A Case for Optimistic Coordination in HPC Storage Systems

Philip Carns,* Kevin Harms,* Dries Kimpe,* Justin M. Wozniak,* Robert Ross,* Lee Ward,† Matthew Curry,†
Ruth Klundt,† Geoff Danielson,† Cengiz Karakoyunlu,‡ John Chandy,‡ Bradley Settlemyer,§ William Gropp¶

*Argonne National Laboratory, Argonne, IL, USA, {carns,dkimpe,wozniak,rross}@mcs.anl.gov, harms@alcf.anl.gov
†Sandia National Laboratories, Albuquerque, NM, USA, {lee,mlcurry,rklundt,gcdanie}@sandia.gov
‡University of Connecticut, Storrs, CT, USA, {cengiz,john.chandy}@uconn.edu
§Oak Ridge National Laboratory, Oak Ridge, TN, USA, settlemyerbw@ornl.gov
¶University of Illinois at Urbana-Champaign, Urbana, IL, USA, wgropp@illinois.edu

***Abstract*—High-performance computing (HPC) storage systems rely on access coordination to ensure that concurrent updates do not produce incoherent results. HPC storage systems typically employ pessimistic distributed locking to provide this functionality in cases where applications cannot perform their own coordination. This approach, however, introduces significant performance overhead and complicates fault handling.**

In this work we evaluate the viability of optimistic conditional storage operations as an alternative to distributed locking in HPC storage systems. We investigate design strategies and compare the two approaches in a prototype object storage system using a parallel read/modify/write benchmark. Our prototype illustrates that conditional operations can be easily integrated into distributed object storage systems and can outperform standard coordination primitives for simple update workloads. Our experiments show that conditional updates can achieve over two orders of magnitude higher performance than pessimistic locking for some parallel read/modify/write workloads.

I. Introduction

Parallel applications rely on high-performance computing (HPC) storage systems to present a shared storage namespace and service concurrent I/O requests from a large number of processes. Because an HPC storage system is a shared resource, access coordination is required to ensure that concurrent updates do not produce incoherent results. In some cases this coordination is handled by the application itself (e.g., by MPI synchronization or explicit data partitioning); in other cases coordination is provided by the storage system (e.g., POSIX file semantics). Storage system synchronization primitives are especially critical in cases where it is difficult for applications to protect consistency on their own. Examples include

- concurrent namespace (file or directory) updates,
- unaligned access in block-based storage systems, and
- applications in which application-level coordination is not feasible (e.g., loosely coupled calculations such as reverse index computation, the GUPS workload, and parallel histogram creation).

The traditional approach to providing access coordination in these cases is through distributed file locking in a parallel file system. This approach has several drawbacks. Distributed locking is pessimistic; it introduces significant overhead even in cases where conflicts are rare. In addition, it introduces shared state on client processes. This shared state complicates fault tolerance and compromises scalability by forcing the storage system to be cognizant of client and application fault conditions.

This problem has been studied extensively in the database and shared-memory literature. A well-known alternative to locking in those arenas is to leverage conditional storage operations, such as compare-and-swap or load-link/store-conditional. These primitives can be used as building blocks to construct a number of higher-level atomic operations. To date, however, conditional storage operations have not been adopted in HPC storage systems for a number of reasons. Notably, the traditional POSIX API does not support conditional storage primitives, and current HPC storage systems do not expose any other API to client nodes. In addition, there is a lack of support for efficient comparison or conditional checks in the local storage abstractions that form the foundation of most HPC storage systems, for example, Trove (PVFS), ldiskfs (Lustre), or ZFS (Lustre).

Fortunately, two storage trends are now converging in a way that makes these problems more tenable. The first trend is a growing consensus in the storage community that alternative storage APIs will be necessary in order to continue to scale HPC storage systems. The most promising low level model for storage access is the distributed object storage model [1] [2] [3]. Such models are not tied to legacy POSIX semantics and can more easily be modified to expose additional synchronization primitives to applications, while still providing a solid foundation for a variety of higher level interfaces (including POSIX). The second trend is toward increased functionality in local storage abstractions in order to support features such as checksumming [4], log-structured storage [5], and provenance [6]. These features inherently require additional metadata support in the local storage component of a distributed storage system, and this same metadata infrastructure can often be reused to support conditional storage primitives with modest incremental complexity.

In this work we evaluate optimistic store-conditional operators as an alternative to traditional pessimistic locking for coordination in HPC storage systems. The remainder of the paper is organized as follows. Section II identifies related work in optimistic coordination primitives. Section III explores the design space for implementing conditional storage primitives in a distributed object storage system. Section IV presents an empirical study comparing optimistic and pessimistic techniques using a parallel read/modify/write benchmark on a prototype object storage system. Section V presents conclusions from our study and outlines directions for future work.

II. Related Work

Optimistic coordination methods have been studied extensively in database systems. Kung and Robinson [7] established both the initial terminology and methodology for optimistic updates in distributed databases. They also identified the conditions in which those techniques would be more efficient than traditional locking algorithms. Agrawal et al. [8] later developed sophisticated models for comparing optimistic and pessimistic algorithms in database systems.

Optimistic coordination techniques have also been evaluated in object storage systems, though previous work has focused on attribute access rather than bulk data access. Devulapalli et al. [9] extended the T10 API to include compare-and-swap (CAS) and fetch-and-add (FA) operations for object attributes. CAS and FA primitives can be used to implement some common data structures in storage systems, such as linked lists, work queues, and delivery counters. Later work [10] also demonstrated that directory operations can be supported on object storage devices by using the CAS primitive to guarantee correctness. Lang et al. [11] explored the possibility of adding similar atomic operations (FA, enqueue, and dequeue) to storage systems by way of file system extended attributes rather than objects. The RADOS object storage API [12] provided as part of the Ceph storage system [13] supports arbitrary atomic object methods through a plug-in framework. This mechanism could be used to implement a variety of coordination primitives.

A number of distributed key-value storage systems have identified a similar need for conditional operators. These include conditional put and delete operations in Amazon SimpleDB [14]; the watch statement in Redis [15]; the condput operation in Hyperdex [16]; and the conditional put, update, and delete operations in Amazon DynamoDB [17].

Amiri et al. [18] evaluated concurrency control protocols for shared block storage devices. They considered three locking techniques as well as timestamp ordering in terms of both scalability and performance under contention.

III. Implementation Strategies

One can choose from multiple techniques when implementing a conditional write operation in a storage system. One approach is to implement a compare-and-swap primitive. To use a compare-and-swap primitive, the caller provides the original data along with the new data for a write operation. The server then compares the original data with the current data on disk. If the buffers match, the server atomically "swaps" the data on disk with the new data buffer. The principal advantage of this approach is that it requires no modification to the `read()` function and no additional metadata infrastructure. The approach has two drawbacks, however. The first is that each update must first read existing data off of disk in order to perform the comparison, thereby limiting performance for large updates. The second is that compare-and-swap primitives are susceptible to the "ABA problem" [19]. The ABA problem is a scenario in which a compare-and-swap operation succeeds even though different clients have updated the buffer and then changed it back to its original value before the compare-and-swap completes. Hence, even though the compare-and-swap will produce consistent results, it cannot be used as a mechanism to detect buffer modification.

We therefore elected to use a version-based conditional update strategy. At read time, the storage system produces a version number for the region being read. The client can then write the same region with an incremented version number. The server will reject the write operation if the incremented version number is not strictly higher than what is already on disk.

The granularity of the version number is a critical design consideration for version-based conditional primitives. A version number could apply to an entire object, a block, or an arbitrary extent. Extent-based versioning allows for smaller granularity and greater concurrency. We elected to use extent-based versioning for this reason. Our prototype builds on previous work on the Transactional Object Storage Device (TOSD, formerly known as the VOSD), which provides efficient, byte-granular atomicity and versioning in order to support the construction of lightweight replication protocols [20]. The TOSD uses log-structured storage for all data. It maintains an index that maps logical offsets to log offsets while simultaneously tracking a unique version number for each extent in the log.

An important practical consideration when relying on byte-granular versioning is how to handle the case where the read step in a read/modify/write operation spans multiple versions. In this case the storage system must either provide a compound version number (by appending multiple version numbers or hashing multiple version numbers) or promote the region to use a single, consistent version number that is greater than or equal to all version numbers contained in that region. We did not implement either approach in our prototype; instead, we allow conditional updates only to byte regions that share a single version number.

Another design consideration when adding conditional operators in an HPC storage system is the question of which level of abstraction should perform the actual conditional comparison to determine if a write will succeed. For example, this comparison could be performed by the storage server or provided natively by the local storage abstraction. The local storage abstraction layer is likely to be the most efficient location for the comparison, and this approach requires

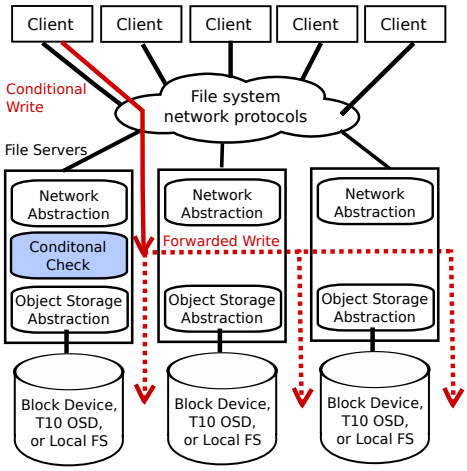

Fig. 1. Overview of optimistic conditional write path

minimal modification of the server or the storage system protocol. The drawback is that it complicates consistency if data is replicated across servers. In particular, it may produce nondeterministic results if concurrent conditional updates are applied in different orders on different replicas. To avoid this problem, we instead used the approach shown in Figure 1. A master server performs the comparison atomically on behalf of all replicas. If the conditional comparison succeeds, then the update is forwarded as a normal write operation to local storage and to replica servers. This approach allows the master server to serialize conflicting conditional operations and prevent nondeterministic ordering across replicas. Note that the master server is assigned on a per object basis. We expect that every server in the storage system will be a master server for some subset of objects.

In this work we demonstrate our conditional write implementation using a prototype distributed object storage system that uses MPI for communication, Aesop [21] as the programming model, and the TOSD [20] for local storage. Objects are distributed across servers algorithmically; clients can access those objects using conventional create, read, write, and delete primitives. The prototype also supports data "forks" that allow multiple independent byte streams to be associated with the same object. TOSD forks are similar to those offered by the Galley parallel file system [22].

IV. Evaluation

The experiments presented here were executed on the Fusion cluster in Argonne's Laboratory Computing Resource Center. Fusion is an IBM iDataPlex dx260 M2 system with a QDR InfiniBand interconnect. Fusion has 320 compute nodes consisting of two Intel Nehalem 2.6 GHz Xeon processors with 36 GB of RAM. Each node contains a local disk drive, but we elected to use a ramdisk on each node for experimental evaluation because of poor performance of the disk subsystem. The prototype software was built with MPICH 1.2.1 and GCC 4.1.2.

A. Benchmark

We constructed a parallel read/modify/write benchmark for use in evaluating our prototype. The read/modify/write access pattern has applications in a variety of namespace and coordination use cases, but our benchmark focuses on a simple 4 KiB block read/modify/write example. MPI is used to start a benchmark process on each compute node, synchronize timing, and aggregate results. Aesop is used within each client process to generate additional concurrency. Each read/modify/write operation selects a random object and a random fork within that object to read a value in, update it, and write the change back to the same block. Forks are used for convenience as opposed to modifying different offsets in a single fork. The benchmark code provides two mechanisms to ensure coherent updates. The first method uses pessimistic locking with a single lock server and a discrete lock for each object. The second method uses optimistic conditional updates enforced by a version number.

When executed in locking mode, each concurrent operation first acquires a lock for the object, reads a value, modifies the value, writes the value, and releases the lock. The locking method utilizes ZooKeeper [23], a distributed coordination service. ZooKeeper runs as a standalone process and has a C client library to link against for applications. ZooKeeper supports high reliability through redundant servers and stores data to disk. It is used by major organizations such as Yahoo!, Rackspace, and Zynga [24]. ZooKeeper itself does not provide a native lock mechanism, but it does offer a set of primitives that can be used to construct a lock. The Apache ZooKeeper suite publishes a recipe (protocol) for how to perform locks in ZooKeeper [25]. We implemented this locking recipe in a thread-safe manner with one deviation from the published recipe: we used a "get" operation to set a watcher instead of the "exists" operation. The reason is that using "exists" can leak a resource if a watch is set on a nonexistent node [26]. ZooKeeper was configured with default settings except for specifying the maxClientCnxns=0 option to prevent it from limiting the number of concurrent connections.

We used a single ZooKeeper lock server in all experiments and assigned a unique lock to each object in the storage system. Note that the performance of the locking infrastructure could be likely be improved through various architectural changes, for example by embedding the lock services in the storage system itself so that the number of lock servers scales automatically with the size of the storage system. This type of lock implementation sensitivity study is beyond the scope of this paper.

The conditional update method was described in Section III. For each read/modify/write update, the client reads a block and retrieves a version number for that block as part of the read operation. It then performs a conditional write of the same block with the version number incremented by one. If the write fails, then the client retries from the read step.

We did not hold the total number of operations in each experiment at a fixed value. Instead, we varied the number of

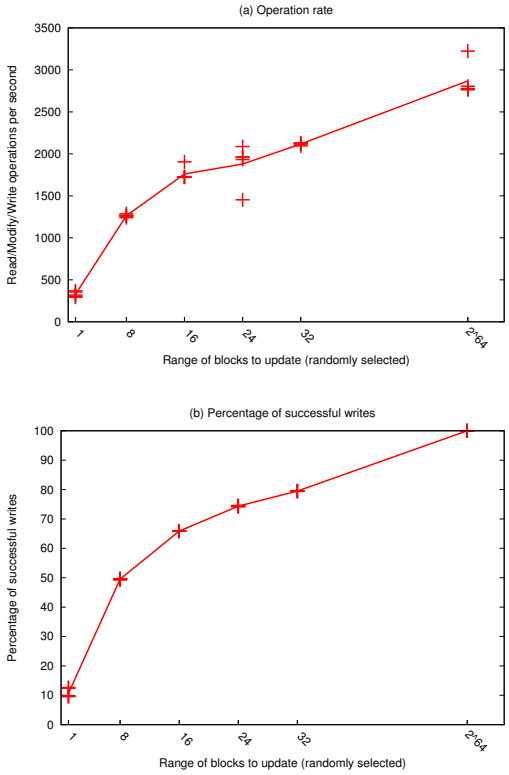

Fig. 2. Sixteen concurrent conditional read/modify/write operations with one client node and one server node. Each point represents an experiment; the solid line shows the average of all experiments.

operations performed in order to measure at least 60 seconds of sustained performance in all test cases. Each experiment was repeated five times.

B. Results

We compared the optimistic and pessimistic coordination techniques in terms of both sensitivity to update conflicts and overall scalability.

1) Sensitivity to conflict rate: Optimistic coordination primitives are most effective in situations with minimal contention. When contention occurs, all but one write will fail, and the client must retry any failed read/modify/write operations. These additional retries will degrade performance. Figure 2 illustrates this effect at small scale. In this experiment, we issued 16 concurrent read/modify/write operations from a single client to a single object on a server. Each concurrent operation in the benchmark modified a randomly chosen block, but we controlled the level of contention by varying the range of the random number generator from 1 (every update modifies the same block; 100% contention) to 2^{64} (every update modifies a different block; 0% contention).

Figure 2(a) shows the performance in terms of aggregate read/modify/write operations per second as the range of target blocks is varied. Figure 2(b) shows the percentage of conditional write operations that were successful for each data point. If all 16 concurrent operations attempt to read/modify/write the same block, then less than 10% of the updates are successful, and we see a corresponding average aggregate rate of 327 ops/s. If all 16 concurrent operations modify different blocks at all times, then 100% of the updates are successful, and we see a corresponding average aggregate rate of 2,869 ops/s. A correlation exists between the percentage of successful write operations and the observed read/modify/write performance.

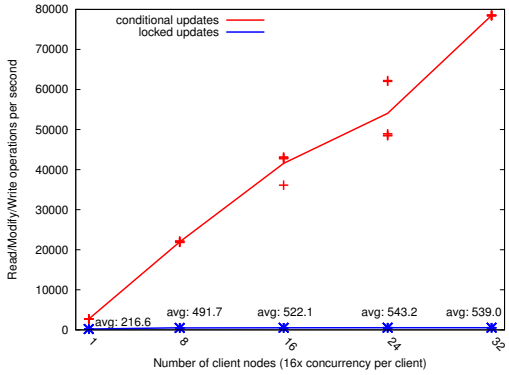

Fig. 3. Random read/modify/write rate with 32 server nodes and 16 concurrent operations per client (random updates to 2^{64} possible blocks, no contention). Solid line indicates average. Locking performance is annotated for clarity.

2) Scalability: We next evaluated the scalability of the optimistic conditional write primitives in comparison with traditional pessimistic locking. Figure 3 shows the ideal scenario, in which the range of blocks being modified is so large that contention never occurs among concurrent updates. In this example we hold the number of servers (and number of objects) constant at 32 and vary the number of client nodes from 1 to 32. Each client node submits 16 concurrent operations so that the aggregate concurrency varies from 16 to 512. The conditional read/modify/write approach attains an average rate of 78,492 ops/s with 512 concurrent operations. The lock-based read/modify/write operation in contrast achieves an average rate of only 539 ops/s. The lock-based approach does not gain any additional performance beyond 256 concurrent operations because the lock server becomes a central bottleneck for all operations. The optimistic approach is limited only by the number of data servers and the distribution of data on those servers.

Figure 4 repeats the previous experiment but limits the total range of blocks being updated to 512 (16 blocks per server). In this configuration, the number of blocks to be updated is constant, but the probability of contention increases as more concurrent operations are generated. The performance of the lock-based approach is not significantly affected by the level of contention produced by the benchmark. It therefore achieves a maximum average rate of 534 ops/s, which is within 1% of the performance attained by using pessimistic locking with no contention. The conditional results show an interesting trend, however. The performance scales well up through 256 concurrent client operations. At 512 concurrent operations, however, the performance drops significantly despite a modest decrease in the percentage of successful write operations. This indicates that the level of contention has a stronger impact on performance as the overall concurrency is increased. Even

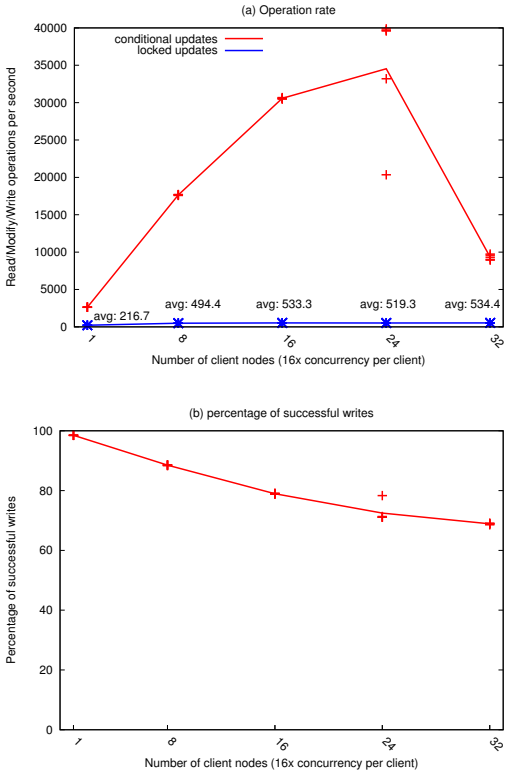

Fig. 4. Read/modify/write operations with 32 server nodes, 16 concurrent operations per client node, and random updates to 512 different blocks. Solid line indicates average. Locking performance is annotated for clarity.

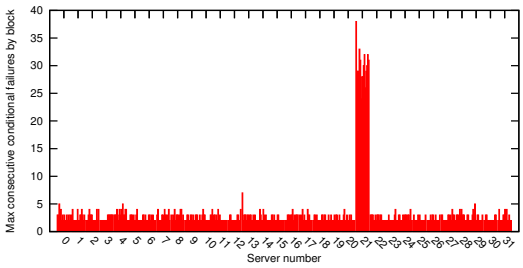

Fig. 5. Maximum consecutive conditional write failures per block with 32 servers, 32 clients, and random updates to 512 blocks.

in this case, however, the conditional write approach still outperforms the lock-based approach by a factor of 17.

We instrumented the object storage server to better understand the cause of the observed performance degradation with 512 concurrent operations. Figure 5 shows the maximum number of consecutive failed write attempts for each block for an example benchmark run that corresponds with the largest conditional examples shown earlier in Figure 4. This example used 32 servers, 32 clients, and 16 concurrent operations per client to modify 512 randomly selected blocks. We see that most blocks experienced no more than 5 consecutive conditional write failures. The 16 blocks hosted on server 21, however, experienced anywhere from 24 to 38 consecutive conditional failures despite the overall random distribution of updates. We believe that this is caused by a cascading contention effect. Once a server begins to experience contention, the additional retry traffic may degrade server performance. Contention on other blocks therefore becomes increasingly likely as the server takes longer to process all read/modify/write operations. This indicates that conditional operation primitives may benefit from a back-off algorithm to alleviate server pressure once high contention is observed. A server could control the back-off strategy itself by delaying acknowledgments for failed conditional update requests. This approach would allow the back-off algorithm to take advantage of statistics gathered by servers to make intelligent back-off policy decisions.

V. Conclusions

In this work we used a prototype system to evaluate the effectiveness of optimistic conditional write primitives as an alternative to traditional, pessimistic distributed locking for coordination in an HPC storage system. The conditional write primitives leveraged a versioning object storage abstraction to provide byte-level granularity for maximum update concurrency. The resulting system scales automatically with the number of servers and objects with no dependency on external services to provide coordination. Our experiments show that conditional update performance scales nearly linearly on a Linux cluster with up to 512 concurrent operations in cases with little contention. We also discovered that conditional update performance may degrade significantly if steps are not taken to throttle the retry rate in high-contention scenarios. Despite this behavior, conditional updates were found to outperform a standard pessimistic locking implementation by a wide margin in all cases. We therefore believe that optimistic coordination primitives are an excellent alternative to pessimistic locking for a wide range of potential HPC storage use cases.

In future work we intend to explore back-off algorithms that automatically account for server processing rate and contention levels in order to prevent server performance degradation in high-contention scenarios. We also hope to explore the use of conditional updates in common file system use cases, such as maintaining a consistent file and directory namespace. We will also evaluate the use of conditional updates in managing the consistency of distributed data structures in scientific data analysis.

Acknowledgments

This material is based on work supported by, or in part by U.S. Department of Energy's Oak Ridge National Laboratory and included the Extreme Scale Systems Center, located at ORNL and funded by the DoD in part by the "Novel Software Storage Architectures" contract. This work also was supported by U.S. Department of Energy, under contracts DE-AC02-06CH11357 and DE-FG02-08ER25835.

We gratefully acknowledge the computing resources provided on "Fusion," a 320-node computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

[1] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan, "Integrating parallel file systems with object-based storage devices," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 27:1–27:10.

[2] S. A. Weil, A. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A Fast, Scalable, and Reliable Storage Service for Petabyte-scale Storage Clusters," in *Proceedings of the 2007 ACM Petascale Data Storage Workshop (PDSW 07)*, Reno, NV, Nov. 2007.

[3] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "Blobseer: Next-generation data management for large scale infrastructures," *J. Parallel Distrib. Comput.*, vol. 71, no. 2, pp. 169–184, Feb. 2011.

[4] S. Narayan, J. Chandy, S. Lang, P. Carns, and R. Ross, "Uncovering errors: The cost of detecting silent data corruption," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 37–41.

[5] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[6] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 4–4.

[7] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, June 1981.

[8] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modeling: alternatives and implications," *ACM Trans. Database Syst.*, vol. 12, no. 4, pp. 609–654, Nov. 1987.

[9] A. Devulapalli, D. Dalessandro, and P. Wyckoff, "Data structure consistency using atomic operations in storage devices," *IEEE International Workshop on Storage Network Architecture and Parallel I/O*, pp. 65–73, 2008.

[10] N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, and P. Sadayappan, "An OSD-based approach to managing directory operations in parallel file systems," in *IEEE International Conference on Cluster Computing*, Sept. 2008.

[11] S. Lang, R. Latham, D. Kimpe, and R. Ross, "Interfaces for Coordinated Access in the File System," in *Proceedings of 2009 Workshop on Interfaces and Architectures for Scientific Data Storage*, Sept. 2009.

[12] "Librados API documentation." [Online]. Available: http://ceph.com/docs/master/api/librados/

[13] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI*, 2006, pp. 307–320.

[14] "Amazon SimpleDB," http://aws.amazon.com/simpledb/.

[15] "Redis," http://redis.io/.

[16] "Hyperdex: A Searchable Distributed Key-Value Store," http://hyperdex.org/.

[17] "Amazon DynamoDB," http://aws.amazon.com/dynamodb/.

[18] K. Amiri, G. A. Gibson, and R. Golding, "Highly concurrent shared storage," in *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*. Washington, DC: IEEE Computer Society, 2000, pp. 298–.

[19] M. Michael, "ABA prevention using single-word instructions," *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

[20] P. Carns, R. Ross, and S. Lang, "Object storage semantics for replicated concurrent-writer file systems," in *Proceedings of 2010 Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS 2010)*. IEEE, 2010.

[21] D. Kimpe, P. Carns, K. Harms, J. Wozniak, S. Lang, and R. Ross, "Aesop: Expressing concurrency in high-performance system software," in *Proceedings of 7th IEEE International Conference on Networking, Architecture, and Storage (NAS 2012)*, 2012.

[22] N. Nieuwejaar and D. Kotz, "The Galley parallel file system," in *Proceedings of the 10th ACM International Conference on Supercomputing*. Philadelphia: ACM Press, May 1996, pp. 374–381.

[23] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIX Association, 2010, pp. 11–11.

[24] "ZooKeeper Users." [Online]. Available: https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy

[25] "ZooKeeper Lock Recipe." [Online]. Available: http://zookeeper.apache.org/doc/r3.3.5/recipes.html#sc_recipes_Locks

[26] "ZooKeeper Watch Jira Request." [Online]. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-442