

# Data Structure Consistency Using Atomic Operations in Storage Devices

Ananth Devulapalli  
Ohio Supercomputer Center  
1224 Kinnear Rd  
Columbus, OH 43212  
ananth@osc.edu

Dennis Dalessandro  
Ohio Supercomputer Center  
1224 Kinnear Rd  
Columbus, OH 43212  
dennis@osc.edu

Pete Wyckoff  
Ohio Supercomputer Center  
1224 Kinnear Rd  
Columbus, OH 43212  
pw@osc.edu

## Abstract

*Managing concurrency is a fundamental requirement for any multi-threaded system, frequently implemented by serializing critical code regions or using object locks on shared resources. Storage systems are one case of this, where multiple clients may wish to access or modify on-disk objects concurrently yet safely. Data consistency may be provided by an inter-client protocol, or it can be implemented in the file system server or storage device.*

*In this work we demonstrate ways of enabling atomic operations on object based storage devices (OSDs), in particular, the compare-and-swap and fetch-and-add atomic primitives. With examples from basic disk resident data structures to higher level applications like file systems, we show how atomics-capable storage devices can be used to solve consistency requirements of distributed algorithms. Offloading consistency management to storage devices obviates the need for dedicated lock manager servers.*

## 1 Introduction

Controlled access to shared resources is a fundamental requirement for any multiprocess system. The resource could be a hardware device, memory region, or block of storage on a disk. The simplest form of controlled access is the exclusive lock, where at most one process is permitted to access the resource. More complex forms, such as multiple-reader locks and queuing systems, are also possible.

There are numerous examples where serialized access to storage devices is essential. It ranges from database transactions to file system metadata operations which demand consistency and correctness guarantees. The work in this paper is at a lower level than the applications. Here we are proposing atomic primitives to be supported directly by the

storage device. We believe that atomics-capable disks will simplify synchronization management at the higher level applications.

Cooperative locking is by necessity implemented using inter-process communication, either by directly sending messages between interested processes, or by changing the state of some shared resource to perform the lock protocol. More complex protocols are frequently built on simple primitives offered by hardware, such as the “test and set” or “load locked, store conditional” operations of many multi-processor systems [12, 15].

There are two major ways to implement resource locking. First is through cooperative algorithms that are distributed across all processes, such as Paxos [14]. The second is through centralized locking, where a single entity arbitrates access to the shared resource. This is commonly found in distributed lock managers [11], where processes make requests for a lock to a central server, which grants access to only one at a time. In centralized locking, the lock arbitration may occur at the resource itself, or at a separate server.

In some situations, only centralized locking where the lock is at the resource is sufficient. Both distributed cooperative locking and centralized locking using an external server have the *split brain* problem. Because the locking operations go across a network that is different from where the resource lives, it is possible for the processes to communicate with the resource, but not to be able to obtain a lock to do so. In practice, this problem is worked around using multiple redundant communication mechanisms, but an easier approach is to implement the locks inside the locked resource, or in our case, on the disks.

This paper discusses ways to implement locking on storage devices, in particular on object-based storage devices (OSDs). We argue that locking at the level of an object is both natural from a programmer’s point of view, and scalable from a resource management point of view. We present in Section 4 design choices for building primitives for locking, which can be used for a broad class of algorithms that

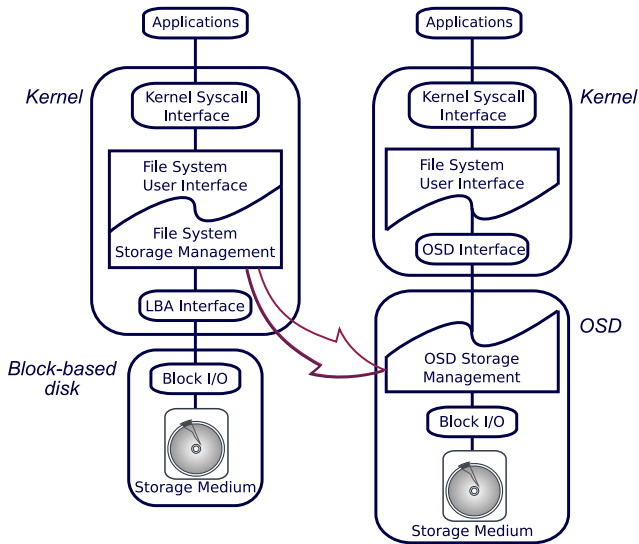
---

This work was supported by NSF through award CCF-0621484.

require atomic state modifications. Examples of how these features can be used are shown in Sections 6 and 7. Experimental evaluation is described in Section 8 followed by our conclusions in Section 10.

## 2 OSD Background

Object-based storage devices (OSDs) place increasing responsibility for data layout and management on the devices themselves. Unlike traditional block-based disks that represent storage as a linear array of bytes and export a logical block address (LBA) interface, OSDs export an object interface. Enabling a high-level object interface allows OSDs to take over low-level data layout management operations from the host operating system as shown in Figure 1. According to the OSD specification [22], an object is a fundamental entity that has attributes and stores data. The object data is made up of a sequence of related bytes. Some object attributes can be user-defined, and all attributes are managed by the device. This capability of extensible attributes gives users powerful semantic control over management of the data.



**Figure 1. Comparison of block-based and object-based disks.**

We use this attribute feature of OSDs as a natural mechanism with which to implement atomic operations. Per-object state is represented as a value in an attribute. These “lock” attributes are interpreted only by applications and do not have any particular meaning to the storage device itself. By invoking the correct atomic primitives to modify these attributes, applications can trust that the OSD will maintain state information needed to implement the exclusivity algorithm.

OSDs offer commands to read and write data, create and destroy objects, and get and set attributes, among others. The attribute modification commands are special in that they can be attached arbitrarily to any other command. This allows particular optimizations to reduce round-trip transfers between an application and the device. The attribute number space on an object consists of two 32-bit identifiers: an “attribute page” and an “attribute number.” The value stored in a particular attribute identified by these identifiers is an arbitrary list of bytes, up to 16 kB in length. The attribute value need not have any particular format, although applications will find it useful to interpret the bytes in certain ways, as an integer or a string, for example.

## 3 Locking on Storage Devices

In this section we describe some of alternatives for enabling atomic primitives on OSDs.

### 3.1 Current approaches

To implement a protocol for exclusive access using storage devices, it is necessary for multiple clients to use storage commands to ensure that only one of the clients exits the algorithm knowing it has acquired the lock. While OSDs do support a highly flexible set of independent attributes on each object, they do not have any special semantics that enable exclusive access. In this respect they behave just like the data in the objects: no guarantees are provided that two separate commands will occur without intervention by a command from another client.

This lack of atomicity is the same issue that prevents locking algorithms being built using today’s block-based devices. If one assumes that data reads or writes are atomic at some granularity, such as at the 512-byte sector level, it is possible to construct series of reads and writes to obtain a conceptual lock. However, such atomic behavior is not guaranteed by the specifications, nor would the multi-step algorithms be very efficient.

There is support for RESERVE and RELEASE commands in some SCSI devices, which allow a client to take over the device for itself and restrict access from other clients. This is a very coarse operation, in that the disk may not process any commands for other clients during the reservation. It also had the problematic behavior that a bus reset or device power cycle will break an existing reservation. This can cause the unexpected loss of a lock when a machine unexpectedly reboots, even one not currently involved in the locking activity. A more recent SCSI primary command standard, version 3 [21], adds “persistent” reservations that optionally allow devices to preserve the state across a power cycle, although they also limit access to only

a single client and thus prohibit use of the entire device (not just a few blocks) during the reservation.

### 3.2 Alternative mechanisms

Due to these fundamental limitations of existing storage devices, and the need to implement scalable resource-resident exclusive access protocols, we seek to extend the OSD specification by adding commands that can be used to build locking algorithms. One approach would be to add a LOCK command that can claim a particular object for a client, much like the SCSI RESERVE, but at object granularity. This would work, and solve many of the issues called out above; however, it requires particular state at each object to track the locks, even across power cycles. In an implementation, it requires that any object access must first check the lock state before proceeding, possibly slowing down all operations.

Instead we leverage the existing attribute infrastructure on OSDs by adding a way to atomically modify a particular attribute on an object. This is not a persistent lock, but simply a pair of operations such as read-modify-write or test-and-set, that can be performed by the device on a single attribute without interruption. This approach adds no overhead to any other operations, requires no separate state (beyond existing attribute support), and is very easy to implement.

Several atomic primitives have been used for synchronization. In shared memory multiprocessors, where atomic operations are widely supported in machine instruction sets, primitives like load-linked, store-conditional, compare-and-swap, test-and-set and read-modify-write provide synchronization mechanisms. However, Mellor-Crummey et al. [15, 16] showed that compare-and-swap (CAS) is the most powerful among these atomic primitives and can be used as a building block for more sophisticated locks, like shared and multimodal locks. This motivated us to include compare-and-swap and fetch-and-add as the two atomic primitives for our OSD implementation. The following section explores the design options for these primitives in the context of OSDs.

## 4 Atomic operations design choices

Even for operations as conceptually simple as compare-and-swap (CAS) and fetch-and-add (FA), there are many options regarding their implementation on OSDs. The two constraints we seek to impose are: atomic operations should be broadly useful by a variety of applications; and, the ability to support atomic operations should not unduly complicate other functions of the OSD. The following paragraphs discuss some of the trade-offs.

**Number of atomic attributes:** The number of attributes per object dedicated for atomic operations significantly affects the implementation of atomic operations. One might use a single attribute per object for atomic operations or might use many attributes. It is very natural to associate one lock per object and indeed several programming paradigms follow that construct. But it unnecessarily constrains non-interfering operations to be serialized. For example, two operations modifying different attributes in an object will be forced to be serialized, when instead they could be executed simultaneously. Therefore, limiting atomic operations to a single attribute might be natural to implement, but induces false sharing.

**Attribute location:** Depending on the number of dedicated atomic attributes, the choice of attribute location might be important. For example, in order to implement single-attribute atomics, we may select one of the “reserved user object pages” [22] and designate attributes within that page for atomic operations. That would place the atomic attributes at a well-known location for each object. It might be sufficient for applications requiring locks on the whole object, but it constrains applications which might want to lock the object using a different section of attribute space.

**Attribute length:** In traditional SMPs or in high-performance interconnects like InfiniBand [9] that support remote-atomic operations, atomic operations are executed on word-size operands. These operands are typically 8 bytes for modern architectures, and a fixed word size would be easier to implement in hardware. However, some applications may prefer longer or variable length attribute values, perhaps to store arbitrary string values as the lock item.

**Undefined attributes:** The traditional CAS operation would atomically swap the value only if the comparison succeeded. However in the OSD case, an undefined attribute has a value of length zero, and is treated differently than defined attributes. The CAS operation can be extended to handle this case as follows: if the attribute value was set (i.e., defined), perform the CAS as previously described; however, if it was not defined, perform the CAS as if the comparison succeeded and always update the value. For FA, an undefined attribute can behave as if it were defined with the value zero. These extensions simplify some algorithms by removing an explicit lock initialization step.

**Advisory or mandatory:** Should it be possible for attributes that are used for atomics to be modified by non-atomic operations? This is akin to asking if the use of CAS is advisory or mandatory. It would be possible, especially with fixed attribute location, for the device to disallow a

simple SET ATTRIBUTES command that targets a atomic attribute. Doing so, however, would require an extra check even for non-atomic operations and goes against the spirit of limiting the interference of atomic operations on normal OSD operations.

**Isolation level:** Obviously the CAS or FA operation itself must be indivisible. But should atomicity guarantee be extended to the optional get and set attribute tasks? An ongoing atomic operation might prevent any other access to the entire object and all its attributes. Or the CAS or FA may be limited in scope to modify just the single attribute in question, allowing other operations (such as READ and WRITE) to proceed simultaneously. While a wide scope for isolation may allow some algorithms to securely fetch multiple attributes in a single operation, this may be more invasive to implement as other non-atomic commands also must be paused during the duration of the compare-and-swap or fetch-and-add.

**CAS effect on other attributes:** Every OSD command in addition to executing the operation can optionally get or set attributes. The OSD specification carefully enumerates the ordering of various tasks during the execution of an OSD command: the command itself is performed first, then attributes are set, and finally attributes are retrieved, as specified by the command. In defining a CAS command, we may extend its semantics so that other set attribute commands in the set attribute list are only performed if the CAS “succeeded,” that is, if the swap was performed. This permits applications to do a one-step operation of acquiring the lock and updating one or more other parameters. If the lock could not be acquired, the other parameters are not updated. The lock can optionally be released in this same command by adding another set attributes entry for the lock parameter itself, allowing single-command modification of multiple attributes under a lock.

## 5 Atomic operations implementation

Based on the above design space and experimentation with many prototypes, we have settled on a single implementation. The code was tested and validated in our OSD initiator and target software [6, 7].

Two new bidirectional OSD commands, CAS and FA, are defined. They are modeled after the existing SET ATTRIBUTES command, which specifies a particular object, and can both set and get attributes in the single command. It does this by providing two lists: attribute locations and values to set, and other attribute locations to retrieve.

For CAS, the compare and swap values are conveyed as the first two attributes in the list of attributes to set. The

OSD will atomically compare the “compare value” against the value already in that attribute, and if identical, it will replace it with the “swap value.” The “original value” may optionally be retrieved by requesting an attribute from the Current Command Attributes Page. This usage style matches how other OSD operations send multiple data items, and how they return results from the operation itself, as distinct from returning data or existing attributes.

The fetch-and-add command, FA, operates similarly, but requires only the single “add value” to be sent by the initiator. This value (possibly negative) is added to the existing attribute, and the original value may optionally be retrieved as for CAS.

Both CAS and FA may operate on any attribute—there is no fixed location. This allows multiple uses of locking in a single or multiple applications to be supported simultaneously, without any external coordination.

The length for the values in a CAS operation is arbitrary, up to the maximum of 64 kB as defined by the OSD specification [22]. For FA, however, we chose to limit its scope to 8-byte words that are treated as signed integers during the computation. The reason for limiting the size to 8-bytes is that at present computation on 8-bytes words is standard and the main purpose of FA is to progressively generate unique tokens and 8-bytes are sufficient for the applications. However, the design is extensible to larger or smaller word sizes if required.

The implementation does treat undefined attributes as described above: CAS will succeed when encountering an undefined value, and FA will treat it as zero. This was observed to simplify our test applications considerably.

Partially due to the lack of a fixed location for attributes, and also driven by the desire to keep the implementation simple and avoid interference with the rest of the OSD, both CAS and FA are purely advisory. Other commands may set or get the attributes affected by the atomic operations, without using them. They are also atomic only at the level of the attribute on which they operate. Other commands that affect other aspects of the object may proceed concurrently.

While conceptually attractive, we decided not to include the ability of using the result of the CAS comparison to potentially disable other attribute modifications in the same command. In the implementation, some complexity was necessary in order to squash the later attribute changes and to carry around the result of the CAS in order to do so. No compelling application example was found, either.

## 6 Distributed data structures

In this section we describe some fundamental data structures that are used on shared storage, and discuss how they could be implemented on object-based storage devices us-

ing the new compare-and-swap and fetch-and-add operations.

## 6.1 Linked list

The classic approach to have objects “point” to other objects is a linked list. Linked lists are used in several popular disk-resident data structures used by filesystems such as hash tables and B-trees. When multiple processes are involved in concurrent modification of a linked list, some form of synchronization is required to ensure consistency.

On OSDs, a natural way to store the items or objects that are being chained together is one per OSD object. Then to link objects together, attributes on the objects can be used to point to other objects, by storing the object identifiers as values of particular attributes. Data structure consistency can be enforced by using a single global lock to protect the entire list, but finer grain concurrency can be supported by using compare-and-swap to modify links. For instance, inserting a new element at a position in the chain involves setting up the pointers on the new element, then changing the forward link on the previous item in the existing list to point to the new element. This is only safe to do if the link has not changed during the time the new element is being constructed. A CAS operation can be used to ensure that the old value of the link is what was expected, achieving a lock-free insert algorithm for this particular data structure.

## 6.2 Work Queue

A work queue is used to regulate processing steps on a set of items. Multiple producers insert new items into the queue, and consumers retrieve the items and process them. Order among the items is not important, but it is crucial to ensure that every item gets processed exactly once by a single consumer, and that no items are lost. It is natural to identify these items with OSD objects.

Producers can allocate new objects using the CREATE operation that guarantees a distinct identifier for each object. Consumers can use the LIST command or an atomic counter (described in the next section) or some other out-of-band mechanism to identify new items. For a consumer to claim an item, it must be sure that no other consumer has done so. This is possible by defining an attribute on the work item to indicate the object has been “claimed” and to require consumers to modify that attribute using CAS to attempt to claim an object. Even if two consumers attempt this operation at the same time, the OSD ensures that only one will succeed. The winner can perform its processing then change the attribute to deliver it to future processing phases.

## 6.3 Delivery Counter

In some situations, there is exactly one resource, and multiple clients must manipulate it exclusively. A common way to ensure this is with a spin lock, easily implemented on an OSD by using a compare-and-swap operation on a well-known attribute of an object. Clients must poll the OSD, continually retrying the CAS operation, to check for availability of the resource.

When there are many clients or the resource occupancy time is high, it is better to have clients queue up for service to avoid the polling overheads on the device and the network. The classic solution to this problem is the bakery [13] algorithm which relies on a unique number generator. The fetch-and-add command can be used to deliver a unique, increasingly ordered “ticket” to a client, as is done to manage a line of shoppers waiting for attention from the delicatessen attendant. When the ticket is “called,” the shopper can make his purchase.

In the OSD case, there is no mechanism for signaling to clients when a ticket is ready, but this can be implemented out-of-band by a number of mechanisms. For instance, ticket holders can place their IP addresses or host names, or some system identifier suitable for the application, into an attribute slot for that ticket. As the current resource owner completes the exclusive section, it can release the lock and signal the ticket holder that is next in line. Some amount of polling will be required to avoid race conditions and account for failed clients, but much less frequently than with the spin lock approach, especially when the waiting queue is long.

## 7 Application case studies

In this section we discuss two applications that use atomic operations on OSDs. This builds on the previous section by highlighting exactly how the new operations can be used.

### 7.1 File system directory

It is possible to implement a traditional Unix-style file system on an OSD. If the file system is to be used in a distributed environment, where multiple clients can access it concurrently without an intervening server, care is required to avoid corrupting file system data structures. This direct-access approach is different than network-attached file systems like NFS that mediate access using servers; it is also different from direct-attached block storage because multiple clients can access the files at the same time.

In a related work [6], we adapted an existing distributed parallel file system to use OSDs natively. Parallel Virtual

File System [3] is a parallel filesystem used mainly in high-performance computing environments serving I/O intensive parallel applications. By using OSDs as the back-end data and metadata store, we are able to eliminate all file system servers completely, letting clients directly access the storage.

We chose to implement directories as OSD objects and directory entries as attributes of that object. The location of the directory entry within the attribute page is determined by the hash value of its name. The contents of the attribute contain the entry name and the object identifier to which it points. Because no two entries in a directory are permitted to have the same name, the algorithm to insert a new directory entry uses CAS to ensure that the slot (and hence the name) was previously empty. This avoids the race condition where two clients attempt to create the same file at the same time. CAS is also used on entry removal to avoid the situation where another client quickly deletes and reinserts a new file, causing the wrong file to be deleted.

One problem that arises when dealing with hash tables is the possibility of a collision. In our case, due to the finite number of entries in an attribute page, multiple directory entries will be hashed to the same location with some non-zero probability. The conventional solution to a collision problem is chaining [4]. When multiple entries collide into the same hash bucket, they are chained into a linked list. This is the approach we follow for collision resolution.

within the lock page is to serialize modifications to linked list in the corresponding hash table entry. There is a wrap-around counter that points to the next available entry in the linked list page. It is implemented as an attribute in some other page.

Each directory entry encoding should uniquely identify a file or directory. The encoded value of the directory entry is stored as an attribute value in the hash table page and, in the case of collisions, in the linked list page. As shown in the figure, this encoded value is made up of a mark bit, next pointer, and the directory entry name that includes the 8-byte object identifier for the entry.

Figure 2 shows a snapshot of the state of different data structures. In the common case, for names “dirent-56” and “dirent-59,” there is no collision and the mark bit and next pointer are zero. Rarely, multiple names will hash to the same attribute slot. Here, slot 455 is shared by “dirent-20,” “dirent-78” and two others. As a result, the next pointer specifies a location in the linked list page that contains the second directory entry for that slot. Further links in the chain are found from there. The mark bit is used to control access to the linked list page so that the algorithm can take the fast path and update directory entries in a single step. All the data structures reside on disk and atomic operations are used extensively to ensure consistency.

## 7.2 Avoiding split brain in fail-over

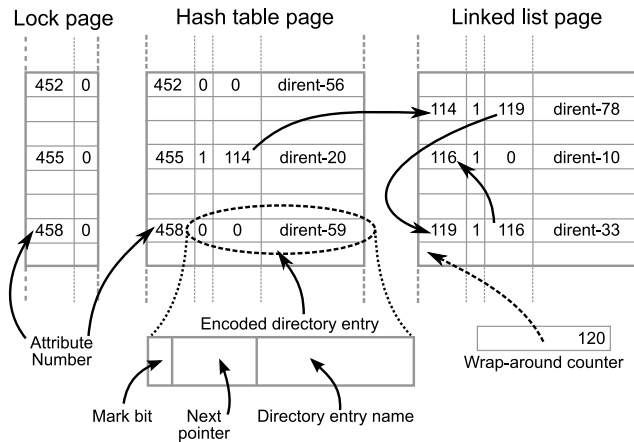


Figure 2. Hash table data structure.

The encoding of the directory entry and the design of the data structures used is influenced by the hash table collision resolution scheme. Figure 2 shows the OSD-resident data structures used by our scheme. The hash table page stores the directory entries. This is the entry point into the hash table. A directory entry name is hashed to get an attribute number in this page. The linked list page stores the remaining nodes of the linked list and is only used in the case of collisions. The lock page stores a lock for each entry corresponding to the hash table entry. The purpose of a lock

As mentioned in the introduction, the classic fault tolerance setup involves two (or more) machines connected to a single disk. One machine is “active,” performing the service and saving its run-time state on the shared disk. In the event that it fails, the other machine notices and takes over the service by reading the state from the disk and continuing. The way that the back-up machine detects failure is usually through “pinging” the host on a separate network. In the case of no response, it declares it dead and takes over the service.

However, the back-up machine will also declare the active machine dead if the network fails, resulting in the case that both machines attempt to use the shared disk and likely data corruption. This is the “split brain” scenario, and it is usually generalized to network partitions involving whole subsets of machines that become isolated from each other.

An alternate approach is to place liveness detection into the storage system. The active client is responsible for updating a timestamp somewhere on the disk. The back-up machine watches the timestamp to ensure progress, and initiates take-over if it infers that the active machine has failed. When multiple machines are involved, the decision on the next leader could be resolved by the one that succeeds in setting up its identifier as the leader on the storage device. The primary on its return should check with the storage the

current consensus on the leader. Traditionally these issues are handled using SCSI reservations that have the problematic issues mentioned earlier. With atomic operations, this process can be modeled as a work queue, defining objects to represent services and using attributes on those to represent ownership and activity timestamps.

## 8 Lock throughput experiment

While this paper focuses exclusively on the use of atomic operations on storage devices directly, the leading approach in practice is to use a distributed lock manager. A DLM is an service implemented on one or more machines, outside of the storage system, that grants access to shared resources. We argue that implementing locking directly on disks is more natural, requires fewer components, and avoids the “split brain” scenario to implement locking directly on disks; however, a server-based DLM approach often is a perfectly reasonable approach for some applications.

This section shows one experiment to gauge the performance of on-disk locking against a DLM. We conducted the experiment on a Linux cluster where each node has two AMD Opteron 250 processors, 2 GB of RAM, 80 GB SATA disk and runs version 2.6.25-rc1 of Linux operating system. The onboard Tigon 3 Gigabit Ethernet NIC is used for communication, with a single SMC 8648T 48-port switch.

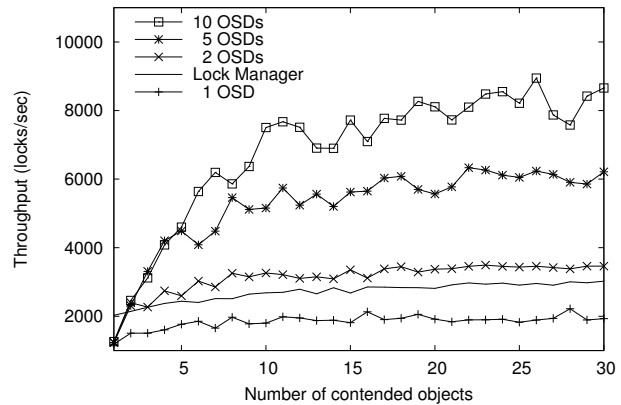
There are quite a few caveats regarding this experiment. It was not designed to demonstrate a performance advantage for OSDs; in fact, due to our use of a software object-based storage device, it is difficult to make any solid claims about expected OSD performance. We implemented a lock manager as a single server process, not as a group of processes sharing the load of many locks. This was intentional to mimic the typical environment that has tens or hundreds of disks for each lock manager server.

The lock manager is implemented using a very light weight protocol and only serves *exclusive locks*. Its software overheads are likely low compared to production-quality implementations. In order to tolerate faults, the lock manager commits every data structure change to its internal disk. This is necessary and common in practice, as otherwise, a fault in the lock manager would not be recoverable. The OSD also commits every attribute change to the disk. For our lock manager implementation we use `O_DIRECT` mode writes for the log file with commit granularity of 512 B. This mechanism bypasses normal file system buffering and delivers the highest throughput to the stable store.

Since locking in OSDs does not require any central server for synchronization, the processes must re-attempt lock acquisition in the case of contention. It is not acceptable for them to busy-loop as that would saturate the network and the OSDs. This problem similar to the col-

lision avoidance problem on shared media, such as Ethernet hubs [17]. The classic solution to that problem is truncated binary exponential backoff. In the context of SMPs, this exponential algorithm has been adapted for lock contention [2, 15], and we use a similar approach in the case of OSDs.

The experiment was conducted as follows. For a given number of OSDs, the root process populates them with a predefined set of objects. A subset of these objects are selected as the locking set. Contention for locks is inversely proportional to the number of objects in the locking set. A set of 10 processes perform a fixed number of iterations across the following steps: select an object at random from the set, acquire the lock for that object, write 256 bytes of data to the object, and finally release the lock. In the centralized lock manager case, locking and unlocking are handled by it alone, and in the case of OSDs, the processes rely on the CAS (Section 5) operation to manipulate locks that exist as attributes on the objects. At the end, the root process stops its timer and calculates an overall throughput value.



**Figure 3. Lock throughput for four OSD configurations, and for a single distributed lock manager, which is shown as the curve without any symbols.**

Figure 3 shows the lock throughput for 5 configurations. The four OSD cases are for the number of drives in the system, where all OSDs share the locking load, because locks are implemented directly on the objects that the test processes are attempting to manipulate. The single DLM case is shown as a solid line without symbols in the figure. It manages all locks for the devices.

In all cases, throughput is low when there is only one object in the set, yielding a very high level of contention for the lock on that object. As more objects are put into play, the chance that multiple processes will randomly select a single object at the same time goes down, and throughput increases. Limited device (or DLM) capacity causes all

curves to flatten out at low contention values (to the right).

The figure shows that the single DLM process provides better throughput than a single OSD. This is due to iSCSI protocol overhead and the slower algorithms involved by using SQLite as the back-end attribute storage system in the OSD; while the lock manager uses a minimal network protocol and direct writes to the disk. In the expected use case where a shared storage system is comprised of many disks, each disk manages the locking for the objects it hosts. This allows the throughput to scale almost linearly as more object-based storage devices are added to the system, as shown in the separate curves for 1, 2, 5 and 10 OSDs.

(There is a further optimization in the case of OSD locks that we do not take advantage of here. It involves release the lock by setting the lock attribute to zero along with the write command. This is safe as the write is performed with the lock held. It saves one round-trip CAS in the OSD case, and pushes the single OSD throughput up to roughly coincide with the lock manager throughput.)

## 9 Related work

Intelligent disks [8] have been proposed earlier to reduce latency and improve the performance of data intensive workloads. The main motivation was to exploit the reducing cost of compute logic by adding computation elements closer to the hard drive, thereby offloading computation to the peripheral and improving the performance of applications dealing with large and streaming data. This idea was earlier popularized by Active Disk [1], Active Storage [18] and iDisk [10] projects, which specifically targeted data streaming applications like decision support systems which are pretty common in database domain. Semantically smart drives [20] and type-safe disks [19] have been proposed to exploit the information that is mined from the filesystem data structure operations on the disk, to improve overall system performance by techniques like prefetching, efficient data layout and better caching support. In contrast, our work is focused in trying to enable atomic operations on a disk, which is clearly complementary and an extension to the previous work.

Other peripherals which support atomic operations include advanced interconnects like InfiniBand [9], where these operations are heavily used for synchronization of cooperative processes. Atomics enabled network peripherals ease the realization of a distributed shared memory system out of a cluster of computers. One of our main motivations in enabling atomic operations on disks is when the cooperation of the clients happens only through the storage device, and enabling atomics operations on disk enables the disk to unify all the serialization tasks.

Machine support for atomic operations are widely found in multi-processor systems [12]. Compare and swap has

been shown [15, 16] to be most generic atomic primitive on which other synchronization primitives could be build. Unlike shared memory systems, distributed lock manager (DLM) [5, 11] is the standard way of providing locking services in distributed memory systems. By enabling atomic operations on the disk, our work complements the existing distributed systems, which can use these disks to address the *split brain* problem.

## 10 Conclusion and Future work

In this work we proposed the addition of two new commands, compare-and-swap and fetch-and-add, to support concurrency control that is integrated in the storage system. The approach extends the existing attribute model of OSDs by providing the capability to modify an attribute atomically within a single command.

Providing primitives at storage devices for the management of concurrency can lead to elegant and scalable distributed data structure algorithms.

We have submitted our work to the OSD technical working group and are working closely with them to incorporate the features described in this paper.

## References

- [1] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, 1998.
- [2] T. E. Andersen. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.
- [3] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.
- [4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.
- [5] R. G. Davis. *VAXcluster Principles*. Digital Press, 1993.
- [6] A. Devulapalli, D. Dalessandro, N. Ali, and P. Wyckoff. Integrating parallel file systems with object-based storage devices. In *Proceedings of SC'07*, Reno, NV, Nov. 2007.
- [7] A. Devulapalli, D. Dalessandro, and P. Wyckoff. Attribute storage design for object-based storage devices. In *24<sup>th</sup> IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, San Diego, CA, Sept. 2007.
- [8] G. F. Hughes. Computers: wise drives. *IEEE Spectrum*, 39(8):37–41, 2002.
- [9] InfiniBand Trade Association. *InfiniBand Architecture Specification*, Oct. 2004.
- [10] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, 1998.

- [11] W. Knottenbelt, S. Zertal, and P. Harrison. Performance analysis of three implementation strategies for distributed lock management. *Computers and Digital Techniques, IEE Proceedings*, 148(45):176–187, 2001.
- [12] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Trans. Program. Lang. Syst.*, 10(4):579–601, 1988.
- [13] L. Lamport. A new solution of Dijkstra’s concurrent programming problem. *Comm. ACM*, 17(8), 1974.
- [14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
- [15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.
- [16] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. *SIGARCH Comp. Arch. News*, 19(2):269–278, 1991.
- [17] R. M. Metcalfe and D. R. Boggs. Ethernet: distributed packet switching for local computer networks. *Commun. ACM*, 19(7):395–404, 1976.
- [18] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB ’98)*, pages 62–73, 1998.
- [19] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe disks. In *OSDI ’06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 15–28, 2006.
- [20] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2<sup>nd</sup> USENIX Conference on File and Storage Technologies (FAST ’03)*, pages 73–88, 2003.
- [21] R. O. Weber. Information technology—SCSI Primary commands - 3 (SPC-2), revision 23. Technical report, INCITS Technical Committee T10/1416-D, May 2005.
- [22] R. O. Weber. Information technology—SCSI object-based storage device commands -2 (OSD-2), revision 3. Technical report, INCITS Technical Committee T10/1729-D, Jan. 2008.