DeclStore: Layering is for the Faint of Heart

Noah Watkins, Michael A. Sevilla, Kathryn Dahlgren, Peter Alvaro, Shel Finkelstein, Carlos Maltzahn†
The University of California, Santa Cruz
{*jayhawk, msevilla, carlosm*†}*@soe.ucsc.edu,* {*kmdahlgr, palvaro, shel*}*@ucsc.edu*

1 Introduction

Popular storage systems support diverse storage abstractions by providing important disaggregation benefits. Instead of maintaining a separate system for each abstraction, *unified* storage systems, in particular, support standard file, block, and object abstractions so the same hardware can be used for a wider range and a more flexible mix of applications. As these large-scale unified storage systems evolve to meet the requirements of an increasingly diverse set of applications and next-generation hardware, *de jure* approaches of the past—based on standardized interfaces—are giving way to domain-specific interfaces and optimizations. While promising, the ad-hoc strategies characteristic of current approaches to co-design are untenable.

The standardization of the POSIX I/O interface has been a major success. General adoption allows application developers to avoid vendor lock-in and encourages storage system designers to innovate independently. However, large-scale storage systems are generally dominated by proprietary offerings, preventing exploration of alternative interfaces. An increase in the number of special-purpose storage systems characterizes recent history in the field, including the emergence of high-performance, and highly modifiable, open-source storage systems, which enable system changes without of vendor lock-in. Unfortunately, evolving storage system interfaces is a challenging task requiring domain expertise and predicated on the willingness of programmers to forfeit the protection from change afforded by narrow interfaces.

Malacology [11] 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 [15], Malacology demonstrates how two real-world services, a distributed shared-log and a file system metadata load balancer, could be constructed using a 'dirty-slate' approach. 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, the technique entails navigating a complex design space and simultaneously addressing often orthogonal concerns (e.g. functional correctness, performance, and fault-tolerance). Worse still, the availability of domain expertise required to build a performant interface is not a fixed or reliable resource. As a result, interface composition, as utilized in techniques such as the Malacology Approach, is sensitive to the change, quickly evolving workloads, and software/hardware upgrades characteristic of unified storage systems. This necessitates massive maintenance overheads.

To address these challenges, we advocate for the use of high-level declarative languages (e.g. Datalog) as a means of programming new storage system interfaces. By specifying the functional behavior of a storage interface once in a relational (or algebraic language), optimizers built around cost models sensitive to storage and access tools and overheads can explore a space of functionally equivalent physical implementations. Much like query planning and optimization in database systems, this approach will logically differentiate correctness and performance and protect higher-level services from lower-level system changes. However, despite the parallels with database systems, this paper demonstrates, and begins to address, fundamental differences in the optimization design space.

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 capable of expanding the space of possible implementations.

2 Programmable Storage

The most common approaches for designing workarounds capable of handling application requirements not met by an underlying storage system roughly fall into among 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. For instance, such classes of limitations inspired many extensions to Hadoop [5, 8, 7, 10].

Application changes. Changing the underlying application with the addition of more data management intelligence or via the integration of domain-specific middleware represents another popular option for adapting to storage system deficiencies. When application adjustments depend on non-standard semantics exposed by the storage system (e.g. relaxed POSIX file I/O or MPI-IO hints) the resulting coupling can be fragile. For example, both SchiHadoop [6] and Rhea [9] do an excellent jobs of partitioning data in Hadoop applications, but may not withstand the test of time for future workloads, since the partitioning is specific to scientific data.

Storage modifications. When these two approaches fail to meet an application's needs, developers may turn their attention to any number of heavyweight 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. For example, HDFS fails to meet many needs of metadata-intensive workloads [12]. Systems builder achieve performance improvement by modifying the underlying architecture or API [3].

Rather than relying on storage systems to evolve or applications to change, the "Malacology Approach" manifests a hybrid technique which embraces interface instability without placing an unmanageable burden on developers.

2.1 The Malacology Approach

Programmable storage systems improve the development experience over the lifecycles of long-lived and co-designed applications and storage systems by exposing a number of existing internal services capable of principled composition. Importantly, the compositionality of the internal functionality aids in the development of higher-level application specific services [11]. Figure 1 shows the architecture of Malacology, a prototype programmable storage system implemented in Ceph, which exposes a variety of low-level internal services such as custom object interfaces, cluster metadata management, and load-balancing. While Ceph natively exposes file, block, and object abstractions, Malacology demonstrates the construction of two real-world services using only the composition of existing interfaces present in Ceph.

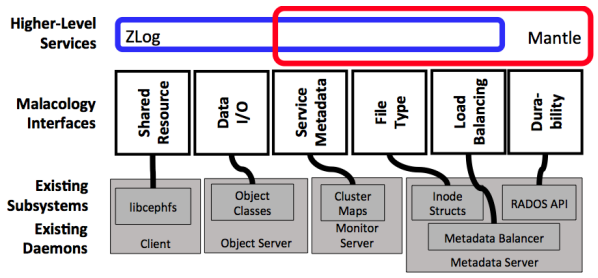


Figure 1: Malacology implementation in Ceph. Existing sub-systems are composed to form new services and application-specific optimizations.

One of these synthesized interfaces is a high-performance distributed shared log that implements the CORFU protocol [2]. This protocol achieves high-throughput by using a soft-state network-attached counter and stripes log content over a cluster of flash devices exposing protocol-specific storage interfaces. While CORFU is a stand-alone system, Malacology is able to instantiate the same abstraction and approximate the same unique optimizations.

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 implements the counter using the metadata server's ability to provide temporary exclusive access to a shared resource (in this case, file metadata). CORFU's storage device interface is constructed using application-specific object interfaces in Ceph. These software-based interfaces are constructed as a composition of low-level I/O interfaces (e.g. an LSM-tree and a bytestream) operating in an atomic context. The technique allows the interface to maintain consistency across native interfaces.

The demonstration of interface synthesis in Malacology suggests a new form of application development capable of significantly reducing the number of lines of code associated with a higher-level application. While such an ability to construct software-defined interfaces is powerful, subsequent analyses demonstrate how access to low-level interfaces can be a double-edged sword providing extended application development powers at the cost of increased maintenance complexity.

3 Design Space

The narrowly-defined interfaces representing the dominating trend in storage systems designs is a boon allowing systems and applications to evolve independently by establishing limitations on the size of the design space where applications couple with storage. Programmable storage lifts the veil on the system and, with it, forces developers of higher-level services to confront a large and expanding set of possible designs.

In this section we elucidate the matter of design space size and complexity in programmable storage. We report on our experience building and optimizing *multiple* functionally equivalent implementations of the CORFU protocol in Ceph, demonstrating that static selection of optimization strategies and tuning decisions can lead to performance portability challenges in programmable storage systems.

3.1 System Tunables and Hardware

A recent version of Ceph (v10.2.0-1281-g1f03205) has 994 tunable parameters. 195 parameters pertain to the object server itself and 95 parameters focus on low-level storage abstractions built on XFS or BlueStore. Ceph also has tunables controlling the characteristics of underlying subsystems, including LevelDB (10 tunables) and RocksDB (5 tunables), in addition to the Ceph-specific key-value stores (5 tunables), object cache (6 tunables), journals (24 tunables), and other optional object stores like BlueStore (49 tunables). Previous investigations exploring the application of auto-tuning [4] techniques to systems exhibiting a large space of parameters met with limited success. Challenges associated with the approach are exacerbated in the context of application-specific modifications and dynamically changing workloads.

Hardware. Ceph is designed to run on a wide variety of commodity hardware as well as new NVMe devices. All these devices encompass specific sets of characteristics and tunables (e.g., the IO operation scheduler type). In our experiments, we tested SSD, HDDs, and NVMe devices and discovered a wide range of behaviors and performance profiles. While we generally observe the expected result of faster devices, 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 interfaces.

Takeaway: The ever-evolving space of hardware and system tunables presents a challenge in optimizing systems, even in static cases with fixed workloads. Programmable storage approaches introduce application-specific interfaces sensitive to changes in workloads and guided by cost models integrating low-level interface performance considerations. In particular, the low-level performance considerations promise to significantly increase the design space and set of concerns to be addressed by programmers.

3.2 Software

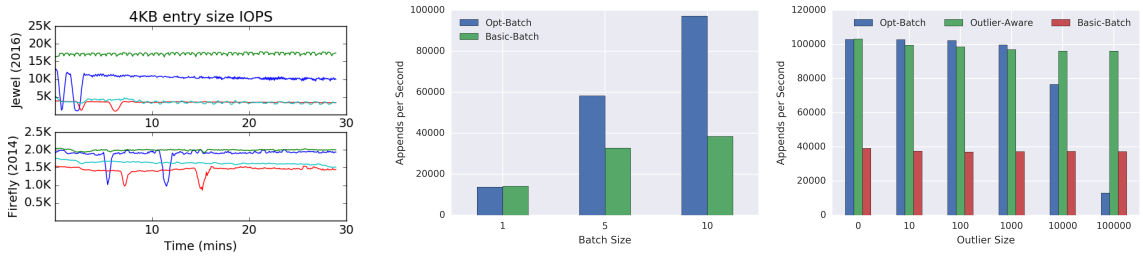
The primary source of complexity in large storage systems is, unsurprisingly, the vast amount of software written to handle challenges like fault-tolerance and consistency in distributed heterogeneous environments. We have found that even routine upgrades can cause performance regressions manifesting in obstacles for adopters of a programmable storage approach to development.

The CORFU protocol stripes a log across a cluster of storage devices, where each device exposes a custom 64-bit write-once address space for reading and writing log entries. While this interface can be built directly into flash devices [14], we constructed four different versions in Ceph each as a software abstraction over the existing object interface. Each of our software-based implementations differ with respect to optimization strategies 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 running on two versions of Ceph from 2014 and 2016. The data indicate performance in general is significantly better in the newer version of Ceph. However, the implementations manifest another interesting observation. The top two implementations, run on a version of Ceph from 2014, perform with nearly identical throughput, but have strikingly different implementation complexities. The performance of the same implementations on the newer version of Ceph illustrates a challenge: developers face a reasonable choice of a simpler implementation in the 2014 version of Ceph and a storage interface which will perform worse in the 2016 version of Ceph, requiring a significant overhaul of low-level interface implementations.

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 batching log appends from independent clients into a single request. The *Basic-Batch* case implements group commit at the re-



(a) **(b)** **(c)**

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

quest level, but processes each sub-request (i.e. 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 *Basic-Batch* case, the *Opt-Batch* implementation is able to achieve significantly higher performance by constructing more efficient I/O requests using range queries and data sieving techniques [**?**] afforded 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 capable of triggering cascading performance impacts. For example, adding artificial delays to increase batch size affects other metrics such as latency. Additionally, while the performance impact of application-specific batching is significant, techniques such as range queries and data sieving are sensitive to outliers resulting from 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 the scope of this challenge. The *Basic-Batch* case handles each request in a batch independently and, while the resulting performance is worse relative to the other techniques, it is not sensitive to outliers. On the other hand, the *Opt-Batch* implementation achieves high append throughput, but performance degrades as the magnitude of the outliers in the batch increases. In contrast, an *Outlier-Aware* policy applies a simple heuristic to identify and handle outliers independently, resulting in only a slight decrease in performance over the best case.

Takeaway: Choosing the best implementation of a storage interface depends on the timing of development (Ceph version), the expertise of programmers and administrators (Ceph features), the tuning parameters and hardware selection, as well as system-level and application-specific workload changes. A direct consequence of such a large design space is forcing engineers to duplicate efforts when hardware and software characteristics change. Such a circumstance necessitates the sub-optimal pursuit of unnecessary on-going work and increases the risk of introducing bugs that, in the best case, affect a single application and, in monolithic designs, are more likely to cause systemic data loss.

We believe a better understanding of application and interface semantics exposes a frontier of new and better approaches with more optimal maintenance requirements than hard-coded and hand-tuned implementations. An ideal solution to these challenges is an automated system search of *implementations*—not simply tuning parameters—based on programmer-produced specifications of storage interfaces in a process independent of optimization strategies and guaranteed to not introduce correctness bugs. Next we'll discuss a candidate approach using a declarative language for interface specification.

4 Declarative Programmable Storage

As we have seen, current ad-hoc approaches to programmable storage restrict use to developers with distributed programming expertise, knowledge of the intricacies of the underlying storage system and its performance model, and use hard-coded imperative methods. This limits 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 is 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 stor-

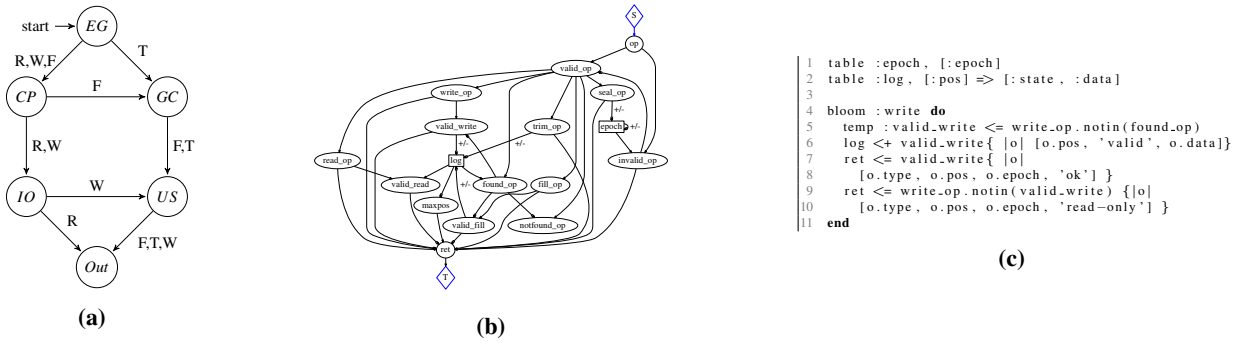


Figure 3: the logical dataflow could not be more consise and still capture the state machine.

age 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: CORFU as a Query

We model the storage interface of the CORFU protocol as query in our declarative langauge in which log data and metadata are represented by two persistent collections (Figure 3 (c) Lines 1-2). The mapping of collections onto physical storage is abstracted at this level, permitting optimizations and implementation details such as log striping and partitioning to be discovered and applied transparently by an optimizer. The declarative specification of the CORFU *write* operation is shown on lines 3-10. This interface is a write-once 64-bit address space and depends on a method for quickly resolving log positions and metadata to physical storage addresses. Since the specification is an invariant across system changes and low-level interfaces, specific access methods and the construction of indexes can be considered by an optimizer.

Amazingly, only a few code snippets can express the semantics of the entire storage interface requirements in CORFU[1]. Figure 3a shows the state transition diagram for the CORFU storage interface, and Figure 3b shows the corresponding dataflow diagram for the entire CORFU protocol expressed in Bloom that can serve as input to an optimizer. 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.

[1]Due to space limitations refer to [13] for a full program listing.

5 Discussion and Conclusion

The current state of the field of storage systems is clearly in the midst of significant change. The sparse collection of guide posts capable of assisting developers navigating such a large, complex, and ever-evolving physical design space serves as the primary source of motivation for the use of declarative languages as a novel method distancing the dynamic nature of storage systems from the more static nature of higher-level applications. While our implementation does not yet map a declarative specification on to a particular physical design, the specification provides a powerful infrastructure for automating such a mapping and achieving other optimizations.

Given the declarative flavor of the interfaces we have defined, deep parallels exist between the physical design challenges described in this paper and the larger body of mature work in query planning and optimization. The Bloom language used as a basis for the declarative specification language of our approach produces a dataflow graph amenable to sophisticated static analyses. We envision the graph will be made fully available for exploitation by the storage system.

We are currently considering the scope of optimizations possible with such a dataflow model in the context of storage systems. For instance, without semantic knowledge of the interface, batching techniques described in Section 3.2.1 are limited to optimizations such as selecting magic values for timers and buffer sizes. Semantic information expands the design space and permits intelligent reordering or coalescing with a dependence upon the relationship between operations beyond what auto-tuning has considered in a storage system context.

Conclusion. With each new application-specific storage interface, one must consider the logical conclusion as a future in which little to no distinction exists between storage and database systems, or not. The roads to both futures are lined by intermediate solutions, such as the one we have proposed are, which explore tools and techniques for managing the dual tradeoffs of complexity and

portability.

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] BALMIN, A., KALDEWEY, T., AND TATA, S. Clydesdale: Structured data processing on hadoop. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 705–708.

[4] 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.

[5] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. D. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 285–296.

[6] BUCK, J., WATKINS, N., LEFEVRE, J., IOANNIDOU, K., MALTZAHN, C., POLYZOTIS, N., AND BRANDT, S. A. Scihadoop: Array-based query processing in hadoop. In *SC '11* (Seattle, WA, November 2011).

[7] EKANAYAKE, J., GUNARATHNE, T., FOX, G., BALKIR, A. S., POULAIN, C., ARAUJO, N., AND BARGA, R. Dryadlinq for scientific analyses. In *2009 Fifth IEEE International Conference on e-Science* (Dec 2009), pp. 329–336.

[8] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., AND FOX, G. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (New York, NY, USA, 2010), HPDC '10, ACM, pp. 810–818.

[9] GKANTSIDIS, C., VYTINIOTIS, D., HODSON, O., NARAYANAN, D., DINU, F., AND ROWSTRON, A. Rhea: Automatic filtering for unstructured cloud storage. In *NSDI'13* (Lombard, IL, April 2-5 2013).

[10] MIHAILESCU, M., SOUNDARARAJAN, G., AND AMZA, C. Mixapart: Decoupled analytics for shared storage systems. In *4th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'12, Boston, MA, USA, June 13-14, 2012* (2012).

[11] SEVILLA, M. A., WATKINS, N., JIMENEZ, I., ALVARO, P., FINKELSTEIN, S., LEFEVRE, J., AND MALTZAHN, C. Malacology: A programmable storage system. In *Proceedings of the 12th European Conference on Computer Systems* (Belgrade, Serbia), Eurosys '17. To Appear, preprint: `https://www.soe.ucsc.edu/research/technical-reports/UCSC-SOE-17-04`.

[12] SHVACHKO, K. V. Hdfs scalability: The limits to growth. *;login: 35*, 2 (2010).

[13] 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.

[14] WEI, M., DAVIS, J. D., WOBBER, T., BALAKRISHNAN, M., AND MALKHI, D. Beyond block i/o: Implementing a distributed shared log in hardware. In *Proceedings of the 6th International Systems and Storage Conference* (New York, NY, USA, 2013), SYSTOR '13, ACM, pp. 21:1–21:11.

[15] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation*, OSDI '06.