# Scalable Concurrency Control and Recovery for Shared Storage Arrays

Khalil Amiri[1], Garth Gibson[2], Richard Golding[3]

February 1999
CMU-CS-99-111

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

1. Department of Electrical and Computer Engineering, can be reached via email at *khalil.amiri@cs.cmu.edu*
2. School of Computer Science, can be reached via email at *garth@cs.cmu.edu*
3. Hewlett-Packard Laboratories, can be reached via email at *golding@hpl.hp.com*

This research is sponsored by DARPA/ITO through ARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Additional support was provided by NSF under grants EEC-94-02384 and IRI-9625428 and NSF, ARPA and NASA under NSF Cooperative Agreement IRI-9411299. We are indebted to generous contributions from the member companies of the Parallel Data Consortium. At the time of this writing, these companies include Hewlett-Packard Laboratories, LSI Logic, Intel, Quantum, Seagate Technology, Wind River Systems, Hitachi, IBM, Novell Inc., Siemens Microelectronics, Clariion Array Development, and 3Com Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

**ACM Computing Reviews Keywords:** B.4.2 Input/output devices, D.4.1. Concurrency, D.4.2. Storage management, D.4.5. Reliability, D.4.7. Distributed Systems.

# Abstract

*Shared storage arrays enable thousands of storage devices to be shared and directly accessed by end hosts over switched system-area networks, promising databases and filesystems highly scalable, reliable storage. In such systems, however, concurrent host I/Os can span multiple shared devices and access overlapping ranges potentially leading to inconsistencies for redundancy codes and for data read by end hosts. In order to enable existing applications to run unmodified and simplify the development of future ones, we desire a shared storage array to provide the illusion of a single controller without the scalability bottleneck and single point of failure of an actual single controller. In this paper, we show how rapidly increasing storage device intelligence coupled with storage's special characteristics can be successfully exploited to arrive at a high performance solution to this storage management problem. In particular, we examine four concurrency control schemes and specialize them to shared storage arrays; two centralized ones: simple server locking, and server locking with leased callbacks; and two distributed ones based on device participation: distributed locking using storage-device-embedded lock servers and timestamp ordering using loosely synchronized clocks. Simulation results show that both centralized locking schemes suffer from scalability limitations. Moreover, callback locking is particularly suspect if applications do not have much inherent locality and if the storage system introduces false sharing. Distributed concurrency control with device support is attractive as it scales control capacity with storage and performance capacity and offers the opportunity to piggyback lock/ordering messages on operation requests, eliminating message latency costs. Simulations show that both storage-optimized device-based protocols exhibit close to ideal scaling achieving 90-95% of the throughput possible under totally unprotected operation. Furthermore, timestamp ordering uses less network resources, is free from deadlocks and has performance advantages under high load. We show how timestamp ordering can be extended with careful operation history recording to ensure efficient failure recovery without inducing I/Os under normal operation. This brings the overhead of concurrency control and recovery to a negligible few percent thereby realizing the scalability potential of the shared array I/O architecture.*

# 1. Motivation

Traditional I/O subsystems use a centralized component to coordinate access to storage when the system includes multiple storage devices (Figure 1(a)). The data may be striped across the devices or stored redundantly such that a single logical I/O operation initiated by an application may involve sending requests to multiple devices. A single component controlling storage (the *storage controller*) receives the application's read and write requests and coordinates them so that applications see the appearance of a single shared disk. Networked clients send requests to the storage controller (a shared file server, storage server or RAID controller) which accesses storage devices on their behalf. Even in redundant configurations where tandem storage controllers are used [Bart81, Bhide91], one controller explicitly or implicitly relinquishes control of storage devices to another host as part of a fail-over or load rebalancing protocol. Similarly, for parallel RAID architectures where intelligence is distributed to the storage devices [Cao93], a single host still controls all the storage devices and serializes all the accesses.

One of the major limitations of today's I/O subsystems is their limited scalability due to the number of shared controllers that data must pass through, typically from client to server, from server to RAID controller, and from RAID controller to device. Emerging shared, network-attached storage arrays (Figure 1(b)) are enhancing scalability by eliminating the shared controllers and enabling direct host access to potentially thousands of storage devices [Gibson98, Lee96, Mass98] over cost-effective switched networks [Boden95, Horst95, Benn96]. In these systems, each host acts as the storage controller on behalf of the applications running on it. Such systems are desirable because they can potentially achieve greater scalability and availability, and can allow large collections of storage to be managed as a single system [Gold95].

Unfortunately, shared storage arrays shown in Figure 1(b) lack a central point to effect coordination. Hosts accessing shared storage can see inconsistent data or corrupt it unless proper concurrency control provisions are taken. This concurrency control can be done in the application, for example, a distributed database, but only for the conflicts that the application is aware of. Lower level functionality added by storage such as parity maintenance for fault-tolerance is transparent to the application and defeats application-level concurrency control protocols. For example, consider two hosts in a cluster, each one is writing to a separate block, but the blocks happen to be in the same RAID stripe, thereby updating the same parity block. Races can occur which would cause the parity protecting both files to be corrupted,

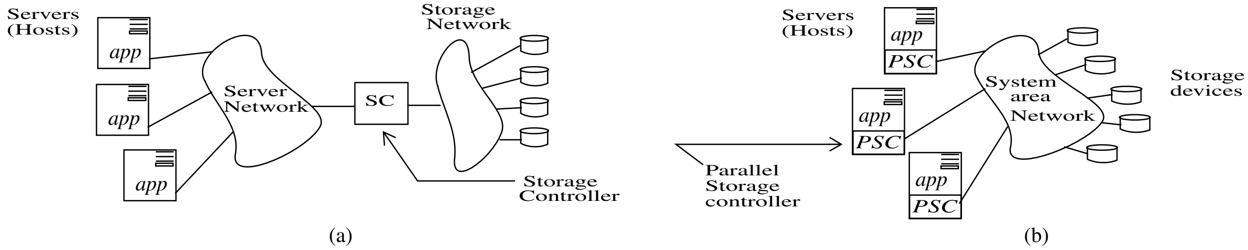



**Figure 1**. Illustrations of a conventional storage system (a), where all host requests are serialized at a single storage controller, and a shared storage array (b) where network-attached storage devices are shared and directly accessed by hosts and where serializability must be ensured by a distributed protocol, implicilty transforming low-level host software into a parallel storage controller.

forsaking the disk fault-tolerance property. Furthermore, several applications can be simplified if the storage system provided serialization of storage level reads and write requests to data partitioned across several storage devices (so that a reader can not see partial effects of an ongoing write, i.e. see the data either before or after a concurrent write by another host). To preserve the consistency of storage layer redundancy codes (RAID parity, mirrors), to allow existing applications to run unmodified and to simplify the development of future ones, we desire the shared storage system to provide the illusion of a single controller.

In this paper, we identify storage's special characteristics and specialize traditional approaches to take advantage of these characteristics. In particular, we examine four concurrency control schemes and specialize them to shared storage arrays; two centralized ones: simple server locking, and server locking with leased callbacks; and two distributed ones based on device participation: distributed locking using storage-device-embedded lock servers and timestamp ordering using loosely synchronized clocks. We evaluate the protocols under various workload and networks conditions and discuss their implications on recovery.

The rest of the paper is organized as follows. Section 2 describes shared storage arrays and the composition of their operations, and formalizes the guarantees that they should provide. Section 3 briefly reviews the traditional approaches to ensuring serializability for database transactions, as we later apply them to shared storage arrays. Section 4 presents and evaluates our specialized protocols. Section 5 discusses the extension of the protocols to provide the proper recovery guarantees. We overview ongoing work in Section 6 and conclude the paper in Section 7.

## 2. Shared storage arrays

Figure 1(b) 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, possibly organized into partitions or objects, and act independently of each other. Hosts perform high-level read and write *operations* (*hostread* and *hostwrite)* to logical ranges that map onto one or more low-level *I/O requests* (*devread* and *devwrite).* A low-level I/O request is a read or write to (contiguous) physical blocks on a single device. Depending on the striping and redundancy architecture, and whether a storage device has failed, *hostread* and *hostwrite* operations may break down into different low-level *devreads* and *devwrites,* and some form of computation may be needed[1]. We assume that each host performs storage access (parity computation and access coordination) on behalf of the applications running on it. We refer to low-level device requests that are part of the same high-level operation as *siblings*.

Large collections of storage commonly employ redundant codes transparently with respect to applications so that simple and common device failures can be tolerated without invoking expensive higher level failure and disaster recovery mechanisms. Maintaining this redundancy is a primary storage management function. For example, in RAID level 5, a redundancy code is computed across a group of data blocks and stored on a separate parity device as illustrated in Figure 2. 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 [Patt88]. The block of parity that protects a set of data units is called a parity unit. A set of data units and their corresponding parity unit is called a parity stripe. Whether the array is in fault-free or in degraded mode, all of the RAID level 5 operations are either single-phase or two-phase (a read phase followed by a write phase) as shown in Figure 3(a). This distinguishing characteristic of storage clusters in fact extends to all the RAID architectures (including mirroring, single failure tolerating parity and double failure tolerating parity [Gibson92, Blaum94] as well as parity declustering [Holl94] which is particularly appropriate for large storage systems). All fault-free and degraded mode high-level operations are composed of either single-phase or two-phase collections of low-level requests. A single phase *hostread* (*hostwrite*) breaks down into parallel *devread (devwrite)* requests,

1. We assume that low-level storage management code in each host performs storage access (parity computation and access coordination) on behalf of the applications running on that host. It can be implemented in software as operating system device drivers or could be delegated to a system-area network card.

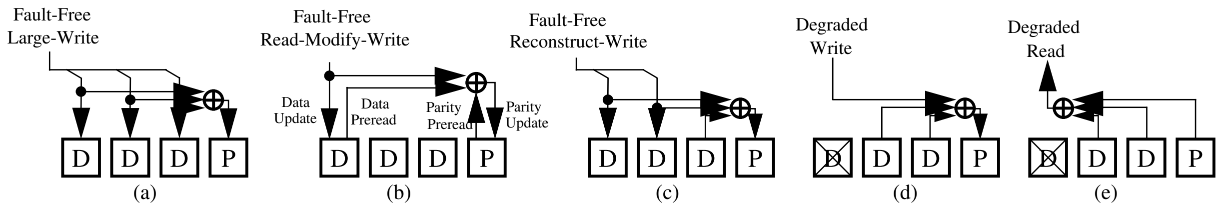



**Figure 2**. RAID level 5 hostwrite and hostread operations in the absence and presence of faults. An arrow directed towards a device represents a devwrite, while an arrow originating at a device represents a devread. The + operation represents bitwise XOR. A device marked with an X represents a failed device. Host write operations in fault-free mode are handled in one of three ways, depending on the number of units being updated. In all cases, the update mechanisms are designed to guarantee the property that after the write completes, the parity unit holds the cumulative XOR over the corresponding data units. In the case of a large write (a), since all the data units in the stripe are being updated, parity can be computed by the host as the XOR of the data units and the data and parity blocks can be written in parallel. If less than half of the data units in a stripe are being updated, the read-modify-write protocol is used (b). In this case, the prior contents of the data units being updated are read and XORed with the new data about to be written. This produces a map of the bit positions that need to be toggled in the parity unit. These changes are applied to the parity unit by reading its old contents, XORing it with the previously generated map, and writing the result back to the parity unit. Reconstruct-writes (c) are invoked when the number of data units is more than half of the number of data units in a parity stripe. In this case, the data units *not* being updated are read, and XORed with the new data to compute the new parity. Then, the new data units and the parity unit are written. If a device has failed, the degraded-mode protocols shown in (d) and (e) are used. Under degraded mode, all the operational devices are accessed whenever any device is read or written.

while a two-phase *hostwrite* breaks down into a first read phase (where *devreads* are sent to the disks) followed by a write phase (where *devwrite*s are issued). While we focus for the rest of this paper on RAID level 5 as our case study and evaluation architecture, our concurrency control protocols equally apply to the other levels.

### *2.1. Storage transactions*

A shared storage array partitions data across multiple devices and uses redundancy codes that cause a single high-level host read or write (*hostread* and *hostwrite*) operation to break down into multiple low-level disk I/O requests (*devread* and *devwrite*) possibly separated with some computation. We call these high-level operations *storage transactions*.

When applications on multiple hosts concurrently perform hostwrite storage transactions across shared devices which are transparently protected by RAID level 5, parity inconsistencies can arise. To illustrate this problem, consider two hosts writing to two different data blocks that happen to be in the same parity stripe, as shown in Figure 3(b). Because both data blocks are in the same parity stripe, 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 unit 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 the device (YB).

In this section, we would like to formalize the guarantees that storage-level transactions should provide to the higher levels (i.e. the guarantees that a shared storage array should provide to the higher levels).

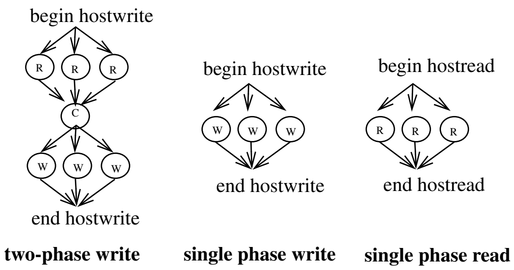



**Figure 3(a).** The general composition of storage-level host operations. Operations are either single phase or two-phase. Two-phase operations have a read phase followed by a write phase. A computation (C) separates the two I/O phases. Host operations are denoted by *hostread* and *hostwrite*, while disk level requests are denoted by *devread* and *devwrite*.

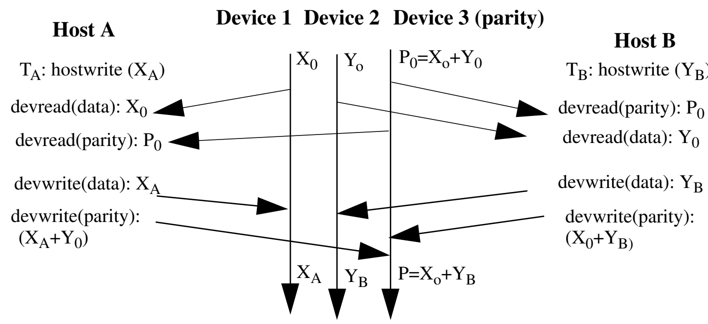



**Figure 3(b).** A timeline showing two concurrent read-modify-writes (two-phase writes) in RAID with no concurrency control provisions taken. Initially, the data units contain on device 1 and device 2 contain $X_0$ and $Y_0$ respectively and the parity unit is their XOR ($X_0+Y_0$). Although host A is updating device 1 and host B is updating device 2, they both need to update the parity, both read the same version of the parity, but Host A writes parity last overwriting B's parity write, leaving parity inconsistent.

Given that storage-level transactions are layered below high-level transactions, it makes little sense for them to replicate the performance-costly guarantees provided by the higher level. Accordingly, we concentrate on the aspects of the storage array that are not accounted for by traditional database transaction theory. Database transactions [Gray75, Eswa76] traditionally provide the ACID (Atomicity, Consistency, Isolation and Durability) properties [Haerder83, Gray93]. We argue that for storage transactions, we need and should provide only durability, consistency and isolation. These notions have a specific meaning for storage-level transactions:

- **Consistency.** Consistency for storage transactions is defined by the particular redundancy code and striping employed, but is well-known to the storage system. For example, in the case of RAID level 5, the consistency requirement is that the cumulative XOR of the data units in the stripe equals the parity block. It must be ensured in the face of concurrent host write requests and in the face of power and host failures.
- **Isolation**. We desire the storage system to ensure serializability for concurrents high-level reads and writes (storage transactions) submitted to it. Providing this guarantee simplifies higher levels and ensures that the consistency of the redundancy codes is preserved, since we know that a serial application of the storage transactions preserves consistency.
- **Durability**. The storage array needs to ensure that data written to it by completed storage transactions persists, never switching back to older data. This is the basic function of storage devices today - we will not address it further.
- **No atomicity**. The recovery property of database transactions, that guarantees that a transaction will appear to have either completed in its entirety or to never have started is called atomicity [Gray75]. Transaction atomicity is ensured by recovery mechanisms in the database [Gray93, Mohan92] in the face of three kinds of failures: transaction failures (user abort), system failures (loss of volatile memory contents) and media (storage device) failures. The recovery mechanism is triggered whenever a failure occurs. It does not assume or require that storage writes are atomic because 1) atomicity of storage writes (even without striping and redundancy) is expensive, and 2) the higher level mechanisms in the database already ensure this function at the granularity of a whole transaction.

Our goal in this paper then is to arrive at a protocol ensuring isolation and consistency for storage transactions. Given that the connectivity of system-area networks enables thousands of storage devices and hosts to be clustered in one shared array, we desire the protocol to scale to such large sizes. Furthermore, the protocol must be free from deadlocks. Finally, new functions required from storage device should be simple and clear, given that devices are manufactured by several independent vendors, and complex systems implemented by diverse organizations are likely to be fragile. In Section 5, we will address preserving consistency after failures. In Section 4, we will specialize techniques used to ensure isolation for database transactions to the problem of ensuring isolation of storage transactions. We briefly review these database techniques we draw upon in the next section.

## 3. Background: Serializability in database systems

The traditional correctness requirement for achieving isolation of database transactions is serializability [Papa79, Benstein87], which has traditionally been ensured using one of three approaches:

- **Locking.** This conservative approach is the most widely used in practice. Locks on shared data items are used to enforce a serial order on execution. Significant concurrency can be achieved with *two-phase locking* [Gray75], a programming discipline where locks may be acquired one at a time, but no lock can be released until it is certain that no more locks will be needed. Locking is the preferred solution when resources are limited and the overhead of lock acquisition and release is low. Two-phase locking, however, is susceptible to holding locks long enough to reduce concurrency and to deadlocks. A deadlock prevention or detection and resolution scheme must be employed in conjunction with two-phase locking.
- **Optimistic methods.** Locking is also known as the pessimistic approach, since it presumes that contention is common and locks data items before accessing them. Optimistic methods are rather aggressive since they assume conflict is rare and do not acquire locks before accessing the data but instead validate via another mechanism that the transaction's execution was serializable when the transaction is ready to commit [Eswa76, Kung81]. If a serializability violation is detected, the transaction is aborted and restarted. Optimistic protocols are desirable when the overhead of locking/unlocking is high (e.g. if it involves network messaging), when conflicts are rare or when resources are plentiful and would be otherwise idle (e.g. multiprocessors).
- **Timestamp ordering.** Timestamp ordering protocols select an a priori order of execution using some form of timestamps and then enforce that order [Bern80]. Most implementations verify timestamps as

transactions execute read and write accesses to the database, but some more optimistic variants delay the checks until commit time [Adya95]. 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 database tags each data item with a pair of timestamps, `rts` and `wts`, which correspond to the largest timestamp of a transaction that read and wrote the data item, respectively. Basically, a read by transaction `T` with timestamp `opts(T)` to data item `v` is accepted if `opts(T)>wts(v)`, otherwise it is immediately rejected. A write is accepted if `opts(T)>wts(v)` and `rts(v)`. If an operation is rejected, its parent transaction is aborted and restarted with a new larger timestamp. In order to avoid cascading aborts, reads are not allowed to read data items written by active (uncommitted) transactions. In fact, when an active transaction wants to update a data item, it first submits a `pre-write` to the database declaring its intention to write but without actually updating the data. The database accepts a pre-write only if `opts(T)>wts(v)` and `rts(v)`. A pre-write is committed when the active transaction T commits and a `write` is issued for each submitted `pre-write`. Only then is the new value updated in the database and made visible to readers. A transaction that issued a `pre-write` may abort, in which case its pre-writes are discarded and any blocked requests are inspected in case they can be completed. If a `pre-write` with `opts(T)` has been provisionally accepted but not yet committed or aborted, any later operations from a transaction `T'` with `opts(T')>opts(T)` are blocked until the pre-write with `opts(T)` is committed or aborted.

```
handle read(v, opts);
if (opts<wts(v)) then {
    return (REJECT);
} else if (opts>min-pts(v)) {
    put request on queue;
    min-rts(v)=MIN(opts, min-rts(v));
else {
    rts(v)= MAX(rts(v), opts)
    inspectQueuedWrites = true;
    return committed value;
}
```

```
handle pre-write(v, opts);
if (opts < rts(v)) then {
    return(REJECT);
} else if (opts < wts(v)){
    return (REJECT);
} else {
    min-pts(v)=MIN(opts,
                   min-pts(v));
    put request on queue;
    return(ACCEPT);
}
```

```
handle write(v, opts, new-value)
if (opts > min-rts(v)) {
    put request on queue;
else {
    write new-value to store;
    wts(v) = MAX(wts(v), opts);
    inspectQueuedReads = true;
}
```

**Figure 4**. Pseudo-code for basic timestamp ordering, showing the actions taken by the database upon receipt of a read (left), a pre-write (center) and a write request (right). *opts* represents the timestamp of the transaction making the request. *min-rts(v)* represents the smallest timestamp of a queued read to data item *v*, while *min-pts(v)* represents the smallest timestamp of a queued pre-write to *v*. When a write is processed, its corresponding pre-write is removed from the service queue. This could increase *min-pts(v)*, resulting in some reads being serviced. Similarly, when a read is processed and removed from the service queue, *min-rts(v)* could increase resulting in some write requests becoming eligible for service. When a read is processed, the *inspectQueuedWrites* flag is set, which causes the database to inspect the queue and see if any writes can be serviced. Similarly, when a write is processed, the *inspectQueuedReads* flag is set so that queued reads are inspected for service. Upon inspection of the queue, any requests that can be serviced are dequeued, possibly leading to computing new values of *min-rts(v)* and *min-pts(v)*.

| Baseline simulation parameters | |
|---|---|
| System size | 20 devices, 16 hosts, RAID level 5, stripe width = 4 data + parity, $10^3$ stripe units per device. |
| Host workload | random think time (normally distributed with mean 80 ms, variance 10 ms), 70% reads, host access address is uniformly random, access size uniformly varies between 1 and 4 stripe units. |
| Service times | Disk service time random (normally distributed with mean positioning of 8 ms, variance 1 ms + 0.06 microsec/byte (transfer). Network bandwidth is 10 MBytes/sec and has a random overhead of 1-2ms. Mean host/lock server message processing/request service time is 750 useconds. |

**Table 1**. Baseline simulation parameters. Host data is striped across the devices in a RAID level 5 layout, with 5 devices per parity group. Host access sizes excersize all the possible RAID write algorithms shown in Figure 2.

# 4. Concurrency control for shared storage arrays

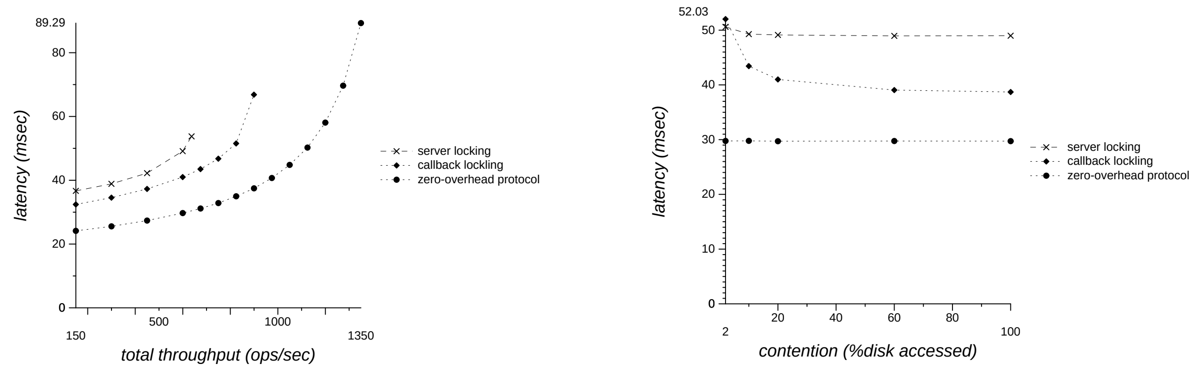
In this section, we present two centralized protocols, server locking and callback locking, and two distributed protocols, device-served locking and timestamp ordering. We show how the distributed protocols benefit from specialization to storage characteristics and technology trends. We evaluate performance relative to ideal performance, that of unprotected concurrent operation, which we call the *zero-overhead protocol* (no concurrency control work). First of all, however, we describe our evaluation environment.

## *4.1. Evaluation environment*

We implemented our protocols in full detail in simulation, using the Pantheon simulator system [Wilkes96]. We simulate a cluster system consisting of hosts and disks connected by a network. Table 1 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 [Seag97] over a fast switched local area network (like FibreChannel). Host clocks were allowed to drift within a practical few milliseconds of real-time [Mills88]. 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.

## *4.2. Centralized locking protocols*

With *server locking,* a centralized lock server provides locking on low-level storage block ranges. A host acquires an exclusive (in case of a *devwrite*) or a shared (in case of a *devread*) lock on a set of target ranges by sending a *lock message* to the lock server. The host may then issue the *devreads* or *devwrites* low-level




(a) (b)

**Figure 5**. (a) Scaling of server and callback locking. The bottom-most line represents the zero-overhead protocol. Centralized locking schemes bottleneck before delivering half of the maximum achievable throughput. The applied load was increased by increasing the number of active hosts in the system until saturation, starting from 4 hosts. The baseline workload (16 hosts) corresponds to the fourth point in the left-hand graph. (b) The right-hand graph shows the effect of contention on the baseline workload's end latency by decreasing the fraction of disk targeted by host accesses. The graph shows that under high contention, the pre-I/O latency of callback locking is as slow as that the latency of callback locking increases to that of server locking. Callback locking still reduces latencies by about half under moderate to high contention even for a random-access workload because the baseline workload is mostly reads (70%) and shared locks are cached at each host.

I/O requests to the devices. When all I/O requests complete, the host sends an *unlock message* to the lock server. As there is only one lock and one unlock message per high-level operation, the protocol is trivially two-phase and therefore serializable. Because all locks are acquired in a single message, latency is minimized and deadlocks are avoided. The lock server queues a host's lock request if there is an outstanding lock on a block in the requested range. Once all the conflicting locks have been released, a response is returned to the host. However, server locking introduces a potential bottleneck at the server and delays issuing the requests low-level I/O requests for at least one round trip of messaging to the lock server.

*Callback locking* [Howa88, Lamb91, Carey94] is a popular variant of server locking, which 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 in the near future and be able to avoid sending a lock message to the server. If a host requests a lock from the lock server that conflicts with a lock cached by another host, the server contacts the host holding the conflicting lock (this is the *callback message*), asking it to relinquish the cached lock so the server can grant a lock to the new owner. One common optimization to callback locking is to have locks automatically expire after a lease period so that callback messaging is reduced. Our implementation uses a lease period of four seconds.

Figure 5 highlights the scalability limitation of server locking protocols. It plots the host-end latency of the protocols against (a) throughput (number of hosts) and against (b) contention. The protocols bottleneck substantially before delivering the full throughput of the I/O system that is attainable with the zero-overhead protocol. This is caused by the fact that the server's CPU is bottlenecked with handling network messaging and processing lock and unlock requests (750usec per message send or receive). Callback locking reduces lock server load and lock acquisition latencies, provided that locks are commonly reused by the same host multiple times before a different host requests a conflicting lock. At one extreme, a host requests locks on its private data and never again interacts with the lock server until the lease expires. At the other extreme, each lock is used once by a host, and then is called back by a conflicting use. This will induce the same number of messages as simple server locking, but the requesting host must wait on two other machines, one at a time, to obtain a lock that must be called back potentially doubling pre-I/O latency. Pre-I/O latency can be even worse if a locks is shared by a large number hosts since all of them need to be called back.

Figure 5 and Figure 6 show 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 as the workload contains little of it, but from the dominance of read traffic which allows concurrent read locks at all hosts, until the next write. While more benefits would be possible if the workload had more locality, the false sharing between independent accesses that share a parity block limits the potential benefits of locality.

### *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[1]. This can be achieved in two ways: a host can send a single request to one of the lock servers which coordinate among themselves (ensuring proper concurrency control and free-

**Summary baseline performance**

| | Server Locking | Callback locking | Device-served locks | Timestamp ordering | Zero-overhead protocol |
|---|---|---|---|---|---|
| Mean latency (msec) | 49 | 41 | 33 | 32.8 | 29.71 |
| Throughput (ops/sec) | 599.6 | 599.6 | 600.2 | 599.7 | 600.4 |
| Average number of messages | 10.67 | 9.57 | 11.86 | 7.79 | 6.68 |
| Fraction of operations delayed (blocked or retried) | 0.62% | 33% | 0.31% | 0.16% | 0% |
| Peak 20-device throughput | 600 | 899.3 | 1275.1 | 1274.7 | 1349.5 |

**Figure 6**. Summary performance of the various protocols under the baseline configuration of 16 hosts and 20 devices. The table shows the mean host latency, throughput, average number of messages and the fraction of operation delayed (blocked or retried) for each protocol. Distributed protocols (device-served locking and timestamp ordering) approximate the ideal performance of the zero-overhead protocol. The performance of the distributed protocols reflects all the piggy-backing and overwrite optimizations discussed in the paper.

dom from deadlock) and return a single response to the host, or hosts can send directly send parallel lock requests to the servers based on the partitioning of locks. As the first approach simply displaces the responsibility at the cost of more messaging, we do not discuss it further. We assume locks are partitioned across the locks servers according to some static or dynamic scheme. When a host attempts to acquire multiple locks managed by multiple servers, deadlocks can arise. This can be avoided if locks are acquired in a specific order, but this implies that lock requests are sent to the lock servers in sequence—the next request is sent only after the previous reply is received—and not in parallel, increasing latency and lock hold time substantially. Alternatively, deadlocks can be detected via time-outs. If a lock request can not be serviced at a lock server after a given time-out, a negative response is returned to the host. A host recovers from deadlock by releasing all the acquired locks, and retrying lock acquisition from the beginning. While parallel lock servers do not have the bottleneck problem, this benefit comes at the cost of more messaging and usually longer pre-I/O latency. This induces longer lock hold time, increasing the window in time of potential conflict compared to an unloaded single lock server and resulting in higher probabilities of conflict.

### *4.4. Device-served locking*

Given the paramount importance of I/O system scalability and the opportunity of increasing storage device intelligence, we investigated embedding lock servers in the devices. The goal is to reduce the cost of a scalable serializable storage array, and by specialization, increase its performance. Our specializations exploit the two-phase nature of RAID storage transactions can be exploited to piggy-back lock messaging on the I/O requests; thereby, reducing the total number of messages, and latency.

In device-served locks, each device serves locks for the blocks stored on it. This balances lock load over all the devices, enhancing scalability. Like any parallel lock server scheme, simple device-served locking increases the amount of messaging and pre-I/O latency. However, often the lock/unlock messaging can be

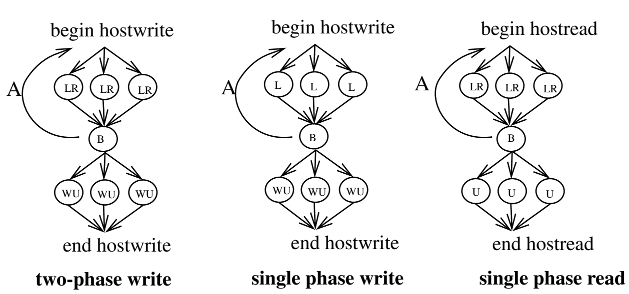



**Figure 7**. The breakdown of a host operation wiht device-served locking and the piggy-backing optimization. A node represents a messge exchange with a device. An "L" node includes a `lock` operation, "LR" represent the `lock-and-devread` operation, while "WU" stands for `devwrite-and-unlock`. The edges represent control dependencies. A "B" node represents a synchronization 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").

1. Faster lock servers can be built using low-latency networks, hardware-specialized transport protocols and symmetric multiprocessor machines. However, this simply increases the maximum attainable throughput but does not eliminate the scalability limitation.

eliminated by piggy-backing these messages on the I/O requests. Lock requests are piggy-backed on the first I/O phase of a two-phase storage transaction. To make recovery simple, we require that a host not issue any *devwrites* until all locks have been acquired, although it may issue *devreads*. Thusly, restarting an operation in the lock acquisition phase does not require recovering the state of the blocks (since no data has been written yet). In the case of single-phase writes, a lock phase must be added, preceding the writes since we can not bundle the lock and write requests together without risking serializability. However, unlock messages can be piggy-backed onto write I/O requests as shown in Figure 7. In the case of single-phase reads, lock acquisition can be piggy-backed on the reads, reducing pre-I/O latency, but a separate unlock phase is required. The latency can be hidden from the application since the data has been received. In the case of 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. This device-supported parallel locking is almost sufficient to eliminate the need for leased callback locks because 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. Given the increase in complexity that comes with callback locking and the increase in complexity when systems depend on identical semantics in devices manufactured by multiple independent vendors, we see pragmatic value in avoiding callback locking.

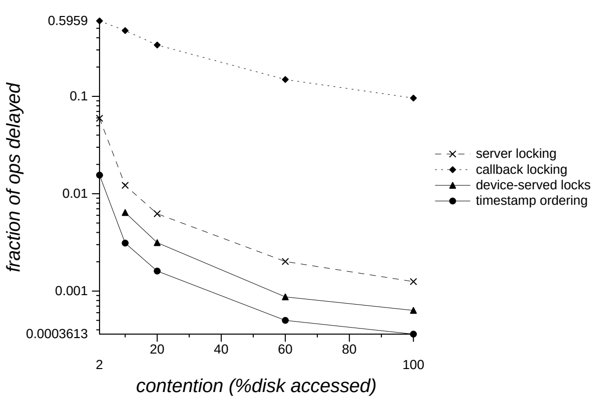



**Figure 8**. The performance of the protocols under high contention. When the hosts are conflicting over 2% of the active portion of the disk, device-served locking suffers from disk queues and latency growing without bound, due of timeout-induced restarts.

Device-served locking is more effective than the centralized locking schemes, as shown in the summary table of Figure 6. 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 has two disadvantages: it uses more messages than centralized locking protocols, as shown in the summary table of Figure 6, and it has performance vulnerabilities under high contention due to its susceptibility to deadlocks.The difficulty of configuring the

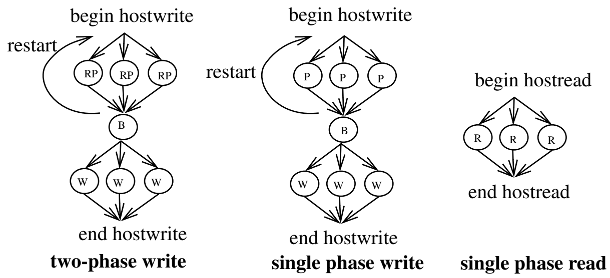



**Figure 9**. The composition of host operations in the optimized timestamp ordering protocol. `devread`, `devwrite`, and `pre-write` requests are denoted by "R", "W" and "P" nodes respectively. "RP" denotes a `read-and-prewrite` request. A "B" node represents a synchronization barrier at the host, where the host blocks until all preceding operations complete, restarting from the beginning if any of them fail. Some of the pre-writes or reads may be rejected by the device because they did not pass the timestamp checks. In a two-phase write, the pre-write requests are piggy-backed with the reads. Hence, both reads and two-phase writes use the minimal amount of messaging. Single phase writes still require a round of messages before the devwrite requests are issue.

time-outs is another disadvantage of device-served locking. Figure 8 graphs the behavior of the protocol under increasing contention, which we induce by limiting the fraction of storage addressed by all hosts. When hosts access no more than 2% of the active disk space, the disk queues start to grow because of increased lock hold times that occur when deadlocks must be detected. Note that although deadlocks may be uncommon, they have two second-order effects; they cause a large number of requests to be queued behind the blocked (deadlocked) requests until time-outs unlock the effected data; and when deadlocks involve multiple requests at multiple devices, time-outs lead to inefficiencies because they restart more operations than necessary to ensure progress. This results in further messaging load on the network and in wasting device resources on more message and request processing.

### *4.5. Timestamp ordering*

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. Following Section 3, timestamp ordering works by having hosts independently determine a total order in which high-level operations should be serialized, and by providing information about that order (in the form of a timestamp) in each I/O request so that the devices can enforce the ordering. 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 can not occur and all allowed schedules are serializable. Instead, out-of-order requests will be rejected causing their parent high-level operation to be aborted and retried with a higher timestamp. As in the database case, since each device is performing a local check, a write request may pass the check in some devices, but the high-level operation may abort due to failed checks in other devices. To avoid complex and expensive undo operations, writes need consensus before allowing changes; splitting the write protocol into a pre-write phase followed by a write phase ensures that all the devices take the same decision. The cluster of hosts share a loosely-synchronized clock used to generate timestamps. New timestamps are generated at a host by sampling the local clock, then appending the host's unique identifier to the least significant bits of the clock value. Each device maintains two timestamps associated with each block (`rts` and `wts`) as discussed in Section 3. The device similarly maintains a queue of requests, reads, writes and pre-writes which are awaiting service (blocked behind a pre-write). As in Section 3, we denote by `min-pts` the smallest timestamp for a pre-write that has been accepted in a specific block's request queue.

4.5.1. Implementation of timestamp ordering

We describe the read-modify-write protocol here since it employs the piggy-backing optimization and is of reasonable complexity. 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 host starts by generating a new timestamp, `opts`, then sends low-level I/O `read` requests to the data and parity devices, tagging each request with `opts`, and bundling each request with a `pre-write` request. The device receiving a "`read-and-pre-write`" request, performs the necessary timestamp checks both for a read and a pre-write, accepting the request only if both checks succeed; that is `opts>rts` and `wts`. An accepted request is queued if `opts>min-pts` because there is no outstanding pre-write with a lower timestamp, otherwise data is returned to the host and `rts` is updated if `opts>rts`. When the host has received all the data from the accepted "`read-and-pre-write`" requests, it computes the new parity and sends the new data and parity in low-level write I/O requests also tagged with `opts`. The devices are guaranteed by the acceptance of the pre-write to update `wts`, discard the corresponding pre-write request from the queue, possibly increasing `min-pts`. The request queue is then inspected to see if any `read` or `read-and-pre-write` requests can now be completed. Under normal circumstances, the read-modify-writes protocol does not place any overhead, just like piggy-backed device-based locking.

We discuss optimizations to the basic timestamp ordering protocol next, which lend them to efficient implementation. These optimizations were implemented in our simulation, and the results reflect their effects.

4.5.2. Implementation optimizations

**Minimizing buffering overhead.** The protocol as presented can induce high buffering overhead at the storage device when there is high overwrite and read activity to overlapping ranges. If the storage device has accepted multiple pre-writes to a block, and their corresponding writes are received out of order, the writes have to be buffered and applied in order to satisfy any intermediate reads. Our approach is to apply

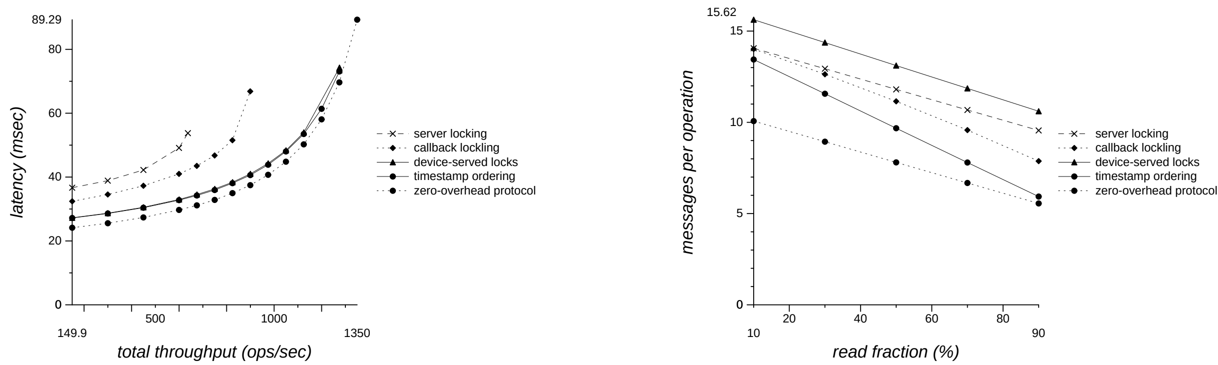



**Figure 10**. Scalability of timestamp ordering compared to the locking variants (left) and the effect of workload (write/read) composition on messaging overhead for the various protocols (right). Under read-intensive workloads, timestamp ordering comes close to the zero-overhead protocol. Its messaging overhead increases as the fraction of writes increases. However, it still performs the least amount of messaging across the range of workloads.

writes as soon as they arrive meeting the stringent scheduling requirements of modern disk drives. As a result, some reads may have to be rejected. Although readers can starve in our protocol if there is a persistent stream of writes, this is unlikely in shared arrays. The advantage of immediate write processing is that storage devices can exploit their ability to stream data at high data rates to the disk surface.

**Avoiding timestamp accesses.** The alert reader may have noticed that the protocols require 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, nearby 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 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. 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 for concurrency control and for efficient timestamp management has been suggested in the Thor client-server object-oriented database management system [Adya95].

In addition to being highly scalable, as illustrated in Figure 10, another advantage of timestamp ordering is that uses the smallest amount of messaging compared to all the other protocols. It has no messaging overhead on reads, and with the piggy-backing optimization applied, it can also eliminate the messaging overhead associated with read-modify-write operations.

### *4.6. Blocking/retry behavior*

When several operations attempt to access a conflicting range, the succeeding operations are delayed until the first one completes. The probability of delay depends on the level of contention in the workload of course. But even for a fixed workload, the concurrency control protocol and environmental factors (e.g. network reordering of messages) can result in different delay behavior for the different protocols. As shown in Figure 6 and Figure 8, the fraction of operations delayed is highest for callback locking because it has the highest window of vulnerability to conflict (lock hold time). Distributed device-based protocols both do better than callback locking and server locking because they exploit piggy-backing of lock/ordering requests on the I/Os thereby avoiding the latency of communicating with the lock server before starting the I/O and shortening the window of vulnerability to conflict.

Both device-based protocols, however, are potentially more sensitive to the message transport layer, precisely message arrival skew. Message arrival skew can cause deadlocks and restarts for device-served locks, and rejections and retries for timestamp ordering because sibling requests are serviced in a different order at different devices. Restarts and retries are also counted as delays and are therefore accounted for in Figure 6 and Figure 8.

To investigate the effect of message skew on the delay and latency behavior of the protocols, we conducted an experiment where we varied the degree of reordering performed by the network and measured its effect on delay and latency.[1] Message latency was modeled as a uniformly distributed random variable over a given window size, extending from 1 to *ws* milliseconds ([*1-ws*]). A larger window size implies highly variable message latencies and leads to a higher probability of out-of-order message arrival. Figure 11 graphs latency and fraction of operations delayed against *ws*. Although timestamp ordering and device-based locking are sensitive to message skew, the end effect on host latency is less noticeable. This is

1. The probability of message arrival skew (out-of-order arrival) may grow because of aberrant host or network behavior. The time between the receipt of an I/O in one device and the receipt of its sibling at another may grow because: (1) a slow host or as a side effect of process scheduling decisions at the host; (2) some I/O requests may get lost, requiring a retransmission by the network protocol, causing some requests to arrive relatively earlier than others; (3) the network may have unpredictable transmission latencies due to bursty traffic, or may dynamically route sibling requests over different links with different transmission latencies or loads.

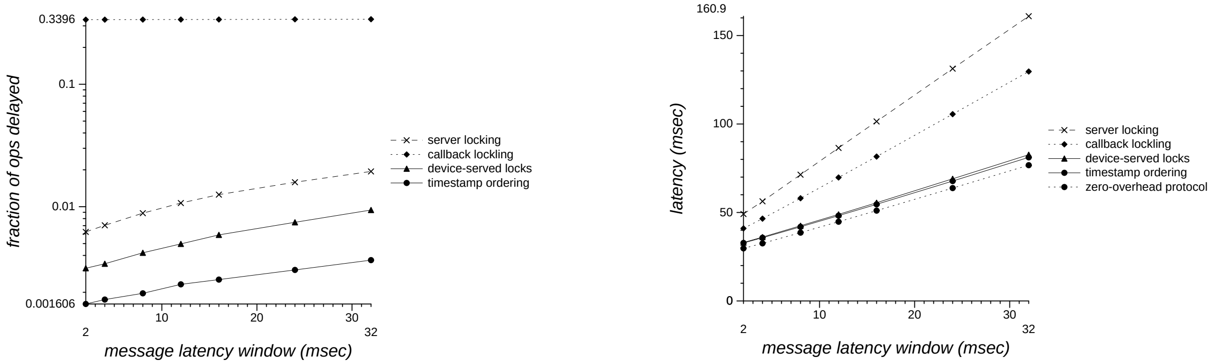



**Figure 11**. Effect of the variability in network message latencies on the fraction of operation delayed or retried (left) and on host-end latency (right) for the various protocols. Message latency is varied uniformly between 1ms and a maximum window size. The graphs plot latency and fraction of operations delayed versus the size of the variability window size. Note that for the left-hand graph, the y-axis is plotted in log scale.

because high message variability in fact plagues the centralized locking variants even more since they perform more pre-I/O messaging.

### *4.7. Effect of faster disks*

Finally, we carried out a final experiment to try to anticipate the performance of the protocols in the near future given the current industry trends. Disk drive performance is expected to keep growing as a result of evolutionary hardware technology improvements (e.g. higher densities leading to higher transfer rates), the introduction of new technologies (e.g. solid-state disks leading to reduced access times) or device economics (dropping memory prices and increased device functionality [Gibson98] leading to larger on-disk cache memory sizes). To simulate the effect of faster disks, we carried out an experiment where we increased the hit rate of the disk cache and measured the scalability of the protocols as the hit rate increased. As shown in Figure 12, callback locking does not keep up with the throughput of the faster disk drives. Device-served locks continues to approximate ideal scaling behavior at 20% cache hit rates. However, at 60% cache hit rates, the devices can support a larger number of hosts, resulting in increased probability of real and spurious deadlocks. As a result, the fraction of operations restarted increases dramatically. Timestamp ordering, however, continues to approximate the performance of the zero-overhead protocol even under high load. This is because under high load, when conflicting requests build up in the disk queues, timestamp ordering naturally ensures that all devices will service sibling requests in the same order (timestamp order), while device-served locking services requests in a potentially random order (device-served locking satisfies a request at a device as soon as the locks it requests are available.)

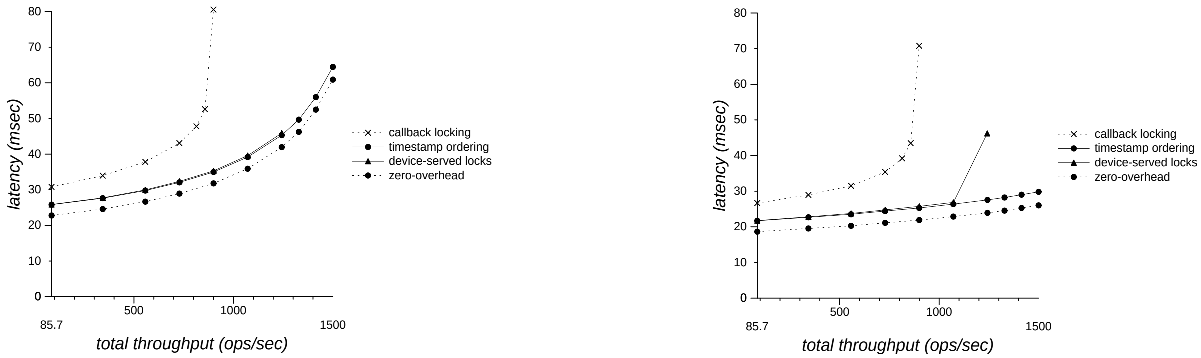



**Figure 12**. The scalability of callback locking, device-served locking and timestamp ordering for 20% (left) and 60% (right) disk cache hit rates. The bottom-most line represents the performance of the zero-overhead protocol. While timestamp ordering continues to approximate ideal performance at higher disk hit rates, callback locking bottlenecks at a fraction of the possible throughput. Server locking (not shown here) bottlenecks at an even lower throughput. The singular point on the right-hand graph corresponds to device-served locking under 58 hosts (the number of devices is kept fixed at 20). Faster disks can support more hosts, but contention among more hosts at some point leads to a sharp increase in the number of restarted operations for device-based locks (the fraction of operations restarted jumps from a negligible value to 17%).

## 5. Recovery for shared storage arrays

Shared storage arrays must ensure their consistency on recovery from a host failure, a partial or a general power failure. Consistency can be ensured by recomputing all the redundancy codes on recovery from a failure.This is prohibitingly expensive, of course. The ideal approach would be to recompute the code only for those stripes that are in fact inconsistent. This requires the recovery process to determine which blocks were in the process of being updated during the failure.

**Consistency under locking.** The locking protocols can be extended to ensure consistency pessimistically in the following way: if a host crashes holding a write lock, then the lock server marks the target range suspect, and recomputes parity on recovery to ensure it is consistent before handing the lock to another host. This requires maintaining the lock state (the outstanding write locks) at the server on stable storage so that on recovery, the lock server can determine the suspect ranges. This makes the lock operation expensive, since each lock and unlock must be recorded on persistent storage. An alternative approach uses "busy bits" that are set before the update and reset once the update completes. Using "busy bits" to mark blocks that are being updates induces two extra synchronous I/Os per write.[1]

**Consistency under timestamp ordering**. The basic timestamp ordering protocol can be extended to provide consistency in one of two approaches. The first is similar to the busy-bit approach. The second avoids

1. This overhead can be reduced, however, using write-ahead logging and group commit [Hag87] as used in the Petal clustered storage system [Lee96] or through the use of NVRAM. Note that the amount of NVRAM required could be large especially in the case of callback locking where locks are held by the hosts for long periods of time.

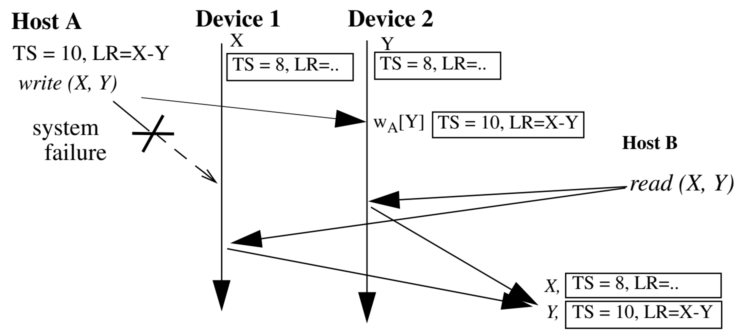



**Figure 13**. An illustration of how linkage records are used to detect writes that did not complete on recovery. Host A intends to write to both blocks X and Y. The host attaches the timestamp 10, and the linkage record X-Y to each write request.The write request is received and processed at device 2, but due to a host or general power failure, the write request sent to device 1 is not processed. Since device 2 processed the write, updating the timestamp and the linkage record in the header of the block together with the data. Upon recovery, Host B reads the suspect blocks and inspects the linkage records. The cross-check fails because the linkage record returned with block Y refers to block X, but block X has a lower timestamp (8<10) which implies that block X has not been updated when B's read was processed.

the use of extra I/Os and NVRAM completely at the expense of slightly more recovery overhead. The first approach uses information in the pre-write queues of the devices to determine writes that were active during the failure. Note that the active writes are easy to determine under timestamp ordering, because if a multi-device write did not complete, then it must have a `pre-write` still in the service queue at some device (for which the corresponding write was not received). The recovery manager can query the devices for the `pre-write` that have been accepted but not committed and deduce from that the blocks that belong to inconsistent stripes and recompute the parity for those stripes. Similar to the "busy bit" approach, this requires maintaining the pre-writes queues on stable storage inducing extra I/Os unless the storage device is endowed with enough NVRAM.

The second approach maintains `pre-write` queues in volatile memory accessing the disk only to write data to the disk surface. It uses an additional data structure called a *linkage record*. A linkage record is, in principle, a list of (device id, block id) pairs. It is used to describe the data that is being written in the *hostwrite* operation. When a *devwrite* is processed, the host data is written to disk and the device also writes in the block header the timestamp and the linkage record associated with the *hostwrite*.[1] Upon recovery, the recovery process can find suspect blocks by simply reading the linkage records of the blocks and cross-checking them, as illustrated by the example of Figure 13. Block data does not need to be

1. Note that the linkage record can be very compact and can be stored in the extra bytes available in the sector headers of modern disk drives (~20 bytes). Linkage records can be compact because hosts issue writes to logically contiguous range of bytes, and hence the linkage record can simply consist of two integers: the address of the first block and the of the last block in the range.

transferred and parity need not be recomputed unless cross-checking actually fails. In this case, the stripe can be made consistent by reading the data units, computing fresh parity, and writing it back to the array.

## 6. Future work

The protocols we presented in this paper assume that storage controllers do not cache data or parity blocks. In our ongoing work, we are evaluating ways to support coherent caching of parity and data blocks.

The protocols also ensure both isolation (serializability) and consistency. Serializability is ensured even if host requests access overlapping data. Several applications perform higher level locking so that hosts never concurrently access overlapping data. If serializability is not required by the higher level, then the only correctness criterion that storage transactions should provide to higher levels is the consistency of redundancy codes. The presented protocols can be further specialized so that they only ensure consistency. Our ongoing work is focussing on specializing the protocols to exploit such weaker guarantees.

## 7. Conclusions

Shared storage arrays promise databases and filesystems highly scalable, reliable, self-managing storage. However, they eliminate the storage controller and, thereby the central point of control from storage systems, causing storage added function such as RAID to be visible over the network, potentially causing inconsistencies as a result of concurrency or failures. Rapidly increasing storage device intelligence, coupled with the special characteristics of storage operations can be successfully exploited to arrive at a high-performance solution to this problem that realizes the scalability potential of the shared array I/O architecture.

In this paper, we examine four concurrency control schemes for shared storage arrays; two centralized ones: simple server locking and server locking with leased callbacks; and two distributed ones with device participation; distributed locking using storage-device-embedded lock servers, and timestamp ordering schemes based on loosely synchronized clocks. Simulation results show that both server-based locking schemes suffer from scalability limitations. Moreover, we found callback locking particularly suspect if applications do not have much inherent locality or if the storage system causes this locality to be disturbed.

Distributed concurrency control with device support is attractive as it scales control capacity with storage and performance capacity, and offers the opportunity to piggyback lock/ordering messages on operation requests, eliminating message latency costs. Distributed locking, like any multiple lock server solution, implies that locks can not be acquired at once making it susceptible to deadlocks. Deadlock avoidance via serial locking stretches the normal case latency and is therefore undesirable. Time-out-based deadlock detection and recovery yields a simple strategy and is shown to impose little performance overhead except for cases of extreme contention.

Timestamp ordering based on loosely synchronized clocks is free from deadlocks, places less stress on the network than distributed locking, is vulnerable to fewer conflicts and has performance advantages under high load. Furthermore, we show how it can be extended with operation history recording using embedded linkage records to ensure proper consistency after failures without inducing extra I/Os under normal operation. We have completed, with little effort, an implementation of timestamp ordering in our prototype storage system.

## References


**[Adya95]** A. Adya, R. Gruber, B. Liskov and U. Maheshwari, "Efficient optimistic concurrency control using loosely synchronized clocks", *Proceeding of the SIGMOD International Conference on Management of Data*, May 1995.

**[Bart81]** J. Bartlett, "A nonstop kernel", *Proceedings of the Eighth Symposium on Operating System Principle*s, December 1981, pp. 19-22.

**[Benn96]** A. Benner, "Fibre Channel: Gigabit Communications and I/O for Computer Networks", McGraw Hill, New York, 1996.

**[Bern80]** P. Bernstein and N. Goodman, "Timestamp based algorithms for concurrency control in distributed database systems", *Proceedings of the 6th International Conference on Very Large Databases*, October 1980.

**[Bern87]** P. Bernstein, V.Hadzilacos, and N. Goodman, Concurrency control and recovery in database systems, Addison-Wesley, Reading, MA, 1987.

[**Bhide91]** A. Bhide, E. Elnozahy, and S. Morgan, "A Highly Available Network File Server", *Proceedings of the 1991 USENIX Winter Conference,* pp. 199-206.

**[Blaum94]** M. Blaum, J. Brady, J. Bruk, J. Menon, "EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures," *Proceedings of the 21st ISCA*, Chicago IL, April 18-21, 1994, pp. 245-254.

**[Boden95]** N. Boden et al., "Myrinet: A Gigabit-per-Second Local Area Network", *IEEE Micro*, Feb. 1995.

**[Cao93]** P. Cao, S. Lim, S.Venkataraman and J.Wilkes, "The TickerTAIP parallel RAID architecture", *Proceedings of the Symposium on Computer Architecture*, 1993, pp. 52-63

**[Carey94]** M. Carey, M. Franklin, and M. Zaharioudakis, "Fine-grained sharing in a page server OODBMS," *Proceedings of the 1994 ACM SIGMOD*, Minneapolis, MN, May 1994.

**[Eswa76]** K. Eswaran, J. Gray, R. Lorie and L. Traiger, "The notions of consistency and predicate locks in a database systems." *Communications of the ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.

**[Gibson92]** G. A. Gibson, Redundant Disk Arrays: Reliable, Parallel Secondary Storage, The MIT Press, 1992.

**[Gibson97]** G. Gibson et al, "File Server Scaling with Network-Attached Secure Disks" *Proceedings of ACM SIGMETRICS*, June 1997.

**[Gibson98]** G. Gibson et al, "Cost-effective high-bandwidth storage architecture" *Proceedings of ACM ASPLOS*, October 1998.

**[Gold95]** R. Golding, E. shriver, T. Sullivan and J. Wilkes, "Attribute-managed storage," Workshop on modeling and specification of I/O, San Antonio, Texas, October 1995.

**[Gray75**] J. Gray, R. Lorie, G. Putzulo, and I. Traiger, "Granularity of locks and degrees of consistency in a shared database", IBM Research Report, RJ1654, September 1975.

**[Gray93]** J. Gray and A. Reuter. Transaction Processing: Concepts and techniques. Morgan Kaufmann, San Mateo, CA, 1993.

**[Haerd83]** T. Haerder and A. Reuter, "Principles of Transaction-oriented Database Recovery", *ACM Computing Surverys,* Vol. 15, No. 4, pp. 287-317.

**[Hag87]** R. B. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, 8-11 November 1987, pp 155-162.

**[Holl94]** M. Holland, G. Gibson, and D. Sieworek, "Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk
Arrays," *Journal of Distributed and Parallel Databases*, Vol. 2, No. 3, pp. 295-335, July 1994.

**[Horst95]** R. Horst, "TNet: A Reliable System Area Network", *IEEE Micro,* Feb. 1995.

**[Howa88]** J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 51-81, February 1988.

[**Kung81]** H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control", *ACM Transactions on Database Systems,* vol. 6, no. 2, June 1981.

**[Lamb91]** C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," *CACM*, Vol. 34, No. 10, October 1991.

**[Lee96]** E. Lee and C. Thekkath, "Petal: Distributed Virtual Disks", *Proceedings of the seventh ASPLOS,* October 1996, pp. 84-92.

**[Mass98]** P. Massiglia, "Changing storage subsystems," *Computer Technology Review*, Jan 1999.

**[Mills88]** D. L. Mills, "Network time protocol: specification and implementation," DARPA-internet report RFC 1059, DARPA, July 1988.

**[Mohan92]** C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, Vol. 17, No. 1, pp. 94-162, 1992.

**[Papa79]** C. Papadimitriou, "Serializability of concurrent updates," *Journal of the ACM*, Vol. 26, No. 4, October 1979, pp. 631-653.

[**Patt88]** D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks", *Proc. of ACM SIGMOD*, June 1988.

**[Seag97]** Seagate Technology, "Cheetah: Industry-Leading Performance for the Most Demanding Applications", *http://seagate.com*, 1997.

**[Wilkes96]** J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system", *ACM Transactions on Computer Systems*, vol. 14, no. 1, February 1996, pp.108-136.