Wonderful : A Terrific Application and Fascinating Paper

Your N. Here
Your Institution

Second Name
Second Institution

1 Introduction

As large-scale unified storage systems evolve to meet the requirements of of an increasingly diverse set of application and next-generation hardware, *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, and storage system designers to innovate independently. However, large-scale storage systems have been dominated by proprietary offerings, preventing exploration of alternative interfaces. 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 [4] 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 interfaces. 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 functionally 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 additional optimizations that can be utilized to expand the space of possible implementations.

2 Programmable Storage

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

Extra services. "Bolt-on" services are intended to improve performance or enable a feature, but come at the expense of additional hardware, software sub-systems and dependencies that must be managed, as well as trusted.

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 (e.g. relaxed POSIX file I/O or MPI-IO hints) the coupling that results can be fragile and result when migrating to new systems or handling upgrades.

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 can require significant cost, domain knowledge, and extreme care when building or modifying critical software that can take years of code-hardening to trust.

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 unmanageable burden on developers.

2.1 Malacology Approach

Malacology is a recently proposed approach to co-design between applications and storage systems 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 custom object interfaces, cluster metadata management, and load-balancing. While Ceph natively exposes file, block, and object abstractions, Malacology demonstrated the construction of two real-world services using only a combination of existing interfaces present in Ceph.

One of these interfaces is a high-performance distributed shared-log that closely based on the CORFU protocol [2]. The CORFU protocol achieves high-throughput through the use of a network-attached counter for high-frequency log position assignment, and depends on a custom storage device interface that exposes a write-once infinite address space for handling fault recovery and reconfiguration. Malacology reproduces the CORFU network-attached counter service using a capability-based mechanism found in the Ceph distributed file system for managing cached metadata. The service models the counter state as exclusive access to file metadata similar to the size of a file in shared write-write mode. The storage device interface in CORFU is constructed using a software abstraction over low-level I/O interfaces in Ceph that use atomic updates to bulk data and indexes.

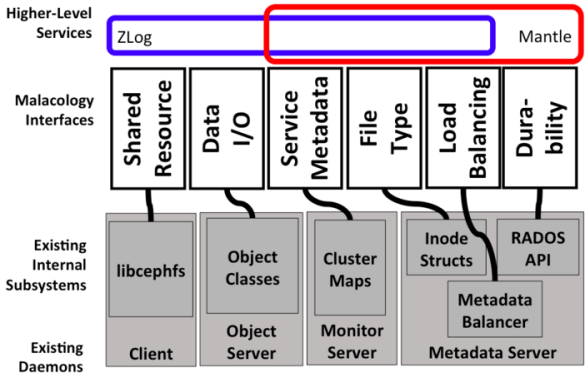

Figure 1: Malacology implementation in Ceph. Existing sub-systems are composed to form new services and application-specific optimizations.

The demonstration of interface synthesis in Malacology suggests a new form of application development that significantly reduces the programming surface area. While this ability to construct software-defined interfaces is powerful, next we will show how access to low-level interfaces can be a double-edged sword, providing its power at the cost of maintenance complexity.

3 Design Space

The narrow interface exposed by storage systems has been a boon in allowing 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 on the system, and with it, forces applications developers to confront a large set of possible designs.

3.1 Software Parameters

To illustrate the design space 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 2a 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 relationship 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.

Takeaway: Choosing the best implementations is dependent on both the timing of the development (Ceph Version) and the expertise of the administrator (Ceph Features). Ceph development is a moving target with multiple stable releases each year, 400+ contributors, and 70–260 commits per week. Disruptive and innovative change in Ceph is also common place, with features such as BlueStore [6] replacing traditional file systems like XFS that have limitations on the workloads generated by Ceph such as double writes and metadata scalability.

3.2 System Tunables

A recent version of Ceph (v10.2.0-1281-g1f03205) has 994 tunables parameters, where 195 of them pertain to the object server itself and 95 of them focus on low-level storage abstractions built on XFS or BlueStore. 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). Auto-tuning [3] techniques have been applied to systems with a large space of parameters with limited success, but the challenge is exacerbated in the context of application-specific modifications and workloads that change dynamically. Next we show an example of an application-specific optimization technique that benefits from dynamic tuning.

3.2.1 Application-specific Group Commit

Group commit is a technique used in database query execution that combines multiple transactions in order to amortize over per-transaction fixed costs like logging. Figure 2b shows the performance impact of using a *group commit*-like technique for combining log appends from independent clients into a single request. The *simple* case implements group commit at the request level, but processes each sub-request append independently using low-level I/O interfaces. The modest performance increase is attributed to a reduction in average per-request costs related to network round-trips. Compared with the *simple* case, the *batching-aware* the *batch-aware* implementation is able to achieve significantly higher performance by constructing more efficient I/O requests using range queries and data sieving techniques provided by the low-level I/O interfaces.

The batched execution technique of group commit can significantly increase throughput, but the story is much more complex. The ability to apply this technique requires tuning parameters such as adding artificial delays to increase batch size that will also affect latency, and parameter tuning is affected dynamically by workload patterns. While the performance impact of application-specific batching is significant, techniques such as range queries and data sieving are sensitive to outliers that can occur with buggy or slow clients.

When outliers occur in a batch, naively building large I/O requests can result in a large amount of wasted I/O. Figure 2c highlights this scope of this challenge. The *simple* case handles each request in the batch independently, and while it performs relatively worse than the other techniques, it is not sensitive to outliers. 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.

Takeaway: System tunables present a challenge in optimizing systems, even in static cases with fixed workloads. Programmable storage approaches that introduce application-specific interfaces that are sensitive to changes in workloads including adversarial cases that must be analyzed and handled greatly increase the design space and set of concerns to be addressed.

3.3 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. While we generally observe the expected result of faster devices resulting in better application performance, choosing the best implementation strategy is highly dependent on hardware. The changes in Ceph required to fully exploit the performance profile of NVMe, persistent memory, and RDMA networks will likely result in new design trade-offs for application-specific interface designs.

4 Declarative Programmable Storage

Current ad-hoc approaches to programmable storage restrict use to developers with distributed programming ex-

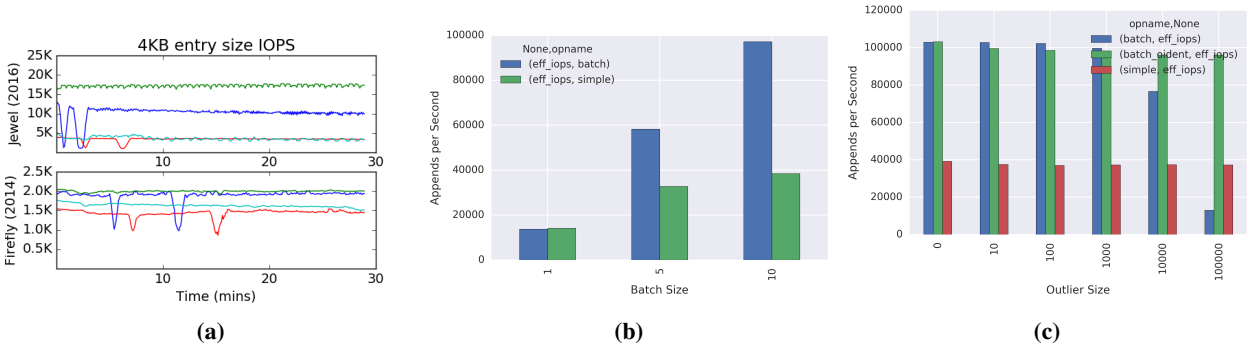

Figure 2: (a) relative performance differences can be drastic after storage software upgrade. (b) total throughput with and without batching. (c) identifying and handling an outlier independently maintains the benefits of batching without the performance degradation of unnecessarily large I/O requests.

pertise, knowledge of the intricacies of the underlying storage system and its performance model, and primarily use hard-coded imperative methods. This restricts the use of optimizations that can be performed automatically or derived from static analysis. Based on the challenges we have demonstrated stemming from the dynamic nature and large design space of programmable storage, we present an alternative, declarative programming model that can reduce the learning curve for new users, and allow existing developers to increase productivity by writing less code that is more portable.

The model we propose corresponds to a subset of the Bloom language which a declarative language for expressing distributed programs as an unordered set of rules [1]. 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, with interfaces being expressed as a collection of *queries* over a request stream that are filtered, transformed, and combined with system state. Next we present a brief example of the CORFU shared-log interface expressed using this model.

4.1 Example: The CORFU Log Interface

The CORFU log protocol achieves high-performance in part by striping the log across a large number of fast storage devices. Using our declarative language we model the log as a single relation, hiding the implementation detail of log partitioning. Lines 2-3 in Listing 1 show the declaration of state for the CORFU interface consisting of two persistent collections: one for the log data, and one for interface metadata. The mapping of collections onto physical storage is abstracted at this level, permitting optimizations discussed earlier to be applied transparently. Note that due to space limitations we only highlight salient features, and the reader may refer to [5] for a full program listing.

Lines 5-10 define operations as scratch collections that are not persistent; they contain state only for the duration of a single time step. Operations may be further sub-divided for specific cases, such as *valid* or *invalid* depending on if they reflect up-to-date system state. An implementation of this guard is shown on lines 12-17, and may select to cache the epoch value in volatile storage because it infrequently changes.

```
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
12   # epoch guard
13   invalid_op <= (op * epoch).pairs{|o,e|
14     o.epoch <= e.epoch}
15   valid_op <= op.notin(invalid_op)
16   ret <= invalid_op{|o|
17     [o.type, o.pos, o.epoch, 'stale']}
18 end
```

Listing 1: State Declaration

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 2 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 2: Write

The CORFU interface depends on applications to mark portions of the log as unused in order to facilitate garbage collection. In Listing 3 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.

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

Listing 3: Trim

Amazingly, only a 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[1]) is written in C++ and the storage interface component comprises nearly 700 lines of code, and uses a hard-coded indexing strategy that has been rewritten multiple times to explore alternative optimization techniques. 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 Discussion

The challenge of navigating the physical design space has served as the primary source of motivation for selection of a declarative language. While our implementation does not yet map a declarative specification on to a particular physical design, the specification provides a powerful infrastructure for automating this mapping and achieving other optimizations. 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.

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 3 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. Similar degrees of freedom can be imagined when handling other approaches to implementing the CORFU sequencer service. Given its soft-state nature heavy-weight processes that enforce durability can be circumvented in favor of shorter code paths that optimize for throughput and latency.

The Bloom language that we use as a basis for a declarative specification language 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.

6 Conclusion

I can't believe this fits.

References

[1] ALVARO, P., CONWAY, N., HELLERSTEIN, J. M., AND MARCZAK, W. R. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR '11* (2011).

[2] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBER, T., WEI, M., AND DAVIS, J. D. Corfu: A shared log design for flash clusters. In *NSDI'12* (San Jose, CA, April 2012).

[3] BEHZAD, B., LUU, H. V. T., HUCHETTE, J., SURENDRA, AYDT, R., KOZIOL, Q., SNIR, M., ET AL. Taming parallel i/o complexity with auto-tuning. In *Proceedings of the International Conference on High mance Computing, Networking, Storage and Analysis* (2013), ACM, p. 68.

[4] 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.

[5] WATKINS, N., SEVILLA, M., JIMENEZ, I., OJHA, N., ALVARO, P., AND MALTZAHN, C. Brados: Declarative, programmable object storage. Tech. Rep. UCSC-SOE-16-12, UC Santa Cruz, 2016.

[6] WEIL, S. A. BlueStore: A New, Faster Storage Backend for Ceph, April 2016.

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