Wonderful : A Terrific Application and Fascinating Paper

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

Your Abstract Text Goes Here. Just a few facts. Whet our appetites.

1 Introduction

As large-scale unified storage systems evolve to meet the requirements of next-generation hardware and an increasingly diverse set of applications, *de jure* approaches of the past—based on standardized interfaces—are giving way domain-specific interfaces and optimizations. While promising, current approaches to co-design are based on ad-hoc strategies that are untenable.

The standardization of the POSIX I/O interface has been a major success, allowing application developers to avoid vendor lock-in, while providing isolation to system developers. However, large-scale storage systems have been dominated by proprietary offerings, preventing exploration of alternative interfaces and complicating migration paths. Recently we have seen an increase in the number of special-purpose storage systems, including high-performance open-source storage systems that are modifiable, enabling system changes without fear of vendor lock-in. Unfortunately, evolving storage system interfaces is a challenging task that involves domain expertise and requires programmers to forfeit the protection from change afforded by narrow interfaces.

Malacology [1] is a recently proposed storage system that advocates for an approach to co-design called *programmable storage*. The approach is based on exposing low-level functionality as reusable building blocks, allowing developers to custom-fit their applications to take advantage of the code-hardened capabilities of the underlying system and avoid duplication of complex and error-prone services. By recombining existing services in the Ceph storage system, Malacology demonstrated how two real-world services could be constructed. Unfortunately, implementing applications on top of a system like Malacology can be an ad-hoc process that is difficult to reason about and manage.

Despite the powerful approach advocated by Malacology, it requires programmers to navigate a complex design space, simultaneously addressing often orthogonal concerns including functional correctness, performance, and fault-tolerance. Worse still, the domain expertise required to build a performant interface can be quickly lost because interface composition is sensitive to changes in the underlying environment such as hardware and software upgrades as well as evolving workloads common place in unified storage systems.

To address this challenge we are actively exploring the use of high-level declarative languages based on Datalog to program storage APIs. By specifying the functional behavior of a storage interface once in a relational (or algebraic language), an optimizer can use a cost model to explore a space of functionality equivalent physical implementations. Much like query planning and optimization in database systems, this approach will separate the concerns of correctness and performance, protecting applications against changes. But despite the parallels with database systems, the design space is quite different.

In this paper we demonstrate the challenge of programmable storage by showing the sensitivity of domain-specific interfaces to changes in the underlying system. We then show that the relational model is able to capture the functional behavior of a popular shared-log service, and finally we explore a few additional optimizations that can be utilized to expand the space of possible implementations.

2 The Programmable Storage Challenge

When application requirements are not met by an underlying storage system the most common approach is to design a workaround that will fall roughly into one of three categories:

Extra services. 'Bolt-on" services are intended to improve performance or enable a feature, but come at the expensive of additional sub-systems and dependencies that the application must manage, as well as trust.

Application changes. The second approach to adapting to a storage system deficiency is to change the application itself by adding more data management intelligence into the application or as domain-specific middleware. When application changes depend on non-standard semantics exposed by the storage system the coupling that results can be fragile and result in lock-in.

Storage modifications. When these two approaches fail to meet an application's needs, developers may turn their attention to any number of heavy-weight solutions ranging from changing the storage system itself, up to and including designing entirely new systems. This approach requires significant domain knowledge, and extreme care when altering code-hardened systems.

Rather than relying on storage systems to evolve *or* applications to change, a hybrid approach that embraces interface instability allows maximum flexibility, so long as it does not impose an unmanagable burden on developers.

2.1 Programmable Storage

Malacology is a recently proposed approach that advocates a design strategy called programmable storage in which existing storage system services are safely exposed such that they can be composed to form application specific services. Figure 1 shows the architecture of Malacology as implemented in Ceph, which exposes a variety of low-level internal services such object interfaces, and cluster metadata management that are in-turn composed to form a set of new system services that support the requirements of multiple real-world applications.

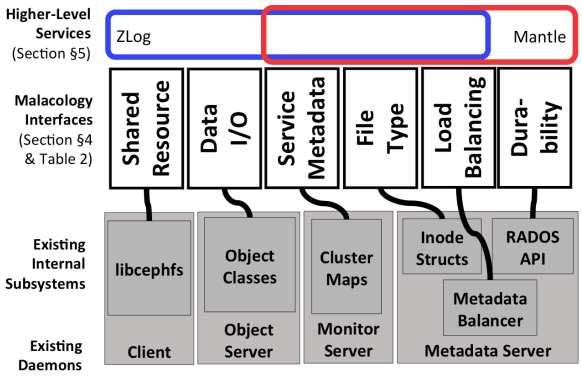

Figure 1: Malacology implementation in Ceph. Existing sub-systems are composed to form new services and application-specific optimizations.

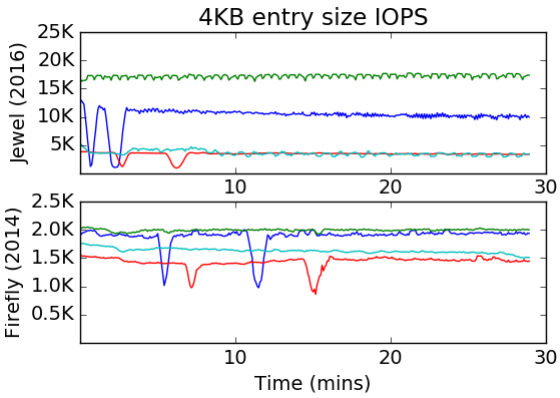

Figure 2: asdf

The Malacology services exposed in Ceph were demonstrated by implementing the CORFU high-performance shared-log abstraction. The design re-used metadata management caching features to construct the high-frequncy network-attached counter, and software-defined object interfaces replicated the write-once semantics of the storage devices, both of which are required by the CORFU protocol. While the ability to build software-defined interfaces is powerful, next we will show how access to low-level interfaces can be a double-edged sword, providing power at the cost of maintenance.

2.2 Everyone Loves A Standard

The narrow interface exposed by storage systems has been a boon in allow systems and applications to evolve independently, in affect limiting the size of the design space where applications couple with storage. Programmable storage lifts the veil the system, and with it, forces applications developers to confront a large set of possible designs.

To illustrate this challenge we implemented as an object interface the CORFU storage device specification, that is a write-once interface over a 64-bit address space. The interface is used as a building block of the CORFU protocol to read and write log entries that are striped over an entire cluster. The implementations differ in their optimization strategy of utilizing internal system interfaces. For instance one implementation uses a key-value interface to manage the address space index and entry data, while another implementation stores the entry data using a byte-addressable interface.

Figure 2 shows the append throughput of four such implementations run on two versions of Ceph from 2014 and 2016. The first observation to be made is that performance in general is significantly better in the newer version of Ceph. However, what is interesting is the re-

lationship between the implementations. Run on a version of Ceph from 2014, the top two implementations perform with nearly identical throughput, but have strikingly different implementation complexities. The performance of the same implementations on a newer version of Ceph illustrate a challenge: given a reasonable choice of a simpler implementation in 2014, a storage interface will perform worse in 2016, requiring significant rework of low-level interface implementations.

3 Design Space

It is easy to underestimate the scale of the design space, even for a relatively simple interface such as a distributed shared-log.

3.1 Software Parameters

Ceph releases stable versions every year (Oct/Nov) and long-term support (LTS) versions every 3-4 months [**?**]. 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 [**?**].

Ceph constantly adds new features and one particular feature that has the potential to greatly improve the performance of log appends is BlueStore [**?**]. 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 flexibility to store the 3 components 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 development (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.

3.1.1 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 stores (5 tunables), its object cache (6 tunables), its journals (24 tunables), and its other optional object stores 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 [**?**] could go a long way. Regardless of the technique that we use, it is clear the number of tunables increases the physical design parameters to an unwieldy state space size.

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

3.1.2 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 **??** 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 **??** (top row, third column) but the behavior and relative performance make this hardware configuration especially tricky.

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.

4 Programming Model

It is important to not restrict use to developers seasoned in distributed programming and those with knowledge

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

of the intricacies of Ceph and its performance model. Any hard-coded imperative method for build storage interfaces today restrict the optimizations that can be performed automatically or derived from static analysis. To this end we present a declarative programming model that can reduce the learning curve for new users, and allow existing developers to increase producitivty by writing less code that is more portable.

The model corresponds to a subset of the Bloom language which a declarative language for expressing distributed programs as an unordered set of rules [**?**]. These rules fully specify program semantics and allow a programmer to ignore the details associated with how a program is evaluated. This level of abstraction is attractive for building storage interfaces whose portability and correctness is critical.

We model 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 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 The CORFU Log Interface

To demonstrate the approach of using a declarative language, we show how the CORFU shared-log protocol can be expressed independent of the interfaces and features of the underlying storage system. We have included the salient components of the specification, but due to space constraints refer to the reader to [**?**] for a full program listing.

Listing 1 shows the declaration of state for the CORFU interface. Lines 1—3 define the schema of the two persistent collections that hold the current epoch value, and the log contents. These collections are mapped onto storage but abstract away the low-level interface Lines 5—6 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    scratch :write_op, op.schema
6    scratch :trim_op,  op.schema
7
8    # op did or did not pass the epoch guard
9    scratch :valid_op,   op.schema
10   scratch :invalid_op, op.schema
11 end
```

Listing 1: State Declaration

Every operation handled by a storage device in CORFU is guarded by comparing the epoch in which it was dispatched with the current configured epoch. Listing 2 shows how an operation is compared against the current state. An implementation of this guard may select to cache the epoch value in volatile storage because it infrequently changes.

```
1  state 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  end
```

Listing 2: Epoch Guard

The write-once 64-bit address space exposed by CORFU storage devices depends on a fast lookup within a potentially large and sparse index. The declarative specification shown in Listing 3 enables an implementation to select an index and storage method independently.

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

Listing 3: Write

The CORFU interface depends on applications to mark portions of the log as unused in order to facilitate garbage collection. In Listing 4 entries are tracked as unused for reclamation, and implementations may take advantage of specific optimizations provided by an index implementation or hardware support found in modern non-volatile memories.

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

Listing 4: Trim

Amazingly, these few code snippets can express the semantics of the entire storage device interface requirements in CORFU. For reference our prototype implementation of CORFU in Ceph (called ZLog[2]) is written in C++ and the storage interface component comprises nearly 700 lines of code. This version 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. Furthermore, the complexity introduced by using the bytestream interface would grow the amount of code written in the optimized version.

But beyond the convenience of writing less code, it is far easier for the programmer writing an interface such as CORFU to convenience 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.

5 Additional Optimizations

While our implementation does not yet map a declarative Brados specification to a particular physical design, the specification provides a powerful infrastructure for automating this mapping and achieving other optimizations.

5.1 Batching

Figure 3 shows the performance benefits of batching log appends for two different implementations. The *simple* implementation uses internal I/O interfaces to process each append independently, and the modest performance benefits are due to the amatorization of the fixed per-request cost. The *batch-aware* implementation is able to achieve much higher performance by constructing more efficient I/O requests using range queries and data sieving.

While the performance impact of application-specific batching is significant, techniques such as range queries and data sieving are sensitive to outliers. For example, handling two requests independently will become more efficient than using a data sieving technique as the size of the range between the requests increases. Figure 4 highlights this challenge. The *simple* implementation has consistent, but relatively worse performance. The *batch-aware* implementation achieves high append throughput, but performance degrades as the magnitude of the batch outlier increases. In contrast, the *batch-oident* applies a simple heuristic to identify the outlier and handle it independently, resulting in only a slight decrease in performance over the best case.

[2]https://github.com/noahdesu/zlog

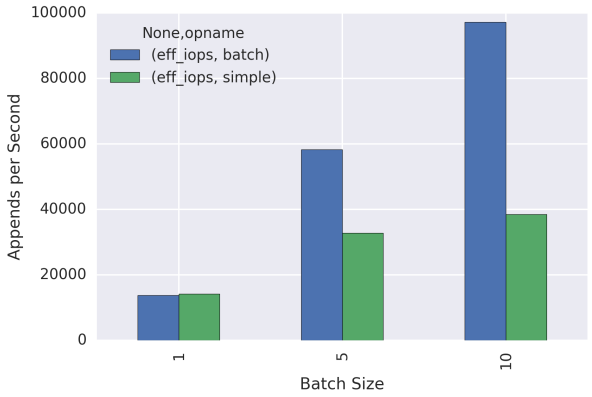

Figure 3: Total throughput with and without batching.

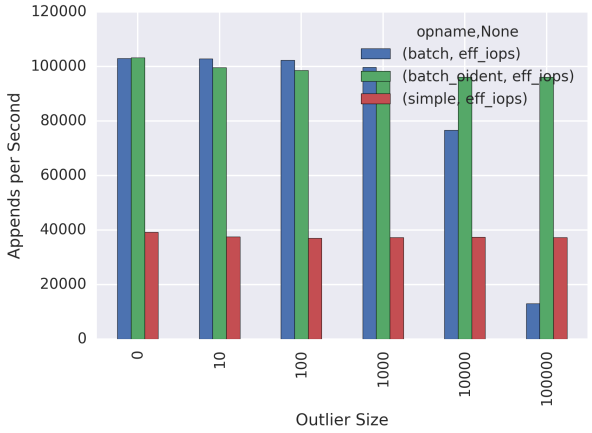

Figure 4: Identifying and handling an outlier independently maintains the beneifts of batching without the performance degredation of unecessarily large I/O requests.

5.2 Physical Design

The challenge of navigating the physical design space has served as the primary source of motivation for selection of a declarative language. Given the declarative nature of the interfaces we have defined, we can draw parallels between the physical design challenges described in this paper and the large body of mature work in query planning and optimization. Consider the simple interface reference counting interface described in Section **??**. The C++ interface makes a strong assumption about a relatively small number of tags, and chooses to fully serialize and deserialize a C++ `std::map` for every request and store the marshalled data as an extended attribute. As the number of tags grows the cost of false sharing will increase to the point that selection of an index-based in-

terface will likely offer a performance advantage. While the monolithic version can outperform for small sets of tags, this type of optimization decision is precisely what can be achieved using a declarative interface definition that hides low-level evaluation aspects such as physical design.

Looking beyond standard forms of optimization decisions that seek to select an appropriate mix of low-level I/O interfaces, data structure selection is an important point of optimization. For instance in Section **??** we showed that using the bytestream for metadata management as opposed to the key-value interface offered superior performance. However the unstructured nature of the bytestream data model imposes no restrictions on implementation or storage layout. Integration of common indexing techniques into an optimizer combined with a performance model will allow our CORFU interface to derive similar optimizations when appropriate.

5.3 Static Analysis

The Bloom language that we use as a basis for Brados produces a data flow graph that can be used in static analysis. We envision that this graph will be made available to the OSD and used to reorder and coalesce requests based on optimization criteria available from a performance model combined with semantic information from the dataflow. For example today object classes are represented as black boxes from the point of view of the OSD execution engine. Understanding the behavior of an object class may allow intelligent prefetching. Another type of analysis that may be useful for optimization is optimistic execution combined with branch prediction where frequent paths through a dataflow are handled optimistically.

References

[1] SEVILLA, M., WATKINS, N., JIMENEZ, I., ALVARO, P., FINKELSTEIN, S., LEFEVRE, J., AND MALTZAHN, C. Malacology: A programmable storage system. In *Eurosys 2017*. To Appear.