DeclStore: Layering is for the Faint of Heart

Noah Watkins, Michael A. Sevilla, Ivo Jimenez, Kathryn Dahlgren,
Peter Alvaro, Shel Finkelstein, Carlos Maltzahn
The University of California, Santa Cruz
{*jayhawk, msevilla, ivo, 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 large-scale unified storage systems continue to 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 has allowed 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 when the need has presented itself. 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 fear of vendor lock-in. Unfortunately, evolving storage system interfaces is a challenging task requiring domain expertise, and is predicated on the willingness of programmers to forfeit the protection from change afforded by narrow interfaces.

Malacology [14] is a recently proposed storage system that advocates for an approach to co-design called *programmable storage*. The approach exposes low-level functionality as reusable building blocks, allowing developers to custom-fit their applications to take advantage of the existing code-hardened capabilities in an underlying system, and avoid duplication of complex and error-prone services. By recombining existing services in the Ceph storage system [21], Malacology demonstrated 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, such an ad-hoc approach can be difficult to reason effectively about and manage.

Despite the benefits of the approach demonstrated by Malacology, the technique requires navigation of a complex design space while 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, the interfaces built with Malacology are sensitive to evolving workloads. This results in burdensome maintenance overhead when underlying hardware and software changes.

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 can explore a space of functionally equivalent physical implementations. Much like query planning and optimization in database systems, this approach will logically differentiate correctness from performance, and protect higher-level services from lower-level system changes [13]. However, despite the parallels with database systems, this paper demonstrates, and begins to address, fundamental differences in the optimization design space.

In the next section we expand on the concept of programmable storage, and then highlight the size and complexity of the design space confronting developers that embark on co-designing applications and storage. Using a distributed shared-log interface as a motivating example, we propose the use of a declarative language capable of capturing functional behavior for defining future stor-

age system interfaces.

2 Programmable Storage

Common workarounds when application requirements are not met by an underlying storage system roughly fall into three categories:

"Bolt-on services" improve performance or enable new features, 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 [6, 9, 8, 12].

Application changes introduce data management intelligence or integrate domain-specific middleware into an application. When application changes depend on non-standard storage semantics (e.g. relaxed POSIX file I/O or MPI-IO hints) the resulting coupling can be fragile. For example, both SciHadoop [7] and Rhea [10] do an excellent job of partitioning data in Hadoop applications, but may not withstand the test of time for future workloads, since the partitioning is specific to the use case. Approaches to I/O optimization in middleware (e.g. MPI-IO) take advantage of an application's structured and partitioned data model, but face portability challenges when mapping parallel I/O onto a bytestream. The challenge is, in part, due to the wide range of optimization strategies that are dependent on low-level storage system *magic numbers* for optimal data partitioning, distribution, and alignment. The PLFS file system takes an approach of virtualizing the POSIX byte stream over a set of logs to address this issue [5].

Storage system modifications are often a last resort because such heavyweight solutions range from merely changing the underlying system to designing entirely new systems. This approach requires at a minimum, a certain level of access to modify the system, 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 [15]. This has led to modifications to its architecture and API [3] to improve performance.

Rather than relying on storage systems or applications to change, Malacology exposes data management services already present in the underlying system, which can be re-used to avoid code duplication and reliance on external services.

The Malacology Approach Malacology is a prototype programmable storage system based on Ceph that improves the development experience of co-designing applications and storage systems by exposing common internal storage services for re-use [14]. Figure 1 shows the architecture of Malacology, along with the set of services that are exposed, such as domain-specific object interfaces, cluster-level 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.

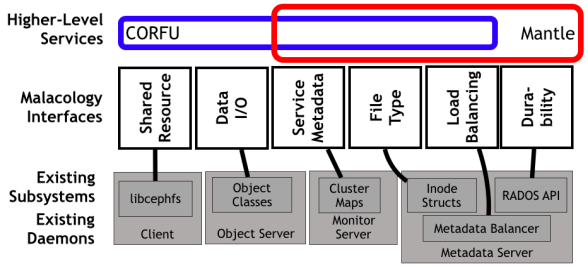

Figure 1: Malacology implementation in Ceph. Existing sub-systems are composed to form new services and application-specific optimizations.

One of these synthesized interfaces is a high-performance distributed shared-log based on the CORFU protocol [2]. While CORFU can be a stand-alone system, Malacology is capable of instantiating the same storage abstraction and approximating the same optimizations. High-performance in CORFU is achieved in part through the use of a soft-state network-attached counter. Malacology approximates this optimization using the capability-based caching mechanisms in the Ceph distributed file system, modeling the counter as a shared resource (i.e., file metadata). Additionally, the co-designed device interfaces used in CORFU are critical to the safety of the protocol, and are replicated in Ceph using custom software-based interfaces to storage objects.

Although powerful, storage interface construction in Malacology (Data I/O interface in Figure 1) is a double-edged sword. The narrowly-defined interfaces dominating systems today have been a boon to developers by limiting the size of the design space where applications couple with storage, allowing systems to evolve independently. Programmable storage lifts the veil on the system and, thereby forces developers of higher-level services to confront a much broader set of possible designs.

3 Design Space

In this section we highlight the size and complexity of the design space of programmable storage, showing how the ad-hoc approach used in Malacology is limited by increases in software design and maintenance of co-designed interfaces. Note that while there are many interfaces in Malacology, we focus on the *Data I/O* interface

for our examples. We report on our experience building multiple functionally equivalent implementations of the CORFU protocol in Ceph, and demonstrate that static selection of optimization strategies and tuning decisions can lead to performance portability challenges.

System Tunables and Hardware. A recent version of Ceph from May 2016 had 994 tunable parameters, controlling all aspects of the system such as the object storage server (195), low-level components such as XFS and BlueStore (95), and sub-systems such as RocksDB and journaling (29). Previous investigations exploring the application of auto-tuning techniques to systems exhibiting a large space of parameters was met with limited success [4]. And challenges associated with this approach are exacerbated in the context of application-specific modifications and dynamically changing workloads which only serve to increase the state space size.

Hardware. Ceph is intended to run on a wide variety of commodity, high-end, and low-end hardware, including newer high-performance non-volatile storage devices. Each hardware configuration encompasses specific sets of performance characteristics and tunables (e.g. I/O scheduler selection, and policies such as timeouts). In our experiments, we tested a variety of hardware and discovered a wide range of behaviors and performance profiles. While we generally observe the expected improvements on faster devices, choosing the best implementation strategy is highly dependent on hardware. This will continue to be true as storage systems evolve to support new technologies such as persistent memories and RDMA networks that may require entirely new storage interfaces for applications to fully exploit the performance of hardware.

Takeaway: Evolving hardware and system tunables presents a challenge in optimizing systems, even in static cases with fixed workloads. Programmable storage approaches that introduce application-specific interfaces are sensitive to changes in workloads and the cost models of low-level interfaces that are subject to change. This greatly increases the design space and set of concerns that must be addressed by programmers.

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 which manifest as obstacles for adopters of a programmable storage approach to development. We use the CORFU shared-log protocol as a motivating example.

Shared-log. In our implementation of CORFU on Ceph [18] the shared-log is striped across a set of objects in Ceph to provide parallel I/O bandwidth. Each object implements the custom storage interface that exposes a 64-bit write-once address space, required by the CORFU protocol. While this interface can be built directly into flash devices [20], we constructed four different versions in software as native object interfaces in Ceph. Each of our implementations differs with respect to which internal interfaces are used (e.g. RocksDB, and/or a bytestream) and how data is striped and partitioned in the system.

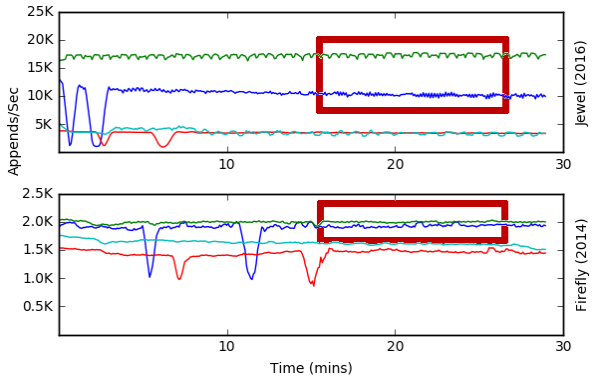

Figure 2: Performance of four shared-log implementations on two different versions Ceph.

Figure 2 shows the append throughput of four such implementations running on two versions of Ceph from 2014 and 2016 using the same hardware, in which the performance in general is significantly better in the newer version of Ceph. However, if we consider other costs such as software maintenance these results reveal another trade-off. The top two best implementations running on the 2014 version of Ceph perform with nearly identical throughput, but have different implementation complexities. When we consider the performance of the same implementations on the newer version of Ceph a challenge presents itself: developers face a reasonable choice between a simpler implementation in the 2014 version of Ceph with little performance difference, and a storage interface which will perform significantly worse in the 2016 version of Ceph, requiring a significant overhaul of low-level interface implementations. We believe that these trade-offs will continue to present themselves as new hardware is supported and internal storage interfaces evolve.

Group commit. In addition to the broad challenge of design and implementation, tuning application-specific interfaces for a static implementation can be challenging. Group commit is a technique used in database query execution that combines multiple transactions in order to amortize over fixed per-transaction costs [11]. We implemented two batching strategies for shared-log appends. The first approach called *Basic-Batch* groups multiple

requests together, but processes each sub-request (i.e. log append) independently at the lowest level. The second approach called *Opt-Batch* examines the requests in a batch and issues efficient low-level I/O requests (e.g. range queries and data sieving [17]). With a batch size of 1 request both approaches achieve approximately 14K appends per second with a single storage node. With a batch size of 5 requests *Basic-Batch* and *Opt-Batch* performance increases by 2.3x and 4.2x, respectively, and with a batch size of 10 requests the increase is 2.7x and 7.0x, achieving 97K appends per second at the high end.

While this batching technique significantly increases throughput, the story is more complex. The effectiveness of this technique requires tuning parameters such as forcing request delays to achieve larger batch sizes, which in turn have a direct effect on latency. While performance of this technique benefited from using range queries and data sieving, these interfaces are sensitive to outliers that generate large I/O requests containing a high percentage of irrelevant data. In Figure 3a 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. The *Opt-Batch* implementation achieves high append throughput, but performance degrades as the magnitude of the outliers in the batch increases due to wasted I/O. 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 (e.g. system version); the expertise of programmers and administrators; tuning parameters and hardware configuration; and system-level and application-specific workload characteristics. A direct consequence of such a large design space is that some choices may quickly become sub-optimal as aspects of the system change. This forces developers to revise implementations frequently, increasing the risk of introducing bugs that, in the best case, affect a single application and, in monolithic designs, may cause systemic data loss.

We believe a better understanding of application and interface semantics exposes a frontier of new and better approaches with fewer 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

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 propose an alternative, declarative programming model which reduces the learning curve for new users, and allows existing developers to increase productivity by writing fewer, more portable lines of code.

The model we propose corresponds to a subset of Bloom, a declarative language for expressing distributed programs as an unordered set of rules [1]. Bloom rules fully specify program semantics and allow developers to ignore the details associated with program evaluation. This level of abstraction is attractive for building storage interfaces whose portability and correctness is critical. We use Bloom to model the storage system state uniformly as a collection of relations, with interfaces expressed as a collection of *queries* over a request stream that are filtered, transformed, and combined with other system state. We present a brief example of the CORFU shared-log interface expressed using this model.

Example: CORFU as a Query We model the storage interface of the CORFU protocol as a query in our declarative language in which the shared-log and metadata are represented by two persistent abstract collections mapped onto physical storage. This transformation permits optimizations and implementation details (e.g. log striping and partitioning) to be discovered and applied transparently by an optimizer. Since the specification of the interface is invariant across system changes and low-level interfaces, the optimizer can automatically render execution decisions and build indexes using the performance characteristics of specific access methods. For example a low-level indexing engine for text will likely be out-performed by other engines for the CORFU 64-bit write-once address space interface. Likewise, an instance of the interface that uses fixed log entries can directly map log entries onto a low-level byte stream, avoiding an explicit index in some situations.

Amazingly, the semantics of the entire storage interface requirements in CORFU[1] are expressible using only a few Bloom code snippets amenable as input to an optimizer. Figure 3b shows the state transition diagram for the CORFU storage interface and Figure 3c shows the

[1] Due to space limitations refer to [19] for a full program listing.

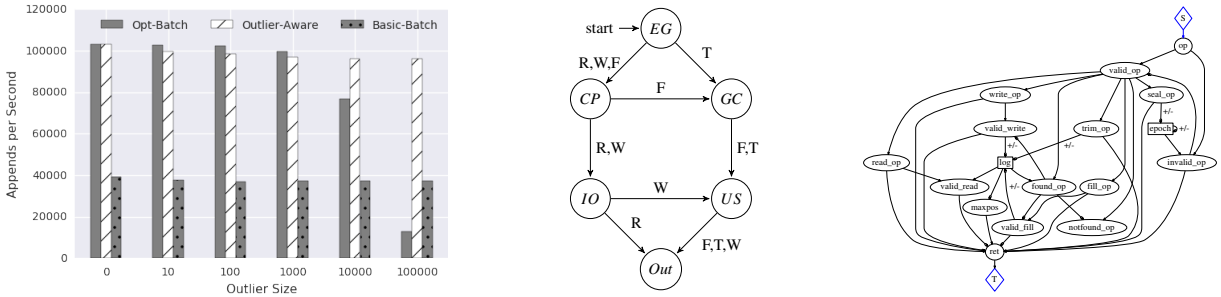

Figure 3: (a) batching performance with and without outlier detection (b) state machine for the CORFU storage device. (c) logical dataflow of the CORFU storage protocol which could not be more concise and still capture the state machine.

corresponding dataflow diagram for the Bloom CORFU protocol. Beyond the convenience of writing less code, the entire experience of designing and writing an interface such as CORFU in a declarative language such as Bloom eases the process of constructing convincingly correct implementations. Specifically, the high-level details of the implementation mask distracting issues related to the physical design and the many other "gotchas" associated with writing low-level systems software.

Our current Bloom specification of CORFU assumes the existence of an external sequencer service to assign log positions. However, we are working towards a specification that defines the sequencer service as a view over the log, whose state is managed in volatile storage. A declarative specification will be critical to providing portability of the service, since storage systems internally utilize volatile storage in many forms (e.g. memory caches and non-replicated data). For example, our work in Malacology showed how inode state in a distributed file system could be used to build a sequencer, but an object-based storage system could place sequencer state in an object cache while providing fast-path access that is difficult to achieve with the consistency and durability requirements of non-volatile object state.

5 Discussion and Conclusion

It's clear that storage systems are currently in the midst of significant change and with few guideposts available to developers navigating a large and complex design space. This has served as the primary source of motivation for our use of declarative languages. And 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. The Bloom language that we use as a basis for a declarative specification produces a dataflow graph that can be used in static analysis, and we envision that this graph will be made fully available to the storage system to exploit before and during runtime.

We are currently considering the scope of optimizations that are possible with such a dataflow model in the context of storage systems. For instance, without semantic knowledge of an interface, batching techniques described in Section 3 are limited to optimizations such as selecting magic values for timers and buffer sizes. Semantic information expands the design space, permitting intelligent reordering or coalescing that depends on relationships between operations, going beyond what autotuning has previously considered.

Finally, we emphasize that new non-volatile memories are exposing code path length as a bottleneck [16], and that achieving a desired performance level while proposing *higher*-level abstractions is a critical concern that must be addressed. We see advancements in main memory databases as an indicator that performance considerations are being addressed in other similar contexts.

Conclusion. Optimizing every new or changed application as storage systems evolve is obviously impractical. A storage system is not the same as a database system, but techniques from database optimization can potentially be leveraged to address complexity, performance and transparent portability for applications running on evolving storage systems. Generalizing from the example we described, we think this approach is innovative and promising.

Acknowledgements: This work was partially funded by the Center for Research in Open Source Software, DOE Award DE-SC0016074, and NSF Award 1450488.

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] BENT, J., GIBSON, G., GRIDER, G., MCCLELLAND, B., NOWOCZYNSKI, P., NUNEZ, J., POLTE, M., AND WINGATE, M. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 21:1–21:12.

[6] 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.

[7] 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).

[8] 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.

[9] 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.

[10] 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).

[11] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[12] 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).

[13] SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1979), SIGMOD '79, ACM, pp. 23–34.

[14] 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`.

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

[16] SWANSON, S., AND CAULFIELD, A. Refactor, reduce, recycle: Restructuring the i/o stack for the future of storage. *Computer 46*, 8 (Aug. 2013), 52–59.

[17] THAKUR, R., GROPP, W., AND LUSK, E. Data sieving and collective i/o in romio. In *Frontiers of Massively Parallel Computation, 1999. Frontiers '99. The Seventh Symposium on the* (Feb 1999), pp. 182–189.

[18] WATKINS, N. Zlog distributed shared-log. `https://github.com/noahdesu/zlog`, 2014–2017.

[19] 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.

[20] 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.

[21] 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.