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.

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.

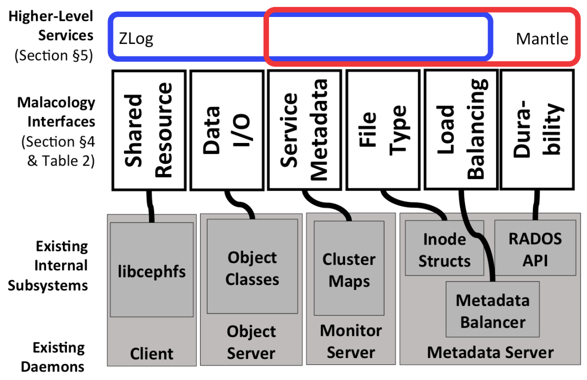

Figure 1: Malacology implementation in Ceph. Existing sub-systems are composed to form new services and application-specific optimizations.

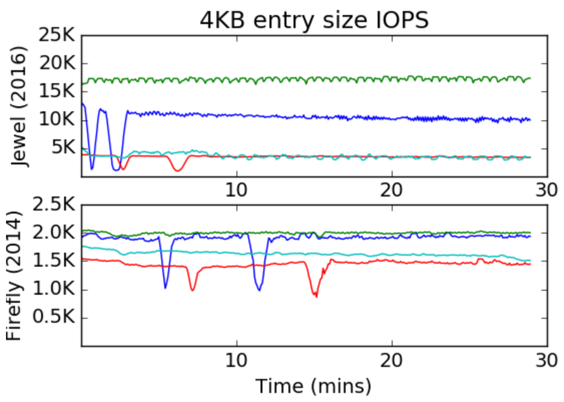

Figure 2: asdf

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 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.

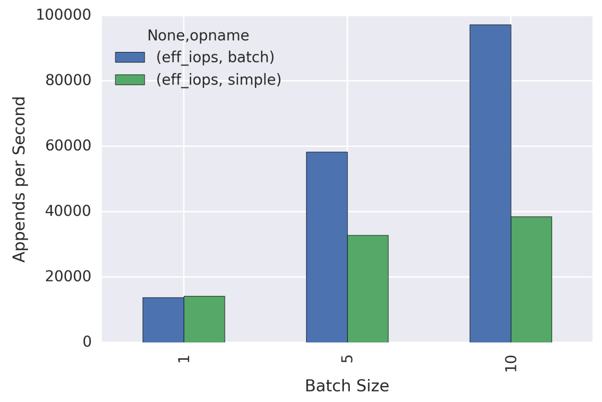

Figure 3: Total throughput with and without batching.

3 Design Space

4 Programming Model

5 Additional Optimizations

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.

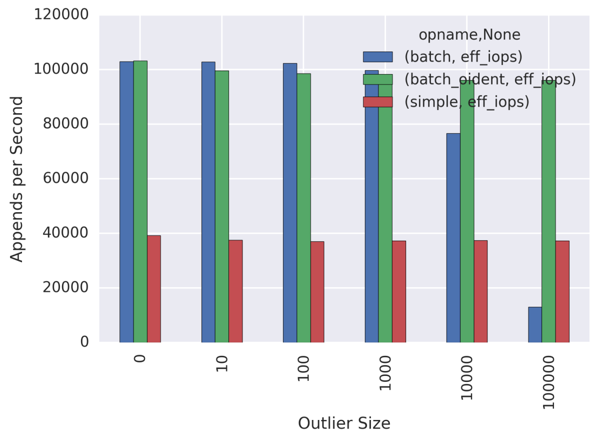

Figure 4: Identifying and handling an outlier independently maintains the beneifts of batching without the performance degredation of unecessarily large I/O requests.

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.