Appears in the Proceedings of the International Conference On Distributed Computing Systems, Taipei, April 2000.

Highly concurrent shared storage

Khalil Amiri, Garth A. Gibson
Carnegie Mellon University, Pittsburgh, PA
{amiri+,garth+}@cs.cmu.edu

Richard Golding
Hewlett-Packard Laboratories, Palo Alto, CA
golding@hpl.hp.com

Abstract[1]

Switched system-area networks enable thousands of storage devices to be shared and directly accessed by end hosts, promising databases and filesystems highly scalable, reliable storage. In such systems, hosts perform access tasks (read and write) and management tasks (storage migration and reconstruction of data on failed devices.) Each task translates into multiple phases of low-level device I/Os, so that concurrent host tasks accessing shared devices can corrupt redundancy codes and cause hosts to read inconsistent data. Concurrency control protocols that scale to large system sizes are required in order to coordinate on-line storage management and access tasks. In this paper, we identify the tasks that storage controllers must perform, and propose an approach which allows these tasks to be composed from basic operations—called base storage transactions (BSTs)—such that correctness requires only the serializability of the BSTs and not of the parent tasks. We present highly scalable distributed protocols which exploit storage technology trends and BST properties to achieve serializability while coming within a few percent of ideal performance.

1. Introduction

Traditional I/O subsystems, such as RAID arrays, use a single centralized component to coordinate access to storage when the system includes multiple storage devices. A single *storage controller* receives an application's read and write requests and coordinates them so that applications see the appearance of a single shared disk. In addition to performing storage access on behalf of clients, the storage controller also performs other "management" tasks. Storage management tasks include migrating data to balance load or utilize new devices [18], adapting storage representation to access pattern [25], backup, and the reconstruction of data on failed devices.

1. This research is supported by DARPA/ITO through DARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Additional support was provided by the members companies of the Parallel Data Consortium, including: Hewlett-Packard Laboratories, Intel, Quantum, Seagate Technology, Wind River Systems, 3Com, Compaq, EMC, and Symbios Logic.

One of the major limitations of today's I/O subsystems is their limited scalability caused by shared controllers that data must pass through, typically from server to RAID controller, and from RAID controller to device. Emerging shared, network-attached storage arrays, like the one shown in Figure 1(a), enhance scalability by eliminating the shared controllers and enable direct host access to potentially thousands of storage devices [9, 18] over cost-effective switched networks [4, 15]. In these systems, each host acts as the storage controller on behalf of the applications running on it, achieving scalable storage access bandwidths [9].

Unfortunately, such shared storage arrays lack a central point to effect coordination. Because data is striped across several devices and often stored redundantly, a single logical I/O operation initiated by an application may involve sending requests to several devices. Unless proper concurrency control provisions are taken, these I/Os can become interleaved so that hosts see inconsistent data or corrupt the redundancy codes. These consistencies can occur even if the application processes running on the hosts are participating in an application-level concurrency control protocol, because storage systems can impose hidden relationships among the data they store, such as shared parity blocks.

For example, consider two hosts in a cluster as shown in the timeline of Figure 1(b). Each host is writing to a separate block, but the blocks happen to be in the same RAID stripe, thereby sharing the same parity block. Both hosts pre-read the same parity block and use it to compute the new parity. Later, both hosts write data to their independent blocks but overwrite the parity block such that it reflects only one host's update. The final state of the parity block is, therefore, not the cumulative XOR of the data blocks. A subsequent failure of a data disk, say device 2, will lead to reconstruction that does not reflect the last data value written to a device. In general, races can occur between concurrent host accesses, or between concurrent accesses and management operations such as migration or reconstruction.

Scalable storage access and management is crucial in today's storage marketplace [11]. In current storage systems, management operations are either done manually after taking the system off-line, use a centralized

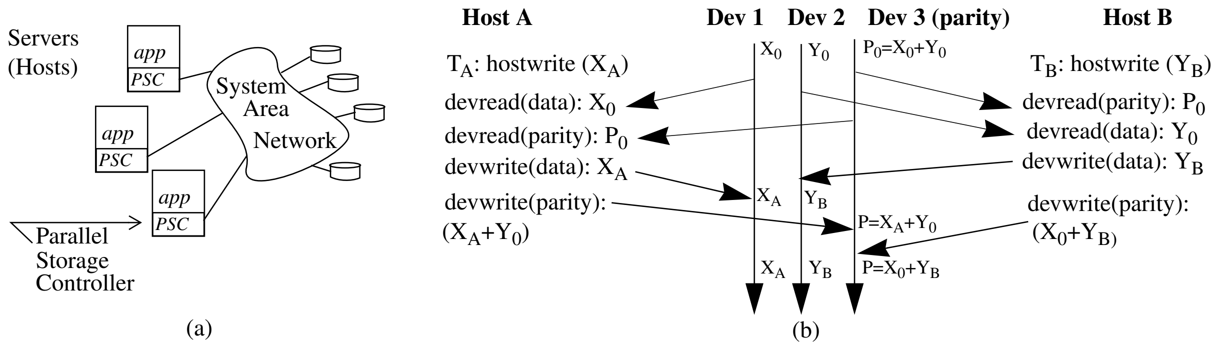

Figure 1: A shared storage system (a), and a timeline showing two concurrent small writes from two hosts (b). No concurrency control provisions are taken. Although host A is updating device 1 and host B is updating device 2, both must read, modify and update the same parity block on device 3. Both hosts read the same version of the parity block and the final writer leaves parity inconsistent.

implementation [25], or assume a simple redundancy scheme [18]. The paramount importance of storage system throughput and availability leads to the employment of ad-hoc management techniques, contributing to annual storage management costs that are 6-12 times the purchase cost of storage [11].

In this paper, we address the challenges of building a scalable distributed storage system that enables high concurrency between access and management tasks while ensuring correctness. In particular, we characterize the tasks that storage controllers perform and break these tasks down into sets of basic two-phased operations, which we call *base storage transactions* (BSTs). We claim that overall correctness requires ensuring only the *serializability* of the component BSTs and not of the parent tasks. We present distributed concurrency control protocols that exploit BST properties and technology trends toward more device functionality to provide serializability with good scalability. The protocols we present come within a few percent of the performance of an ideal zero-overhead protocol that would perform no concurrency control work and provide no correctness guarantees. We further argue that only a limited form of atomicity, and not "all or nothing" atomicity, is required from BSTs.

The rest of the paper is organized as follows: Section 2 describes in more detail the kind of tasks that are carried out at the storage layer (by the storage controllers). In Section 3, we show how these tasks can be composed out of a few BSTs. We further show that the serializability of the BSTs ensures correctness for the parent tasks. In Section 4, we present distributed concurrency control protocols specialized to BSTs. We compare their performance to centralized variants and to the zero-overhead protocol. We conclude the paper in Section 5.

2. Storage system description

Large collections of storage commonly employ redundancy that is transparent to applications, so that simple and common device failures can be tolerated without invoking expensive higher-level failure and disaster recovery mechanisms. For example, in RAID level 5, a parity-based redundancy code is computed across a group of data blocks and stored on a separate parity device. This allows the system to tolerate any *single self-identifying device failure* by recovering data from the failed device using the other data blocks in the group and the redundant code [23]. The block of parity that protects a set of data blocks is called a parity block. A set of data blocks and their corresponding parity block is called a parity stripe. Because it is one of the most complex of common storage redundancy schemes, we focus for the rest of this paper on RAID level 5 as our case study and evaluation architecture.

Figure 1(a) shows the kind of system that concerns us. A shared storage system is composed of multiple disks and hosts, connected by a scalable network fabric. The devices store uniquely named blocks and act independently of each other. Each host acts as a storage controller for its applications. The controller function can be implemented in software as an operating system device driver or could be delegated to a network interface card.

Hosts perform exactly four operations, divided into *access tasks* and *management tasks*. The access tasks are reads and writes (**hostread** and **hostwrite** operations). These tasks provide semantics essentially identical to reading and writing a disk drive or array. The management tasks are reconstruction and data migration (**reconstruct** and **migrate** operations respectively). Each high-level task is mapped onto one or more low-level I/O requests to (contiguous) physical blocks on a single device (**devread** and **devwrite**). Depending on the striping and redundancy policy, and whether a storage device has failed, a **hostread** or **hostwrite** may break down into different low-level **devreads** and **devwrites**, and some form of computation may be needed, such as computing parity. We refer to low-level device requests that are part of the same high-level task as *siblings*.

The **hostread** and **hostwrite** tasks are addressed to *virtual objects*, which may be files, or whole volumes.

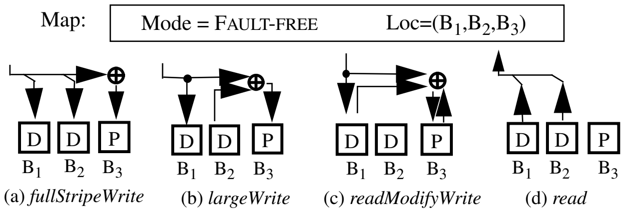

Figure 2: BSTs used in FAULT-FREE mode. Only one narrow stripe is shown for simplicity. A **hostwrite** invokes one of three BSTs (a-c), and a **hostread** invokes the *read* BST (d). An arrow directed towards a device represents a **devwrite**; an arrow away from a device represents a read; all reads precede all writes.

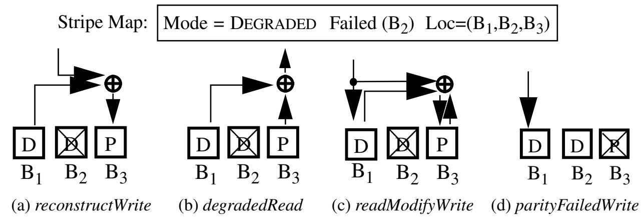

Figure 3: BSTs invoked in DEGRADED mode. The *reconstructWrite* BST (a) is invoked when the failed disk is being updated. In this case, the data blocks *not* being updated are read and XORed with the new data to compute and update the parity block. The *degradedRead* BST reconstructs the data on the failed block using all the other data and parity blocks in the stripe.

Blocks within a virtual object are mapped onto physical block ranges on one or more physical storage devices. The representation of a virtual object is described by a *stripe map* which specifies how the object is mapped, what redundancy scheme is used, and what BSTs to use to read and write the object. Stripe maps are cached by storage controllers to enable them to carry out the access tasks. Each host's controller performs access tasks on behalf of the applications running on it.

Management functions will occasionally change the contents of stripe maps—for example, during data migration or reconstruction. However, hosts cache copies of the maps, which must be kept coherent. There are several ways to maintain this coherence, such as using leases and invalidation callbacks on the cached data or other methods [10]; we do not consider this issue further in this paper.

3. Shared storage management

Concurrency is an essential property of shared storage. In large clustered systems, for example, many clients often use stored data at the same time. In addition, the reconstruction or copying of large virtual objects can take a long time. However, it is hard to ensure correctness when concurrent storage controllers share access to data on multiple devices Device failures, which can occur in the midst of concurrently executing tasks, complicate this further.

Transaction theory, which was originally developed for database management systems, handles this complexity by grouping primitive read and write operations into transactions that exhibit ACID properties (Atomicity, Consistency, Isolation and Durability) [13]. Databases, however, must correctly perform arbitrary transactions whose semantics are application-defined. Storage controllers, on the other hand, perform only four tasks, and the semantics of these tasks are well known. This a priori knowledge enables powerful specialization of transaction mechanisms for storage tasks [7].

In the following discussion, we will describe the consistency of storage controller tasks in terms of a durable serial execution, where tasks are executed one at a time and failures only occur while the system is idle (in the gaps between task executions). We then discuss how traditional atomicity and isolation properties can be interpreted for storage tasks to cope with concurrent execution and failure.

3.1. Base Storage Transactions

BSTs are transactions specialized to storage controller tasks. An access task, **hostread** or **hostwrite**, is executed using one BST. Which BST is chosen depends on the state of the system and exactly which blocks are being accessed. Storage management tasks, **reconstruct** and **migrate** are usually composed of a series of short-running BSTs. This reduces the impact on time-sensitive host access tasks and enables a variety of performance optimizations [14, 20].

Each virtual object is in one of four modes: FAULT-FREE (the usual state), DEGRADED (when one device has failed), RECONSTRUCTING (when recovering from a failure) or MIGRATING (when moving data). The first two modes are *access* modes, where only access tasks are performed, and the second two are *management* modes, where both management and access tasks are allowed. Different BSTs are used in different modes, partly to account for device failures and partly to exploit knowledge about concurrent management and access tasks. Table 1 shows the BSTs used to perform each allowed task in each of these modes.

The BSTs for FAULT-FREE and DEGRADED modes are straightforward, and are shown in Figure 2 and Figure 3 respectively. The other two modes are described below.

3.1.1. RECONSTRUCTING mode. This mode is used when recovering from a disk failure (Figure 4). The system declares a new block on a new disk to be the replacement block, then uses the **reconstruct** task to recover the contents of that block. This can occur in parallel with **hostread** and **hostwrite** tasks. All these tasks are aware of both the old and new mappings for the stripe, but the read BSTs use the "original array", ignoring the replacement block altogether. **Hostwrite** tasks use BSTs that behave as if the original array were in DEGRADED mode, but also update the replacement block whenever the failed block is written to.

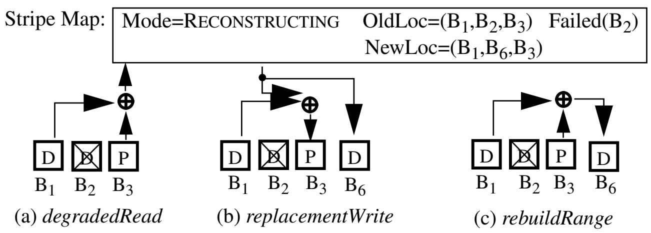

Figure 4: BSTs invoked in RECONSTRUCTING mode. Block B_6 is allocated as the replacement block. The *degradedRead* BST (a) is invoked by a **hostread** task if the failed device is being accessed. The *readModifyWrite* BST (shown in Figure 3) is invoked by a **hostwrite** task if the failed device is not being updated. The *replacementWrite* BST, which is a *reconstructWrite* BST that in addition updates the replacement block, is invoked if the failed device is being updated. The *rebuildRange* BST (c) is invoked by the **reconstruct** task to reconstruct the data on the failed device and write to the replacement one.

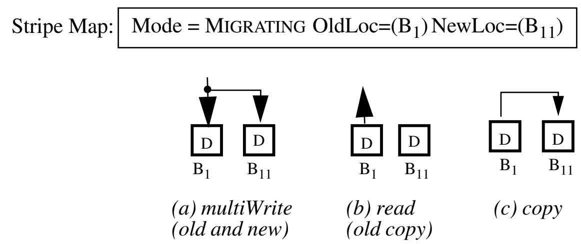

Figure 5: BSTs invoked in MIGRATING mode. The stripe map specifies the old and the new locations of a physical block. A **hostwrite** invokes the *multiWrite* BST (a) which writes to both the old and new locations; a **hostread** reads data from the old location (b). The **migrate** task invokes the *copy* BST (c) which copies data from the old location to the new location.

	MODE			
Task	**FAULT-FREE (Figure 2)**	**DEGRADED (Figure 3)**	**RECONSTRUCTING (Figure 4)**	**MIGRATING (Figure 5)**
hostread	*read*	failed block: *degradedRead* else: *read*	failed block: *degradedRead* else: *read* (old location)	*read* (old location)
hostwrite	small: *readModifyWrite* large: *largeWrite* full stripe: *fullStripeWrite*	failed block: *reconstructWrite* other failed: *readModifyWrite* parity failed: *parityFailedWrite*	failed block: *replacementWrite* other failed: *readModifyWrite* parity failed: *reconstructWrite* (new location)	*multiWrite* (and cloning)
reconstruct	—	—	*rebuildRange*	—
migrate	—	—	—	*copy*

Table 1 : BSTs used for different tasks in different modes. The BST chosen depends on the “size” of the write, as a fraction of the blocks in the stripe, and on whether the block being accessed has failed, some other data block has failed, or the parity block has failed.

The **reconstruct** task rebuilds the data on the replacement block using the *rebuildRange* BST, which reads the surviving data and parity blocks in a stripe, computes the contents of the failed data block and writes it to the replacement disk. When the **reconstruct** task is done, the replacement block will reflect the data from the failed block, parity will be consistent, and the stripe enters FAULT-FREE mode. Note that reconstruction concurrent with host accesses may result in unnecessary, but still correct, work.

3.1.2. MIGRATING mode. To simplify exposition, assume a non-redundant virtual object as shown in Figure 5. In this mode, the stripe map for the virtual object specifies the old and new physical locations. **Hostwrite** tasks update the old and new physical locations by invoking a *multiWrite* BST. Thus, at any point during the migration, the target physical blocks are either empty (not yet written to) or contain the same contents as their associated source physical blocks. **Hostread** tasks invoke the *read* BST, which reads the physical blocks from their old locations. The *read* BST does not access the target physical blocks because they may be still empty. The **migrate** task can be ongoing in parallel using the *copy* BST. This can be easily generalized to a redundant scheme by cloning the part of each write BST that updates the source to also update the target with the same value.

3.2. BST Properties

BSTs specialize general transaction ACID properties to the limited tasks of shared storage controllers.

3.2.1. BST Consistency. In the context of shared redundant storage, consistency means that redundant storage blocks contain data that correctly encodes the corresponding data block values. For RAID 5, this means that after each BST, the value of the parity block (P) is the XOR of all the values of the last writes to each of the corresponding data blocks (D). Each of the BSTs shown in Figures 2—5 has the property that, provided storage is consistent when they start and does not fail while they are executing, and provided that BSTs execute one at a time, then storage is consistent after the completion of the BST.

3.2.2. BST Durability. Because the primitive operations of BSTs transform stable storage (typically magnetic disks), durability of changes is not difficult. Storage regions written by a successful BST maintain the last written data and never switch back to older data values. Durability is preserved even after single device failures: RAID 5 reconstruction, combined with BSTs preserving data-parity consistency, ensures that the data values reconstructed on new storage are the same as the last written values.

3.2.3. BST Atomicity. This is the property that a transaction, once started, either completes entirely or terminates without changing any storage values. In the absence of knowledge of the function of a transaction, database systems must provide full atomicity by logging values to be changed before making any changes. It preserves the logs until all changes are made, and re-applies the log of committed changes to recover from a failure [13].

However, for storage tasks we have full knowledge both of the specific semantics needed by storage tasks and the structure of the BSTs that will implement them. Specifically, all BSTs can be represented as directed acyclic graphs (Figure 7), in which it is possible to ensure that no device write begins until after all device reads are complete [7]. This point, immediately before the initiation of any writes, is the *commit point*. The notion of the commit point allows us to relax our previous assumption that failures occur only between BST executions. Note that we still assume a single device failure, and the occurrence of more than one failure, or other untimely host failures, is catastrophic and can result in data loss.

Before reaching the commit point, any BST encountering a device failure (during a read) simply terminates, and its parent task reissues a new BST for DEGRADED mode. After the commit point, a BST encountering a single device failure (during a write) simply completes. This is correct because an observer cannot distinguish between a single failure occurring after the commit point and the failure of that device immediately after the BST completes.

If a host fails after the commit point but before data has been sent to all devices, atomicity will be violated because some blocks have been written, and the values that should have been written to the others have been lost with the loss of the host's memory. All existing (non-database) storage systems have this problem. Our system, in this case, protects parity consistency by detecting the host failure and initiating a *rebuildRange* BST to update the parity to correspond to this possibly corrupted data. (Note that to detect a failure and take these actions, storage system code not on the failed host must know which BST was active. This is accomplished by piggybacking notification on the concurrency control protocol of Section 4.)

3.2.4. BST Isolation. We can now relax our assumption that BSTs execute serially. In showing that consistency, durability, and atomicity are correctly maintained when executing BSTs, we only required isolation between the effects of one BST's execution and those of others. *Serializable* execution of BSTs both allows concurrency and provides this level of isolation by ensuring that the results of the concurrent execution are identical to the results of some serial execution sequence [22].

The following sections analyze four serializability algorithms for shared storage systems.

4. Concurrency control algorithms for BSTs

A protocol that provides serializability is commonly called a concurrency control protocol. We discuss two commonly used centralized concurrency control protocols, *server locking* and *callback locking*, and present our new device-supported distributed protocols, *device-served locking* and *timestamp ordering*. We show how the distributed protocols benefit from specialization to BST properties and storage technology trends. We evaluate performance relative to the ideal performance of a zero-overhead protocol, which performs no concurrency control.

4.1. Evaluation environment

We implemented our protocols in fully detailed simulation, using the Pantheon simulator system [25]. We simulate a cluster system consisting of hosts and disks connected by a network. Table 2 shows the baseline parameters of the experiments. Although the protocols were simulated in detail, the service times for hosts, controllers, links and storage devices were derived from simple distributions based on observed behavior of Pentium-class servers communicating with 1997 SCSI disks [24] over a fast switched local area network (like FibreChannel). Host clocks were

Baseline simulation parameters	
System size	20 devices, 16 hosts, RAID level 5, stripe width = 4 data + parity, 1000 blocks per device.
Host workload	random think time (normally distributed with mean 80 ms, var. 10ms), 70% reads, access size uniformly random between 1-4 blocks, target address random.
Service times	Disk: 8ms positioning, 16MB/sec transfer rate. Network: 1-2ms overhead per message, 10 MBytes/sec switched Ethernet. 750 µsec mean host/lock server message processing time.

Table 2 : Baseline simulation parameters. Host data is striped across the parity group. We assume modern disk drives and a switched network of 10MB links. Runs are repeated five times. Each run lasts 500 seconds to keep variance small.

allowed to drift within a practical few milliseconds of real-time [19]. We compared the performance of the protocols under a variety of synthetically generated workloads and environmental conditions. The baseline workload represents the kind of sharing that is characteristic of OLTP workloads and cluster applications (databases and file servers), where load is dynamically balanced across the hosts or servers in the cluster resulting in limited locality and mostly random accesses. This baseline system applies a moderate to high load on its storage devices, yielding about 50% sustained utilization. We report the performance of the protocols in FAULT-FREE mode since it is representative of their general performance. Moreover, the relative performance of the protocols under DEGRADED mode was found to be similar to that under FAULT-FREE mode.

4.2. Centralized locking protocols

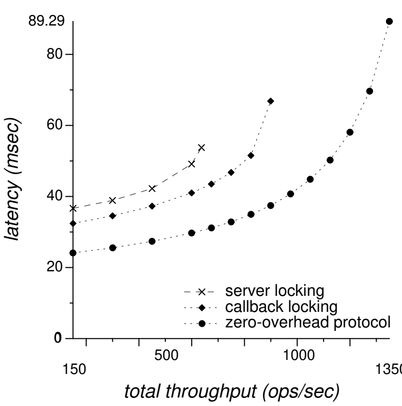

Figure 6: Scaling of server and callback locking. The baseline workload (16 hosts) corresponds to the fourth point from the left in the graph.

Locking is the most commonly employed mechanism for implementing concurrency control. Although it provides a task with exclusive access to a data item, execution may not be serializable and deadlocks can occur when a task needs to lock multiple blocks and does so carelessly. Further, locking can severely limit concurrency. In practice, serializability of tasks that obtain multiple locks is achieved with *two-phase locking*, a programming discipline in which no lock can be released before the last lock has been obtained [12]. The effect on concurrency can be limited by locking only the minimum necessary items, and holding the locks for as short a time as possible. Deadlock is handled either by avoidance, such as a discipline of acquiring locks in a strict published order in all tasks, or by a detection and recovery mechanism, which detects (likely) deadlock and aborts some lock-holding tasks. We discuss two locking algorithms: simple server locking and its caching variant, callback locking. The protocols have the two-phase property and hence ensure serializability. They are also free from deadlocks.

4.2.1. Server locking. Under this scheme, a centralized lock server provides locking on low-level storage block ranges. A BST executing at a host acquires an exclusive (for a **devwrite**) or a shared (for a **devread**) lock on a set of target ranges by sending a single *lock message* to the lock server. The lock server queues a host's request if there is an existing lock on any part of the requested range. Once conflicting locks have been released, the server grants the request. The host may then issue the low-level I/O requests to the devices (**devreads** or **devwrites**). When all I/O requests in the BST complete, the host sends an *unlock message* to the lock server. As there is only one lock and one unlock message per BST, the protocol is trivially two-phase and therefore serializable. Because all locks are acquired in a single request, lock acquisition latency is minimized and deadlocks are avoided. However, server locking introduces a potential bottleneck at the server and delays issuing low-level I/O requests for at least one round trip of messaging to the lock server.

4.2.2. Callback locking. This is a popular variant of server locking that delays the unlock message, effectively caching the lock at the host, in the hope that the host will generate another access to the same block(s) in the near future and be able to avoid sending subsequent lock messages to the server [16, 17, 6]. If a host requests a lock from the lock server that is currently cached by another host, the server asks the host holding the cached lock (this is the *callback message*) to relinquish it before granting the lock to the newly requesting host. A common optimization to callback locking is to have locks automatically expire after a "lease" period so that callback messages are sent only to reclaim recently cached locks. Our implementation uses a lease period of four seconds.

4.2.3. Performance. Figure 6 highlights the scalability limitation of centralized locking protocols. It plots the average host-end task latency against total offered throughput (by varying the number of hosts) using the simulation parameters of Table 2. The plots of Figure 6 show that the locking protocols deliver only 30% or 50% of the full throughput of the I/O system, as defined by the zero-overhead protocol (which performs no concurrency control work and provides no consistency guarantees). The protocols bottleneck at a fraction of the attainable throughput because the lock server's CPU saturates handling lock and unlock requests (750 µsec per message sent or received.) While there are several ways to increase lock server throughput, by streamlining the server's network processing and request handling for example, they do not eliminate the bottleneck.

Callback locking can reduce lock server load and lock acquisition latencies when locks are commonly reused by the same host multiple times before a lease expires. The false sharing induced by shared parity blocks can reduce this benefit somewhat. Figure 6 shows that at the baseline workload, callback locking reduces latency relative to simple locking by 20% but is still 33% larger than the zero-overhead protocol. This benefit is not from locality—the workload contains little of it—but from the dominance of read traffic, which allows concurrent cached read locks at all hosts until the next write. In the worst case, however, each lock is used once by a host, and then is called back by a conflicting use at another host. This will induce the same number of messages as simple server locking, but lock acquisition latency can be much worse: for example, a write lock request that conflicts with a read lock shared by many hosts must wait for all of those hosts to respond to their callbacks. Callback locking also is more sensitive to contention than server locking. In other experiments, we found that callback locking gives worse latency than server locking at high contention due to its longer lock hold times.

4.3. Parallel lock servers

The scalability bottleneck of centralized locking protocols can be avoided by distributing the locking work across multiple parallel lock servers. Locks can be partitioned across multiple lock servers according to some static or dynamic scheme, and hosts send lock requests, consistent with two-phase locking rules, directly to the appropriate servers.

However, multiple lock servers lack a key benefit of centralized locking: simple deadlock avoidance. Using parallel lock servers, deadlocks can be avoided by acquiring locks one at a time, but this increases the locking latency and lock holding time substantially. This in turn increases the probability of lock conflicts.

Instead, deadlocks can be detected via request time-outs. If a lock request cannot be granted at a lock server within a given time, that server presumes deadlock and denies the request. The host BST then releases any acquired locks, and retries from the beginning. We present a specialized implementation of this scheme in the following section.

4.4. Device-served locking

Given the opportunity of increasing storage device intelligence [9, 18], we investigated embedding lock servers in the storage devices [21]. Device-served locking reduces the cost of a scalable serializable storage array by eliminating the need for dedicated lock server hardware, and decreases latency and the total number of messages by exploiting the two-phase nature of BSTs to piggy-back lock messaging on I/O requests.

In device-served locks, each device serves locks for the blocks stored on it. This enhances scalability by balancing lock load over all the devices. While widely-parallel locking like this can introduce many lock and unlock messages, device-served locking mitigates this somewhat by piggy-backing these messages on I/O requests as shown in Figure 7.

For single-phase reads, lock acquisition can be piggy-backed on the reads, reducing pre-I/O latency, but a

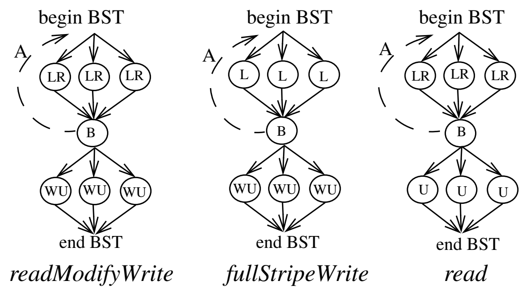

Figure 7: The implementation of BSTs with device-served locking and the piggy-backing optimization. A node represents a message exchange with a device. An "L" node stands for a **lock** operation; a "U" node stands for an **unlock** operation. "LR" represents the **lock-and-devread** operation, while "WU" represents the **devwrite-and-unlock**. The edges represent control dependencies. A "B" node represents a commit point at the host, where the host blocks until all preceding operations complete, restarting from the beginning if any of them fail. Lock operations can fail if the device times out before it can grant the lock ("A").

separate unlock phase is required. Fortunately, the second phase latency can be hidden from the application since the data has been received. For two-phase writes, locks can be acquired during the first I/O phase (by piggy-backing the lock requests on the **devread** requests) and released during the second I/O phase (by piggy-backing the unlock messages onto the **devwrite** requests) totally hiding the latency and messaging cost of locking. We require that a host not issue any **devwrites** until *all* locks have been acquired in order to preserve atomicity, although it may issue **devreads**. Restarting a BST in the lock acquisition phase, therefore, does not require undoing writes (since no data has been written yet).

The efficiency of device-supported parallel locking eliminates the need for leased callback locks. Two-phase writes have no latency overhead associated with locking, and the overhead of unlocking for single phase reads is not observable. Only single phase writes would benefit from lock caching, and we feel that the simplicity of simple device-served locks outweighs the possible performance improvement. Our experiments show that device-served locking is more effective than the centralized locking schemes. With the baseline workload, it achieves latencies only 10% larger than minimal and a peak throughput equal to 94% of maximum.

Despite its scalability, device-served locking is vulnerable to poor performance under high contention because of its susceptibility to deadlocks and the difficulty of properly tuning the critical lock request timeout. Under low to moderate contention, which we induced by controlling the fraction of storage addressed by all hosts, device-served locking did better than centralized locking, but when contention was high (hosts accessing only 2% of the active disk space), the system became unstable. This was because many BSTs restarted, either because of actual deadlocks or a too-short timeout setting, thus causing device overload.

4.5. Timestamp ordering

Developed for highly concurrent databases, timestamp ordering protocols are an attractive mechanism for distributed concurrency control over storage devices since they place no overhead on reads and are not susceptible to deadlocks. These protocols select an a priori order of transaction execution using some form of timestamps and then enforce that order [5]. Most database implementations verify timestamp order each time a transaction executes an access to the database, but more optimistic variants delay all checks until commit time [1].

In the simplest timestamp ordering approach, each transaction is tagged with a unique timestamp at the time it starts. In order to verify that reads and writes are proceeding in timestamp order, the blocks are tagged with a pair of timestamps, *rts* and *wts*, which correspond to the largest timestamp of a transaction that read or wrote the block, respectively. A read by transaction *T* with timestamp *opts(T)* to block *v* is accepted if *opts(T)>wts(v)*, otherwise it is immediately rejected. A write is accepted if *opts(T)>wts(v)* and *opts(T)>rts(v)*. If an access is rejected, its parent transaction is aborted and restarted with a new larger timestamp.

In order to avoid cascading aborts, in which the abort of one transaction causes a rippling abort of many others, reads are not allowed to access data written by active (uncommitted) transactions. When an active transaction wants to update a block, it first submits a **prewrite** to storage declaring its intention to write but without actually updating the data. Storage accepts a **prewrite** only if *opts(T)>wts(v)* and *opts(T)>wts(v)*. When the active transaction *T* commits, a write is issued for each submitted **prewrite**. Only then is the new value updated and made visible to other readers. A transaction that issued a **prewrite** may abort, in which case its **prewrites** are discarded and appropriate blocked requests are unblocked. Readers are blocked behind any active **prewrite** request at a device until the write commits or aborts.

4.5.1. Timestamp ordering for BSTs. As described above, timestamp ordering works by having hosts independently determine a total serial order to which the effect of concurrent execution should be equivalent. BST timestamp ordering depends on devices capable of maintaining tags and queues as described above, and on BSTs providing information about that order (in the form of a timestamp) in each I/O request. Since I/O requests are tagged with an explicit order according to which they have to be processed (if at all) at each device, deadlocks cannot occur and all allowed schedules are serializable. Instead, out-of-order requests will be rejected, causing their parent transaction (BST) to be aborted and retried with a larger timestamp.

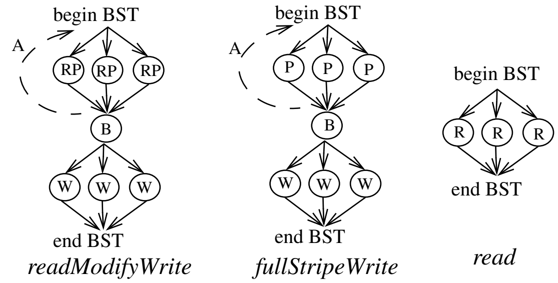

Figure 8: The composition of host operations in the optimized timestamp ordering protocol. **Devread**, **devwrite**, and **prewrite** requests are denoted by "R", "W" and "P" nodes respectively. "RP" denotes a **read-and-prewrite** request. A "B" node represents the commit point at the host, where the host blocks until all preceding operations complete, restarting from the beginning if any of them fail. In a two-phase write, the **prewrite** requests are piggy-backed on the reads. Hence, both reads and two-phase writes use the minimal amount of messaging.

As in the general case, since each device is performing a local check, a write request may pass the check in some devices, but the BST may abort due to failed checks in other devices. Because of the simple structure of BSTs, splitting the write protocol into a prewrite phase followed by a write phase ensures that the host has all device decisions before issuing any write, allowing it to reach a commit point without changing the contents of any storage. New timestamps are generated at a host by sampling a local clock which is loosely synchronized with the rest of the cluster, then appending the host's unique identifier to the least significant bits of the clock value.

As an example, consider the *readModifyWrite* BST since it employs the piggy-backing optimization and is of reasonable complexity, as shown in Figure 8. This protocol reads data and parity in a first phase, uses this data together with the "new data" to compute the new parity, then updates both data and parity. The BST execution starts with the host locally generating a new timestamp, *opts*, then sends low-level **devread** requests to the data and parity devices, tagging each request with *opts,* and bundling each request with a **prewrite** request.

The device receiving a **read-and-prewrite** request performs the necessary timestamp checks both for a read and a **prewrite**, accepting the request only if both checks succeed; that is, *opts>rts* and *opts>wts* for each affected block. An accepted request is queued if there is an outstanding **prewrite** with a lower timestamp, otherwise data is returned to the host and *rts* is updated if *opts>rts*. When the host has received all requested data, it computes the new parity and sends the new data and parity in **devwrites** also tagged with *opts*. The devices are guaranteed by the acceptance of the **prewrite** to do the write, update *wts*, and discard the corresponding **prewrite** request from the queue. The request queue is then inspected to see if any **read** or **read-and-prewrite** requests can now be completed.

Under normal circumstances, the *readModifyWrite* BST does not induce any overhead, just like piggy-backed device-based locking, because reads arriving while other writing BSTs are in progress is rare. We discuss optimizations to the basic timestamp ordering protocol next, which lend themselves to efficient implementation. These optimizations were implemented in our simulation, and the results reflect their effects.

4.5.2. Avoiding timestamp accesses. Our protocol requires that the pair of timestamps, *rts* and *wts*, associated with each disk block be durable: read before any disk operation and written after every disk operation. A naive implementation might store these timestamps on disk, near the associated data. However, this would result in one extra disk access after reading a block (to update the block's *rts),* and one extra disk access before writing a block (to read the block's previous *wts).*

Doubling the number of disk accesses is not consistent with our high-performance goal. Because all clocks are loosely synchronized and message delivery latency should be bounded, a device need not accept a request timestamped with a value much smaller than its current time. Such a transaction would have timed out, aborted, and restarted with a later timestamp. Hence, per-block timestamp information older than *T* seconds, for some value to *T*, can be discarded and a value of "current time minus *T*" used instead. Moreover, if a device is reinitiating after a "crash" or power cycle, it can simply wait time *T* after its clock is synchronized before accepting requests, or record its initial synchronized time and reject all requests with earlier timestamps. Therefore, timestamps only need volatile storage, and only enough to record a few seconds of activity. The use of loosely synchronized clocks and efficient timestamp management for concurrency control has been demonstrated in the Thor client-server object-oriented database management system [1].

4.5.3. Performance

As shown in Figure 9, timestamp ordering is highly scalable: the average task latency for timestamp ordering and device-served locking is only 10% higher than that of the zero-overhead protocol. In addition, it uses the smallest amount of messaging compared to all other protocols. It has no messaging overhead on reads, and with the piggy-backing optimization, it can also eliminate the messaging overhead associated with two-phase write BSTs, resulting in, at worst, the same number of messages as the best locking protocol.

4.6. Blocking/retry behavior

In any of the protocols, when several BSTs attempt to access a conflicting range, some of them will be delayed either directly on a lock, or indirectly by suffering an abort and a retry.

The probability of delay depends in part on the level of contention. Shown in Figure 10, the fraction of BSTs delayed is highest for callback locking because it has the largest window of vulnerability to conflict (lock hold time). Both the distributed device-based protocols do better than centralized locking approaches because they exploit piggy-backing of lock/ordering requests on the I/O requests, thereby avoiding the latency of communicating with the lock server before starting the I/O and shortening the window of vulnerability to conflict.

This delay also depends on environmental factors, such as network reordering of messages. Reordering can cause deadlocks and restarts for device-served locks, and rejections and retries for timestamp ordering because

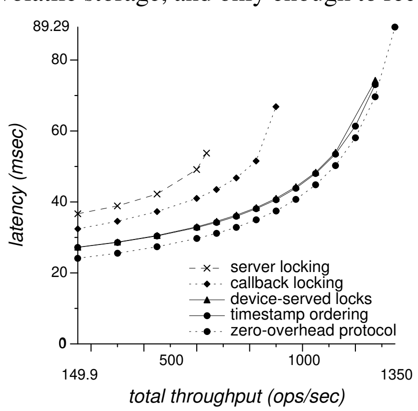

Figure 9: Scalability of timestamp ordering compared to the locking and the zero-overhead protocols.

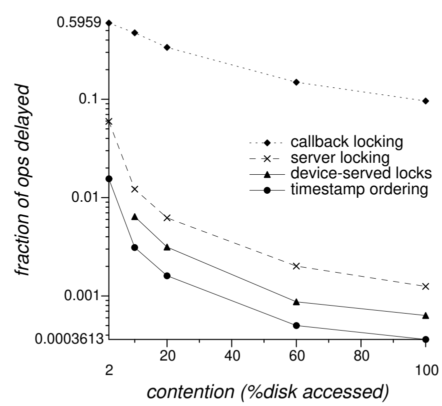

Figure 10: The performance of the protocols under high contention. When the hosts are restricted to only 2% of the active portion of the disk, device-served locking suffers from disk queues and latency growing without bound due to timeout-induced restarts.

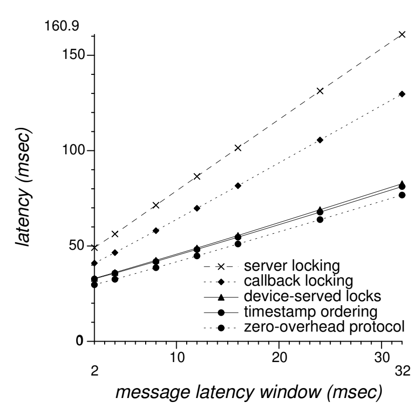

Figure 11: Effect of the variability in network message latencies on end latency.

sibling requests are serviced in a different order at different devices.

To explore the effect of reordering on the delay and latency behavior of the protocols, we conducted an experiment where we changed the variability of network message latency and measured its effect on delay and latency. Message latency was modeled as a uniformly distributed random variable over a given window size, extending from 1 to w milliseconds. A larger window size implies highly variable message latencies and leads to a higher probability of out-of-order message arrival. It also increases the mean message latency. Figure 11 show the increase in average latency as the message variability is increased. Although timestamp ordering and device-based locking are sensitive to message reordering, the end effect on host latency is less noticeable; indeed, the gap between the zero-overhead protocol and these two protocols remains approximately constant, indicating little effect from reordering.

5. Conclusions

Shared storage arrays enable thousands of storage devices to be shared and directly accessed by end hosts over switched system-area networks. In such systems, concurrent host tasks can lead to inconsistencies in redundancy codes and for data read by end hosts. In this paper, we propose a novel storage-specialized transactional approach that enables high concurrency between access and management tasks in a distributed storage system. Our approach breaks down the storage access and management tasks performed by storage controllers into simple two-phased transactions (BSTs) in which data logging is not needed because no disk value is changed before the commit point. We present two distributed concurrency control protocols—device-served locks and timestamp ordering—that exploit intelligence in storage devices to provide serializability for BSTs with high scalability. These protocols use message batching and piggy-backing to reduce BST latencies relative to centralized lock server protocols. In particular, both device-served locking and timestamp ordering achieve 40% higher throughput than server and callback locking for a small (20 device) system. Both distributed protocols exhibit superior scaling, falling short of the ideal protocol's throughput by only 5-10%. At very high contention, timestamp ordering is more robust than device-served locking because it does not depend on a timeout mechanism for deadlock detection.

Acknowledgments. We would like to thank John Wilkes, Elizabeth Borowsky, and Dave Anderson for their valuable feedback. We are grateful to Paul Mazaitis for maintaining our cluster during simulations. We are also grateful to Fay Chang, Joan Digney, the members of the PDL at CMU, the members of the SSP at HP labs, and our anonymous reviewers for improving the clarity of the presentation.

References

[1] A. Adya, R. Gruber, B. Liskov and U. Maheshwari, "Efficient optimistic concurrency control using loosely synchronized clocks," *Proc. SIGMOD*, May 1995.

[2] K. Amiri, G. Gibson and R. Golding, "Scalable concurrency control and recovery for shared storage arrays," Technical Report CMU-CS-99-111, Carnegie Mellon University, Feb. 1999.

[3] K. Amiri, "Scalable dynamic function placement in distributed storage systems," in preparation, Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University.

[4] A. Benner, "Fibre Channel: Gigabit Communications and I/O for Computer Networks," McGraw Hill, New York, 1996.

[5] P. Bernstein and N. Goodman, "Timestamp based algorithms for concurrency control in distributed database systems," *Proc. 6th VLDB*, Oct. 1980.

[6] M. Carey, M. Franklin, M. Zaharioudakis, "Fine-grained sharing in a page server OODBMS," *Proc. 1994 SIGMOD*, May 1994.

[7] W. Courtright, "A transactional approach to redundant disk array implementation," Ph.D. dissertation, issued as Technical Report CMU-CS-97-141, Carnegie-Mellon Univ., Apr. 1997.

[8] K. Eswaran, J. Gray, R. Lorie and L. Traiger, "The notions of consistency and predicate locks in a database systems," *Comm. of the ACM* 19(11), 11, Nov. 1976.

[9] G. Gibson et al., "Cost-effective high-bandwidth storage architecture," *Proc. ACM ASPLOS*, Oct. 1998.

[10] R. Golding and E. Borowsky, "Fault-tolerant replication management in large-scale distributed storage systems," *Proc. 18th IEEE Symp. on Reliable Distributed Systems*, Oct. 1999.

[11] R. Golding, E. Shriver, T. Sullivan, J. Wilkes, "Attribute-managed storage," *Wkshp. on modeling and specification of I/O*, 1995.

[12] J. Gray, R. Lorie, G. Putzulo, and I. Traiger, "Granularity of locks and degrees of consistency in a shared database," IBM Research Report RJ1654, Sep. 1975.

[13] T. Haerder and A. Reuter, "Principles of Transaction-oriented Database Recovery," *ACM Computing Surveys* 15(4), Dec. 1983.

[14] M. Holland, G. Gibson, D. Siewiorek, "Fast on-line failure recovery in redundant disk arrays," *Proc. 23rd FTCS*, 1993.

[15] R. Horst, "TNet: A Reliable System Area Network," *IEEE Micro,* Feb. 1995.

[16] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System," *ACM TOCS* 6(1), Feb. 1988.

[17] C. Lamb, G. Landis, J. Orenstein, D. Weinreb, "The Object-Store Database System," *Comm. of the ACM*, 34(10), Oct. 1991.

[18] E. Lee and C. Thekkath, "Petal: Distributed Virtual Disks," *Proc. 7th ASPLOS,* Oct. 1996.

[19] D. L. Mills, "Network time protocol: specification and implementation," DARPA-internet RFC 1059, DARPA, July 1988.

[20] R. Muntz, J. Lui, "Performance analysis of disk arrays under failure," *Proc. 16th VLDB*, 1990.

[21] M. O'Keefe, "Shared file systems and fibre channel," *Proc. 6th Goddard Conf. on Mass Storage Sys. and Technologies*, 1998.

[22] C. Papadimitriou, "Serializability of concurrent updates," *J. ACM,* 26(4), Oct. 1979.

[23] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks," *Proc. ACM SIGMOD*, June 1988.

[24] Seagate Technology, "Cheetah: Industry-Leading Performance," *http://seagate.com*, 1997.

[25] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," *ACM Trans. on Computer Systems*, vol. 14, no. 1, Feb. 1996.