D-SPTF: Decentralized Request Distribution in Brick-based Storage Systems

Christopher R. Lumb*
Carnegie Mellon Univeristy

Richard Golding
IBM Almaden Research

Gregory R. Ganger
Carnegie Mellon University

ABSTRACT

Distributed Shortest-Positioning Time First (D-SPTF) is a request distribution protocol for decentralized systems of storage servers. D-SPTF exploits high-speed interconnects to dynamically select which server, among those with a replica, should service each read request. In doing so, it simultaneously balances load, exploits the aggregate cache capacity, and reduces positioning times for cache misses. For network latencies expected in storage clusters (e.g., 10–200μs), D-SPTF performs as well as would a hypothetical centralized system with the same collection of CPU, cache, and disk resources. Compared to popular decentralized approaches, D-SPTF achieves up to 65% higher throughput and adapts more cleanly to heterogenous server capabilities.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems, Performance evaluation*

General Terms

Management, Performance

Keywords

Storage systems, Brick Based Storage, Distributed Systems, Disk Scheduling, Decentralized Systems

1. INTRODUCTION

Many envision enterprise-class storage systems composed of networked "intelligent" *storage bricks* [7, 8, 9, 14]. Each brick consists of a few disks, RAM for caching, and CPU for request processing and internal data organization. Large storage infrastructures could have hundreds of storage bricks. The storage analogue of cluster computing, brick-based systems are promoted as incrementally scalable and (in large numbers) cost-effective replacements for todays high-end, supercomputer-like disk array systems. Data redundancy across bricks provides high levels of availability and reliability (à la the RAID arguments [20]), and the aggregate resources (e.g., cache space and internal bandwidth) of many bricks should exceed those of even high-end array controllers.

An important challenge for brick-based storage, as in cluster computing, is to effectively utilize the aggregate resources. Meeting this challenge requires spreading work (requests on data) across storage bricks appropriately. In storage systems, cache hits are critical, because they involve orders of magnitude less work and latency than misses (which go to disk). Thus, it is important to realize the potential of the aggregate cache space; in particular, data should not be replicated in multiple brick caches. During bursts of work, when queues form, requests should be spread across bricks so as to avoid inappropriate idleness and, ideally, so as to reduce disk positioning costs. Achieving these goals is further complicated when heterogeneous collections of bricks comprise the system. In traditional disk array systems, the central disk array controller can provide all of these features. Existing protocols for decentralized storage cannot.

D-SPTF is a request distribution protocol for brick-based storage systems that keep two or more copies of data. It exploits the high-speed, high-bandwidth communication networks expected for such systems to achieve caching, load balancing, and disk scheduling that are competitive with like-resourced centralized solutions. Briefly, it works as follows: Each READ and WRITE request is distributed to all bricks with a replica. WRITE data goes into each NVRAM cache, but all but one brick (chosen by hash of the data's address) evict the data from cache as soon as it has been written to disk. Only one brick needs to service each READ request. Bricks explicitly *claim* READ requests, when they decide to service them, by sending a message to all other bricks with a replica. Cache hits are claimed and serviced immediately. Cache misses, however, go into all relevant local queues. Each brick schedules disk requests from its queue independently and uses CLAIM messages to tell other bricks to not service them. *Pre-scheduling* and *service time bids* are used to cope with network latencies and simultaneous scheduling, respectively.

D-SPTF provides all of the desired load distribution properties. During bursts, all bricks with relevant data will be involved in processing of requests, contributing according to their capabilities. Further, when scheduling its next disk access, a brick can examine the full set of requests for data it stores, using algorithms like Shortest-Positioning-Time-First (SPTF) [15, 22]. Choosing from a larger set of options

*Currently at IBM Alamaden Research

ASPLOS'04, October 7–13 ,2004, Boston, Massachusetts, USA.

significantly increases the effectiveness of these algorithms, decreasing positioning delays and increasing throughput. For example, in a brick-based system keeping three replicas of all data (e.g., as in FAB [8]), D-SPTF increases throughput by 12–27% under heavy loads and also under periods of transient load imbalance. The improvement grows with the number of replicas.

D-SPTF also provides the desired cache properties: exclusivity and centralized-like replacements. Ignoring unflushed NVRAM-buffered writes, only one brick will cache any piece of data at a time; in normal operation, only one brick will service any READ and, if any brick has the requested data in cache, that brick will be the one. In addition to exclusive caching, D-SPTF tends to randomize which brick caches each block and thus helps the separate caches behave more like a global cache of the same size. For example, our experiments show that, using D-SPTF and local LRU replacement, a collection of storage brick caches provide a hit rate within 2% of a single aggregate cache using LRU for a range of workloads.

This paper describes and evaluates D-SPTF via simulation, comparing it to the centralized ideal and popular decentralized algorithms. Compared to LARD and hash-based request distribution, D-SPTF is as good or better at using aggregate cache efficiently, while providing better short-term load balancing and yielding more efficient disk head positioning. It also exploits the resources of heterogeneous bricks more effectively.

The remainder of this paper is organized as follows. Section 2 describes brick-based storage and request distribution strategies. Section 3 details the D-SPTF protocol. Section 4 describes our simulation setup. Section 5 evaluates D-SPTF and compares it to other decentralized approaches and the centralized ideal. Section 6 discusses additional related work.

2. BRICK-BASED STORAGE SYSTEMS

Most current storage systems, including direct-attached disks, RAID arrays, and network filers, are centralized: they have a central point of control, with global knowledge of the system, for making data distribution and request scheduling decisions.

Many now envision building storage systems out of collections of federated smallish bricks connected by high-performance networks. The goal is a system that has incremental scalability, parallel data transfer, and low cost. To increase the capacity or performance of the system, one adds more bricks to the network. The system can move data in parallel directly from clients to bricks via the switched network. The cost benefit is expected to come from using large numbers of cheap, commodity components rather than a few higher-performance but custom components.

Bricks are different from larger centralized systems in several ways: bricks are small, have moderate performance, and often are not internally redundant. Moderate performance and size means that the system needs many bricks, and must be able to use those bricks in parallel. The lack of internal redundancy means that data must be stored redundantly across bricks, with replication being the most common plan. In addition, incremental growth means that, over time, a storage system will tend to include many different models of bricks, likely with different storage capacity, cache size, and IO transfer performance.

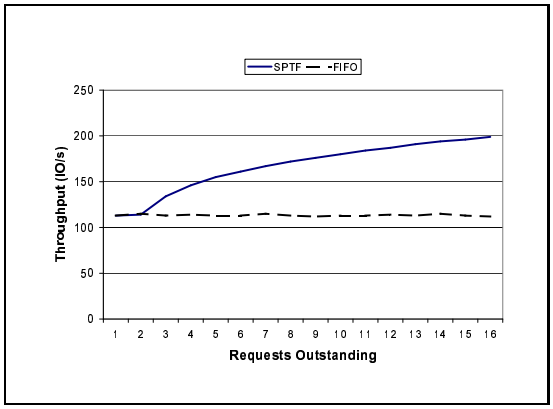

Figure 1: Disk throughput with SPTF scheduling, as a function of queue depth. The data shown are for a closed synthetic workload (see Section 5) with a constant number of pending small requests to random locations on a Quantum Atlas 10K disk.

Composing a system from independent bricks means that each brick does a small fraction of the overall work and that there is no central control. As a consequence, each brick has only a local viewpoint when making decisions. We focus here on three issues made more difficult by the lack of global information: disk head scheduling, cache utilization, and inter-brick load balancing. Existing mechanisms address one or two of these problems at the expense of the others. Each of these problems is compounded when the population of bricks is heterogeneous. The D-SPTF protocol addresses all three problems by exploiting the high-speed networks expected in brick-based systems, allowing bricks to loosely coordinate their local decisions.

2.1 Disk head scheduling

Disk scheduling has a significant effect on performance, because the mechanical positioning delays of disk drives depend on the relative distance between consecutive media accesses. When the sequence of operations performed by a drive cannot be orchestrated (e.g., when using FIFO scheduling), the drive spends almost all of its time seeking and waiting for the media to rotate into position. The importance of scheduling well continues to grow as the density of data on media increases: the time required to transfer a block of data off media decreases much faster than the positioning delays.

The shortest-positioning-time-first (SPTF) scheduling discipline [15, 22] is the state-of-the-art. It works by considering all requests in the queue and selecting the one that the head can service fastest (i.e., with the shortest total seek plus rotation delay). SPTF schedules work best when the request queue has many items in it, giving it more options to consider. Figure 1 illustrates this effect of queue depth on SPTF's ability to improve disk throughput. With only one or two operations pending at a time, SPTF has no options and behaves like FIFO (with two pending, one is being serviced and one is in the queue). As the number of pending requests grows, so does SPTF's ability to increase throughput—at 16 requests outstanding at a time, throughput is 70% higher than FIFO.

Brick-based systems tend to distribute work over many bricks, which decreases the average queue length at each brick. This, in turn, results in less opportunity for scheduling the disk head well. Avoiding this requires increasing the queue depths at bricks making scheduling decisions. One way to do this is to direct requests to only a few bricks; doing so would increase the amount of work at each brick, but leaves other bricks idle and thereby results in less overall system performance than if all disks were transferring at high efficiency. Another solution is to send read requests to all bricks holding a copy of a data item, and use the first answer that comes back. However, this approach duplicates work; while it can improve one request's response latency, overall system throughput is the same as a single brick.

2.2 Load balancing

When there is a choice of where data can be read from, one usually wants to balance load.[1] Centralized systems can do this because they know, or can estimate, the load on each disk. For example, the AutoRAID system directs reads to the disk with the shortest queue [25].

There are simple ways to spread requests across bricks to get balanced load, over the long term. For example, the system can determine where to route a request for a data block by hashing on the block address, or by using other declustering techniques [13, 12]. Then, as the system reads and writes data, the load should (statistically) be approximately even across all the bricks.

However, this is not as good as a centralized system can do: spreading requests gives balance only over the long term, but semi-random distribution of requests can cause transient imbalances, where some bricks get several requests while others get none. Moreover, having different kinds of bricks in the system makes this problem more difficult. With heterogeneous bricks, request distribution algorithms must try to route more requests to faster bricks and fewer to slower ones—where "slower" and "faster," of course, depend on the interaction between the workload, the amount of cache and the specific disk models that each brick has, and any transient issues (e.g., one brick performing internal maintenance).

2.3 Exclusive caching

Maximizing the cache hit rate is critical to good storage system performance. Hence, the cache resources must be used as efficiently as possible. The cache resources in a brick-based system are divided into many small caches, and replacement decisions are made for each cache independently. This independence can work against hit rates. In particular, the system should generally keep only one copy of any particular data block in cache, to maximize cache space and increase hit rate.[2] Distributed systems do not naturally do so: if clients read replicas of a block from different bricks, each brick will have a copy in cache, decreasing the effective size of the aggregate cache in the system. Always reading a particular data item from the same brick will solve this problem at the cost of dynamic load balancing and dynamic head scheduling.

[1]Note that there is a data placement component of load balancing in large-scale systems, which occurs before request distribution enters the picture. Clearly, request distribution can only affect load balancing within the confines of which bricks have replicas of data being accessed.

[2]In other environments, such as web server farms, there is value in having very popular items in multiple servers' caches. For storage, however, repeated re-reading of the same data from servers is rare, since client caches capture such re-use. Thus, cache space is better used for capturing more of the working set.

2.4 Achieving all three at once

Any one of these concerns can be addressed by itself, and LARD [19] and hash-based request distribution schemes can provide both long-term statistical load balancing and exclusive caching. However, no existing scheme provides all three. Further, a heterogeneous population of bricks complicates most existing schemes significantly, given the vagaries of predicting storage performance for an arbitrary workload.

The fundamental property that current solutions share is that they cannot efficiently have knowledge of the current global state of the system. They either choose exactly one place to perform a request, but without knowledge of the current state of the system, or they duplicate work and implicitly get global knowledge at tremendous performance cost.

The D-SPTF approach increases inter-brick communication to make globally-effective local decisions. It involves all bricks that store a particular block in deciding which brick can service a request soonest. When a request will not be serviced immediately, D-SPTF also postpones the decision of which brick will service it. By queueing a request at all bricks that store a copy of the block, the queue depth at each brick is as deep as possible and disk efficiency improves. Further, by communicating its local decision to service a request, a brick ensures that only it actually does the work of reading the data from disk. This naturally leads to balanced load and exclusive caching. If a brick already has a data item in cache, then it will respond immediately, and so other bricks will not load that item into cache. If one brick is more heavily utilized than others, then it will not likely be the fastest to respond to a read request, and so other bricks will pick up the load.

D-SPTF also naturally handles heterogeneous brick populations. Under light load, the fast disks will tend to service reads and slow disks will not, while writes are processed everywhere. Under heavy load, when all bricks can have many requests in flight, work will be distributed proportional to the bricks' relative speeds. The same balancing will occur when the brick population is dynamic due to failures or power-saving shutdowns.

3. THE D-SPTF PROTOCOL

The D-SPTF protocol supports read and write requests from a client to data that is replicated on multiple bricks. While a read can be serviced by any one replica, writes must be serviced by all replicas; this leads to different protocols for read and write.

The protocol tries to always process reads at the brick that can service them first, especially if some brick has the data in cache, and to perform writes so that they leave data in only one brick's cache. Bricks exchange messages with each other to decide which services a read.

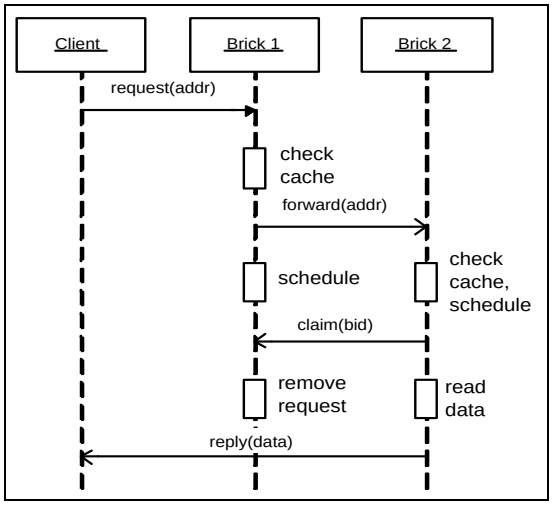

Figure 2: Read operation in D-SPTF. The client sends a read request to one brick, which forwards it to others. The brick that can schedule the read first aborts the read at other bricks and responds to the client.

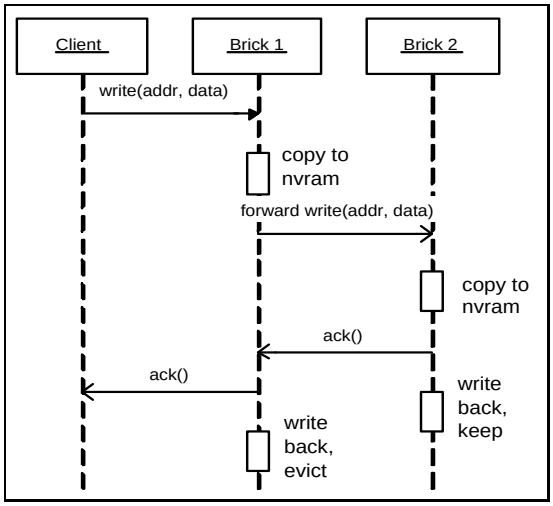

Figure 3: Write operation in D-SPTF. The client sends a write request to one brick, which forwards it to others. Each brick determines whether it is the one to keep the data in cache.

For reads (Figure 2), when a storage brick receives the request from a client, it first checks its own cache. If the read request hits in the brick's cache, then it is immediately returned to the client and no communication with other bricks is required. If not, then the brick places the request in its queue and forwards the request to all other bricks that have replicas of the data requested.

When a brick receives a forwarded read request, it also checks to see if the data is in its own cache. If the request hits in cache, then the brick immediately returns the data to the client and sends a CLAIM message to all other bricks with a replica so that they will not process the request. If the read request does not hit in cache, then the brick places the request in its own disk queue.

When it comes time to select a request for a disk to service, a brick scans its queue and selects the request that the disk can service with the shortest positioning time (i.e., each brick locally uses SPTF scheduling). If the request is a read, the brick then sends a CLAIM message to the other bricks with replicas so that they know to remove the request from their queues.

When a brick receives a CLAIM message, it scans its queue for the request and removes it. If a CLAIM message is delayed or lost, a request may be handled by more than one brick, which will have two effects. First, some resources will be wasted servicing the request twice—this should be very rare in the reliable, high-speed networks of brick-based systems. Second, the client will receive more than one reply—clients can simply squash such duplicate replies.

Once a request completes at a brick, that brick returns the data to the client. If a brick fails after claiming a read, but before returning the data to the client, the request will be lost. Clients timeout and retry if this occurs.

The write protocol (Figure 3) is different. When a brick receives a write request from a client, it immediately forwards the request to all the other bricks with a replica of the data. When a brick receives a write request, either directly or forwarded, the brick immediately stores the data in its local NVRAM cache. Bricks that receive a forwarded write request send an acknowledgment back to the first brick when the data is safely stored; the first brick waits until it has received acknowledgments from all other replicas, then sends an acknowledgment back to the client. If the original brick does not hear from all bricks quickly enough, some consistency protocol must address the potential brick failure. The basic D-SPTF protocol should mesh well with most consistency protocols (e.g., [1, 8, 10]).

At some point after the data is put in the NVRAM cache, it must be destaged to media by placing a write request in the brick's disk queue. Some time later, the brick's local SPTF head scheduler will write the data back to disk, after which all but one brick can remove the data from its cache. Bricks ensure exclusive caching by only keeping the block in cache if $hash(address) mod |replicas| = replica\ id$.

3.1 Concurrent CLAIM messages

One problem with the base protocol above is that, while one brick's CLAIM message is being transmitted across the network, another brick could select the same read and start servicing it. Both bricks would then waste disk head time and cache space. This would occur, in particular, any time the system is idle when a read request arrives.

D-SPTF avoids this problem by pre-scheduling and waiting for a short period (two times the one-way network latency) after sending the CLAIM message. Assuming an approximate bound on network latency, waiting ensures that a brick almost always sees any other brick's CLAIM message before servicing a request. If, during the wait period, the brick receives a CLAIM message from another brick, then only the one of those two that can service the request fastest should do so. To enable this decision, each CLAIM message includes a *service time bid* (the SPTF-predicted positioning time); with this information, each brick can decide for itself which one will service the request. Our current approach is for the request to be serviced by whichever brick submits the

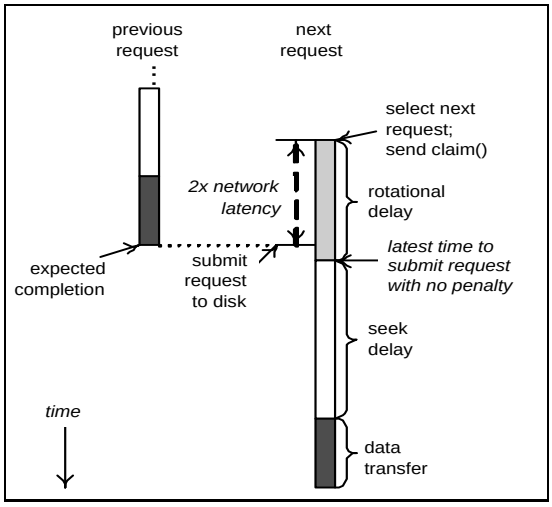

Figure 4: Overlapping CLAIM communication with rotational latency. Issuing the request to disk can be delayed up to the time when seek latency must begin to reach the target track before the intended data passes under the read/write head. The rotational latency gives a window for exchanging CLAIM messages with no penalty.

lowest service time bid, ignoring when CLAIM messages were sent. If two or more service time bids are identical, then the tie is broken by selecting the brick with the lowest brick id. This will work well for high-speed networks, but may induce disk head inefficiency when network latencies are significant fractions of mechanical positioning times.

With pre-scheduling, the wait period can usually be overlapped with the media access time of previous requests. That is, the system does not wait until one disk request completes to select the next one; instead, the system makes its selection and sends CLAIM messages a little more than the wait period before the current request is expected to complete. If the disk is idle when a request enters its queue, the brick will compute the expected seek and rotational latencies required to service that request, and will only wait for competing CLAIM messages as long as the expected rotational latency before issuing the request to disk. As illustrated in Figure 4, this does not impact performance, until network latency is a substantial fraction of rotational latency, since the brick is effectively shifting the rotational latency to before the seek instead of after.

4. EXPERIMENTAL SETUP

We use simulation to evaluate D-SPTF. The simulation is event driven, and uses the publicly-available DiskSim disk models [6] to simulate the disks within bricks. The DiskSim simulator accurately models many disks [3], including the Quantum Atlas 10K assumed in our system model.

We implemented four approaches: D-SPTF, LARD, decentralized hashing, and a centralized system for comparison. The models for the first three systems are similar. Each brick is modeled as a single disk, processor, and associated disk block cache, that is connected through a switched network to all bricks that share an overlapping set of replicas. The one-way network latency is a model parameter, set to 50 microseconds by default.

Each brick contains a request queue. Whenever the disk is about to become idle, the brick scans the request queue and selects the request with the shortest positioning time as the next request for the disk. The simulation model assumes an outside-the-disk SPTF implementation, which has been demonstrated as feasible [5, 18, 26], in order to allow the abort-from-queue capability needed for D-SPTF.

Each brick has its own cache, 512 MB in size unless otherwise stated. All simulated caches are non-volatile and use an LRU replacement policy.

The only difference between the D-SPTF system and the decentralized hashing system is how requests are routed. The D-SPTF implementation follows the protocol outlined in Section 3, with requests broadcast to all replicas and one replica claiming each read. In the decentralized hashing system, a read request is serviced by only one replica, determined by hashing the source LBN to get the brick's id. If a brick receives a read request for another brick, then it will forward the request to that brick without placing the request in its own queue. Hashing on the LBN provides both exclusive caching and long-term load balancing. However, since each replica does not see all requests for the replica set, it will have a reduced effective queue depth for SPTF scheduling; also, decentralized hashing does not adapt to short-term load imbalances.

Locality-Aware Request Distribution (LARD) [19] requires a central front-end to distribute requests among servers. Our implementation achieves this by routing all client requests for a given set of brick replicas to a single brick. When an address is initially requested, the lowest loaded replica is selected to service it. Every additional reference to that address would go to that replica. This optimizes cache hits and balances load in the common case. If the load on a replica exceeds a threshold then addresses would be moved to lower loaded replicas to decrease the load. Our implementation uses the thresholds and tuning values reported in [19].

The centralized system is designed differently from the decentralized systems. The centralized system contains one single cache with the same aggregate cache space as all the bricks in the decentralized systems. It also contains a single request queue that contains all requests. When some disk is about to complete a request, the system selects the next request for that disk. To represent an ideal centralized system, we modeled a centralized version of D-SPTF (via a single outside-the-disk SPTF across disks).

Current disk array controllers are not designed like the idealized centralized system against which we compare D-SPTF, though they could be. Instead, most keep a small number of requests pending at each disk and use a simple scheduling algorithm, such as C-LOOK, for requests not yet sent to a disk. Without a very large number of requests outstanding in the system, the performance of these systems degrades to that of FCFS scheduling. So, for example, the throughput of our idealized centralized system is up to 70% greater than such systems when there are 8 replicas.

5. EVALUATION

This section evaluates D-SPTF relative to a centralized ideal and two popular decentralized schemes. As expected, D-SPTF matches the former and outperforms the latter across many workloads and system setups, because it of-

	Throughput (IOs/sec)	Idle Time	Service Time (ms)
Central SPTF	188	0%	5.28
D-SPTF	187	0%	5.30
Hashing	133	6.5%	7.05
LARD	134	5.9%	7.04

Table 1: Throughput with homogeneous bricks and random workload. This table shows the per-brick throughput of an 8-brick system using each of the four schemes, given a random workload of 90% reads. D-SPTF and Central SPTF outperform Hashing and LARD, because of superior dynamic load balancing and disk scheduling.

fers all of cache exclusivity, dynamic load balancing, and effective disk head scheduling.

5.1 Systems of homogeneous storage bricks

To understand D-SPTF's base performance, we experiment with small-scale systems of identical storage bricks. First, we examine random workloads (i.e., those with no locality of reference) to focus on the load balancing and disk head scheduling aspects. Then, we use traces of four real environments to examine decentralized cache behavior.

5.1.1 Random workloads

Table 1 shows the performance of the four schemes for a synthetic random workload on an 8-replica system of storage bricks. The workload is a closed loop with 16 IOs outstanding at any time, with 90% reads, request sizes drawn from an exponential distribution with mean 4KB, and no locality (all data blocks are equally likely). The table shows per-brick throughput, the average percent of time that each brick is idle (due to transient load imbalances), and the average service time for cache misses.

Two primary insights can be gained from Table 1. First, D-SPTF provides almost the same performance as the centralized ideal, which is the end goal. Second, D-SPTF provides 40% higher throughput than either decentralized hashing or LARD, which provide similar performance. D-SPTF's advantage has two sources: dynamic load balancing and superior disk head scheduling.

Dynamic load balancing. Because all bricks see all queued requests for replicas they hold, a brick will never be idle while another brick has queued requests that it could service. With hashing and LARD, requests are routed to a specific replica-holder as they arrive, and so short-term load imbalances are not uncommon. Even with our relatively heavy synthetic workload, roughly 6% of each brick's time is idle even though the 8-brick system always has 16 requests pending. As expected, the long-term average balance of work across bricks is perfect for all of the schemes. LARD will adapt to heterogeneous server rates, as discussed in Section 5.2, but does not eliminate these transient imbalances.

This transient imbalance effect varies with both load and read:write ratio. For example, for a read-only closed workload of 16 requests at a time, LARD and hashing leave each brick idle 15% of the time. Recall that writes go to every brick with a replica, which can hide much of the potential transient idleness. At 60% reads, the idle percentage drops to 2%. With lower loads, transient imbalances are more evident because there are fewer requests being semi-randomly assigned to bricks at once. With higher loads, transient imbalances result in less idle time, because there are more chances for each brick to be assigned a pending request.

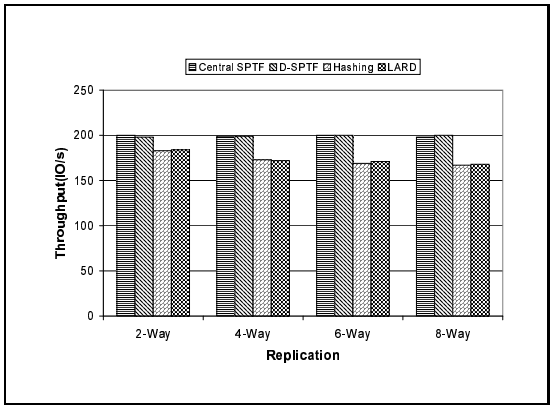

Figure 5: Scheduling Efficiency as a function of replication. This graph shows the per-brick throughput of each algorithm as a funtion of the number of replicas. A random workload of 67% reads was used with 16 requests per replica group.

Disk head scheduling. D-SPTF enables superior disk head scheduling, because it allows each brick's scheduler to consider more pending requests. Specifically, hashing and LARD partition requests as they arrive (i.e., before they are queued), whereas D-SPTF queues cache misses at all replicas and partitions them as they are selected for service. As described in Section 2.1, disk scheduling is much more effective when given more options. For the experiment reported in Table 1, the average queue length when scheduling occurs is approximately nine, as compared to approximately two for hashing and LARD.

More generally, D-SPTF allows a brick to select among all pending read requests that it could service that are not yet already being serviced by another brick. So, for read-only workloads, the average queue depth considered will be approximately the number of pending requests *minus* the number of replicas. For hashing and LARD, read requests are divided evenly among the bricks with replicas, so the average queue length of reads considered will be the number of pending read requests *divided by* the number of replicas.

Figure 5 illustrates the effect of the number of replicas on the disk scheduling benefit. The same synthetic random workload is used, but with a read:write ratio of 2:1 and a range (from 2–8) of numbers of replicas per data block. The number of requests in the system is 16 in all cases.

With basic mirroring (two copies), D-SPTF provides 9% higher throughput than hashing and LARD. As the number of replicas increases, D-SPTF's margin increases. At eight replicas, D-SPTF provides 20% higher throughput. In every case, the media performance of D-SPTF is close to that of the centralized ideal.

The 8-replica improvements are lower than the values from Table 1 because of the difference in read/write ratio. Recall that writes go to all bricks with replicas in all schemes. With 90% reads, each brick will average 1.6 writes and ei-

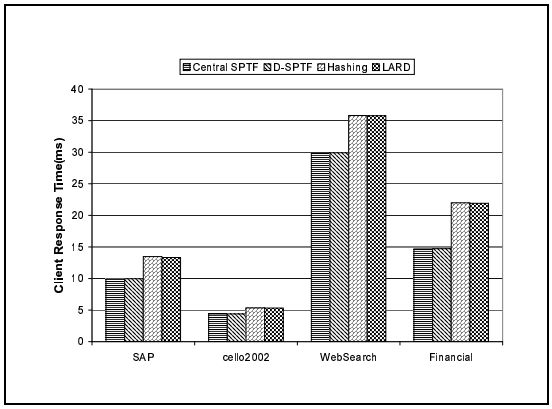

Figure 6: Response times for four traces. This graph shows the response times of the four traces on a two-replica system with various request distribution schemes.

	SAP	Cello2002	WebSearch	Financial
R/W ratio	62%	68%	90%	85%
Size	10KB	11.5KB	8.2K	9K
Requests	4512361	5248101	4579809	5334987
Duration	15hrs	24hrs	4hrs	12hrs

Table 2: Trace characteristics.

	SAP	Cello2002	WebSearch	Financial
Central SPTF	74%	76%	68%	71%
D-SPTF	73%	72%	67%	69%
Hashing	72%	71%	66%	68%
LARD	72%	71%	66%	68%

Table 3: Cache hit rates. This table shows the cache hit rates for the different configurations.

ther 7.4 reads (for D-SPTF) or 1.8 reads (for hashing or LARD) pending; with one request being serviced, the queue depth comparison is 8 for D-SPTF verses 2.4 for hashing and LARD. With 67% reads, the comparison is 8 versus approximately 5.6. The throughput values in Figure 1 for different queue depths match those observed in Table 5, for all of the different schemes and numbers of replicas.

For real workloads, we expect to see similar performance gains for D-SPTF. To verify this expectation, we use four traces (Table 2). The *SAP* trace [11] captures the I/O activity of a SAP-based system running an Oracle DB supporting 3000+ clients accessing billing records. The *Cello2002* trace [11] captures the I/O activity from the software development activities of a 20-person research lab. The *WebSearch* trace [23] captures the performance of a system processing web search queries. The *Financial* trace [23] captures the performance of a system running a financial transaction processing system. All traces were run for the duration of each trace.

Figure 6 shows the results for two-way replication. D-SPTF matches Central SPTF, and both outperform Hashing and LARD. D-SPTF provides an 11% reduction in client side response time for the SAP trace which is in line with the synthetic workloads running 2-way replication. There is no increase in the throughput of the system, because the simulated system is powerful enough to complete all trace requests within the trace time frame.

Among these traces, the SAP and the Financial traces have the greatest locality. This results in D-SPTF's performance improvement being between 35-50% over hashing an LARD compared to 20% for the other traces. Cello2002 also has good locality but has very low load reducing it's performance impact. Greater locality increases the importance of rotational latencies relative to seek times, increasing the value of SPTF scheduling. Since D-SPTF provides higher effective queue depths, it is able to reduce disk service times further when requests are localized.

5.1.2 Caching effects

To explore the caching implications of the different schemes under workloads with locality, we again use the traces. Recall that each brick has its own cache, and the goal in decentralized brick-based storage is exclusive caching: if each brick's cache holds different data, the aggregate cache space is the same as a centralized cache of the same total size. Combined with semi-random distribution of cache misses across bricks, this should result in behavior much like a global LRU. All three decentralized schemes provide both properties, so we expect them to perform similarly to one another and to the centralized ideal.

Table 3 shows the overall cache hit rates for the four traces applied to an 8-replica system. The system contain 50 LUNs and 16GB worth of cache. Centralized caching always achieves the best hit percentage, and the three decentralized schemes have hit rates that are 2–4% lower. This matches the expectations.

5.2 Systems of heterogeneous storage bricks

D-SPTF excels with heterogeneous collections of storage bricks as well as homogeneous collections. Since storage bricks are designed to be incrementally added to the system, an IT staff does not have to deploy a monolithic storage system. Instead the designer can select the storage system that is needed to meet current demands. If the demands increase later in the life of the system, then instead of replacing the entire storage system, one can add new bricks to the system to increase the capacity, performance, and reliability as needed. However, one challenge introduced by this incremental scaling is that there will be bricks of different performance in these systems. Having heterogeneous bricks increases the difficulty of achieving centralized performance. This section shows that D-SPTF is able to adapt to the heterogeneity more effectively than existing request distribution schemes.

5.2.1 Static heterogeneity

Static heterogeneity is the simplest case; for example, consider a two brick system that contains one "fast" device and one "slow" device. One could modify a hashing-based scheme so that, if the slow brick was 50% the speed of the fast brick, the weighted hash function would send 2/3 of the requests to the fast disk and 1/3 to the slow disk. LARD and D-SPTF handle static heterogeneity via load-based request distribution.

	Central SPTF	D-SPTF	Hashing	LARD	Weighted Hash
Fast	195	194	137	182	180
Slow	138	137	137	134	135

Table 4: Throughput balance between a fast and a slow replica. The disk in the slow replica is half the speed of the disk in the fast replica. D-SPTF and a centralized system exploit the full performance of the fast disk, while random hashing paces the fast disk to match the slow disk.

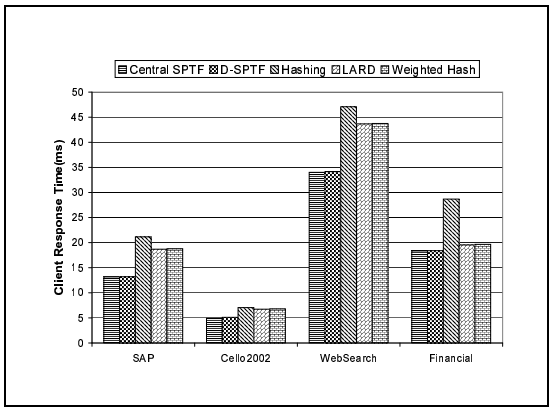

Figure 7: Impacts of static heterogenous devices on trace workloads. This graph shows the client response time of four trace workloads with staticly heterogenous workloads.

Table 4 shows the "fast" and "slow" brick's throughput results under the same synthetic random workload as in 5.1.1. Both D-SPTF and the centralized ideal have the desired load balancing property: as the slow disk gets slower, the throughput of fast brick remains nearly unaffected.

Hashing, however, does not do so well: as the slow disk gets slower, it drags down the performance of the fast disk to match. The load between the two disks quickly becomes unbalanced because hashing assumes that both devices are of equivalent speed and thus sends half the requests to each disk. As a result, the rate at which requests are sent to the fast disk is governed by the rate at which the slow disk can service its requests, and the performance advantage of the fast disk is wasted.

Both LARD and a weighted hash function do a better job of adapting to heterogeneous bricks than standard hashing. This is because weighted hashing accounts for the relative performance of the devices. LARD functions well, also, because it initially allocates requests based upon the load at the device, which will function similarly to weighted hashing. As shown in the next section, both LARD and the weighted hash function have difficulty with adapting to device performance changes.

Figure 7 shows how the slow/fast two-replica systems perform for the traces. D-SPTF is close to Central SPTF and outperforms the other schemes. LARD and weighted hashing are able to adapt to the load balancing characteristics but provide inferior disk scheduling. The Financial trace proves the exception to this rule because LARD and weighted hashing provide similar performance. This is due to relatively high queue depths at the devices which minimize the scheduling advantage of D-SPTF. In general, unmodified hashing performs worst, since it both lacks the ability to properly balance the load and has less efficient scheduling.

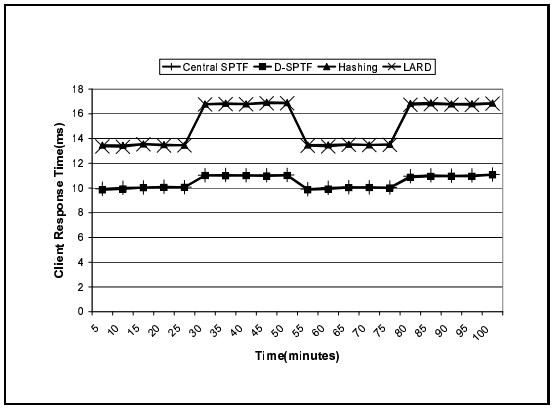

Figure 8: Performance in face of transient performance breakdowns. This graph shows the performance of each scheme in a two-replica system where one replica's performance varies, halving at time 25, correcting at time 50, and halving again at time 75. D-SPTF and Central SPTF adapt best to the transient degradations. Hashing and LARD do not adapt well to this dynamic heterogenous behavior. For LARD, the transient load imbalance is not long enough to trigger a rebalance action.

5.2.2 *Dynamic heterogeneity*

Dynamic heterogeneity can exist even when bricks are homogeneous in construction. For example, it can occur when one brick is performing integrity checks, reorganizing its on-disk layout, or coping with an component failure. To examine the impact of D-SPTF in such situations, we dynamicly varied the performance of a two-replica system. The base system has two identical replicas. The performance of one replica was altered as follows. Initially each replica was operating at full speed,[3] after one quarter of the run had passed, the first replica experienced a 50% degradation of performance. Once half of the time had passed, the degradation ceased and both replicas were again at equal performance. Three quarters of the way through the experiment, the second replica suffers 50% degradation and remains that way until the end of the experiment. The workload that was used to evaluate the system was the SAP trace. To compared the effectiveness of D-SPTF to other schemes, we measured the client side response time at specific time intervals.

Figure 8 shows the results. The average response time of both D-SPTF and Centralized SPTF in non-degraded mode is around 7.8ms per request, while the response time of both hashing and LARD are around 9ms. This 15% difference in

[3]Prior to the experiment, LARD was warmed up long enough to be in steady state where all addresses had been previously allocated, 1/2 to each replica, since they are homogeneous.

response time is due to the improved scheduling efficiency of D-SPTF and central SPTF. All systems are able to effectively balance load in non-degraded mode.

Once the performance of one replica is degraded, however, we see the performance disparity increase. D-SPTF/Central SPTF's response times rise to 8.6ms while Hashing/LARD's response times increase to 11.2ms. This 30% difference occurs because, when the load imbalance occurs, the performance of the LARD and Hashing systems are limited by the slowest replica in the set. Neither adapts to the short-term degradation.

LARD does not adapt to it because the degradation does not continue long enough to trigger rebalancing of the system given the default parameters. With a shorter rebalance window, LARD does adapt but still suffers for one window after the degradation starts and a second after it ends. Also, with short windows, the cache locality degrades. An advantage of D-SPTF is that it responds quickly to changes in replica load and balances the load automatically without requiring external tuning of scheme parameters.

5.3 Communication Costs

D-SPTF provides brick-based storage systems with comparable performance to an ideal centralized SPTF. However, it does so by relying on extra communication bandwidth and low-latency messaging. This section quantifies the extra bandwidth required and D-SPTF's sensitivity to network latency.

5.3.1 Protocol message overheads

D-SPTF involves sending every read request and every CLAIM message to each replica holder, which can result in many more control messages than for hashing-based request distribution schemes. The number of messages sent per request depends on the level of activity in the system. The common case occurs when only one brick attempts to service a request, after completing its last disk request. In this case, the total number of messages will be $2N$, where N is the number of replicas in the system. The maximum number of messages would be sent when all devices are idle when a new request arrives, causing them all to decide to try to service the new request. In this case, N^2 CLAIM messages per request will move through the system.

To validate these expectations, we use the synthetic workload with 16 requests kept outstanding in each replica group. Table 5 shows the number of messages per request for different numbers of bricks and different degrees of replication. As expected, the number of messages per request is very close to $2N$. The message count is unaffected by the number of bricks, since only replica holders for a particular block are involved in communication regarding requests on that block.

The results are not exactly $2N$ because some requests find multiple bricks idle when they arrive. In this case, the idle bricks each attempt to CLAIM the request, generating N messages. This happens rarely in systems that have non-light workloads. For additional insight, Table 6 shows the percentage of such conflicts as a function of the number of replicas, with the workload held steady at 16 requests per replica set. The percentage increases as the number of replicas increase because, with a lower ratio of requests to replicas, it is more likely that two or more of bricks will attempt to service the same request at the same time. Even at with 16 requests for eight bricks, fewer than 2.5% of all requests experience such conflict, and the result is a minimal increase in the number of messages sent.

	2-Way	4-Way	6-Way	8-Way
16 bricks	4.003	8.02	12.07	16.15
32 bricks	4.005	8.01	12.09	16.20
64 bricks	4.007	8.03	12.05	16.17
128 bricks	4.001	8.05	12.08	16.13

Table 5: Number of messages sent per request. The number of messages, in the common case, is two times the number of replicas. The number of messages is not dependent on the number of bricks in the system. There are slightly more than $2N$ messages because of situations when multiple bricks attempt to schedule the same request.

	2-Way	4-Way	6-Way	8-Way
Conflicts	0.15%	0.67%	1.44%	2.44%

Table 6: Number of conflicts per request versus replication level. This table shows the percentage of requests that multiple bricks attempt to claim. As one can see, as the replication level increases, the number of such conflicts increase, because the per-brick workload drops. (Sixteen requests are pending at a time for each configuration.)

In the eight-replica case, the number of D-SPTF messages sent per second per brick was 3230. With a non-data message size of 50 bytes, this results in 160KB/s for messages. If every request were for only 4KB of data, the data communication would consume 800KB/s and the total bandwidth consumed per brick would be 960KB/s. In other words, in this case, the D-SPTF protocol adds approximately 20% more traffic to the network (independent of the number of actual bricks in the system) even with very small requests. For larger requests, of course, the overhead will be lower. For example, with an average request size of 64KB, the added bandwidth would be only 1.2%. The network must support this increase in traffic, without significant performance degradation, in order to effectively support D-SPTF.

5.3.2 Sensitivity to network latency

The D-SPTF protocol relies on relatively low latency messaging to function well. Recall that D-SPTF pre-schedules and overlaps CLAIM communication with disk head positioning time. Doing so hides network latencies that are smaller than rotational latencies.

Figure 9 evaluates the effect of network latency. This experiment uses the 8-replica system and workload from Section 5.1.1, but varies the one-way network communication latency from 0 ms to 3 ms. The results show that the D-SPTF protocol has effectively the same performance as the centralized ideal up to about 1 ms one-way network latency. Even at 1.75 ms network latency, D-SPTF shows less than 5% decrease in throughput. For context, FibreChannel networks have 2–140μs latencies [21], depending on load, and Ethernet-based solutions can provide similar latencies. Thus, D-SPTF should function at full efficiency in storage clusters and many other configurations.

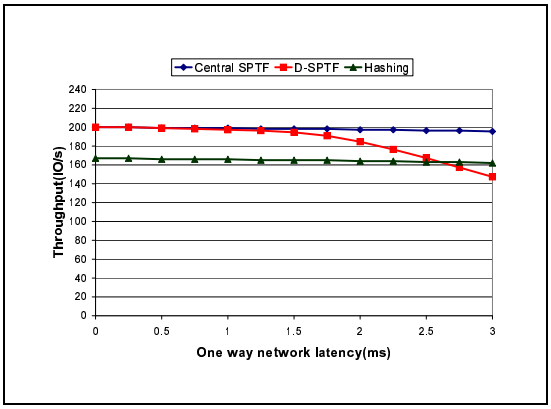

Figure 9: Effect of network latency on D-SPTF throughput. Throughput is shown per replica. The one-way network latency is varied from 0 ms—instantaneous communication—to 3 ms.

D-SPTF performance drops off for very slow interconnects because of the wait period for CLAIM message propagation. Recall that every request is scheduled two one-way network latencies before it is issued. So long as the current request does not complete in less time than two network latencies, no performance is lost. If a request's media time is less than two network latencies, then the system will wait and the disk will go idle until two network latencies have passed. As network latency grows and more requests complete in less than two network latencies, the disks are forced to wait and performance drops.

6. ADDITIONAL RELATED WORK

Several closely related works have been discussed in the context of the paper. This section discusses additional related work.

Several groups have explored the centralized multi-replica SPTF approach, labelled "Central SPTF" in our evaluations, to which we compare D-SPTF. For example, Lo [17] proposed and explored Ivy, a system for exploiting replicas by routing requests to the disk whose head is closest to a desired replica. Wilkes et al. [25] explored a similar approach, but routing requests based on predicted positioning time. Most recently, Yu et al. [26] described an approach similar in spirit to D-SPTF across mirrored local disks, for use in evaluating their SR-Array system. D-SPTF builds on this prior work by bringing its benefits to a decentralized context and simultaneously achieving effective exclusive caching and load balancing.

Several groups have explored explicitly cooperative caching among decentralized systems [4, 24]. These systems introduce substantial bookkeeping and communication that are not necessary if requests are restricted to being serviced by their data's homes. However, these techniques could be used to enhance load balancing and memory usage beyond the confines of a scheme like D-SPTF, which focuses on the assigned replica sites for each data block.

Striping and hashing are popular techniques for load balancing. More dynamic schemes that migrate or rebalance load based on feedback are popular for activities like process executions. With a front-end distributing requests across a set of storage servers, feedback-based load distribution works well [2, 12, 16]. Clusters of web servers often use a load-balancing front-end to distribute client requests across the back-end workers. For example, LARD [19] provides such load balancing while maintaining locality.

7. CONCLUSIONS

D-SPTF distributes requests across heterogeneous storage bricks, with no central point of control, so as to provide good disk head scheduling, cache utilization, and dynamic load balancing. It does so by exploiting high-speed communication to loosely coordinate local decisions towards good global behavior. Specifically, D-SPTF provides all replicas with all possible read requests and allows each replica to schedule locally. Limited communication is used to prevent duplication of work. Overall, given expected communication latencies (i.e., 10–200μs roundtrip), D-SPTF matches the performance of an idealized centralized system (assuming equivalent aggregate resources) and exceeds the performance of LARD and decentralized hash-based schemes.

Acknowledgements

This work is partially funded by the National Science Foundation, via grant #CCR-0205544. We thank the members and companies of the PDL Consortium (including EMC, Engenio, Hewlett-Packard, HGST, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support.

8. REFERENCES

[1] Khalil Amiri, Garth A. Gibson, and Richard Golding. Highly concurrent shared storage. *International Conference on Distributed Computing Systems* (Taipei, Taiwan, 10–13 April 2000), pages 298–307. IEEE Computer Society, 2000.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Jospeh M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: making the fast case common. *Workshop on Input/Output in Parallel and Distributed Systems* (Atlanta, GA, May, 1999), pages 10–22. ACM Press, 1999.

[3] John S. Bucy and Gregory R. Ganger. *The DiskSim simulation environment version 3.0 reference manual.* Technical Report CMU–CS–03–102. Department of Computer Science Carnegie-Mellon University, Pittsburgh, PA, January 2003.

[4] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: using remote client memory to improve file system performance. *Symposium on Operating Systems Design and Implementation* (Monterey, CA, 14–17 November 1994), pages 267–280. IEEE, 1994.

[5] Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. Design and implementation of semi-preemptible IO. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 145–158. USENIX Association, 2003.

[6] The DiskSim Simulation Environment (Version 3.0). http://www.pdl.cmu.edu/DiskSim/index.html.

[7] EqualLogic Inc. PeerStorage Overview, 2003. http://www.equallogic.com/pages/products_technology.htm.

[8] Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. FAB: enterprise storage systems on a shoestring. *Hot Topics in Operating Systems* (Lihue, HI, 18–21 May 2003), pages 133–138. USENIX Association, 2003.

[9] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU-CS-03-178. Carnegie Mellon University, August 2003.

[10] Jim N. Gray. Notes on data base operating systems. In , volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[11] HP Labs SSP traces. http://www.hpl.hp.com/research/ssp.

[12] Hui I. Hsiao and David J. DeWitt. A performance study of three high availability data replication strategies. *Parallel and Distributed Information Systems International Conference* (Miami Beach, FL, 04–06 December 1991), pages 18–28. IEEE, 1991.

[13] Hui-IHsiao and David J. DeWitt. Chained declustering: a new availability strategy for multiprocessor database machines. *International Conference on Data Engineering* (Los Angeles, CA), 1990.

[14] IBM Almaden Research Center. Collective Intelligent Bricks, August, 2003. http://www.almaden.ibm.com/-StorageSystems/autonomic_storage/CIB/index.shtml.

[15] David M. Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position*. Technical report HPL–CSP–91–7. Hewlett-Packard Laboratories, Palo Alto, CA, 24 February 1991, revised 1 March 1991.

[16] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[17] Sai-Lai Lo. *Ivy: a study on replicating data for performance improvment*. TR HPL-CSP-90-48. Hewlett Packard, December 1990.

[18] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 275–288. USENIX Association, 2002.

[19] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):205–216, November 1998.

[20] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD International Conference on Management of Data* (Chicago, IL), pages 109–116, 1–3 June 1988.

[21] Steven Schuchart. High on Fibre. *Network Computing*, 1 December 2002.

[22] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC, 22–26 January 1990), pages 313–323, 1990.

[23] Storage Performance Council traces. http://traces.cs.umass.edu/storage/.

[24] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Madison, WI). Published as *Performance Evaluation Review*, **26**(1):33–43, June 1998.

[25] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.

[26] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading capacity for performance in a disk array. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 243–258. USENIX Association, 2000.