very Large Databases (VLDB '98)*, New York, NY, 1998.

[24] M. T. Runde, W. G. Stevens, P. A. Wortman, and J. A. Chandy. An active storage framework for object storage devices. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012.

[25] M. Uysal, A. Acharya, and J. Saltz. Evaluation of active disks for decision support databases. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 337–348, 2000.

[26] Sage A. Weil. BlueStore: A New, Faster Storage Backend for Ceph, April 2016.

[27] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI'06*, Seattle, WA, November 2006.

[28] Sage A. Weil, Andrew Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. Reno, NV, November 2007.

[29] R. Wickremesinghe, J.S. Chase, and J.S. Vitter. Distributed computing with load-managed active storage. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC 2002)*, pages 13–23, 2002.

[30] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Darrell D. E. Long, Yangwook Kang, Zhongying Niu, and Zhipeng Tan. Design and evaluation of oasis: An active storage framework based on t10 osd standard. In *MSST '11*, Denver, CO, April 24-29 2011.