brados: declarative, programmable object storage

Noah Watkins, Michael Sevilla, Ivo Jimenez,
Neha Ojha, Peter Alvaro, Carlos Maltzahn
University of California, Santa Cruz

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 [36, 37]. 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].

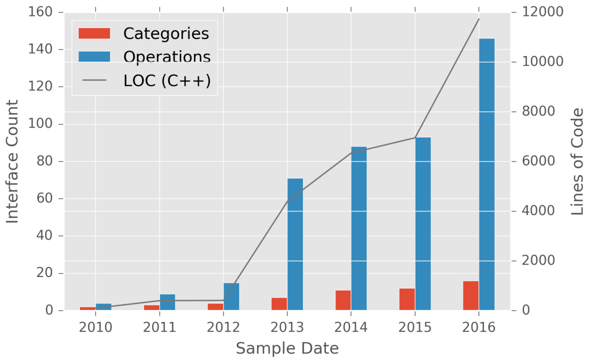


Figure 1: [source] Growth of officially supported, custom object interfaces in RADOS over 6 years. An *operation* is a function executed in the context of an object, and operations are grouped into different *categories* corresponding to applications or utilities, such as reference counting

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 inclusion of a mechanism for dynamically defining extensions using Lua [3] (currently pending review) 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.

Unfortunately the size of the design space for even simple interfaces can be very large, and it can be difficult to choose a design that future proofs an implementation against hardware and software upgrades. Previous work related to storage interface design has largely been in the context of standardization efforts and active storage. While the former is primarily concerned with fixed interfaces, application-specific interfaces built using active storage techniques have not to the best of our knoweldge addressed portability concerns that arise from defining interfaces imperatively. We address this gap by using a declarative language we call *brados* to define application-specific storage interfaces in object-based storage systems, decoupling interface definition from implementation and allowing interfaces to adapt to system changes without modification.

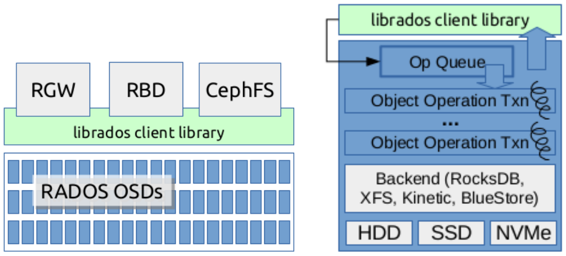
To demonstrate the use of the *brados* language we will show how a high-performance shared-log based on the CORFU protocol can be built in Ceph, as well as existing real-world interfaces used in the Ceph storage system. First we discuss the Ceph storage system which our prototype is built upon and the motivating system, CORFU. We then present the challenges that programmers face when navigating the design space during the process of building an application-storage interface. The *brados* language is then described using the CORFU system and an existing Ceph-specific interface as motivating examples. We conclude by discussing the optimization opportunities that we can realize using analysis of interfaces.

2 Background

In this section we describe the salient components of Ceph that we use to construct application-specific extensions, and provide an overview of the CORFU distributed shared-log which is the primary motivating example used in this paper. The content of this section sets the stage for a discussion of the complexities developers face building application-specific storage interfaces.

2.1 Ceph Basics and Storage Interfaces

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.



(a) Ceph Cluster **(b)** OSD

Figure 2: The Ceph cluster is composed of object storage devices (OSD) that provide access to objects on behalf of client requests. An OSD supports a wide variety of hardware and software configurations, as well as the transactional composition of object operations for defining application-level interfaces.

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 an isolated database associated with each object, and the bytestream interface supports random byte-level access similar to a standard file interface. At a low-level each of these abstract I/O interfaces map to hardware storage devices through a pluggable object backend shown in Figure 2b. 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 [19, 18] via XFS or other supported file systems. Several backend implementations exist for storing data in a range of targets such as HDD, SSD, as well as Ethernet-attached disks, and NVMe devices using optimized access libraries.

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 [7]. 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 [24]. 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 [6, 10].

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.

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.

Storage Programmability 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 in Section 2.1. Thus, by using software-based object interfaces offered by RADOS flash devices in CORFU can be replaced by software-defined interfaces 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.

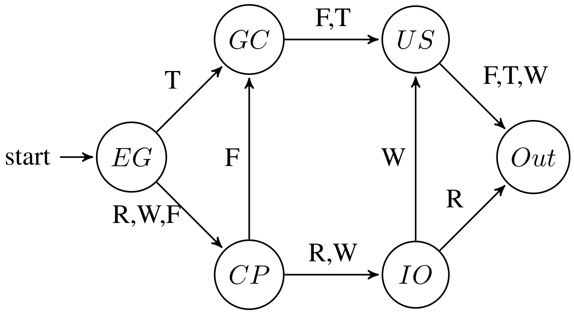


Figure 3: State transition diagram for read (R), write (W), fill (F), and trim (T) CORFU operations. The states epoch guard (EG), check position (CP), and update state (US) access metadata. The I/O performs a log entry read or write, and garbage collection (GC) marks entries for reclamation.

3 Physical Design Space

As we have seen, Ceph provides a rich storage API and places few requirements 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 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 focusing on 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).

The top row of Figure 4 shows the expected performance of different sized log appends without metadata

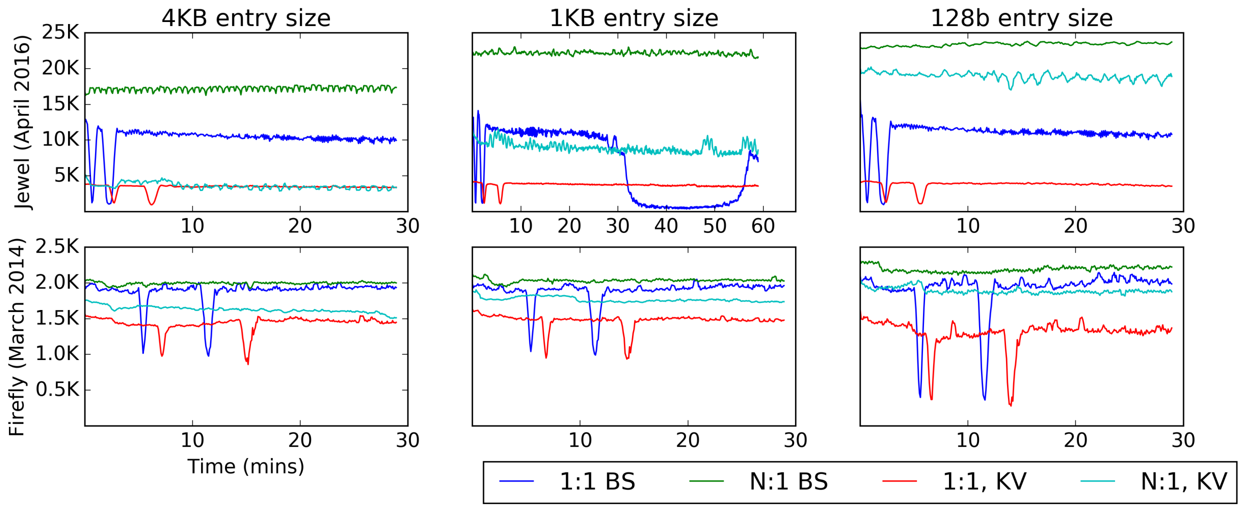


Figure 4: The cluster-specific optimizations impact performance... a lot. This caption is quite terrible.

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.

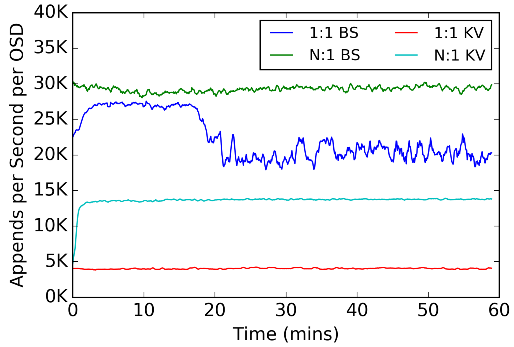


Figure 5: The expected random read performance from a log with 1KB entries performs best on the same strategy that the writes perform best on - the (N:1, BS) append strategy.

management overhead using each of the five strategies defined in Table 2. Starting with Jewel and 4KB entry sizes, we see that both one-to-one strategies have relatively poor performance. Jewel with 1KB entry sizes sees a 2.5× performance increase for the (N:1,KV) mapping. The large period of reduced throughput for corresponds to the OSD splitting file system directories in order to maintain a maximum directory size, and will occur in (1:1, KV) although the threshold number of objects is not reached in this example due to the reduced overall throughput of the (1:1, KV) strategy.

The second lesson that we can learn from the Jewel row of Figure 4 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 2×.

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 5 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

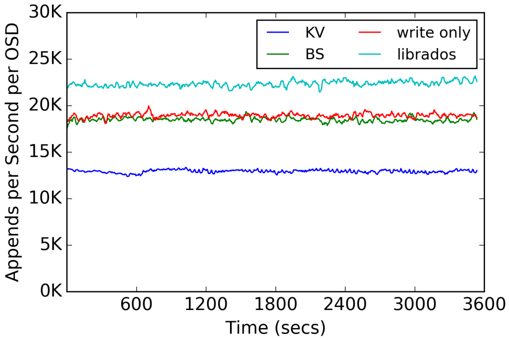


Figure 6: 1hr,1k,cls_wr,jewel: this graph shows the overhead of class

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 6 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 6 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 6 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 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 these contexts, we describe the software, system tunable, and hardware parameters that affect the performance of our CORFU implementation. The complexity and breadth of our parameter sweep, and the fact that our state space grew so large, motivates brados in section 4.

3.4.1 Software Parameters

Ceph releases stable versions every year (Oct/Nov) and LTS versions every 3-4 months [12]. The head of the master branch moves quickly because there are over 400 contributors and an active mailing list. Over the past calendar year, there were between 70 and 260 commits per week [11].

Ceph Versions: to show the evolution of Ceph's performance and behavior, we ran the same benchmark, which measures append throughput, with the same configurations, hardware, and tunables on 2 versions of Ceph: Jewel (April 2016) - the newest stable release and Firefly (May 2014) - a long term support version that introduced cache tiering and erasure coding. The Jewel row of Figure 4 shows that the size of the log entries has a clear impact on performance but the Firefly release shows no such trend. The fact that the appends per second are an order of magnitude slower on Firefly indicates that the the benchmark is bottlenecked by the Ceph software instead of the external log implementation.

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 [35]. 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.

Takeaway: choosing the best implementations is dependent on both the timing of the developemnt (Ceph Version) and the expertise of the administrator (Ceph Features). Different versions and features of Ceph may lead the administrator to choose a suboptimal implementation for the system's next upgrade. The software parameters must be accounted for and benchmarked when making design decisions.

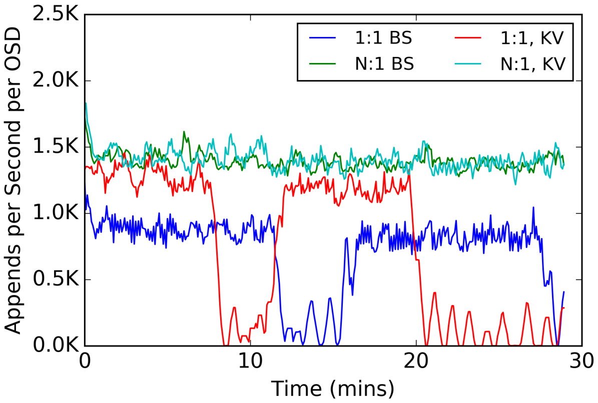


Figure 7: shows that with a HDD, it don't matter if you choose omap!!!!!

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 subystems it uses, like LevelDB (10 tunables), RocksDB (5 tunables), its own key-value 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 [9] 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.

Takeaway: the number and complexity of Ceph's tunables make brute-force parameter selection impossible.

3.4.3 Hardware Parameters

Ceph is designed to run on a wide variety of commodity hardware as well as new NVMe devices. All these devices have their own set of characteristics and tunables (e.g., the IO operation scheduler type). In our experiments, we tested SSD, HDDs, NVMe devices and discovered a wide range of behaviors and performance profiles. As an example, Figure 7 shows the write performance of 128 byte log entries using Jewel and a single HDD. Performance is 10× slower than its SSD counterpart in Figure 4 (top row, third column) but the behavior and relative performance make this hardware configuration especially tricky.

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

The behavior of the 1:1 implementations shows throughput drops lasting for minutes at a time – this limits our focus to the N:1 implementations. The performance of `(N:1, BS)` implementation is almost identical to `(N:1, KV)` (within 1% mean throughput). Regardless of the true bottleneck, it is clear that choosing `(N:1, KV)` is the better choice because of the resulting implementation should be less complex and there is minimal performance degradation.

Takeaway: choosing the best implementations is dependent on hardware. Something as common as an upgrade from HDD to SSD may nullify the benefits of a certain implementation.

3.5 Discussion

What started out as a pragmatic approach to designing a complex service on Ceph quickly proved to be unwieldy. While finding the best solution for a specific set of configurations (i.e. software, tunable, and hardware parameters) can be done, finding a solution that is future-proof is still unsolved.

This physical design process, of first thinking about the design trade-offs and then doing parameters sweeps, is insufficient for large general-purpose systems like Ceph. Had we built ZLog from the ground up (like CORFU) the story might be different but to deal with 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 The Brados Programming Model

As Ceph continues to develop into a target for building distributed services and applications, it is important to not restrict use to developers seasonsed in distributed programming and those with knowledge of the intracacies of Ceph and its performance model. To this end we present Brados, a declarative language with relational semantics for defining the behavior of RADOS object interfaces. Brados uses a simple declarative data model, can reduce the learning curve for new users, and allows existing developers to increase productivity by writing less code that is more portable.

The Brados language corresponds to a subset of the Bloom language, and models the storage system state uniformly as a collection of relations [5]. 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 device classes (e.g., whether to use HDDs or SDDs), physical layout details (e.g. a 1:1 or N:1 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 Basics

4.2 The CORFU Log Interface

We now show the implementation of CORFU using Brados and use this as a vehicle for describing additional aspects of the language of semantics. Listing 1 shows the declaration of state for the CORFU interface. Lines 1 and 2 define the schema of the two persistent collections that hold the current epoch value, and the log contents. These collections are mapped onto storage within Ceph but abstract away the low-level interface (e.g. bytestream vs key-value). Lines 5-9 define the input and output interfaces. We use a generic schema for the input operation to simplify how rules are defined that apply to all operation types. Lines 11-15 define named collections for each operation type. The scratch type indicates that the data is not persistent, and only remains in the collection for a single execution time step. The remaining scratch collections are defined to further subdivide the operations based on different properties which we'll describe next.

```
1  state do
2    table :epoch, [:epoch]
3    table :log, [:pos] => [:state, :data]
4
5    interface input, :op,
6      [:type, :pos, :epoch] => [:data]
7
8    interface output, :ret,
9      [:type, :pos, :epoch] => [:retval]
10
11   scratch :write_op, op.schema
12   scratch :read_op,  op.schema
13   scratch :trim_op,  op.schema
14   scratch :fill_op,  op.schema
15   scratch :seal_op,  op.schema
16
17   # op did or did not pass the epoch guard
18   scratch :valid_op,   op.schema
19   scratch :invalid_op, op.schema
20
21   # op's position was or was not found in the log
22   scratch :found_op,    op.schema
23   scratch :notfound_op, op.schema
24 end
```

Listing 1: State Declaration

Initialization is performed in Listing 2 which acts as a demux for the operation type and properties. Lines 3-7 show 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.

Lines 10 and 11 populate the found_op and notfound_op collections that allow operations behavior to be predicated on if the position associated with a request is found in the log. Finally the remaining lines in Listing 2 populate each of the specific operation collections.

```
1  bloom do
2    # epoch guard
3    invalid_op <= (op * epoch).pairs{|o,e|
4      o.epoch <= e.epoch}
5    valid_op <= op.notin(invalid_op)
6    ret <= invalid_op{|o|
7      [o.type, o.pos, o.epoch, 'stale']}
8
9    # op's position found in log
10   found_op <= (valid_op * log).lefts(pos => pos)
11   notfound_op <= valid_op.notin(found_op)
12
13   # demux on operation type
14   write_op <= valid_op {|o| o if o.type == 'write'}
15   read_op  <= valid_op {|o| o if o.type == 'read'}
16   fill_op  <= valid_op {|o| o if o.type == 'fill'}
17   trim_op  <= valid_op {|o| o if o.type == 'trim'}
18   seal_op  <= valid_op {|o| o if o.type == 'seal'}
19 end
```

Listing 2: Setup

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 input operation. Next an aggregate is computed over the log to find the maximum position written, and this value is returned, typically to a client performing a reconfiguration of the system or following the failure of a sequencer.

```
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*. In practice the removal of a log entry may trigger garbage collection, but we model it here as an update for brevity.

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

Listing 4: Trim

The write and fill interfaces are implemented similarly, and are both shown in Listing 5. 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 the 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

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 5: Write and Fill

Finally the read interface is shown in Listing 6, and structured in a similar way to the write and fill interfaces. First we create a collection containing a valid read operation (named ok_read) that is in the log and does not have the filled or trimmed state set. The data read from the log is returend in the case of a valid read operation, otherwise an error is returend through the output interface.

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

Listing 6: Read

Amazingly these few code snippets express the semantics of the entire storage device interface requirements in CORFU. For reference the ZLog implementation of the CORFU storage interface in C++ comprises nearly 700 lines of code and was designed to store log entries and metadata in the key-value interface, thus requiring a lengthy rewrite to realize the performance advantage available by using the bytestream interface, as was highlighted in Section **??**. Furthermore, the complexity introduced by using the bytestream interface would grow the amount of code written in the optimized version.

But beyond the convience of writing less code, it is far easier for the programmer writing an interface such as CORFU to convinence herself of the correctness of the high-level details of the implementation without being distracted by issues related to physical design or the many other gotchas that one must deal with when writing low-level systems software.

4.3 Optimization

Choosing implementation details - the refcount example is good. small data kept in ceph object bytestream until it becomes large then use omap.

- could use the dataflow graph to do interesting things like understand how to do prefecting of things like keys or data that are often only known at execution time

4.4 Other Interfaces

As we have seen the subset of Bloom used in Brados is powerful enough to express the semantics of the CORFU storage interface. We briefly sketch the implementation of an additional interface found in Ceph that used by the S3-compatible RADOS-Gateway service to perform object reference counting.

The reference count implementation that we describe does not use a typical counter, but rather uses a tag-based interface in which each *reference* to an object is identified by a application-level tag. Each tag represents one or more references supporting an idempotent property useful for applications that may replay reference count operations out of order. Otherwise the semantics are straightforward: clients record references to an object by inserting a tag, and remove a reference by removing a tag. When the number of references (i.e., tags) drop to zero the object is garbage collected.

The reference implementation in Ceph is 300 LOC written in C++. The version written in Brados is shown in Listing 7. Line 10 shows the implementation of requesting a reference which is an idempotent merge of the tag into the persistent `refs` collection. Unlike the CORFU interface that took highlighted how collections are unified with persistent storage, the reference counting interface requires access to RADOS runtime facilities that allow objects to be removed. This is modeled in Brados using the Bloom *channel* abstraction that is used to transmit facts between entities in the system; we effectively send a message to the runtime with our removal request. The `remove_chn` channel may be populated with messages and the RADOS runtime will processes these asynchronously.

```
1  state do
2    table :refs, [:tag]
3    channel :remove_chn
4    interface input, :get_op, [:tag]
5    interface input, :put_op, [:tag]
6    interface output, :ret, [:status]
7  end
8
9  bloom :get do
10   refs <= get_op
11   ret  <= get_op{|op| ['ok']}
12 end
13
14 bloom :put do
15   temp :found <= (put_op * refs).matches
16   refs <- found
17   ret <= put_op{|op| ['ok']}
18
19   temp :remove <= found.notin(refs)
20   remove_chn <~ remove
21 end
```

Listing 7: Reference counting interface

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) [30, 17, 39]. Solutions to safety concerns have examined using managed lanuages, sandbox technologies, as well as restricting extension installation to vendors [23, 39, 31].

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 [34, 15, 26, 20] Others have collected and used statistics to optimize the location of computation in general workloads as well as database systems [14, 13, 29].

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 [38], and optimization that took advantage of read-only data to reorder operations [22]. 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.

Declarative Specifications. Declarative interfaces have been used for specifying distributed analytic tasks [28, 33], information extraction [32] and distributed systems [16, 8]. The use of declarative specifications has also been explored in other contexts such as cloud recovery testing [21], bug reproduction [25] and cloud resource orchestration [27].

7 Conclusion and Future Work

Much work remains to be done. While we demonstrated some of the immediate benefits of separating logical and physical specifications of component compositions, the “dream of data independence is to take the physical design question completely off the table for programmers. We are currently exploring both cost-based optimizers and autotuners to automate the selection of a high performance physical design consistent with a given logical specification.

The database literature on materialized view maintenance and selection can be brought to bear in the context of programmable storage as well. Recall the sequencer logic of Corfu described in Section XX. The Bloom specification (Figure YY) simply states that the current value of the sequencer is the result of evaluating an aggregate query over the log relation. This means that the system may choose any strategy to maintain a sequence value that is consistent with this constraint. The physical design decision to store and update a memory-resident sequence value rather re-computing the maximal value at each request is a view materialization decision that could similarly be made by a compile-time optimizer.

The future of programmable storage is likely to witness a further decomposition of services to allow greater flexibility in customizing deployments. More than likely, not all of these services will assume or provide strongly-consistent operations by default. Instead, programmers will have the flexibility (and burden!) of trading off between performance (offered by weakly consistent subsystems) and programmability. Languages such as Bloom—which provide static analyses to determine when loosely-ordered operations nevertheless produce consistent outcomes—will become increasingly relevant in this domain.

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] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR '11*, 2011.

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

[7] 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.

[8] Michael Edward Bauer. *Legion: Programming Distributed Heterogeneous Architectures with Logical Regions*. Doctoral, Stanford University, Palo Alto, California, 2014.

[9] 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.

[10] 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.

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

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

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

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

[15] 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.

[16] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 1:1–1:14, New York, NY, USA, 2012. ACM.

[17] 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.

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

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

[20] 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.

[21] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 238252, Berkeley, CA, USA, 2011. USENIX Association.

[22] 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.

[23] 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.

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

[25] Kaituo Li, Pallavi Joshi, Aarti Gupta, and Malay K. Ganai. ReproLite: A lightweight tool to quickly reproduce hard system bugs. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 25:125:13, New York, NY, USA, 2014. ACM.

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

[27] Changbin Liu, Boon Thau Loo, and Yun Mao. Declarative automated cloud resource orchestration. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, page 26:126:8, New York, NY, USA, 2011. ACM.

[28] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08.

[29] 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.

[30] 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.

[31] 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.

[32] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07.

[33] A. Thusoo, J.S. Sarma, N. Jain, Zheng Shao, P. Chakka, Ning Zhang, S. Antony, Hao Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.