D-SPTF: Decentralized request distribution in brick-based storage systems

CHRISTOPHER R. LUMB

December 7, 2005
CMU-PDL-05-111

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis committee

Prof. Gregory R. Ganger, Chair
Dr. Richard Golding, IBM Almaden Research
Prof. Babak Falsafi
Prof. Mor Harchol-Balter

Keywords: Decentralized Storage, storage systems, array scheduling, scalablity

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. This central control allows for good cache performance, load balancing and scheduling efficiency. However, many now envision building storage systems out of collections of federated "bricks" connected by high-performance networks. The goal of brick-based storage is a system that has incremental scalability, parallel data transfer, and low cost. However, with bricks, there is no centralized point of control to provide request distribution. This lack of central control makes achieving good scheduling efficiency, load balancing and cache performance a challenge in decentralized brick-based storage.

Distributed Shortest-Positioning Time First (D-SPTF) is a decentralized request distribution protocol designed to address this challenge. 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 aggregate cache capacity, and reduces positioning times for cache misses. For network latencies of up to 0.5ms, D-SPTF performs as well as would a hypothetical centralized system with the same collection of CPU, cache, and disk resources. Compared to a popular decentralized approach, hash-based request distribution, D-SPTF achieves up to 65% higher throughput and adapts more cleanly to heterogenous server capabilities.

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.

Contents

Figures

Tables

1 Introduction

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.

This centralized control and associated global knowledge of the system allow for a number of optimizations. One of these optimizations is centralized caching, which pools cache resources to achieve good cache hit ratios. Centralized caching is enabled by having requests arrive at a centralized point of control and checked against a single cache for cache hits. Additionally, centralized control and global knowledge of the system also allow for the optimal replica to be selected during a cache miss. Selecting the optimal replica for a request allows the system to maximize load balancing and device scheduling efficiency. These decisions are made by this central control and require no coordination or communication by the underlying devices.

However, many envision enterprise-class storage systems composed of net-

worked "intelligent" *storage bricks* [equallogic03; Frolund03; Ganger03a; CIB03]. Each brick consists of a few disks, RAM for caching, and a 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 (ala the RAID arguments [Patterson88]), 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. If data is replicated in multiple brick caches, then the total available cache space is effectively reduced and lower cache hit ratios are possible. During bursts of work, when queues form, requests should be spread across bricks so as to avoid inappropriate idleness and, ideally, to reduce disk positioning cost [Denning67; Worthington95; Ruemmler94].

Achieving these goals is further complicated when heterogeneous collections

of bricks comprise the system. Traditional disk array systems do not face these challenges since all of these features can be provided by the central disk array controller. Prior protocols for decentralized storage systems can provide a subset of these attributes, but no previous protocol provides all of them.

1.1 Thesis statement

With reasonable communication costs, a decentralized storage protocol can utilize storage resources (spindles, cache and capacity) nearly as efficiently as a centralized storage system.

D-SPTF is a request distribution protocol for brick-based storage systems that keep two or more copies of data. The goal of D-SPTF is to bring the advantages of centralized disk array systems to brick-based storage systems. 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 [Baker92; Ruemmler93], 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 by the one brick that has the READ data in cache. Cache misses, however, go into

all relevant local disk 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 both long-term and short-term load imbalances. 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) [Jacobson91; Seltzer90b]. Choosing from a larger set of options significantly increases the effectiveness of these algorithms, decreasing positioning delays and increasing throughput.

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.

1.2 Validation

To validate the thesis statement, this dissertation describes an evaluation of the D-SPTF protocol. The design tradeoffs, including conflict resolution schemes, were evaluated and experiments designed to show the following:

(1) D-SPTF is able to load balance as well as an idealized centralized system and as well or better than existing decentralized approaches.

(2) D-SPTF is able to achieve disk scheduling efficiency as well as an idealized centralized system and as well or better than existing decentralized approaches.

(3) D-SPTF is able to maintain cache performance of existing decentralized systems and nearly that of an idealized centralized system.

(4) D-SPTF is able to adapt to heterogenous systems as well as an idealized centralized system and better than existing decentralized approaches.

(5) The communication costs and latency tolerance of D-SPTF are reasonable for brick-based environments.

This dissertation describes and evaluates D-SPTF via simulation, comparing it to a centralized ideal and popular existing decentralized algorithms, such as LARD [Pai98] and hash-based request distribution. D-SPTF is as good or better than these algorithms 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 by balancing the load to utilize each of the bricks optimally.

2 Existing systems

Current storage systems are going through a transition from centralized architectures to more decentralized and distributed architectures. This transition is being made to take advantage of the scalability and cost advantage of decentralized designs, while trying to maintain the performance of existing centralized systems.

2.1 Centralized 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.

This centralized control and associated global knowledge of the system allow for a number of optimizations. With centralized control, requests are routed through a central controller that decides which replica will service a request. Decisions are made solely by this central control and require no coordination by or communication between the underlying devices.

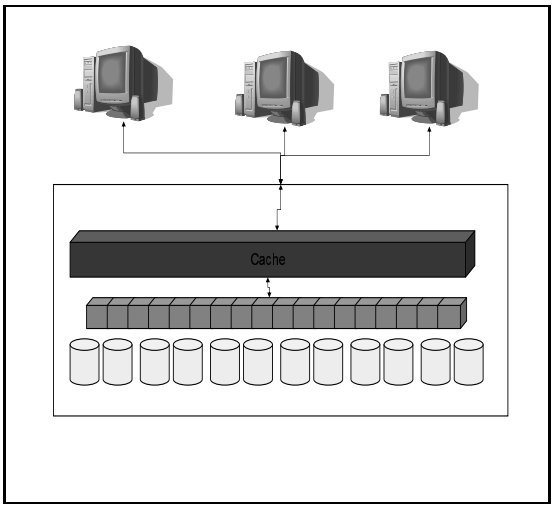

Fig. 2.1: **Picture of a centralized storage system.** A simple picture of a centralized storage system. Centralized storage systems can be thought of as a single cache, single scheduling queue, and a large number or devices to service requests from media.

A typical centralized system is illustrated in Figure 2.1. Request flow is straightforward. When any client decides to issue a read request, it first sends the request to the central storage controller. Once a read request arrives at the central controller, the controller will check its cache. If the requested data is in the cache, the controller will complete it immediately. If the request is not in the cache, then the controller will place the request in its disk queue and wait for a scheduling opportunity. Whenever a disk needs a new request, the controller looks in its queue and selects a request for the device. Since it knows all the requests that any device can service, the controller is able to select the optimal request for a device. When a read request completes at the device, the controller places the data in its cache and sends it to the client.

For a write request, the protocol is similar. When a write request arrives, it

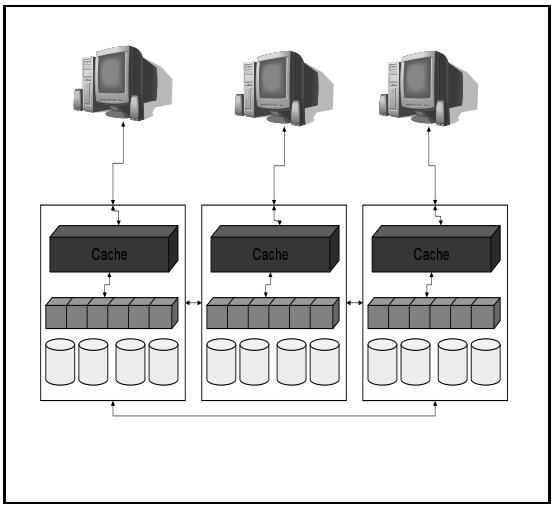

Fig. 2.2: **Picture of a decentralized storage system.** A simple picture of a decentralized storage system. A decentralized storage system is composed of a number of storage bricks. Each storage brick has its own cache, its own queue and a small number of commodity devices to service media requests. Brick are connected to each other through high speed networks.

is first cached in the controller's NVRAM cache and completion is reported back to the client. Only one copy is kept in cache and remains there until all replicas of the data have been written to disk. Once the write has been written back, the cache copy may be evicted or it may remain in cache, depending upon the caching policy.

Existing centralized systems have an advantage over decentralized systems. Their centralized control includes both centralized queueing of requests and centralized caching of requests. With centralized queueing, all requests are placed into a single central queue. When a lower level device in the system needs a new request, the central queue selects one from its queue for that device to service. Since all requests are stored in a central queue, load imbalances between devices with replicas of the same data do not occur. Whenever an replica needs work, a

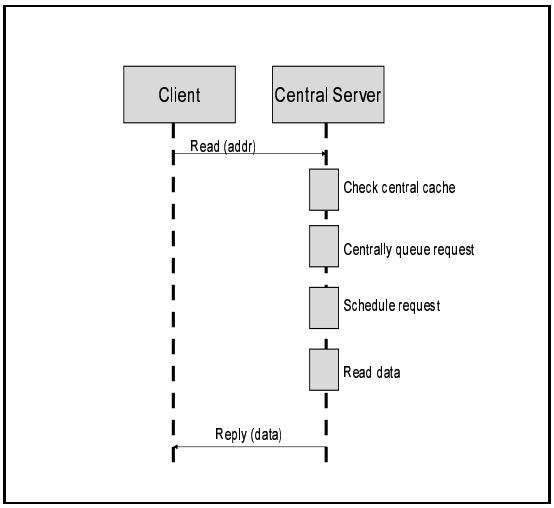

Fig. 2.3: **Operation of a centralized storage system.** To service a request in a centralized storage system, the client first sends a request to the centralized server. The centralized server first checks its centralized cache. If the request is not in cache then the centralized server will place it in its centralized queue. When it is time to send a request to media, the centralized server will scan its centralized queue and select the best request. That request is then sent to one of the storage devices who completes the request. Once the request is complete the centralized server informs the client.

request is removed from the central queue and sent to the replica. Only if there is no work in the central queue, for any data stored by a device, does that device go idle.

Centralized control also allows for efficient cache decisions. With centralized systems there is only one cache for all data in the system. If an underlying storage device services a request, the data will be stored once in the cache, regardless of which replica in a group services the request. This centralized cache makes efficient use of all space and prevents wasted cache space.

In addition to centralized control, these systems provide a central location to store knowledge about the system. This knowledge includes the queue sizes at the

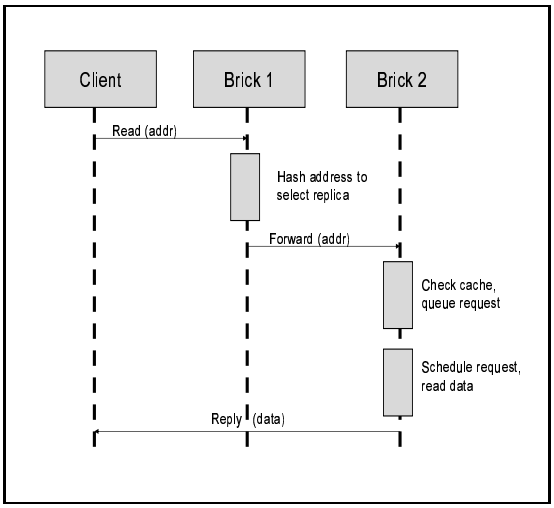

Fig. 2.4: **Operation of a hashed decentralized system.** The operation of a hashed decentralized system, begins when a client sends a request to any of the bricks in the system. The first step that a brick takes when it receives a request is to hash on the LBN of the request to determine which brick will service that request. The request is then forwarded to the brick who is supposed to service it. When it arrives at that brick, the brick first checks its cache. If the request is not in cache then the brick will place it in its queue. Once it is time to send a request to media, the brick will check its queue and select the best request. That request is sent to one of the devices who completes the request. Once the request is complete, the brick that serviced the request will complete the request to the client.

underlying storage devices, the locations of all replicas, and the location of data on each disk. By having such complete state, the controller is able to make accurate decisions about which storage devices should service which requests. They have the capability both to load balance requests and to optimally schedule these requests. With Shortest Positioning Time First (SPTF) algorithms, the ability to schedule using large centralized queues results in improved disk access times relative to systems that use smaller partitioned queues.

While large scale centralized storage systems have a number of advantages, they are not without their disadvantages. Large centralized systems can run into scalability limitations. Additionally, these systems can be very expensive to de-

velop and require one to purchase many years of storage at once rather than incrementally a needs evolve. As a result, decentralized brick-based systems are starting to emerge as viable alternatives to large monolithic centralized storage.

2.2 Decentralized systems

Many now suggest 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. It is expected that brick-based storage will exhibit a significant cost benefit due to brick-based storage using large numbers of less expensive commodity components rather than a few higher-performance but custom components.

Brick-based systems are different from larger centralized systems in several ways: bricks are small, have moderate performance, and often are not internally redundant. Each brick generally consist of a CPU, modest amount of cache, and a few to a few tens of commodity disks. The moderate performance and size of bricks 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 one method. These brick-based systems are designed with incremental growth in mind. As a result, over time, a storage system will tend to include many different models of bricks. These brick-

based storage systems must be able to adapt to heterogenous collections of bricks automatically.

The challenge of a brick-based storage system is to achieve the advantages of the centralized system, including scheduling efficiency, load balancing and cache exclusivity, without significantly increasing the costs of the system.

One example of the operation of a based brick system that uses hashing for request distribution is shown in Figure 2.4. A client can potentially send a request to any brick in the system, since there is no central request aggregation point. When a read request arrives at a brick, the request address(the data's block number) is hashed and the request is sent to the replica that matches the hash to service it. Once the request arrives at that replica brick, the replica's cache is checked. If the request is in cache, then it is completed immediately. If it is not in cache, then the request is placed in the replica brick's queue. When a device in a brick needs a request to service, the brick scans its request queue and selects the best request for that device. When a read request completes at a device, the data is cached in the brick and then delivered to the client.

Write requests in brick-based storage are slightly different. When a write arrives in the system, it must be written to all replica bricks. Once it arrives at each brick, it is placed in the replica brick's NVRAM cache. Once all replica bricks have cached it, then completion is reported to the client. However, there is now a copy of the data in each replica brick's cache, unlike the single copy that exists with centralized storage systems. This can cause a reduction in effective cache size, which can lead

to lower cache hit ratios.

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. Three crucial aspects are made more difficult by the lack of global information: disk head scheduling, cache utilization, and inter-brick load balancing. Existing mechanisms can address one or two of these aspects but do so at the expense of the others. Each of these aspects is complicated when the population of bricks is heterogenous. The D-SPTF protocol addresses all three aspects by exploiting the high-speed networks expected in brick-based systems, allowing bricks to loosely coordinate their local decisions and adapt well to heterogenous brick capabilities.

2.2.1 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 [Jacobson91;

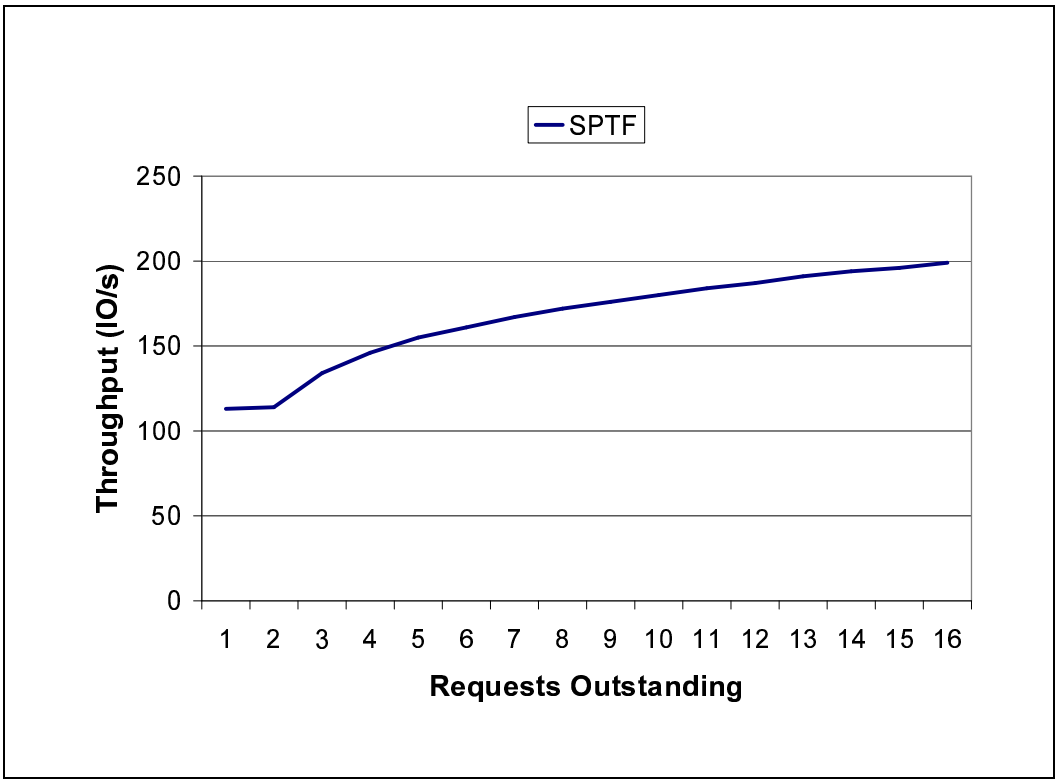

Fig. 2.5: **Disk throughput with SPTF scheduling, as a function of queue depth.** The data shown are for a closed synthetic workload (see Chapter 6) with a constant number of pending small requests to random locations on a Quantum Atlas 10K disk.

Seltzer90b] 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 (that is, 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 2.5 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 generally 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, which results in higher media response times. Maintaining low media access time requires increasing the queue depths at bricks making scheduling decisions. One way to do this is to direct requests to only a subset of the 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 would be the same as a single brick. The centralized approach to this problem is simple: by maintaining all requests in one centralized queue, the maximum scheduling queue depth is obtained and by having global knowledge of the devices in the system, the centralized system is able to select the optimal request for an individual device from the global queue. D-SPTF is able to achieve disk scheduling results similar to a centralized system. D-SPTF's approach enables it to increase the scheduling queue depth of the system without causing duplicate work. This allows D-SPTF to achieve low media access times while utilizing all devices.

2.2.2 Load balancing

In addition to disk head scheduling, balancing the load among all bricks to prevent idle resources is a significant challenge. 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 [Wilkes96]. This allows the centralized system to balance the load in all conditions, preventing both transient and long-term load imbalances.

There are many simplistic ways to spread requests across bricks to achieve long-term load balancing. 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 [Hsiao90; Hsiao91]. 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 among the bricks gives balance only over the long term, but semi-random distribution of requests can cause transient imbalances. Some bricks will get several requests while others get none. Moreover, having different kinds of bricks in the system makes this problem more difficult. With heterogenous 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 interac-

[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.

tion between the workload, the amount of cache and the specific disk models that each brick has, and any transient issues (such as one brick performing internal maintenance).

2.2.3 Exclusive caching

Maximizing the cache hit rate is critical to good storage system performance. Hence, 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 effective cache space and increase the 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. Centralized systems achieve exclusive caching easily. Since all data is stored within one single cache, there will never be duplicate entries in the cache.

[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.

2.3 Achieving all three at once

Any one of these concerns can be addressed individually and some proposed schemes have been able to achieve more than one of these properties. For example LARD [Pai98] 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 heterogenous population of bricks complicates most existing schemes significantly, given the vagaries of predicting storage performance for an arbitrary workload.

The fundamental property that current distributed 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 without knowledge of the current state of the system, or they duplicate work and implicitly get global knowledge at a performance cost.

The D-SPTF approach increases inter-brick communication to make globally-effective local decisions without creating duplicate work. It involves all bricks that store a copy of 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 heterogenous 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.

D-SPTF is not, however, without cost. By increasing the communication between bricks to achieve load balancing, scheduling efficiency and cache exclusivity, the network costs are increased over simple request distribution protocols. D-SPTF exploits high-speed networks and small message sizes to keep these network costs low.

3 Related work

D-SPTF strives to achieve good scheduling efficiency, load balancing and cache exclusivity, while providing the added flexibility of using a choice of head scheduling algorithms. These properties have all been individually addressed in related work but have not been combined in decentralized storage. This chapter discusses the related work for each of the three topic areas: disk scheduling, caching and load balancing.

3.1 Disk head scheduling

The ability of a disk drive to schedule well is influenced by the disk head scheduling algorithm. The goal of many such algorithms is to minimize the disk positioning overheads. D-SPTF has the ability to use any head scheduling algorithm. It will improve performance for any disk's scheduling algorithm that exhibits improved disk efficiency as the number of requests increase, including SPTF, SSTF, SCAN, and CLOOK.

The SPTF disk head scheduling algorithm that forms the heart of D-SPTF is

often used in modern disk drives today. Many studies have been conducted to compare and contrast the performance of SPTF and its predecessors (SSTF, SCAN, CLOOK and FCFS) [Denning67; Frank69; Coffman72; Teorey72; Gotlieb73; Oney75; Wilhelm76; Hofri80; Coffman82; Wong83; Leffler89; Bitton89]. The goal of the Shortest Positioning Time First (SPTF) algorithm is to minimize the total positioning time that a request incurs when it is serviced. The SPTF algorithm does this by scanning through the queue of requests and computing the seek time and the rotational time that each request would incur from the current disk head position. To select the next request to service, SPTF chooses the request with the minimum sum of seek and rotational latency. This results in an algorithm that will service requests with the shortest positioning time (seek + rotational latency).

One complaint about SPTF and related algorithms is that a request could suffer from starvation, because these algorithms seek to minimize the service times without considering how long a request has been in the system [Geist87; Seltzer90; Jacobson91]. Some algorithms have been proposed to address the potential starvation of requests. One prominent example is the age-weighted SPTF [Seltzer90; Jacobson91]. Age weighting considers how long a request has resided in the queue in addition to it positioning time. This age factor is weighted by an administrator-determined amount. As a request resides in the queue longer and longer, it is more likely to be selected even if it incurs a larger positioning time than a younger request in the queue. The larger the weight factor the more likely older requests will be selected.

For the D-SPTF system, the age-weighted SPTF algorithm was not chosen partially because the age factor is a tuning parameter. Since D-SPTF was designed the have as few tunable parameters as possible, this was not desired. In addition, the performance difference of age-weighted SPTF is not significant. Only for a few starved requests does it have a noticeable impact in response time [Seltzer90; Jacobson91].

3.2 Caching

Several groups have explored explicitly cooperative caching among decentralized systems [Felten91; Schilit91; Franklin92; Dahlin94; Feeley95; Voelker98]. These systems introduce substantial bookkeeping and communication that are not necessary if requests are restricted to being serviced by a single location. However, these techniques could be used to enhance load balancing and memory usage beyond D-SPTF, which focuses on the assigned replica sites for each data block.

The Berkeley NOW project [Anderson95d] implemented a form of cooperative caching that would scan the memory of the other nodes in the system before fetching data from disk. The principle was that it was more efficient to retrieve data from network memory than the local disk. The Globally-Managed Memory system [Voelker98] also used this form of cooperative caching with some enhancements to allow for application-driven caching and prefetching. xFS [Wang93b] also use cooperative caching for their distributed file system, relying on multi-processor cache coherence protocols for consistency and management.

One of the attributes of the D-SPTF system is that its cache cooperation is limited to only the aggregate cache space of the replica group. However, many disk array controllers already use cache partitioning [Manjikian95] to minimize cache conflicts between IO streams. This cache partition will limit the cache space available to individual IO streams to prevent one IO stream from flushing the entire cache. This increases the hit percentage of the other IO streams. D-SPTF's cache cooperation only between replica sets will also have a similar effect.

3.3 Load balancing

Another challenge of storage systems is effectively balancing the load among bricks to prevent any one brick from becoming overloaded. There have been a number of techniques developed to specifically address the challenge of load balancing. These are hashing, round robin distribution, and LARD.

Striping and hashing are popular techniques for load balancing. With striping and hashing, the system uses a hash on the logical block number (LBN) of a request to determine which brick receives the request. The hash ensures that all requests for an LBN will go to the same replica. Hashing provides long-term, statistical load balancing; short-term load imbalances can exist either due to imbalances in the workload, imperfections in the replica placement, or imperfections in the hash function.

Round robin request distribution schemes and their weighted variations [IBM-net; CiscoDirector] are in use in many systems. A round robin system achieves load

balancing by sending requests to each of the replicas in sequence. This provides each replica with the same amount of work as any other replica. However, with disk drives, all work is not created equal. In fact, some requests take considerably longer to service than other requests. As a result, a weighted round robin approach was utilized. In a weighted round robin approach, the existing load of the device is also considered when distributing requests. For instance, suppose that device A has 4 requests and device B currently has 3 requests. The weighted round robin system would send the next request to device B to balance the load, even if the prior request was also sent to device B. The goal is to keep the number of requests at each replica approximately equal.

There are also more dynamic schemes that migrate or rebalance load based on feedback. These are popular for activities like process or transaction executions. Clusters of web servers often use a load-balancing front-end to distribute client requests across the back-end workers. With a front-end distributing requests across a set of storage servers, feedback-based load distribution works well [ArpaciDusseau99; Hsiao91; Lee96].

LARD [Pai98] is one load balancing scheme that uses feedback, which combines the load balancing of a round robin replica selection scheme with high locality for improved caching. Round robin request distribution will result in good load balancing since requests are delivered round robin to devices that can service them, weighted by the load at the store to prevent load imbalances. However, this load balancing technique does not take into account the caching impact and, as

a result, can induce poor cache locality, since a request might not be routed to a store that has the data cached. Locality-based hashing has the opposite problem. With this technique, the workload is partitioned among all devices that can service it. This type of system will always route the same request to the same back-end store resulting in good cache hit ratios. However, if the workload is partitioned incorrectly, transient and long-term load imbalances can occur.

LARD improves both load balancing and locality by partitioning the workload initially and then readjusting those partitions if a load imbalance occurs. Specifically, when a request arrives for the first time in the system, LARD selects the server with the lowest load to service that request. LARD records which server was selected to service that request. If the request arrives in the system again, it will be forwarded to the same server that serviced the previous request in order to increase the probability of cache hits.

This is very similar to the locality-based hashing technique. The difference is that LARD redistributes when load imbalances occur. If the load between servers is greater than an administrator-specified threshold, then the LARD system will repartition requests to rebalance to the load. During a load imbalance, when a request arrives it will be redirected to lowest-loaded server instead of going to the server that last serviced it. Again, the new location is recorded so that future requests will be routed to that server. This rebalancing continues until the load imbalance drops below the threshold.

Obviously LARD is sensitive to the threshold tuning parameter that the ad-

ministrator sets. If the threshold is set too low, then LARD will tend to behave like a weighted round robin load balancer, because it will be continuously distributing the load to the least-loaded server. If the threshold is set too high, then it will behave like a locality-based hashing system, because LARD will be less likely to distribute the workload for transient load imbalances. Each of these systems is unacceptable. Choosing this parameter in itself is non-trivial, since a load imbalance under one set of conditions may not be a load imbalance under another set of conditions.

One of the major advantages that D-SPTF has over LARD is that D-SPTF has no tuning parameters. If the performance of the system changes, then D-SPTF will automatically adapt to the new system performance without external intervention. In addition, while LARD can adapt to load imbalances and generate good cache performance, it has no provision to provide the improved scheduling efficiency that D-SPTF provides. Thus, D-SPTF is able to provide all of the benefits of LARD, without requiring tuning, while also providing improved disk scheduling efficiency.

3.4 Replica selection systems

Several previous studies have explored replica selection in multi-disk environments, including Ivy [Lo90], HP AutoRAID [Wilkes96] and SR-Array [Yu00]. D-SPTF is not the first system to propose using multi-disk scheduling, though it is the first to do it in a decentralized setting.

Lo [Lo90] first attempted a technique similar to D-SPTF in a multiple device

system in Ivy. Lo realized that if a system kept multiple copies of a file, read performance could be improved by balancing the load across storage nodes and choosing the one that gives the fastest response. However, the update cost of such a system could incur significant overhead. The approach that they took was to alter the number of replicas of data based upon its access patterns. They hoped to obtain the advantages of wide replication for read-only workloads [Satyanarayanan89; Mann89] and avoid update limitations for replicating read and write data [Morris86; Treese88]. This expands upon user-directed variable schemes [Walker83; Purdin87] to create an automatic and dynamic replication system. Ultimately, Ivy tried to achieve a dynamic replica control system that maximizes read performance and minimizes the overhead incurred due to updates of replicated data.

Ivy's particular relevance to D-SPTF lies in how a replica is selected to service a request. When a file is stored with multiple copies on multiple disks, Ivy sends a read request to the replica disk with the shortest response time using centralized knowledge, including service time plus controller overheads. This is similar to D-SPTF in principal, except that the underlying stores use FCFS, rather than SPTF, head schedulers. Since Ivy uses FCFS, no reordering of requests occurs when new requests arrive. Ivy does suggest that CLOOK could be used instead of FCFS, but this would allow request reordering which makes prior service time predictions unreliable.

D-SPTF is different from Ivy in that it assumes a high performance SPTF disk scheduler at the device rather than FCFS. This SPTF disk scheduler has a

significant increase in scheduling efficiency over a FCFS scheduler. D-SPTF also sends a request to all replicas that can service it, rather than using a centralized decision making service. D-SPTF waits until a request is to be serviced to decide which replica is responsible for that request. This delayed decision allows for better tolerance for transient load imbalances. Ivy also has no notion of caching at each of the nodes; it assumes a central cache above the individual storage devices.

The HP AutoRAID [Wilkes96] is another system that suggests a distributed request selection algorithm. The HP AutoRAID system provides a two-level storage hierarchy in a single disk array controller. The upper level of the storage hierarchy provides high-performance mirrored storage for active data. The lower level of the storage hierarchy provides space-efficient RAID [Lawlor81; Park86; Patterson88; Patterson89; Chen93; Chen94; Menon93] storage for inactive data. For replica selection, the HP AutoRAID system chose to use random selection when a request arrives in order to keep an even and balanced load. However, Wilkes et al. also suggested that one could select the disk to service each request by choosing the disk that could service the block in the shortest time using the SPTF algorithm. However, no experimentation was done using the SPTF algorithm. Some experimentation was done using a SSTF algorithm with a centralized controller and full system knowledge, but Wilkes et al. came to the conclusion that the complexity associated with building a distributed Shortest Seek Time First(SSTF) wasn't worth the gains it provided with basic mirroring.

More recently, Yu et al.[Yu00] presented a similar replica selection algorithm to

D-SPTF. They explored multi-disk array scheduling in the SR-Array, which was designed to trade off storage capacity for improved media performance. This work builds upon the prior work in striping [Matloff87], mirroring [Bitton88; Dishon88; Polyzois93; Orji93] and rotational replicas [Ng91]. The SR-Array combines the techniques used in striping and rotational replicas to achieve a reduction in overall request latency. The SR-Array's relevance to D-SPTF is due to modifications that were made to a mirrored array to compare to the SR-array. The changes that they made to traditional mirroring are very similar to the distributed SPTF algorithm used in D-SPTF, though in a centralized controller. What they proposed was to change mirroring so that instead of partitioning the requests between the mirrors, all requests would be sent to all mirrors. Once a mirror scheduled a request, it would be removed from the other mirror. This technique was only used in a alternative ideal mirror system to compare to SR-Array. The SR-Array could not use such scheduling since all its replicas were made on a single disk. While Yu et al. never explored this technique in detail nor did they consider replication levels greater than 2-way mirroring, this scheme is similar to the centralized ideal against which we compare to D-SPTF.

4 D-SPTF Design

The D-SPTF request distribution protocol was designed with a set of goals in mind: to maximize read performance of a replica-based system, to minimize wasted cache space, and to limit protocol communication costs. The challenge of D-SPTF lies in creating a protocol that is effective for both read and write requests without incurring excessive communication. Additionally, the D-SPTF protocol should prevent multiple bricks from servicing the same request, which adds complexity to the problem. This chapter describes the read and write protocols for D-SPTF, followed by D-SPTF's CLAIM conflict resolution mechanism for preventing duplicate work.

4.1 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 always tries to process reads at the brick that can service

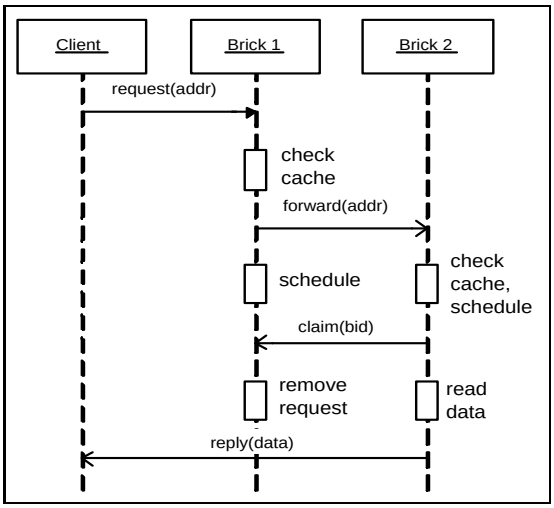

Fig. 4.1: **Read operation in D-SPTF.** The client sends a read request to one brick, which forwards it to others. The brick that can schedule the read first aborts the read at other bricks, and responds to the client.

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 request.

4.1.1 Read processing

The challenge of replica selection is in selecting which brick will service the request so that all the resources are optimally utilized. Figure 4.1 illustrates the steps to service a read request in D-SPTF.

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 the request does not hit in the brick's cache, then the brick places the request in its disk queue

and forwards that request to all other bricks that hold replicas of that data.

When a brick receives a forwarded read request, it first 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 service 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. The first effect is that duplicate work will be done, which will cause some resources to be wasted. The second effect is that the client will receive more than one reply—clients can simply squash such duplicate replies. Message losses should be very rare in the reliable, high-speed networks of brick-based systems. Even if a CLAIM message is lost, conflicting CLAIM messages are rare, so duplicate work caused by lost messages will be even rarer.

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.

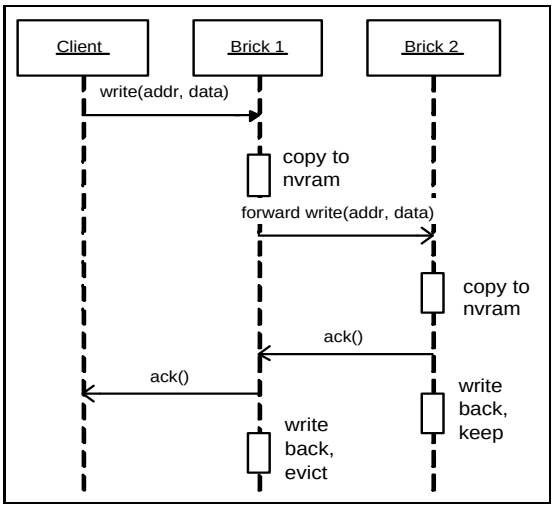

Fig. 4.2: **Write operation in D-SPTF.** The client sends a write request to one brick, which forwards it to others. Each brick determines whether it is the one to keep the data in cache.

4.1.2 Write processing

The write protocol (Figure 4.2) is somewhat different from the read protocol. When a brick receives a write request from a client, it immediately forwards the request to all 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 (which should be very rare in the reliable, high-speed networks of brick-based systems), a consistency protocol must address the potential brick failure. The basic D-SPTF protocol should mesh well with

most consistency protocols (e.g., [Amiri00; Frolund03; Goodson02; Goodson03a; Gray78]). A basic approach would be to timeout requests and resend the request once the timeout has expired. A brick, upon receiving the duplicate request, would return an acknowledgment and overwrite the existing data in cache. If a brick fails to send an acknowledgement after repeated time outs then the request will be reported as failed to the client and/or administrator. Alternatively, the brick could first timeout and resend the request to those bricks that failed to respond. If a few retries fail, then the request can be marked as failed.

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, despite independent invalidation decisions, by only keeping the block in cache if $hash(address) mod |replicas| = replica\ id$.

4.1.3 Concurrent CLAIM messages

With the base protocol, while one brick's CLAIM message is being transmitted across the network, another brick could select the same read and attempt to service it. If both bricks actually service the same request, then duplicate work would occur. Both bricks would then waste disk head time and cache space. This is most likely to occur any time the system is idle when a read request arrives, in which case, all replicas that can service the request will attempt to service it. This

causes each replica to try to CLAIM that request. The system must be designed so that only one CLAIM succeeds to prevent duplicate work. The greatest problem duplicate work creates is the potential for continuing conflict. If two bricks service the same request, then they will be at approximately the same head position to select the next request. Since SPTF is utilized, it is very likely the next request that both bricks select will again be the same. This creates a cycle of continuing conflicts resulting in duplicate work. Duplicate work can be a serious problem.

D-SPTF can use one of two mechanisms to avoid duplicate work: a pessimistic approach (prescheduling) and an optimistic approach (latency scheduling). The pessimistic approach assumes that conflicts will occur and avoids conflicts by prescheduling requests by waiting for a short period (two times the one-way network latency) after sending the CLAIM message to actually send the request to disk. 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 for the corresponding brick); with this information, each brick can decide for itself which one will service the request. The current approach is for the request to be serviced by whichever brick submits the lowest service time bid, ignoring when CLAIM messages were sent. If the service time bids are identical for the competing CLAIM messages, then the brick with the

lowest brick ID will service the request. The brick that lost the bid immediately schedules another request and sends a new claim message. 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 prescheduling, the wait period can almost always 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 wait one network round trip before issuing the request. With no work at the device, this will increase the minimum possible latency of the request by two network round trips for conflict avoidance. As illustrated in Figure 4.3, this does not impact performance unless 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.

Prescheduling does introduce a compromise. If two times the network latency is greater than the service time, then the system will delay servicing the request until two network latencies have occurred. This can allow the device to go idle while the request is waiting and can add to the request latency. However, since many existing networks provide low latencies, this should be rare.

The second approach is more optimistic by assuming that conflicts do not

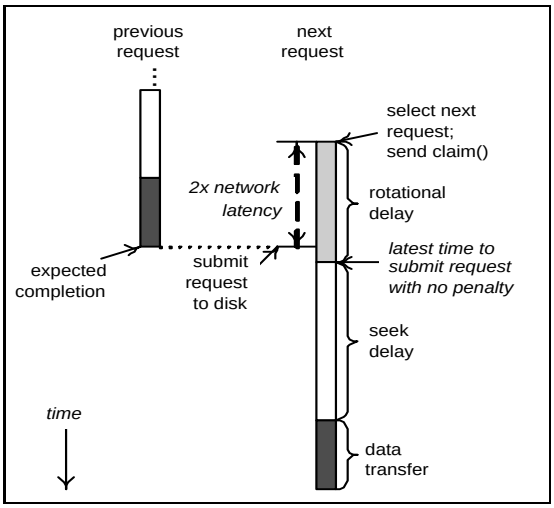

Fig. 4.3: **Overlapping** CLAIM **communication with rotational latency.** Issuing the request to disk can be delayed up to the time when seek latency must begin to reach the target track before the intended data passes under the read/write head. The rotational latency gives a window for exchanging CLAIM messages with no penalty.

occur with great frequency. In this case, it is permissible for duplicate work to occur, so long as it does so with low frequency. Instead of prescheduling a request and waiting two network latencies to issue the request, the optimistic scheme schedules the next request as soon the first one completes. The scheduler only waits for conflicting CLAIM messages as long as the disk would take to rotate into position to service this request before sending the request to disk. Since the disk is rotating continuously, delaying sending the request to the device during rotational latency does not increase request latency. Since network latency tends to be a small fraction of rotational latency, there is sufficient time for competing CLAIM messages to be received before the request is issued.

The benefit of this approach is that one will never increase request latency by waiting for potential conflicting claim messages. But if the network latency is

higher than rotational latency, two bricks may service the same request, resulting in duplicate work. This duplicate work wastes disk head time and runs the risk of creating a continuing conflict. The wasted time will result in increased request latency. To prevent continuing conflict, if a conflict has been detected, one scheduler would need to be temporarily modified to select a different request.

5 Experimental setup

The evaluation of D-SPTF was performed using a simulated environment of bricks, disks, and network infrastructure. To compare against D-SPTF, decentralized algorithms and an ideal centralized system were also simulated. This chapter details the simulation environment utilized and the additional algorithms studied. For all experiments, confidence intervals were computed based upon techniques for an unknown mean and unknown standard deviation.

5.1 Simulation Environment

The D-SPTF system was built within a simulated environment. The simulator is event-driven and is built upon the publicly-available DiskSim disk models [disksim] to simulate the disks within bricks. The DiskSim simulator accurately models many disks [Bucy03], including the Quantum Atlas 10K assumed in our system model. Simulation was chosen for experimentation due to the ease of construction and the accurate modeling provided by the DiskSim diskmodel component.

Four algorithms were implemented—D-SPTF, LARD, decentralized hashing,

and a centralized ideal—to compare the different aspects of replica selection in a brick-based storage systems. The system model for the first three algorithms are similar. Each system contains multiple replica groups, from 2-50 replica groups depending the experiment under test. Each replica group consisted of 2-8 bricks. Each brick is modeled as a single disk, processor, and associated disk block cache. Bricks are connected through a switched network. The one-way network latency is a model parameter, set to 50 microseconds by default, but was varied between 50-5000 microseconds.

Each brick contains a request queue. Whenever a brick's disk is about to become idle, the brick scans its request queue and makes a local decision to select the request with the shortest positioning time as the next request to be serviced. The simulation model assumes an outside-the-disk SPTF implementation, which has been demonstrated as feasible [Dimitrijevic03; Lumb02; Yu00], in order to allow the D-SPTF protocol to abort operations in the queue when an operation is claimed by another brick.

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. Each brick is composed of six 9 GB disks. Each disk is represented as its own volume with no internal striping or redundancy.

5.2 Request distribution algorithms compared

D-SPTF is compared against the two decentralized protocols, decentralized hashing and LARD, and the centralized system. The primary differences 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 4.1, with requests broadcast to all replicas and one replica claiming each read request.

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 that should be serviced by a different brick, 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. In addition, while decentralized hashing provides statistical load balancing, it is susceptible to short-term load imbalances.

The Locality-Aware Request Distribution (LARD) [Pai98] algorithm is designed to maintain good cache performance while preventing load imbalances. LARD achieves this in the following manner. When a request first arrives in the system, LARD looks at the bricks in the system and selects the brick that has the shortest queue depth to service that request. After the LBN has been serviced, LARD records which brick the LBN was routed to so that all future requests for that LBN will go to the same brick. Sending all future requests back to the same

brick allows for good cache hit performance. To prevent load issues, LARD monitors the brick queues and, if any queue reaches a certain set threshold, all new requests will be routed to bricks with lower queue depths and future accesses to those LBNs would go to the new brick locations. While the adaptability will help prevent load imbalances, changing which brick services an LBN can reduce the cache hit ratio. LARD's tuning parameters are an exercise in compromise, since making the system more adaptable improves load balancing at the cost of reducing cache hits. A less adaptable system will have better cache hit ratio but greater potential for load imbalances.

LARD was originally proposed for use in a centralized server front-end to distribute requests. Since decentralized brick-based storage has no centralized control, the LARD protocol was modified for brick-based storage. The brick-based LARD system does this by routing all requests to a replica group through one brick. This brick behaves like the centralized server in that it keeps track of every replica's queue depth and which requests have been routed to which replicas. This central brick also decides when to rebalance and how. The brick-based implementation uses the thresholds and tuning values reported in [Pai98].

The centralized ideal is obviously 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 stores all requests. When a disk is about to complete a request, the centralized 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.

6 Evaluation

This chapter evaluates how well D-SPTF is able to achieve its performance goals. These goals are to provide load balancing for long-term and transient load imbalances, to provide for good disk scheduling efficiency, to maintain cache performance, and to adapt quickly to heterogenous systems, all without requiring administrator tuning of the system. In addition, the system should strive to minimize the communication costs associated with the decentralized request distribution protocol.

Comparing D-SPTF to the performance of existing alternative systems requires one to consider three challenges: load balancing, scheduling efficiency, and cache performance. Each of these challenges has been addressed individually with existing systems but no systems have been able to address all three simultaneously before D-SPTF.

6.1 Load balancing

Load balancing is the equal distribution of work among bricks. This provides each brick with an equal share of the total work in the system and prevents any one brick from becoming overloaded, slowing down the entire system. When any one brick services a disproportionate share of the work in the system, a load imbalance occurs and the entire system's performance can be degraded. There are two types of load imbalances that can occur: long-term load imbalances and transient load imbalances. Long-term load imbalances occur generally because of a misconfiguring of the system, while transient load imbalances occur because of some temporary event that slows the performance of one or more bricks.

Load balancing is a significant problem in storage systems today. Improperly balanced load can result in devices being underutilized or even idle when work exists in the system. Replica-based systems increase the problem of load balancing because traditionally only one replica is selected to receive the request. If the incorrect replica is chosen, then a load imbalance can occur. An example of this would be to send equal amount of work to devices of unequal performance. For the purposes of this dissertation, a load imbalance is measured as the case where one replica is idle while other replicas have work queued that the idle replica could service.

One factor in load balancing is deciding when to select the replica to service the request so that the most informed load balancing decision can be made. If the

decision is made too early, then conditions can change resulting in the previously correct decisions causing a load imbalance.

D-SPTF solves the problem of load balancing in replica selection by delaying the choice of which brick services each request until the request is to be serviced. This is the last possible moment that the load balancing decisions can be made, resulting in the most informed load balancing decision.

D-SPTF implements this by sending a request to all bricks that can service it. This allows all bricks see all requests that they can service, ensuring that no brick will ever be idle when work exists in another brick's queue that it could service. With hashing and LARD, requests are routed to specific replica-holders as they arrive. These distribution schemes are designed to statistically prevent long-term load imbalances but do not prevent transient load imbalances. As a result, a request may be periodically routed to a busy device rather than an idle device because of the statistical load balancer, wasting the idle device's resources.

Load imbalances are also affected by properties of the workload in addition to the request selection algorithm. These properties are the read/write ratio of the workload and the number of requests outstanding in the system. The read/write ratio affects the load balance of the system because writes go to all bricks in a replica group, which reduces load imbalance. Since all bricks receive an equal amount of work, load imbalance is reduced as the number of writes in the system increases.

The number of requests outstanding in the system also has an impact on

load balancing due to the statistical nature of traditional load balancers. In these systems, with few requests outstanding, statistical load balancing may cause a replica to receive additional work, even when there are idle devices, creating a load imbalance. With large numbers of requests outstanding, it is less likely that a device will be without work.

To evaluate how much better D-SPTF is able to prevent transitory load imbalance than the other schemes, I conducted two experiments. To determine which schemes balanced the load most effectively, the average time a device spends idle is measured. A system that is perfectly load balanced will have zero idle time at any of its devices when there exists enough work to fully saturate the system. A system that suffers from transient load imbalances will have some of its devices periodically go idle. The first experiment examines the impact of workload read/write ratio on device idleness, and the second experiment varies the number of requests outstanding in the system.

The first experiment was designed to determine the sensitivity of transient load imbalances to read/write ratio. This experiment uses a synthetic random workload. This workload operates in a closed loop with zero think-time. The LBNs are drawn uniformly from the LBN space of the LUNs tested and the request sizes are drawn from a Zipf [Zipf35] distribution with a 4KB average. 16 requests were maintained outstanding in the system. The read/write ratio was varied between write-only and read-only in 10% increments. Eight replicas are used in the system.

Figure 6.1 show that, irrespective of read/write ratio, D-SPTF never exhibits

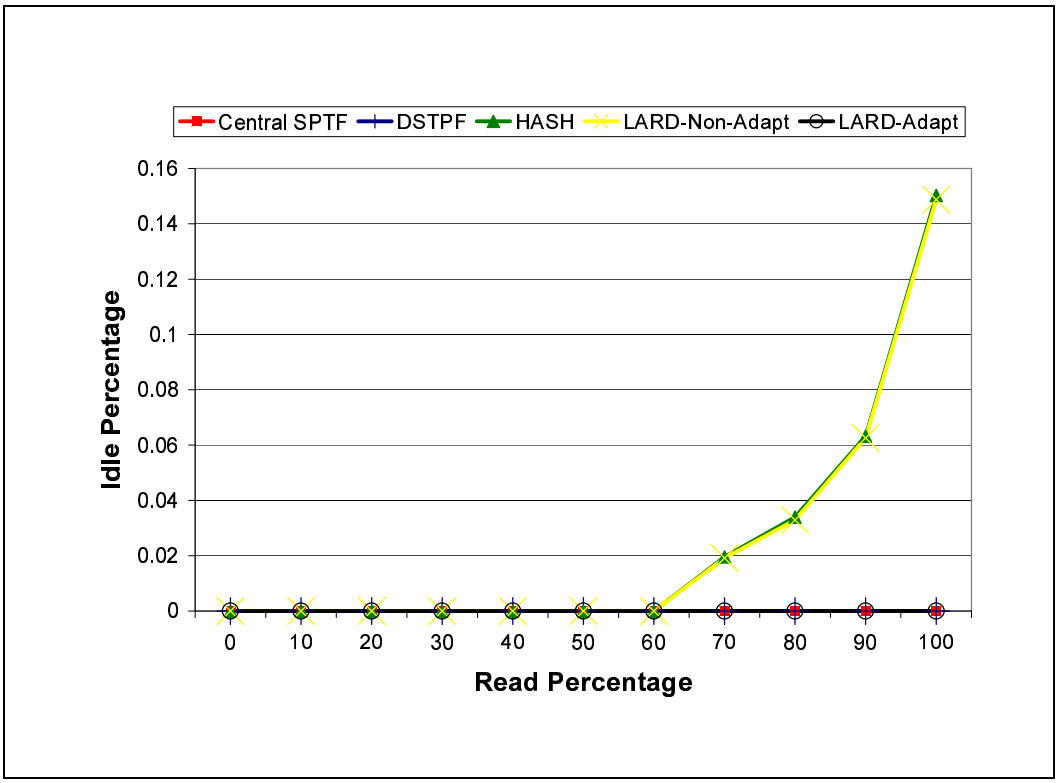

Fig. 6.1: **Brick idle percentage as a function of Read/Write ratio.** This graph show the percentage of idle bricks with each specific algorithm. These results were normalized to the performance of the ideal Centralized SPTF implementation. The 95% confidence intervals for the results were all within 1% idle percentage.

any idle periods in the system. LARD and hashing however do exhibit periods of idle behavior. In a completely read-only workload, they exhibit up to 15% idle periods. However, as the write ratio increases, the idle percentage decreases to almost zero at 60% read and writes. By changing LARD's parameters to be more adaptable, LARD can adapt to transient load imbalances by balancing brick queues more aggressively, but will suffer additional cache penalties.

The second experiment explores the effect of queue depth on transient load imbalance. The same workload as before was used with this experiment, with the

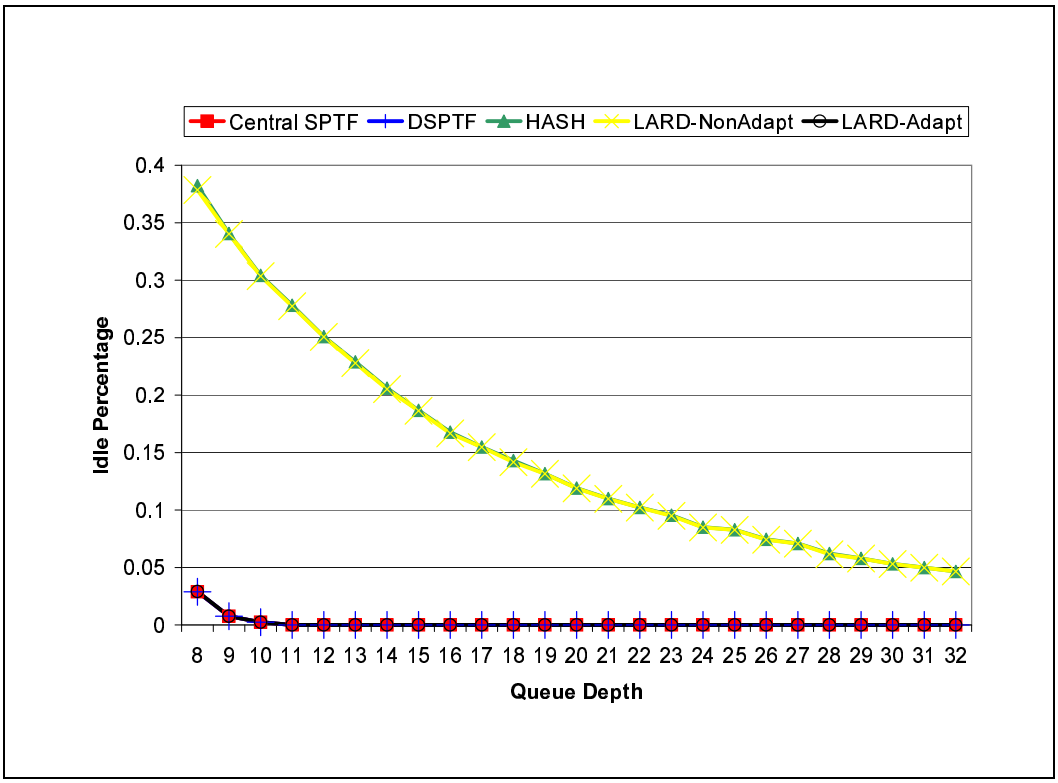

Fig. 6.2: **Idle percentage as a function of queue depth.** This shows the brick idle percentage as a function of the system queue depth for each of the different algorithms. Since there were 8 replicas in the system, a minimum of queue depth of 8 was used to provide at least one request for each replica. At a queue depth of 8 the ideal load balanced systems were slightly idle whenever a request completed at a replica, due to the system having to wait for completion to be reported to a client and a new request to be sent. The 95% confidence intervals for the results were all within 1% idle percentage.

following exceptions: the read/write ratio was fixed at read-only and the queue depth varied from 8 to 32 requests. This experiment utilized 8 replicas.

Figure 6.2 shows that D-SPTF is able to effectively balance the load in the system and prevent idle devices, irrespective of the queue depth. D-SPTF only fails to fully load balance is at queue depths of 8 to 10. At these queue depths, when any of the bricks completes its IO, that brick is idle until the client sends a new request into the system, since there is insufficient work to keep all bricks busy.

LARD and hashing do not fare as well as D-SPTF. These algorithms are able to prevent idle devices at high system queue depths, but at low system queue depths they have up to 35% of their head positioning time wasted by idleness. As before, the more highly adaptive LARD is able to adapt to transitory load imbalances. However, its cache hit ratio will be affected by increasing adaptability.

These experiments show that, for varying workload conditions, D-SPTF avoids transient load imbalances while still maintaining long-term load balancing. Hashing and LARD, while balancing load in the long term, do exhibit transient load imbalances where a request is routed to a busy brick instead of being routed to an idle brick. Since D-SPTF routes all requests to all bricks that can service them, it will never create a situation where work exists at one brick that could be serviced by another. This allows D-SPTF to effectively prevent both transient and long-term load imbalances.

6.2 Scheduling efficiency

With traditional mechanisms, one must trade off between load balancing and scheduling efficiency. Improved load balancing decreases scheduling efficiency and vice versa. However, D-SPTF does not have to make this trade-off. It achieves both at once. To understand why this is the case, one must understand how traditional load balancers impact scheduling efficiency.

The goal of traditional load balancing schemes is to evenly distribute the requests among all the bricks in the system. This gives each brick an equal fraction

of work. Given N requests in a system and K replicas, the traditional system will give each replica N/K requests. But, as shown in Figure 6.4, the more requests that a device has to schedule, the greater its disk scheduling efficiency. The ideal scheduling efficiency occurs when all N requests are sent to one replica instead of evenly distributed for optimal load balancing. But, sending all N requests to one replica will result in the other replicas being idle, wasting resources. The increased scheduling efficiency does not offset the the loss of throughput from idle devices. To maximize system throughput, traditional systems provide load balancing at the expense of disk scheduling efficiency.

D-SPTF does not have to choose between providing both scheduling efficiency and load balancing. D-SPTF does this by sending a request to all bricks that can service it. Since no partitioning occurs at request arrival time, each brick sees all the requests that it could service. This results in a maximally sized queue for disk scheduling. To prevent duplicate work, whenever one brick selects a request to service, that request is removed from the other bricks' request queues. While D-SPTF does maximize the scheduling queue it does not increase the amount of work that each replica services. Each of K replicas still services N/K of the workload on average.

In Section 6.1, D-SPTF was shown to provide effective load balancing. In this section, the experiments show that D-SPTF is also able to provide good disk scheduling efficiency. Scheduling efficiency is sensitive to three workload characteristics: request locality, number of requests outstanding in the system, and

read/write ratio. To evaluate D-SPTF's ability to provide scheduling efficiency a number of experiments were conducted. The first experiment is a baseline experiment. Additional experiments modified the baseline workload's queue depth, read/write ratio and locality to gauge D-SPTF scheduling efficiency under varied workload conditions.

For the baseline experiment, a synthetic random workload was used. This workload operates in closed loop format with zero think-time. The requested LBNs are drawn uniformly from the LUNs and the request sizes are drawn from a Zipf distribution with a 4KB average. The requests are 2/3 reads, and 16 requests were maintained outstanding in the system. Additionally, for the baseline, replication was varied from 2-8 replicas.

With basic mirroring (two copies), D-SPTF provides 9% higher throughput than hashing and LARD (Figure 6.3). As the number of replicas increases, D-SPTF's performance advantage increases. At eight replicas, D-SPTF provides 20% higher throughput. In every case, the media performance of D-SPTF is equivalent to the centralized ideal.

Recall that writes go to all bricks with replicas in all schemes. So, with 90% reads, each brick will average 1.6 writes and either 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 thus 8 verses 2.4. With 67% reads, the comparison is 8 versus approximately 5.6. The throughput values in Figure 6.4 for different queue depths match those observed in Figure 6.3, for all of the different schemes and numbers

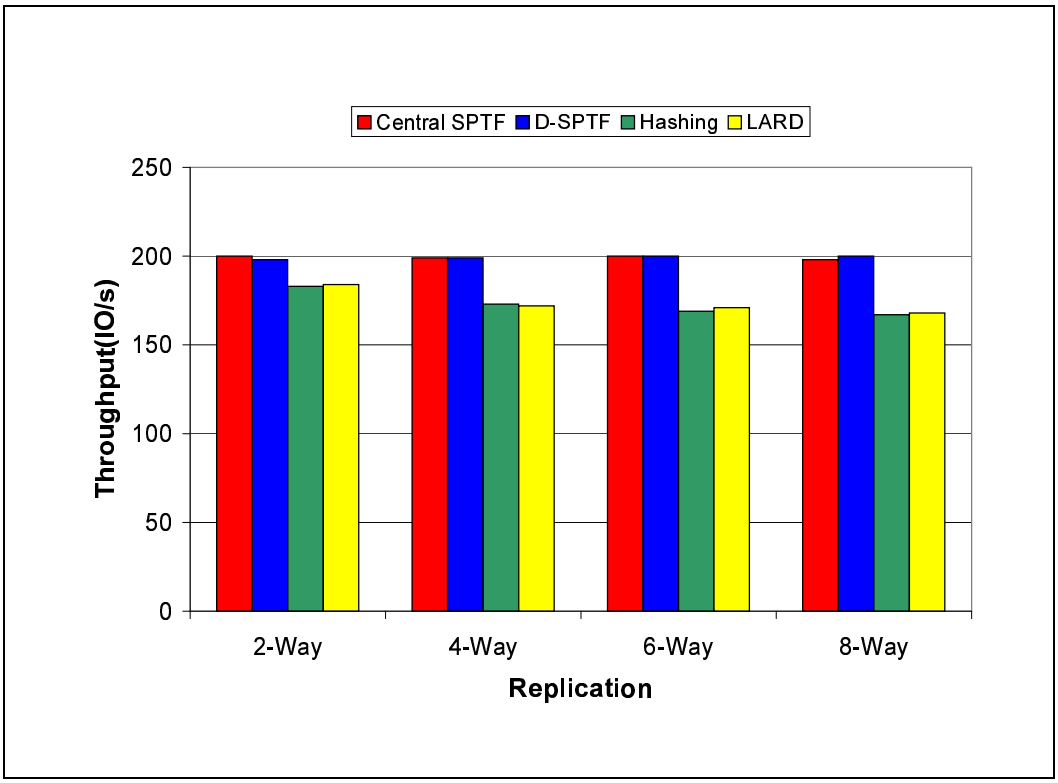

Fig. 6.3: **Throughput comparison of protocols, varying number of replicas.** Throughput is shown per replica (disk or brick). The 95% confidence intervals for the results were all within 2 IO/s.

of replicas.

6.2.1 Effect of number of outstanding requests

Evaluating the impact of number of requests outstanding is important, since the expectation is that as queue depth increases, the performance advantage of D-SPTF will decrease.

To evaluate this effect, the baseline workload was adjusted to vary the effective number of requests outstanding per replica group, while the degree of replication

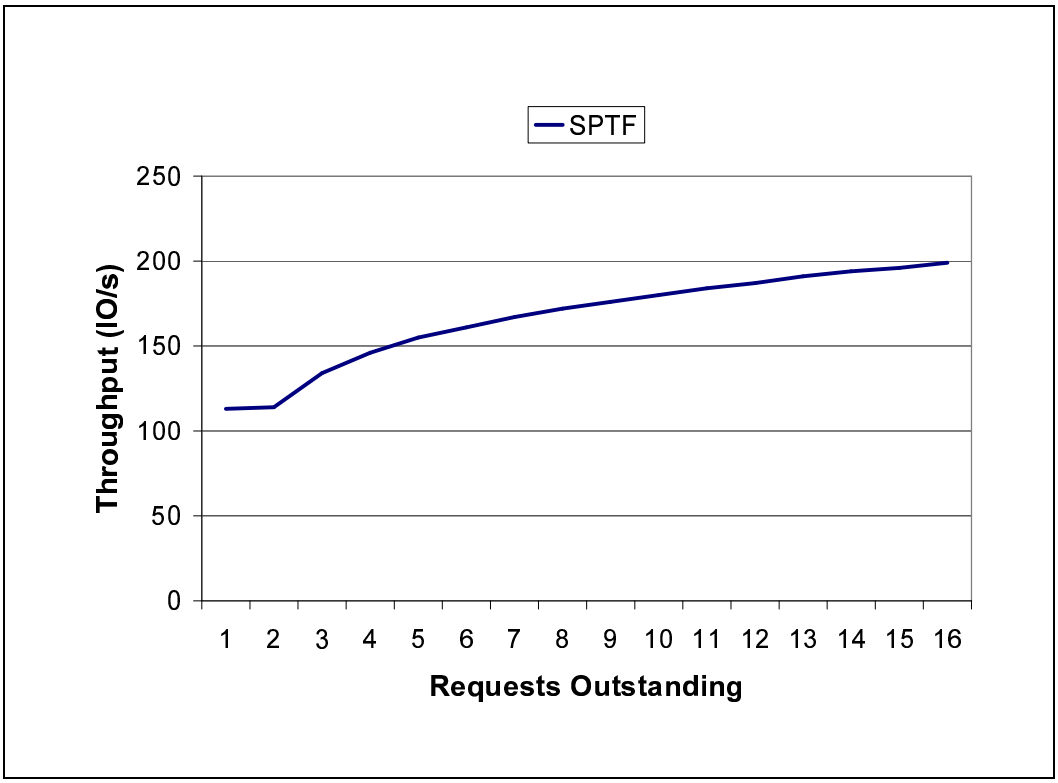

Fig. 6.4: **Disk throughput with SPTF scheduling, as a function of queue depth.** The data shown are for a closed synthetic workload (see Chapter 6) with a constant number of pending small requests to random locations on a Quantum Atlas 10K disk.

was fixed at two replicas. The number of requests outstanding per replica group was varied from 2-32 requests outstanding per replica group with the remainder of the workload parameters kept constant. The results in these experiments were normalized against the central SPTF algorithm to show the performance difference of the competing schemes in greater detail.

By examining Figure 6.5, one can see that, as the number of requests outstanding rises, the performance impact of D-SPTF goes down for a 2 replica system. This is to be expected, because of the diminishing returns that traditional SPTF

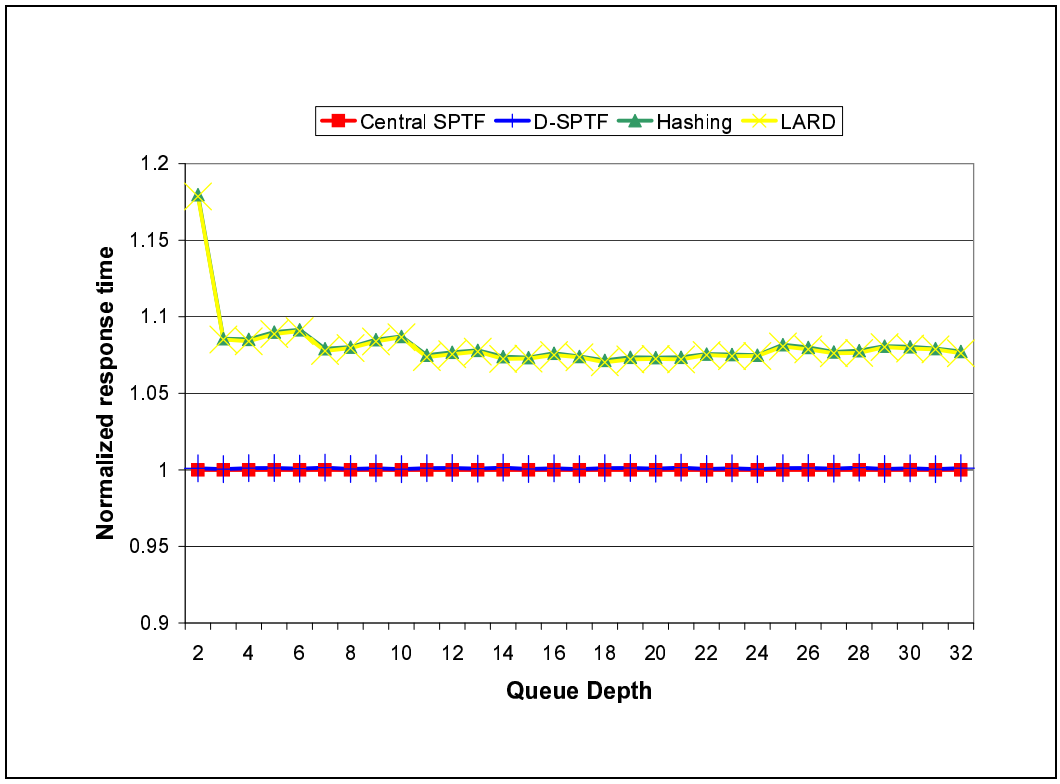

Fig. 6.5: **Response time as a function of queue depth for 2-way replication.** The differences in response time for the differing algorithms is show as queue depth increases. These results were normalized to the performance of the ideal Centralized SPTF implementation. The 95% confidence intervals for the results were all within 40 microseconds.

exhibits as shown in Figure 6.4. It is after a queue depth of 2 that the scheduling advantage of D-SPTF drops below 10%. Device manufacturers have often cited a queue depth of 4 as common for disks. D-SPTF's advantage grows with higher replication levels. By increasing replication to 8, D-SPTF's effectiveness stays at 13% for the baseline case. This is shown in Figure 6.6. The reason that D-SPTF is more effective with greater replication is that each replica tends to stay within a certain range of the disk, allowing it to further reduce the positioning times.

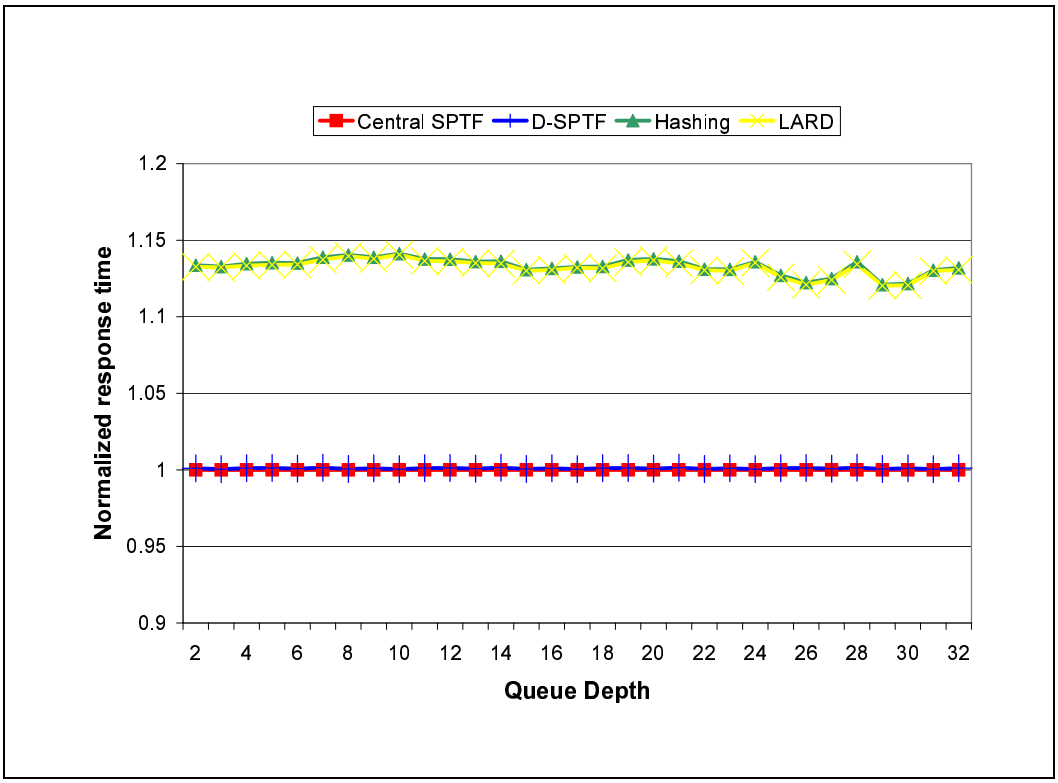

Fig. 6.6: **Response time as a function of queue depth for 8-way replication.** The differences in response time for the differing algorithms is show as queue depth increases. These results were normalized to the performance of the ideal Centralized SPTF implementation. The 95% confidence intervals for the results were all within 45 microseconds.

6.2.2 Effect of read/write ratio

It is important to evaluate the impact of the read/write ratio on the scheduling efficiency of D-SPTF. Since the scheduling impact of D-SPTF is derived from the increase of scheduling queue depth, the more writes there are in the system the lower D-SPTF's advantage will be. This is due to writes being sent to every brick in the replica group, which increases the scheduling queue depth of each brick for each write in the system.

To evaluate the impact of read/write ratio on scheduling efficiency, the baseline

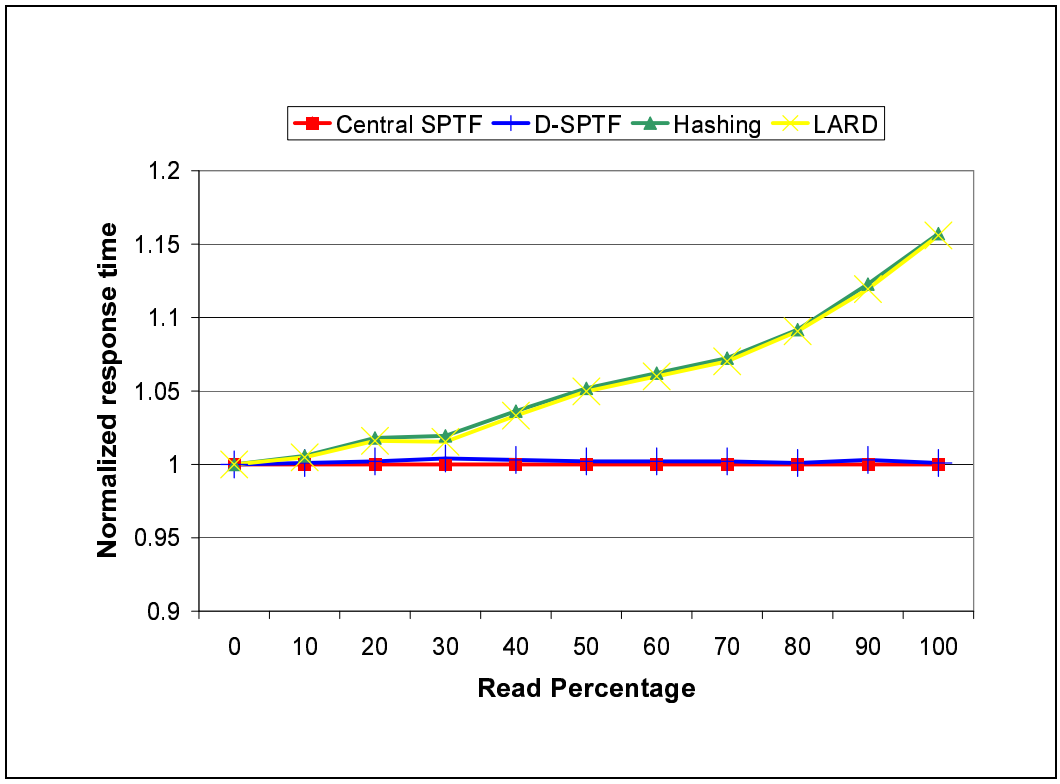

Fig. 6.7: **Response time as a function of read/write.** The differences in response time for the differing algorithms is show as read/write ratio increases. These results were normalized to the performance of the ideal Centralized SPTF implementation. The 95% confidence intervals for the results were all within 40 microseconds.

workload was modified to change the read/write ratio while all other parameters were kept constant. The read/write ratio was varied from write-only to read-only in 10% read/write ratio increments.

Figure 6.7 shows that, as expected, D-SPTF extracts the greatest scheduling efficiency advantage with a read-only workload. The scheduling advantage is reduced significantly once writes are introduced into the system, but D-SPTF is still effective at a common read/write ratio of 67% At a read ratio of less than 90%, the improved scheduling performance of D-SPTF drops below 10%.

The reason that D-SPTF performs best for the read-only workload is because traditional schemes partition the workload. Given a mirrored system (2 bricks), with 10 read requests outstanding, using LARD or hashing should result in 5 requests in each brick's queue on average. Using D-SPTF on the same system will result in 10 requests in each brick's queue, since each brick sees all requests it can service. This results in a 2x greater queue depth for D-SPTF to schedule from with the read-only workload. If, instead of the workload being 10 reads, the ratio was 6 reads to 4 writes, each brick in LARD and hashing would receive all 4 writes and 3 of the reads. This would result in a scheduling queue of 7 for LARD and hashing. Each brick in the D-SPTF would also receive all 4 writes and all 6 reads. This results in a queue depth of 10 requests and only a 43% greater queue depth for D-SPTF.

Increasing the number of writes in the system will reduce the scheduling queue advantage of D-SPTF as a result. At 100% write, D-SPTF's advantage is reduced to zero, as shown in Figure 6.7.

6.2.3 Effect of locality

A third factor that can affect scheduling efficiency is locality. The baseline study utilized a random workload. In real systems, however, workloads often exhibit a degree of locality.

To evaluate the impact of locality on D-SPTF's performance, the baseline workload was modified to introduce localized requests. A localized request is de-

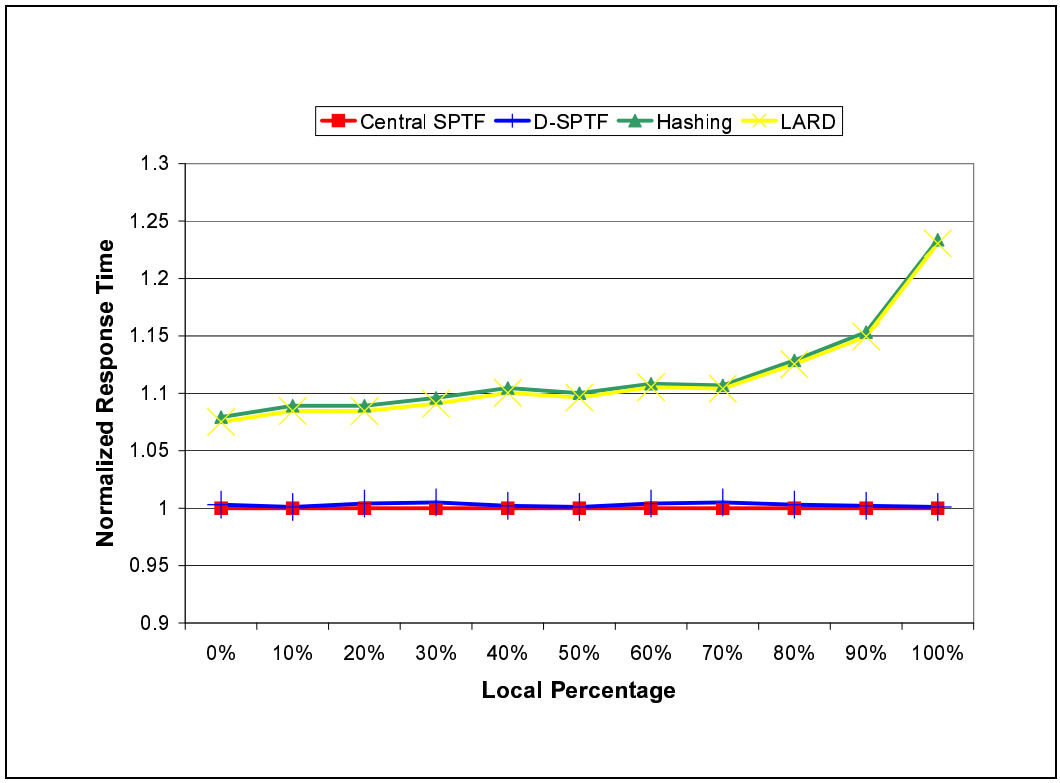

Fig. 6.8: **Normalize response time as a function of request locality.** The differences in response time for the differing algorithms is show as request locality increases. These results were normalized to the performance of the ideal Centralized SPTF implementation. A local request is defined as a request that is within 10 disk tracks of the last request sent to the system. The 95% confidence intervals for the results were all within 30 microseconds.

fined as one that is within ten disk tracks of the last request. The percentage of localized requests was varied between 0-100% of requests, in 10% increments.

Figure 6.8 shows that the performance advantage of D-SPTF increases as the fraction of local requests increase. For a purely random distribution of requests (0% local), D-SPTF achieves 9% better scheduling efficiency than Hashing or LARD. However with a high-locality IO stream, the D-SPTF scheduling advantage increases to 24% over LARD and hashing.

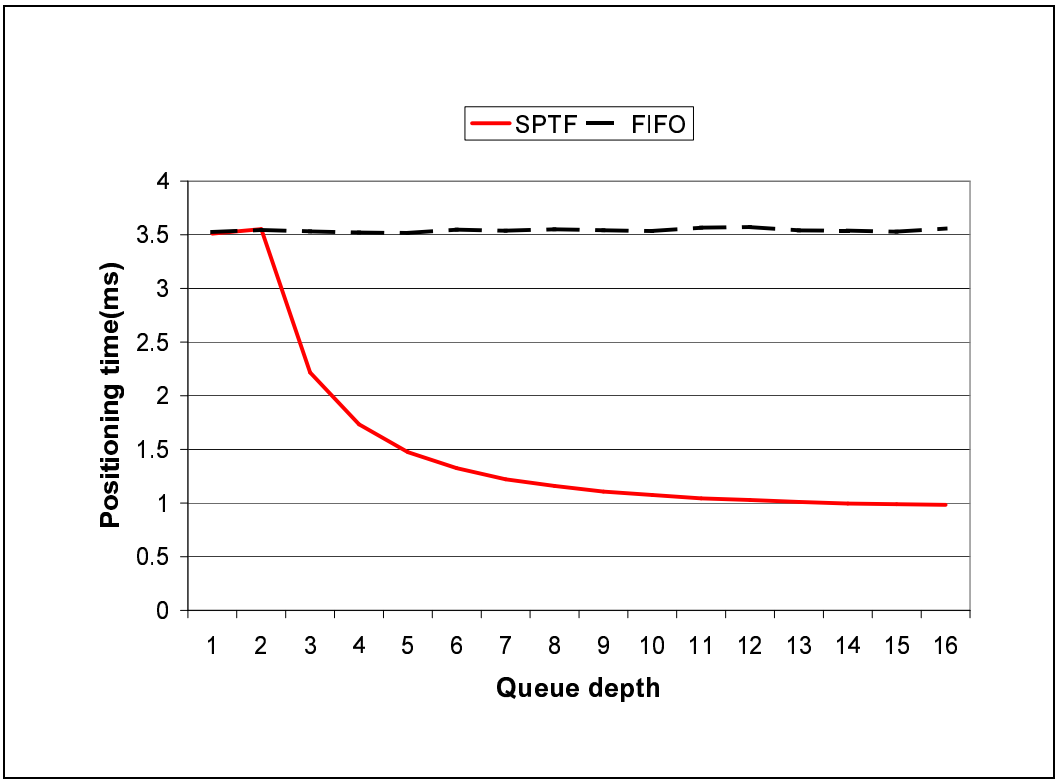

Fig. 6.9: **Local request throughput with SPTF scheduling, as a function of queue depth.** The data shown are for a closed synthetic workload (see Section 6) with a constant number of pending small requests to random local locations on a Quantum Atlas 10K disk.

The reason that D-SPTF benefits from more localized access is the behavior of SPTF scheduling in the presence of local requests. Figure 6.9 demonstrates this behavior. If one compares Figure 6.9 to Figure 6.4, one can see that, for localized requests, SPTF improves positioning time overhead by 3x at a queue depth of 16 relative to a queue depth of 2. On the other hand, with random requests, SPTF only schedules 1.8x better at a queue depth of 16 requests than at a depth of 2. One can also see the the slope of the curve with localized requests is initially greater than with random requests. The reason for this is that localized requests reduce

the seek times significantly. As a result, SPTF is mostly minimizing rotational latency, which can be reduced more easily than seeks.

6.2.4 Effect of combinations

Each of these three attributes (number of outstanding requests, read/write ratio, and locality) can have a significant effect on its own. However, it is important to understand the impact on the system when these effects are combined. This section explores the impacts of read-only workloads with varied queue depths and high-locality workloads with varied queue depth. Also, read-only workloads with high locality and varied queue depth are examined.

Figure 6.10 shows the effect of varying the number of outstanding requests for a read-only workload. The primary impact of increasing queue depth is that the performance of D-SPTF has been increased, making its performance advantage more resilient to increased queue depth. In this case, D-SPTF is provides at least 10% scheduling performance improvement for all measured queue depths. This is considerably better than the benefits for the 67% read/write ratio shown in Figure 6.5.

Figure 6.11 shows the impact of a localized workload with varied queue depths. As in the prior example, D-SPTF's increased advantage allows the system to maintain increased performance in the light of increasing queue depth. Localized requests allow D-SPTF to maintain at least a 10% scheduling performance advantage for the entire range of queue depths, whereas the baseline workload(shown in

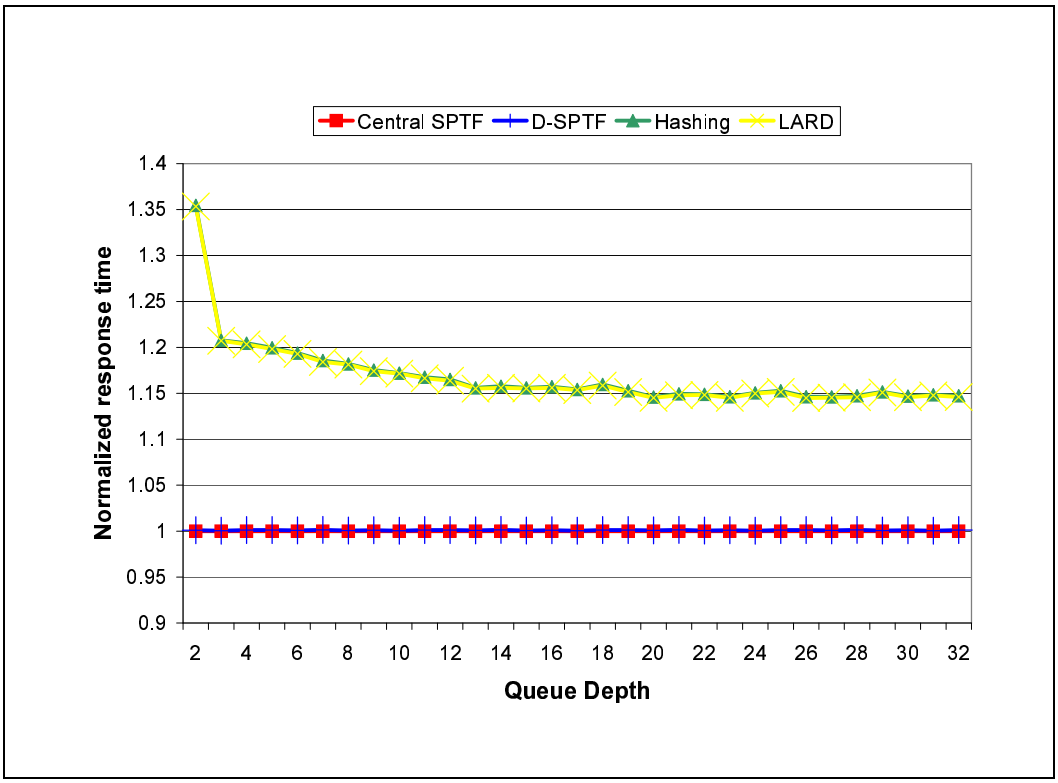

Fig. 6.10: **Response time as a function of queue depth.** The differences in response time for the differing algorithms is show as queue depth increases with a read only workload. The 95% confidence intervals for the results were all within 50 microseconds.

Figure 6.5) reaches at least 10% performance improvement only at 2 requests.

The final experiment demonstrates the impact of increasing queue depth on a read-only, local workload. Figure 6.12 shows that D-SPTF produces much greater benefits in the presence of a read-only localized workload with low replica queue depth. Combining locality and high read ratio results in a maximum performance advantage of 57% for queue depths of below 10. This is considerably better than the baseline maximum of 18% performance improvement shown in Figure 6.5.

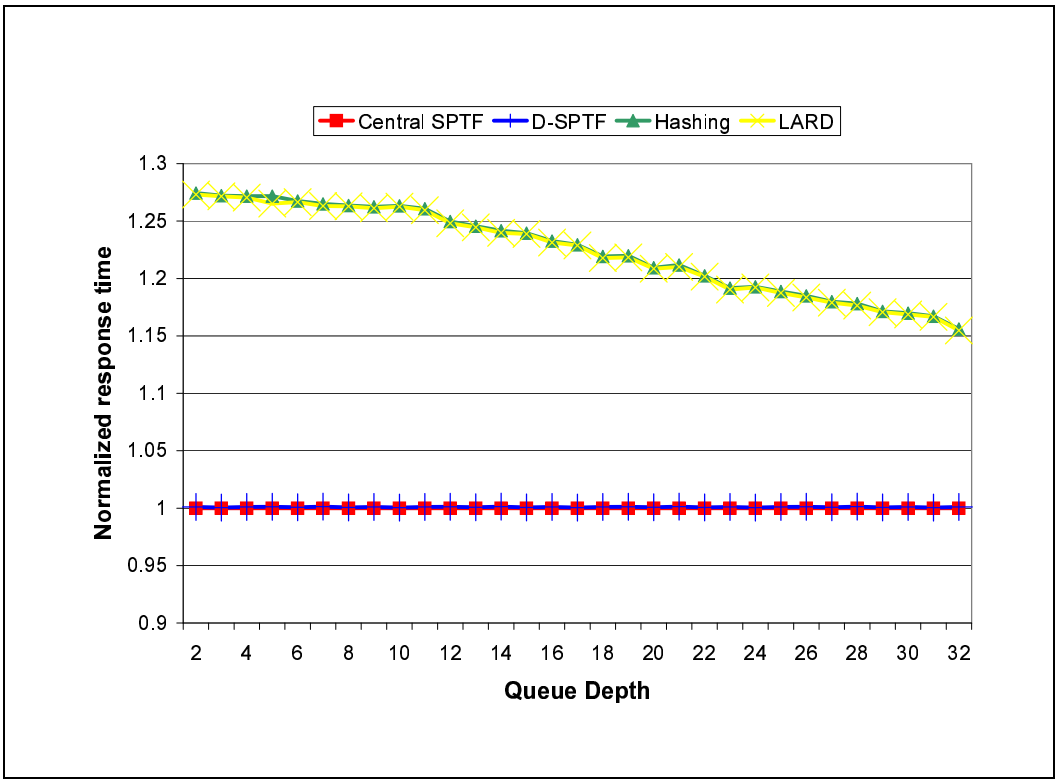

Fig. 6.11: **Response time as a function of queue depth.** The differences in response time for the differing algorithms with a highly local request pattern is shown as queue depth increases. These results were normalized to the performance of the ideal Centralized SPTF implementation. The 95% confidence intervals for the results were all within 35 microseconds.

6.2.5 Traced workloads

In addition to running synthetic workloads, trace workloads were used to examine the impact of D-SPTF in real systems. For real workloads, we expect to see performance gains similar to synthetic workloads for D-SPTF. To verify this expectation, we use four traces (Table 6.1). The *SAP* trace [HPLtraces] captures the I/O activity of a SAP-based system running an Oracle DB supporting 3000+ clients accessing billing records. The *Cello2002* trace [HPLtraces] captures the I/O activity from the software development activities of a 20-person research lab.

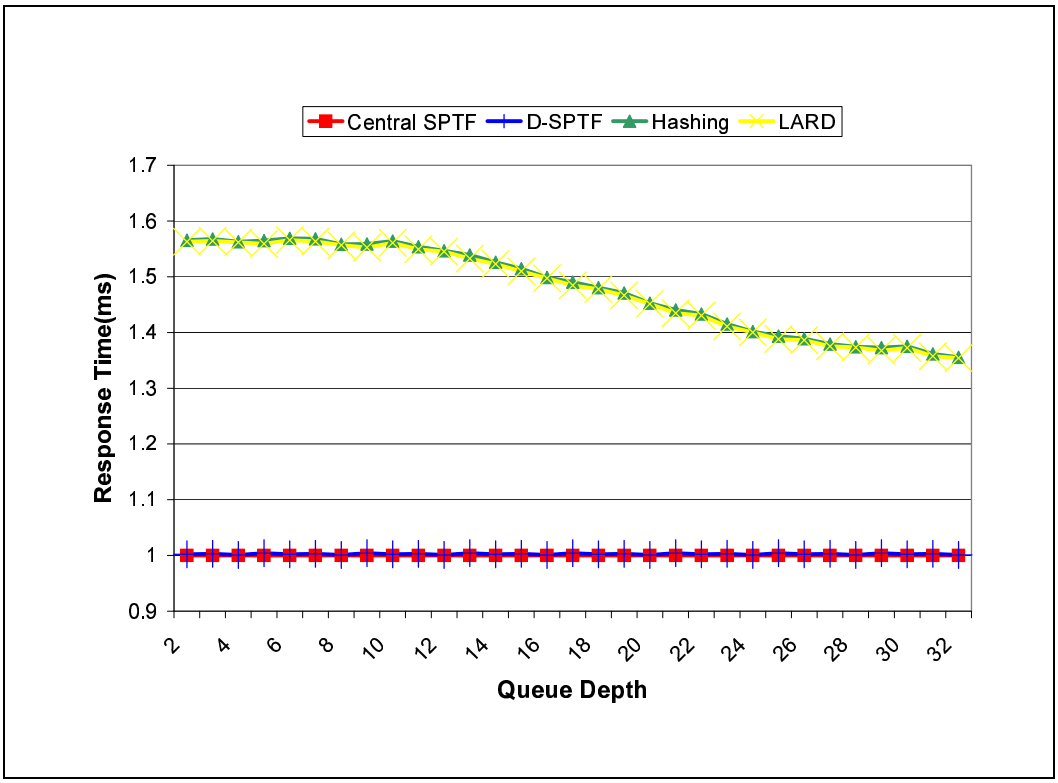

Fig. 6.12: **Response time as a function of queue depth.** The differences in response time for the differing algorithms under a read only, highly localized workload is shown as queue depth increases. These results were normalized to the performance of the ideal Centralized SPTF implementation. The 95% confidence intervals for the results were all within 40 microseconds.

The *WebSearch* trace [SPCtraces] captures the performance of a system processing web search queries. The *Financial* trace [SPCtraces] captures the performance of a system running a financial transaction processing system.

Figure 6.13 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 observed response times for the SAP trace, which is in line with the synthetic workloads running 2-way replication. There is no change in the throughput of the system, because the simulated system is powerful enough to

	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 6.1: **Trace characteristics.**

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 and LARD compared to 20% for the other traces. Cello2002 also has good locality but has very low load, reducing the potential impact of any scheduling algorithm. 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 show higher locality.

One last issue to consider is the impact of request sizes on D-SPTF. D-SPTF functions best with low request sizes, where positioning time dominates media service time. As the requests increase in size, media transfer time becomes a larger percentage of response time. As the role of positioning time is reduced, the scheduling impact of D-SPTF is also reduced. Today there are many workloads with small IOs that are limited by positioning delays. This trend is likely to continue in the future. This is because capacity doubles every 18 months or so while media response time increases only about 10% a year. What is considered a large IO this

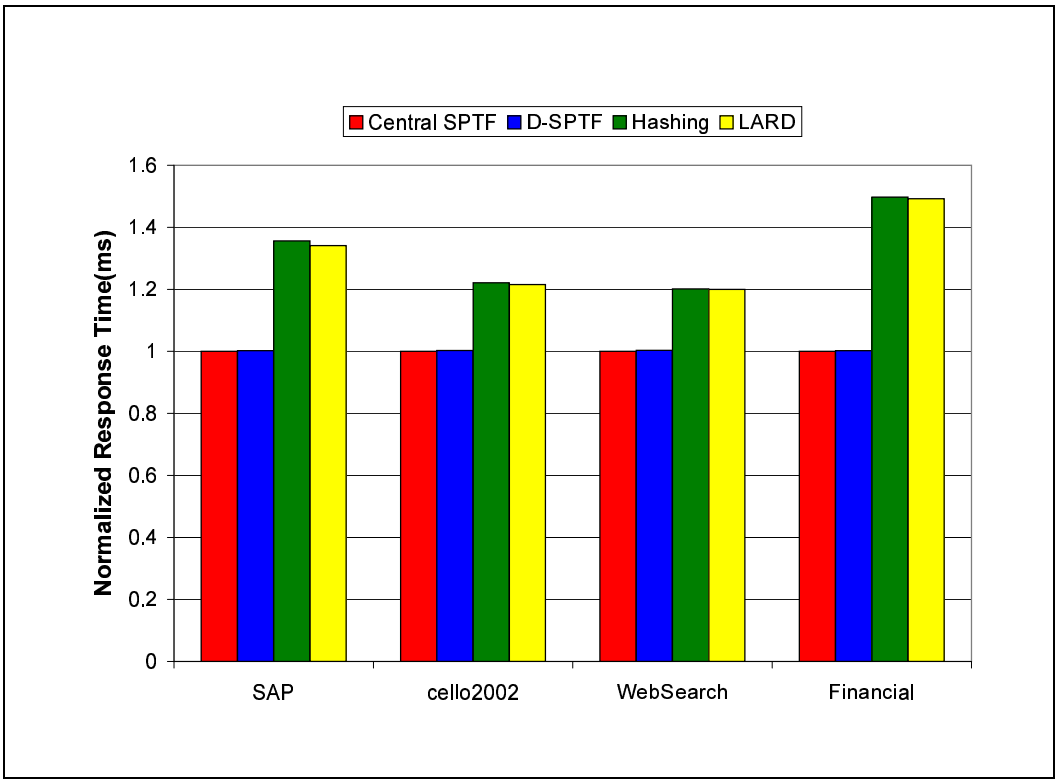

Fig. 6.13: **Response times for four traces.** This graph shows the response times of the four traces on a two-replica system with various request distribution schemes.

year may not be so next year.

6.2.6 Summary

Overall, D-SPTF exhibits a significant increase in scheduling efficiency compared to traditional schemes such as hashing and LARD. This is due to D-SPTF forwarding all requests to all replicas that can service them which maintains load balancing and maximizes scheduling efficiency since each replica has the greatest possible queue to schedule requests from. Unlike traditional systems, D-SPTF does not have to choose between scheduling efficiency and load balancing. D-SPTF can

achieve both simultaneously.

6.3 Cache performance

One of the more challenging aspects of brick-based storage is building a system where many small independent caches behave like one large cache. If the caches fail to work cooperatively, then system performance can suffer from a lack of cache exclusivity and correspondingly lower cache hit rates. Lack of cache exclusivity occurs when two bricks each decide to cache the same data. This effectively reduces the size of the aggregate cache and can cause a lower cache hit ratio. Additionally, lower cache hit ratios can be caused by load imbalances between bricks. Load imbalances can cause one brick to service more requests than another brick, causing the cache of that brick to be cycled too quickly causing lower cache hit ratios.

These two problems do not exist within a traditional centralized system with a single centralized cache. With a centralized cache, only one copy of data is maintained, preserving cache exclusivity and effective cache size. Additionally, with a single centralized cache, there are no load balancing problems resulting in lower cache hit rates.

D-SPTF provides a solution to these problems. D-SPTF is able to both maintain cache exclusivity and keep a high cache hit ratio. For the first problem, cache exclusivity in read requests, D-SPTF sends a request to all replicas that can service that request. When a brick receives each request, it checks its cache, and if it hits in cache, it informs the other bricks so that they do not attempt to service that

request. In the event that two bricks service the same request, conflict resolution after the fact will only save one copy of the data. For write requests, only one brick is selected to cache the new data after writing it to disk media, thus preventing wasted cache space.

For the second problem, D-SPTF aids in maintaining a high cache hit ratio by effectively maintaining balanced load between replicas. As a result, no hot or cold replicas exist, which prevents one cache cycling too quickly.

Hashing and LARD also effectively address both of these issues. Both hashing and LARD, in the common case, send a request for an LBN to a specific replica and only that replica. Since a request is only serviced by a single replica brick, multiple copies in multiple bricks caches are prevented. This maintains exclusive caching and maximizes use of cache space. Additionally, while hashing and LARD may suffer from transient load imbalances, they do effectively provide long-term load balancing. This prevents hot and cold caches. Since all three decentralized schemes provide both caching properties, one would expect them to perform similarly to one another and to the centralized ideal.

To explore the caching implications of the different schemes under workloads with locality, we use the same traces described in Section 6.2.5. Trace characteristics are shown in Table 6.1.

Table 6.2 shows the overall cache hit rates for the four traces applied to an 8-replica system in which each brick has 512MB. The centralized ideal is equipped with an equivalent total memory capacity of 4GB. Centralized caching always

	SAP	Cello2002	WebSearch	Financial
Central SPTF	74.03%	76.12%	68.24%	71.32%
D-SPTF	72.67%	71.82%	66.58%	68.83%
Hashing	72.23%	71.34%	66.25%	68.41%
LARD	72.41%	71.39%	66.37%	68.47%
Adaptive LARD	35.21%	35.87%	32.41%	32.79%

Table 6.2: **Cache hit rates.** This table shows the cache hit rates for the different configurations.

achieves the best hit rate, and the three decentralized schemes have hit rates that are 2–4% lower. This matches the expectations.

The evaluations in previous sections include a variant of LARD that adapts to load imbalances quickly to prevent transient load imbalances. Very small differences in queue depths of bricks would cause a redistribution of load. However, the cost of the load balancing is that this LARD configuration often changes which replica services requests for a given block. This aggressive load rebalancing reduces the cache hit ratios, since the data could have been in cache at the brick that last serviced a request, but not in cache at the new brick. This can also cause two bricks to have the same data in cache, violating exclusive caching, when the new brick reads the data into the cache before it has been flushed from the cache of the previous brick. As a result, the cache hit ratio of Adaptive LARD is significantly reduced. One can see that the cache hit ratio of Adaptive LARD drops from 66-72% to 32-35%. While Adaptive LARD does a better job at transient load balancing, it results in a much worse cache hit ratio.

6.4 D-SPTF communication costs

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

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, either because it is a cache hit or after completing its last disk request. In this case, the total number of messages will be $2N$, where N is the number of replicas of the data. Those $2N$ messages include the original request and the request completion. The maximum number of messages would be sent when all devices are idle when a new read request arrives, causing them all to decide to try to service the new request. In this case, N^2 CLAIM messages per request would move through the system. The minimum number of messages sent occurs during a cache hit on the original brick contacted. In this case, only 2 messages are sent: the request and its completion.

	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 6.3: **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.

To validate these expectations, we use the synthetic workload with 16 requests kept outstanding in each replica group. Table 6.3 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 with normal loads. For additional insight, Table 6.4 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 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
Conflicts	0.15%	0.67%	1.44%	2.44%

Table 6.4: **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 the overhead would 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.

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 6.14 shows the effect of network latency. This experiment uses the 8-replica system and workload from Section 6.1, but varies the one-way network

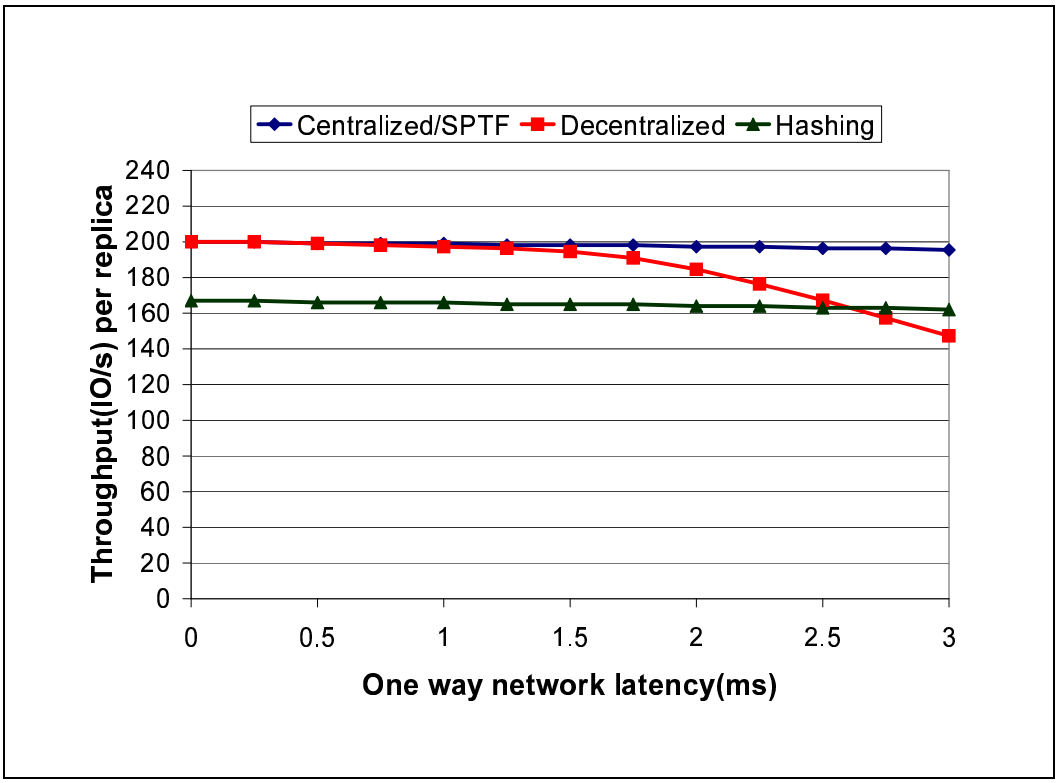

Fig. 6.14: **Effect of network latency on D-SPTF throughput.** Throughput is shown per replica. The network latency varies from 0 ms—instantaneous communication—to 3 ms.

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 [Schuchart02], 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.

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.

D-SPTF's ability to tolerate network latency will decrease if disk positioning times improve at a more rapid pace than network latencies. However, current trends indicate disk head performance improvement significantly less per year than network latency improvement. This suggests that as time goes on, network latency will be an even smaller fraction of media access time than in current systems. This should increase D-SPTF's ability to tolerate network latency in the future.

6.5 Systems of heterogenous storage bricks

D-SPTF excels with heterogenous 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 a storage system size that is appropriate to meet current demands. If the demands increase later in the life of the 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 are likely to be bricks of differing performance in these systems. Hav-

ing heterogenous bricks increases the difficulty of achieving the performance of a centralized system. This section shows that D-SPTF is able to adapt to heterogeneity more effectively than existing request distribution schemes. Two types of heterogenous systems are evaluated. These are static and dynamically heterogenous systems.

Static heterogeneity

A static heterogenous system is a system with devices of differing performance that do not change. A simple example of this is an old brick and a new brick: a new brick will often provide higher performance than an older brick. Since brick-based storage is designed for incremental addition, it is likely that systems will have devices of differing performance. Failure of D-SPTF to adapt to static heterogeneity can result in load imbalances and under-utilization of resources.

Consider a two-brick system that contains one "fast" brick and one "slow" brick. 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 brick and 1/3 to the slow brick. LARD and D-SPTF handle static heterogeneity via this type of load-based request distribution.

To explore the impact of static heterogeneity on the system, a synthetic experiment was run. For this experiment, a synthetic random workload is used. This workload operates in closed loop with zero think-time. The LBNs are drawn uniformly from the LBN space, and the request sizes are drawn from a Zipf dis-

tribution with a 4KB average. The requests are 67% reads, and 16 requests are maintained outstanding in the system. Two replicas were used. One of the replicas in the group functioned at normal speed(fast) and the second replica had 20% longer media response times(slow).

The algorithms tested are the centralized SPTF, D-SPTF, hashing and LARD. Additionally a weighted hashing algorithm was designed. This algorithm sends fewer IOs to the slower replica than the faster one (66% of the faster brick's load) to give a more even load distribution.

Table 6.5 shows the "fast" and "slow" bricks throughput results for this experiment. Both D-SPTF and the centralized ideal have the desired load balancing property: as the speed of the brick decreases, 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 bricks are of equivalent speed and thus sends half the requests to each disk. The problem is that the fast brick completes requests more quickly than the slow brick and, as a result, all of the requests end up queued at the slow brick. This causes the fast brick to have to wait for the slow brick to complete requests before it can receive new work. 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.

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

Table 6.5: **Throughput balance between a fast and a slow replica.** The disk in the slow replica is two-thirds 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. The 95% confidence intervals for the results were all within 1 IO/s.

Both LARD and a weighted hash function do a better job of adapting to heterogenous bricks than standard hashing. This is because weighted hashing accounts for the relative performance of the bricks. LARD functions well, also, because it initially allocates requests based upon the load at the brick, 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 brick performance changes.

Figure 6.15 shows how the slow/fast two-replica systems perform under traced workloads. D-SPTF is close to Central SPTF and outperforms the other schemes. LARD and weighted hashing are able to adapt to the performance difference but provide inferior disk scheduling. In general, unmodified hashing performs worst, since it both lacks the ability to properly balance the load and has less efficient scheduling.

Dynamic heterogeneity

In a dynamic heterogenous system, the performance difference between bricks changes over time. The difference could come from bricks servicing different applications, such as background activity, from short-term internal housekeeping, such

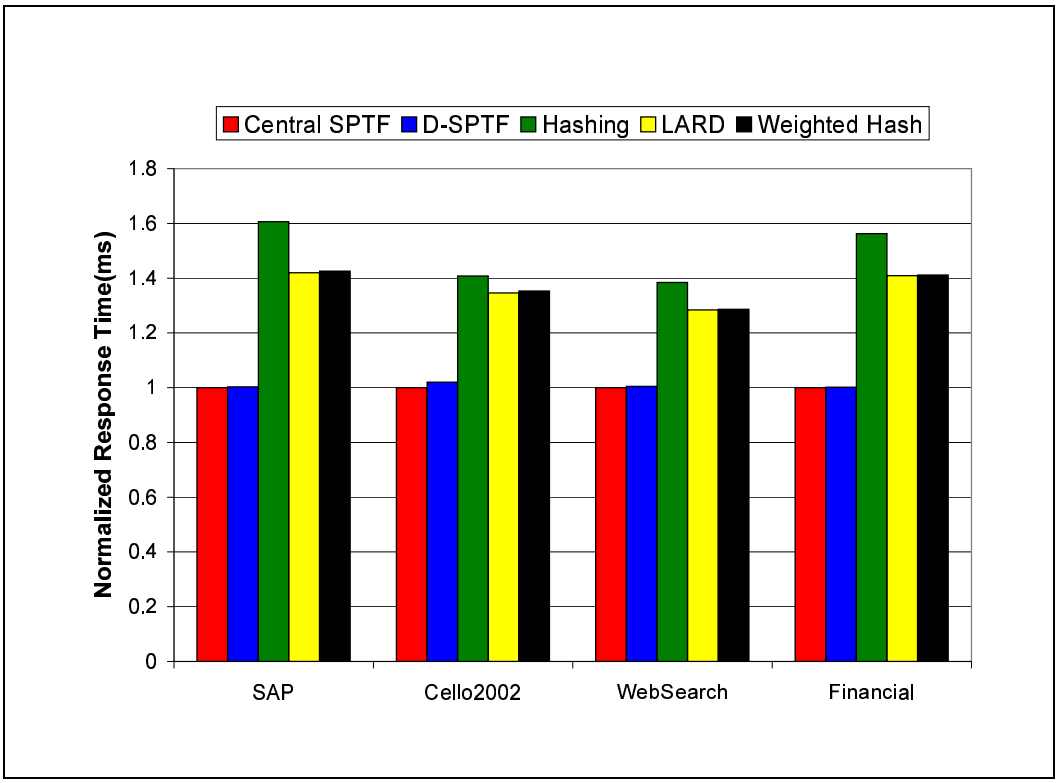

Fig. 6.15: **Impacts of static heterogenous devices on trace workloads.** This graph shows the client response time of four trace workloads with statically heterogenous workloads. These results were normalized to the performance of the ideal Centralized SPTF implementation.

as defragmentation, and from differences in how bricks respond to different access patterns. A system that fails to adapt to this dynamic heterogeneity can result in load imbalances and under-utilization of resources.

To examine the impact of D-SPTF in such situations, I varied the performance of a two-replica system dynamically. The base system had two identical replicas. The performance of one replica was altered as follows. Initially each replica was operating at full speed;[1] after one quarter of the run had passed, the first replica

[1]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.

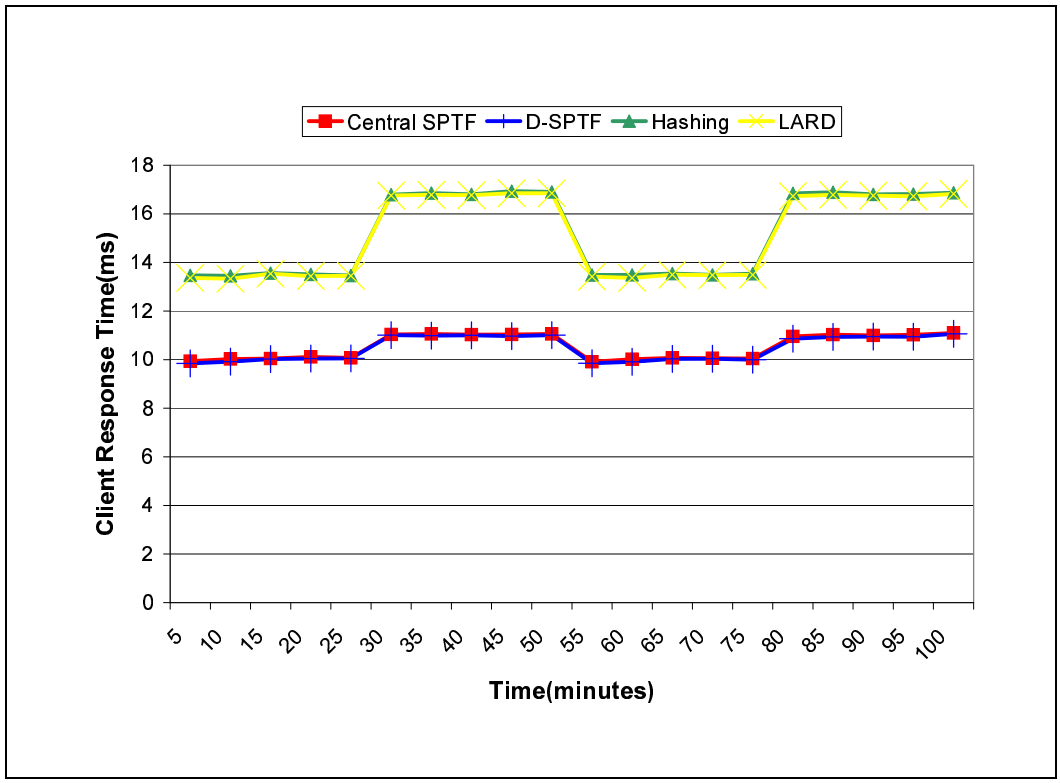

Fig. 6.16: **Performance in face of transient performance breakdowns.** This graph shows the performance of each scheme in a two-replica system where one replica's performance varies, halving at time 25, correcting at time 50, and halving again at time 75. D-SPTF and Central SPTF adapt best to the transient 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. The 95% confidence intervals for the results were all within 40 microseconds.

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 suffered 50% degradation and remained that way until the end of the experiment. The workload that was used to evaluate the system was the SAP trace. I measured the client-side response time at specific time intervals to judge the effectiveness of each scheme.

Figure 6.16 shows the results. The average response time of both D-SPTF

and Centralized SPTF, in non-degraded mode, is around 10ms per request. The response time of both hashing and LARD are around 13.75ms. This 37% difference in 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 11.2ms while Hashing/LARD's response times increase to 17.3ms. This 55% 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 the change because the degradation does not continue long enough to trigger rebalancing of the system given the default parameters. With a more adaptive parameters, LARD is able to adapt quickly to the load imbalance caused by the transient heterogeneity. But, by increasing adaptability, cache performance is negatively impacted. 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 or hurting cache hit rates.

6.6 Little's Law Check

To verify the simulated results of D-SPTF, I applied Little's law for closed systems: $N = X * E[Ts]$. Where N is the total number of jobs the system, X is the

	2-Way replication	4-Way replication	6-Way replication	8-Way replication
Requests(N)	16	16	16	16
Throughput(X)	397 IO/s	799 IO/s	1204 IO/s	1595 IO/s
Calculated Response($E[Ts]$)	40.3ms	20.02ms	13.29ms	10.03ms
Measured Response	40.35ms	20.04ms	13.32	10.07ms

Table 6.6: **Little's Law verification of D-SPTF system.**

throughput, and $E[Ts]$ is the expected service time of each request. Using Little's Law, I calculated the expected service time of each request ($E[Ts] = N/X$) and compared it with the observed service time of each request. I used the baseline experiment to demonstrate that the observed results agreed with the computed results from Little's Law as shown in Table 6.6.

6.7 Conclusions

The experiments show that D-SPTF meets all our goals. D-SPTF provides both long-term and transient load balancing, preventing bricks from experiencing up to 35% idle during periods of low queue depth. D-SPTF also provides significantly better disk scheduling efficiency than the other request distribution algorithms. This improved scheduling efficiency of D-SPTF allows for a 50% increase in media performance. In addition to these benefits, D-SPTF maintains effective caching in the system, without causing the significant decreases in cache effectiveness that occur with the Adaptive LARD algorithm. All of these features are gained with only 1.2%-20% additional bandwidth, depending upon the average size of data

requests.

7 D-SPTF Design Decisions

Chapter 6 evaluates D-SPTF assuming one set of design choices being made. However, those choices are not the only possible options. This chapter explores three design decisions for D-SPTF protocol instantiation. First, the mechanisms for resolving request CLAIM conflicts will be evaluated, comparing both the conservative and optimistic approaches introduced in Chapter 4. The goal is to identify the scheme that scales best with increasing request latency. Second, the choice of disk scheduling algorithm will be evaluated. It will be determined how much of the performance advantage of D-SPTF comes from SPTF scheduling, versus SSTF or CLOOK scheduling. If much of the performance improvement of D-SPTF can be achieved using SSTF or CLOOK, then it may be easier to adapt this approach to existing systems. Third, a new request abort mechanism will be compared against the default abort mechanism to determine the increase in conflicts that would arise from aborting requests after they have been completed, rather than before they are issued. The advantage of aborting requests after they complete at a device is that it will minimize the device changes required to implement D-SPTF.

7.1 Conflict Handling

One of the substantial challenges with developing a distributed replica selection protocol is deciding the best method to handle conflicts when two or more bricks decide to service the same request at the same time. Many schemes (e.g., hashing and LARD) don't worry about this problem since each brick sees an exclusive subset of all requests. In D-SPTF, however, all bricks in a replica group see all requests and so the potential for conflicts exists. When conflicts occur, bricks should be able to resolve them to prevent duplicate work from occurring too frequently.

Two solutions are proposed to address such conflicts: a pessimistic one and an optimistic one. The first approach, referred as prescheduling, is pessimistic. It assumes that conflicts might occur frequently. All experiments in Chapter 6 used this approach. It works by coordinating requests before scheduling. Specifically it selects a request, sends a CLAIM message, and then waits two network latencies to receive responses. This approach results in good overall performance if the disk response time is greater than network delay but erodes performance as network latency approaches average response time.

A more optimistic approach, which is called "latency scheduling", is to assume that conflicts do not occur often, so full prescheduling may be unnecessary. Instead of prescheduling, a request is selected when the previous request completes, and the scheduler waits for however long the rotational latency for that request would be to receive conflicting CLAIM messages. The advantage of this approach

is that no additional request latency is added to wait for potential messages. The disadvantage is that, if network latency exceeds rotational latency, then two bricks could service the same request.

To compare the effectiveness of these two approaches, three attributes are measured: the latency of requests to the client, the number of conflicts and the number of duplicates.

For this evaluation, a synthetic workload was used.. This workload has 66% reads, a Zipf distributed set of request sizes with a mean of 4K. The requests accessed randomly selected blocks uniformly chosen from the LBN space. Caches are disabled to test the two types of protocols. For this experiment, the simulations vary the interbrick network latencies between 0-5ms to evaluate how effectively each approach tolerates increasing network latency.

Figure 7.1 shows that, when network latency is under 1ms, each approach provides the same performance. However, above 1ms, latency scheduling tolerates latency better than prescheduling. Prescheduling forces each request to wait at least two network latencies, even if the service time is shorter, whether there is a conflict or not. Latency scheduling only suffers if conflicts and duplicate work actually occur. Since duplicate work does not occur for every request, latency scheduling tends to perform better at high network latencies.

This is seen by examining Figure 7.2. One sees that the percentage of requests that experience conflicts increases as network latency increases. What one sees is that prescheduling encounters few conflicts (max 1.5%) in the system, because

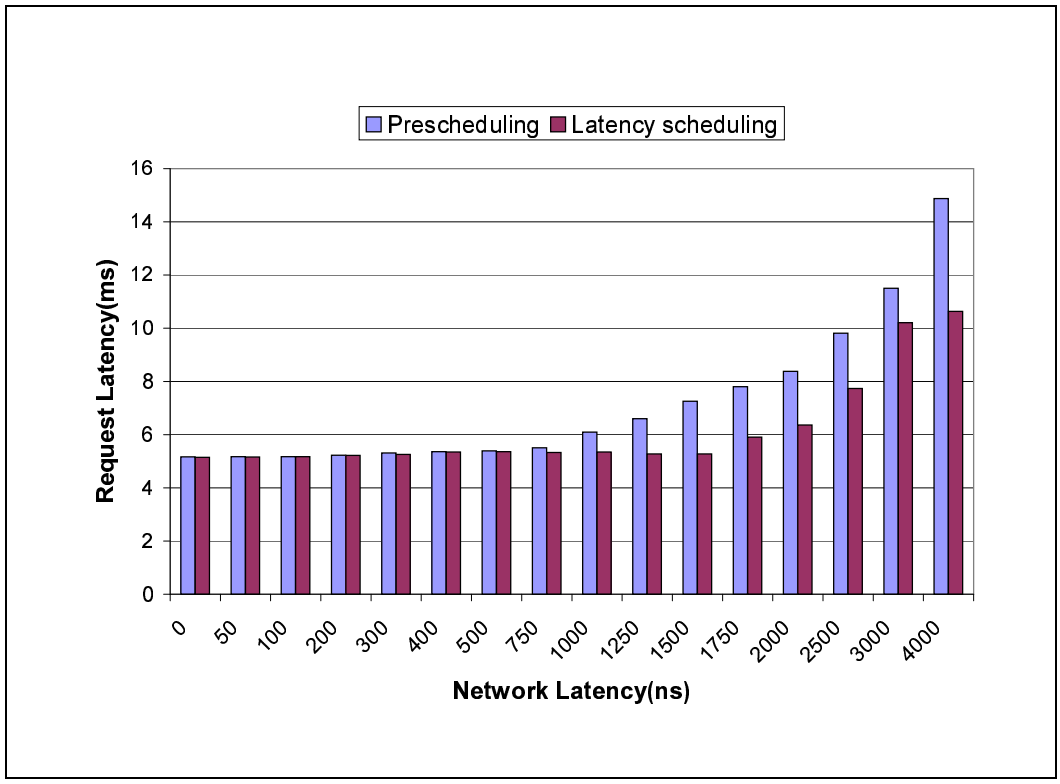

Fig. 7.1: **Request latency as a function of network latency.** The data shown here evaluates the impact of increasing network latency on both prescheduling and latency scheduling. As one can see, up until 1 ms of network latency, there is little difference in the two protocols. As network latency increases latency scheduling performs increasingly better. The 95% confidence intervals for the results were all within 30 micorseconds.

it throttles progress, for network latencies up to 4ms. Latency scheduling results in many more conflicts, up to 43% of all request conflicting at 4ms. The reason that latency scheduling has 43% of all requests conflicting is that, once two bricks service the same request, they can continue to service the same requests. We call this a conflict cycle. A conflict cycle is defined as when two or more bricks continuously select the same requests to service. Prescheduling prevents conflict cycles by preventing two or more bricks from servicing the same request. Latency

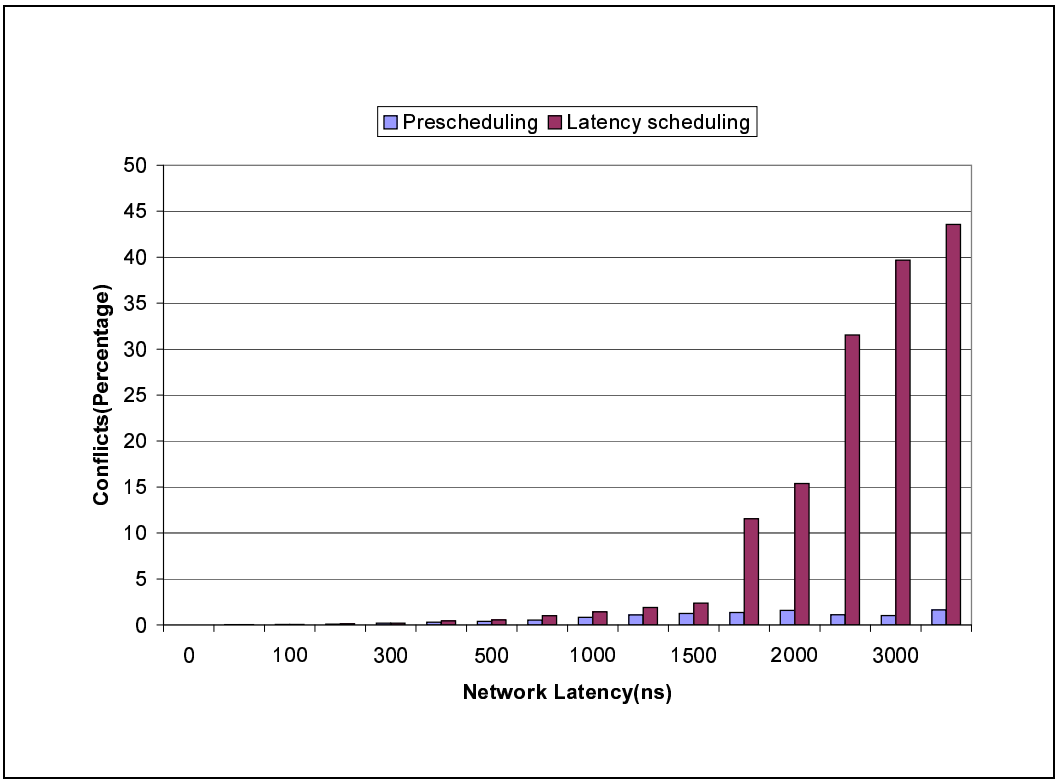

Fig. 7.2: **Conflicts as a function of network latency.** This graph shows the increasing number of conflicts as network latency increases. Clearly, prescheduling results in the fewest number of conflicts

scheduling will allow transient conflict cycles to exist, but will break out of them, eventually, so long as the rotational latency time period is less than twice the network latency.

Additional insight is provided by examining duplicate request rates in Figure 7.3. As expected, prescheduling never results in duplicate requests. On the other hand, latency scheduling has almost half its requests as duplicate during high network latencies, since each conflict results in duplicated disk access and client response. This results in wasting almost half of the bandwidth of the sys-

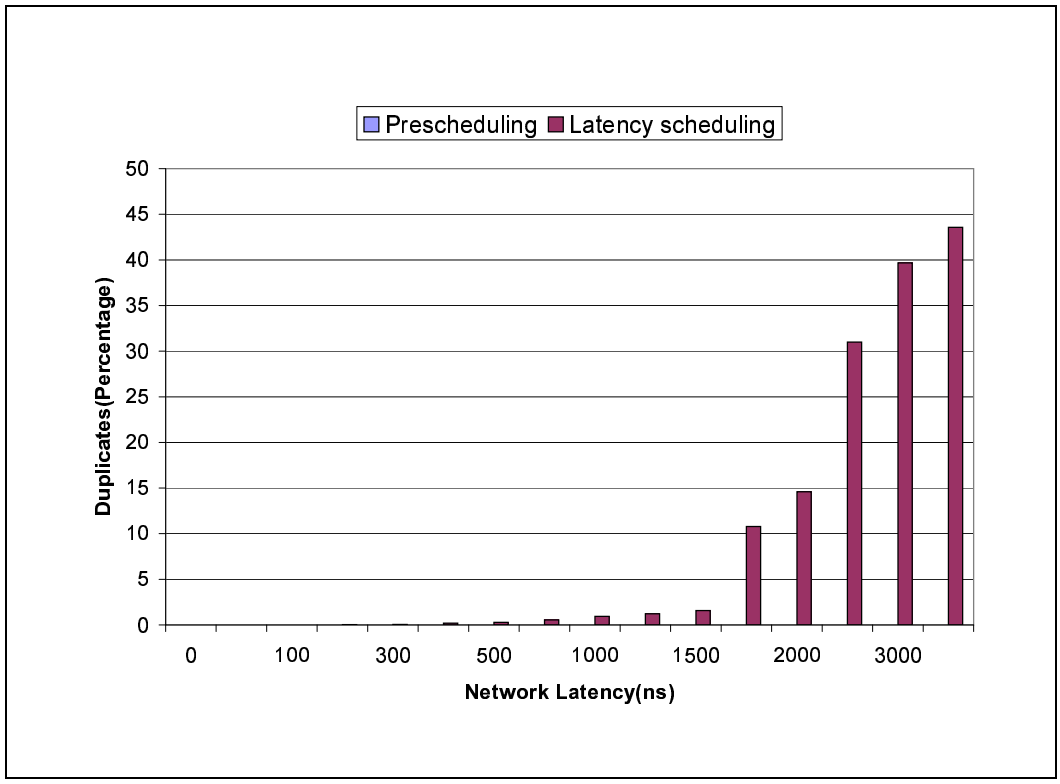

Fig. 7.3: **Duplicates as a function of network latency.** This graph shows the increasing number of duplicates as network latency increases. Prescheduling results in no duplicate requests while, latency scheduling's number of duplicate requests increase considerably as latency increases.

tem.

These results suggest that the optimistic approach, in its purest form, may not be the best approach because it suffers from a significant number of conflicts. However, the majority of these conflicts are due to the system not being able to break out of a conflict cycle once one starts. Thus, a continuous cycle of conflicts occurs. To provide better conflict resolution, rotational latency scheduling was modified so that, if duplicate work occurred, the brick that should not have scheduled the request would schedule its next request by selecting a random request from its

queue. This allows for the conflicting bricks to select different requests for the next access. When a different request is selected, the conflict cycle is broken. This provides a proactive method to break out of conflict cycles quickly and results in significant latency tolerance.

Figure 7.4 shows that rotational latency scheduling with random selection has better network latency tolerance than both prescheduling and regular rotational latency scheduling. This is due to its lack of duplicate requests. Figure 7.5 shows that, by adding the random conflict resolution, duplicate requests are reduced by 10x. In fact, this enhancement reduces duplicates from 43% at 4ms to 4.6%.

With the enhanced optimistic scheme, D-SPTF functions well in high latency environments. Overall, one can see that rotational latency scheduling with random conflict resolution results in good overall performance for a variety of network latencies.

7.2 Beyond SPTF

One of the remaining questions with D-SPTF is how much of the performance benefit is derived from using SPTF scheduling at the device. How much performance could be lost by using alternative local request scheduling algorithms, such as SSTF or CLOOK, which are simpler to implement? Fortunately, the D-SPTF protocol is designed to be modular and changing local schedulers is easy.

To quantify how much performance is gained due to SPTF scheduling, I compare D-SPTF to a Distributed Shortest Seek Time First (D-SSTF) and a Dis-

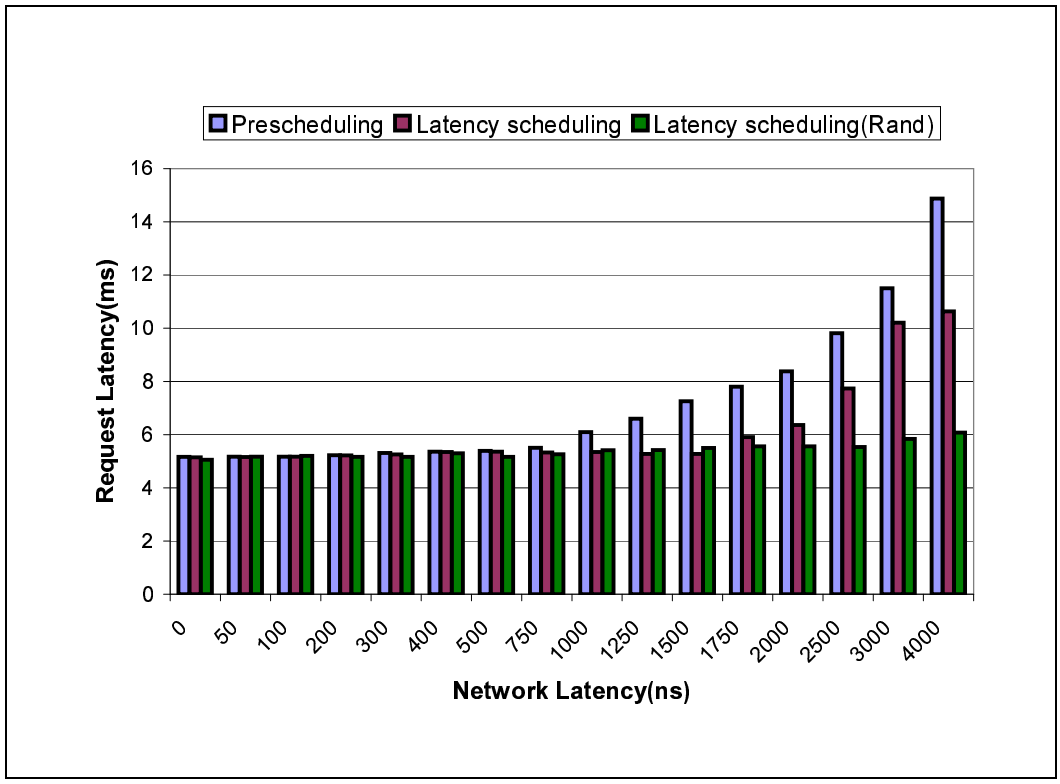

Fig. 7.4: **Request latency as a function of network latency.** The data shown here evaluates the impact of increasing network latency on prescheduling, latency scheduling, and random backoff of latency scheduling. As one can see, up until 1 ms of network latency, there is little difference in the three protocols. As network latency increases latency scheduling performs increasingly better. However adding random backoff to latency scheduling results in a scheme that tolerates network latency very well. The 95% confidence intervals for the results were all within 40 micorseconds.

tributed CLOOK algorithm. The expectation is that D-SPTF should have the best performance, followed by D-SSTF and then D-CLOOK.

To evaluate the performance impact, I use a synthetic workload. This workload operates in closed loop with zero think-time. The LBNs are drawn uniformly from the LBN space, and the request sizes are drawn from a Zipf distribution with a 4KB average. The requests are 67% reads and the queue depth is varied from 2-32 requests.

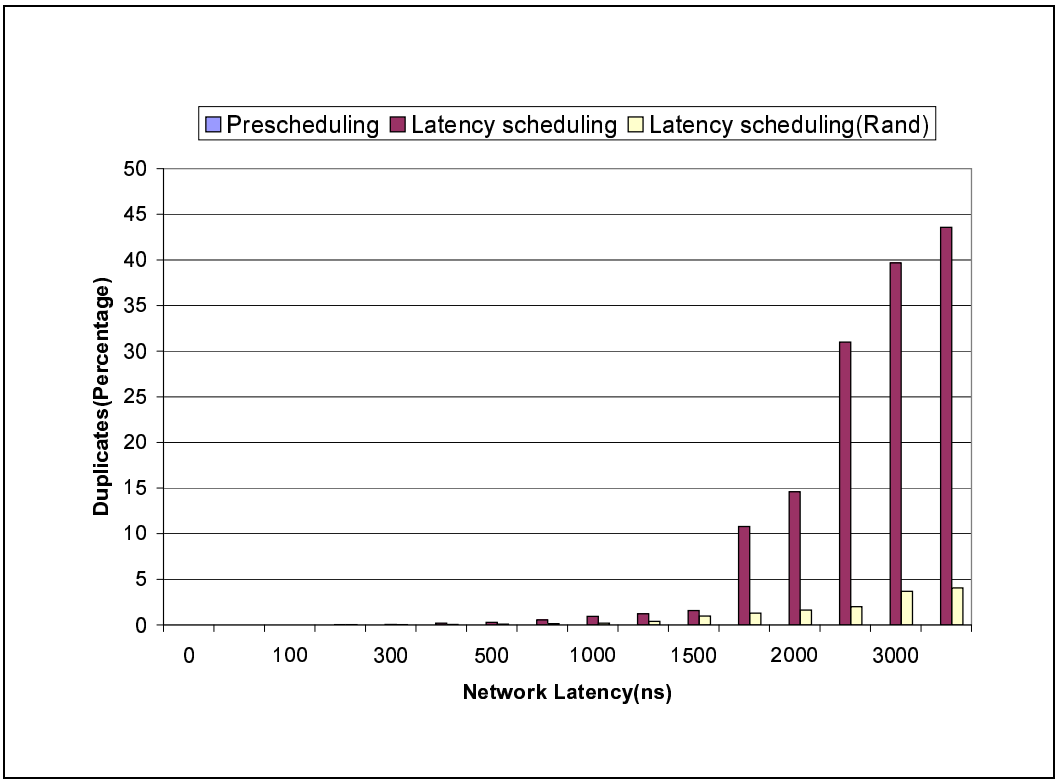

Fig. 7.5: **Duplicates as a function of network latency.** This graph shows the increasing number of duplicates as network latency increases. Prescheduling results in no duplicate requests while, latency scheduling's number of duplicate requests increase considerably as latency increases. Adding random backoff to latency scheduling results in a dramatic reduction in duplicate requests.

Figure 7.6 shows that the different algorithms performed as expected. D-SPTF had the best performance advantage, followed by D-SSTF and D-CLOOK. D-SPTF performs on average 20% better than D-SSTF. Even though D-SSTF performs worse than D-SPTF, D-SSTF does perform better than hashing based request distribution. In environments with dynamic heterogenous systems, D-SSTF can be an advantage over using traditional request distribution schemes.

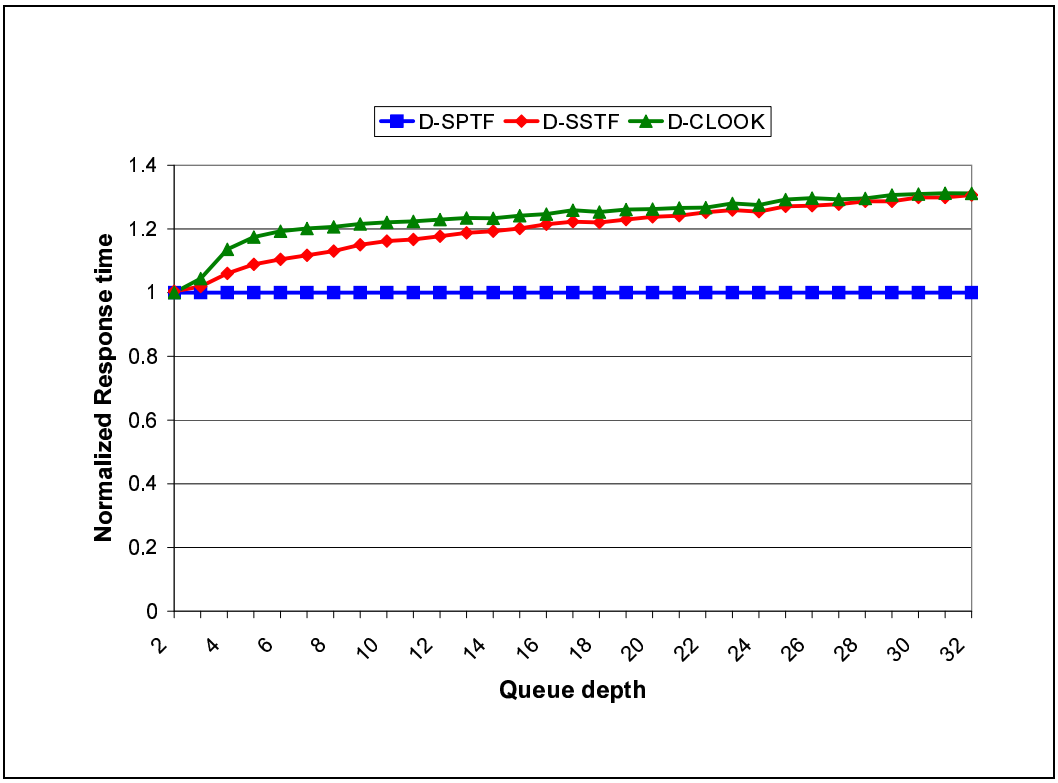

Fig. 7.6: **Client latency comparison of protocols, varying number of replicas.** As expected, the performance difference of the three protocols is similar at low queue depths but increased considerably as the queue depths increase. The 95% confidence intervals for the results were all within 35 microseconds.

7.3 Post request abort

One of the challenges with implementation of D-SPTF in a real system is convincing device manufactures to provide interfaces that will allow full realization of the D-SPTF protocol. To implement D-SPTF in a real system, one would need two commands, an abort command and a scheduled request command. The abort command would be able to remove a request from the queue. This abort command is used to prevent duplicate work by removing requests completed by other bricks.

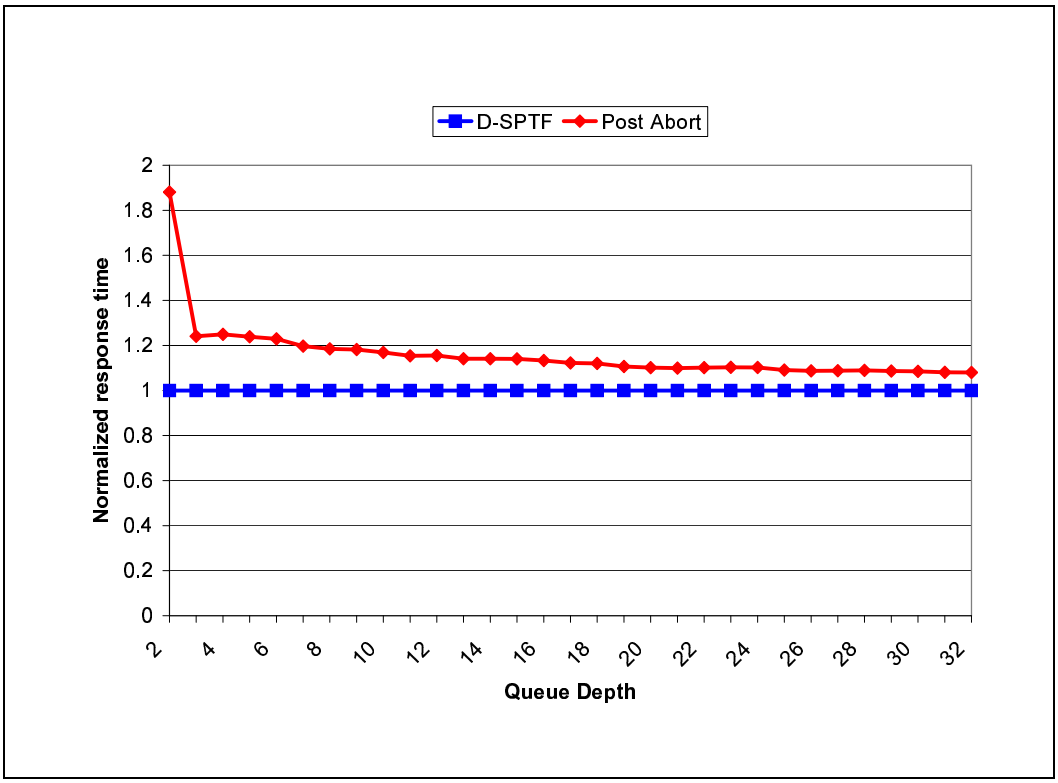

Fig. 7.7: **Latency comparison D-SPTF versus Post Abort D-SPTF, varying queue depth.** One can see that, as queue depth increases, the performance difference of D-SPTF and post abort maintains rough similarity. The 95% confidence intervals for the results were all within 45 microseconds.

The scheduled request command would be needed so that devices can be asked to schedule the request in advance and reveal how long it will take. This command could be used to construct claim messages when a request is scheduled. No current disk interface contains both required commands. As a result, both commands would be needed to be added to disk interfaces before D-SPTF could be fully implemented without an outside-the-disk scheduler.

As an interim measure, one can use the existing SCSI protocol since it has an abort request command. Implementing D-SPTF using only an abort command

is not the ideal situation and will require some changes to the D-SPTF protocol. Currently, the D-SPTF protocol aborts request at other bricks before a request is sent to a disk to be serviced. However, since no schedule command exists to inform a brick which request a local disk scheduler will service next, the brick will only know what request a disk serviced after that disk completes the request. As a result, the brick can only send an abort message to the other members of its replica group after a request has been serviced. This is a post completion abort, called "post abort" for short, instead of a prescheduling abort. This post abort increases potential for conflicts and duplicate work to occur. This is because the window of vulnerability when two bricks can service the same requests expands to include the time it takes to service the request.

The question that must answered is, how much performance is lost by post aborting due to the increased conflict potential? To answer this question, the simulation environment was changed to abort requests after they completed rather than before they were issued to the device.

To evaluate the performance impact, a synthetic workload was used. This workload operated in closed loop format with zero think-time. The LBNs were drawn uniformly from the LBN space and the request size was drawn from a Zipf distribution with a 4KB average. The requests were read only and the queue depth was varied from 2-32 requests.

Figure 7.7 shows the comparison of traditional D-SPTF with "post abort". One can see that the performance of "post abort" is between 8-25% lower than

traditional D-SPTF, except at a queue depth of 2. This is due to the increased number of conflicts observed by "post abort". The problem with "post abort" is that the window for conflicts is much larger, increasing their frequency. In addition, by the time that a conflict has been detected, the next request has been scheduled. This will result in a second conflict occurring after the first. Only after these two conflicts occur can the cycle be broken.

Even though "post abort" does perform worse than D-SPTF, its performance is still superior to the performance of traditional approaches like hashing-based request distribution. This suggests that, while "post abort" may not be optimal, it could be used to gain some of the benefits of D-SPTF with minimal implementation costs.

8 Conclusions

D-SPTF demonstrates that,

With reasonable communication costs, a decentralized storage protocol can utilize storage resources (spindles, cache and capacity) nearly as efficiently as a centralized storage system.

To validate this claim, this dissertation demonstrated the following five properties of D-SPTF.

(1) In Section 6.1, I demonstrate that D-SPTF was able to balance load as effectively as an idealized centralized system and 37% better than existing request distribution algorithms. D-SPTF is able to load balance so effectively because it does not partition requests amongst replicas that can service them. Instead, all replicas see all requests and work is performed in the replica that can service it first. Other decentralized schemes partition requests among the replicas, which will periodically cause transient load imbalances.

(2) In Section 6.2, I show that D-SPTF achieves disk scheduling efficiency that is equivalent to an idealized centralized system and 67% better than exist-

ing request distribution schemes. Because D-SPTF sends all requests to all replicas, the replicas to have much larger disk queues to schedule with. This improves the disk scheduling efficiency of the replicas and decreases the media response time. This efficiency is gained without increasing the queuing time of requests and without negatively impacting load balancing. Unlike other decentralized request distribution protocols, D-SPTF does not have to make a compromise between scheduling efficiency and load balancing.

(3) In Section 6.3, I demonstrate that D-SPTF matches the cache performance of existing request distribution schemes and approaches the performance of the idealized centralized system. The reason that D-SPTF does not match the performance of the centralized system is because it is not working with one single cache. Instead, it works with the decentralized per brick caches with only a small number that function cooperatively with each other.

(4) In Section 6.5, I demonstrate that D-SPTF automatically adapts to heterogenous systems as well as the idealized centralized system. D-SPTF performs better than existing request distribution schemes because it can make a separate choice for each IO request. That choice is made as late as possible and done so with as much global knowledge as possible. This ensures maximum performance. Competing schemes don't adapt on a per-request basis.

(5) In Section 6.4, I show that, while the D-SPTF protocol does increase the communication costs of brick-based systems, these costs are not large enough to

preclude the implementation of the D-SPTF protocol with modern networking hardware.

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. 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 is often able to match the performance of an idealized centralized system (assuming equivalent aggregate resources) and exceeds the performance of LARD and decentralized hash-based schemes.

Bibliography

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.

Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, **15**(1):54–64, February 1995.

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.

Henry G. Baker. The Treadmill: real-time garbage collection without motion sickness. *SIGPLAN Notices*, **27**(3):66–70, March 1992.

Dina Bitton and Jim Gray. Disk shadowing. *International Conference on Very Large Databases* (Los Angeles, CA), pages 331–338, August 1988.

D. Bitton. Arm scheduling in shadowed disks. *IEEE Spring COMPCON* (San Francisco, CA, 27 February–30 March 1989), pages 132–136. IEEE, 1989.

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.

J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. *ACM Symposium on Operating System Principles* (Asheville, NC), pages 120–133, 5–8 December 1993.

Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, **26**(2):145–185, June 1994.

IBM Almaden Research Center. Collective Intelligent Bricks, August, 2003. http://www.almaden.ibm.com/StorageSystems/autonomic_storage/C IB/index.shtml.

Cisco Systems Inc. LocalDirector. www.cisco.com.

E. G. Coffman, L. Klimko, and B. Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal of Computing*, **1**(3):269–279, September 1972.

E. G. Coffman, Jr and Micha Hofri. On the expected performance of scanning disks. *SIAM Journal on Computing*, **11**(1):60–70, February 1982.

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.

Peter J. Denning. Effects of scheduling on file memory operations. *AFIPS Spring Joint Computer Conference* (Atlantic City, New Jersey, 18–20 April 1967), pages 9–21, April 1967.

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.

Y. Dishon and T. S. Liu. Disk Dual Copy Methods and Their Performance. *International Symposium on Fault-Tolerant Computing (FTCS '88)* (Tokyo, JPN, June 1988), pages 314–319. IEEE Computer Society Press, 1988.

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

PeerStorage Overview, 2003. http://www.equallogic.com/pages/products_technology.htm.

Mike Feeley. Methods and models for management of distributed persistent data. Department of Computer Science and Engineering, University of Washington, Seattle, 10th January 1995.

Edward W. Felten and John Zahorjan. *Issues in the implementation of a remote memory paging system.* 91-03-09. Department of Computer Science and Engineering, University of Washington, March 1991.

H. Frank. Analysis and optimization of disk storage devices for time-sharing systems. *Journal of the ACM*, **16**(4):602–620, October 1969.

Michael J. Franklin, Michael J. Carey, and Miron Livny. *Global memory management in client-server DBMS architectures.* Computer Science Technical Report 1094. University of Wisconsin–Madison, June 1992.

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.

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.

Robert Geist and Stephen Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems*, **5**(1):77–92. ACM Press, February 1987.

Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *Decentralized storage consistency via versioning servers.* Technical Report CMU–CS–02–180. Carnegie Mellon University, September 2002.

Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *A protocol family for versatile survivable storage infrastructures.* Technical report CMU-PDL-03-103. CMU, December 2003.

C. C. Gotlieb and G. H. MacEwen. Performance of movable-head disk storage systems. *Journal of the ACM*, **20**(4):604–623, October 1973.

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

M. Hofri. Disk scheduling: FCFS vs. SSTF revisited. *Communications of the ACM*, **23**(11):645–653, November 1980.

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

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.

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.

IBM Corporation. IBM interactive network dispatcher. http://www.ics.raleigh.ibm.com/ics/isslearn.htm.

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.

F. D. Lawlor. Efficient mass storage parity recovery mechanism. *IBM Technical Disclosure Bulletin*, **24**(2):986–987, July 1981.

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.

Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *Design and implementation of the 4.3BSD UNIX operating system.* Addison-Wesley, 1989.

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

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.

Naraig Manjikian and Tarek S. Abdelrahman. Array Data Layout for the Reduction of Cache Conflicts. *8th International Conference on Parallel and Distributed Computing Systems*, 1995.

Timothy Mann, Andy Hisgen, and Garret Swart. *An algorithm for data replication.* Report #46. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, June 1989.

Norman S. Matloff. A Multiple-Disk System for both Fault Tolerance and Improved Performance. *IEEE Transactions on Reliability*, **R-36**(2):199–201. IEEE Computer Society Press.

James Roche Jai Menon and James M. Kasson. Floating Parity and Data Disk Arrays. *Journal of Parallel and Distributed Computing*, **17**(1-2):129–139.

James H. Morris, Mahadev Satyanarayanan, Michael H. Connor, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, **29**(3):184–201, March 1986.

Spencer W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, **40**(1):22–30. IEEE, January 1991.

W. Oney. Queueing analysis of the scan policy for moving-head disks. *Journal of the ACM*, **22**(3):397–412, July 1975.

Cyril U. Orji and Jon A. Solworth. Doubly distorted mirrors. *ACM SIGMOD International Conference on Management of Data* (Washington, DC), pages 307–316, May 1993.

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). Published as *SIGPLAN Notices*, **33**(11):205–216. ACM, 3–7 October 1998.

Arvin Park and K. Balasubramanian. *Providing fault tolerance in parallel secondary storage systems.* CS–TR–057–86. Department of Computer Science, Princeton University, November 1986.

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, 1–3 June 1988), pages 109–116, 1988.

David A. Patterson, Peter Chen, Garth Gibson, and Randy H. Katz. Introduction to redundant arrays of inexpensive disks (RAID). *IEEE Spring COMPCON* (San Francisco, CA), pages 112–117, March 1989.

Christos A. Polyzois, Anupam Bhide, and Daniel M. Dias. Disk mirroring with alternating deferred updates. *International Conference on Very Large Databases* (Dublin, Ireland, 24–27 August 1993), pages 604–617. Morgan Kaufmann, 1993.

Titus Douglas Mahlon Purdin. *Enhancing file availability in distributed systems (the saguaro file system).* PhD thesis.

Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 405–420, 1993.

Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.

M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, **7**(3):247–280. ACM Press, August 1989.

B. N. Schilit and D. Duchamp. *Adaptive remote paging for mobile computers.* Technical report. Department of Computer Science Columbia University, New York, U.S., 1991.

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

Margo Seltzer. A comparison of data manager and file system supported transaction processing. Computer Science Div., Department of Electrical Engineering and Computer Science, University of California at Berkeley, January 1990.

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

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

T. J. Teorey and T. B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, **15**(3):177–184, March 1972.

G. Winfield Treese. Berkeley UNIX on 1000 Workstations : Athena changes to 4.3 BSD. *USENIX Winter Conference* (Dallas, TX, USA, January 1988), pages 175–182, 1988.

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.

Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Theil. The LOCUS distributed operating system. *ACM Symposium on Operating System Principles* (Bretton Woods, New Hampshire). Published as *Operating Systems Review*, **17**(5):49–70, October 1983.

Randolph Y. Wang and Thomas E. Anderson. xFS: a wide area mass storage file system. *Workshop on Workstation Operating Systems* (Napa, CA, 14–15 October 1993), pages 71–78. IEEE Computer Society Press, 1993.

N. C. Wilhelm. An anomaly in disk scheduling: a comparison of FCFS and SSTF seek scheduling using an empirical model for disk accesses. *Communications of the ACM*, **19**(1):13–17, January 1976.

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.

C. K. Wong. *Algorithmic studies in mass storage systems.* Computer Science Press, 11 Taft Court, Rockville, MD 20850, 1983.

Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada), pages 146–156, May 1995.

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.

George Kingsley Zipf. *Psycho-Biology of Languages.* Houghton-Mifflin.