11-2003

D-SPTF: Decentralized Request Distribution in Brick-based Storage (CMU-CS-03-202)

Christopher R. Lumb
Carnegie Mellon University

Gregory R. Ganger
Carnegie Mellon University

Richard Golding
Carnegie Mellon University

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

Christopher R. Lumb, Gregory R. Ganger, Richard Golding*

November 2003
CMU-CS-03-202

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

*IBM Almaden Research

Abstract

Distributed Shortest-Positioning Time First (D-SPTF) is a request distribution protocol for decentralized systems of storage servers. D-SPTF exploits high-speed interconnects to dynamically select which server, among those with a replica, should service each read request. In doing so, it simultaneously balances load, exploits the aggregate cache capacity, and reduces positioning times for cache misses. For network latencies 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 existing decentralized approaches, such as hash-based request distribution, D-SPTF achieves up to 65% higher throughput and adapts more cleanly to heterogeneous server capabilities.

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

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

1 Introduction

Many envision enterprise-class storage systems composed of networked "intelligent" *storage bricks* [8, 9, 10, 14]. Each brick consists of a few disks, RAM for caching, and CPU for request processing and internal data organization. Large storage infrastructures could have hundreds of storage bricks. The storage analogue of cluster computing, brick-based systems are promoted as incrementally scalable and (in large numbers) cost-effective replacements for todays high-end, supercomputer-like disk array systems. Data redundancy across bricks provides high levels of availability and reliability (à la the RAID arguments [20]), and the aggregate resources (e.g., cache space and internal bandwidth) of many bricks should exceed those of even high-end array controllers.

An important challenge for brick-based storage, as in cluster computing, is to effectively utilize the aggregate resources. Meeting this challenge requires spreading work (requests on data) across storage bricks appropriately. In storage systems, cache hits are critical, because they involve orders of magnitude less work and latency than misses (which go to disk). Thus, it is important to realize the potential of the aggregate cache space; in particular, data should not be replicated in multiple brick caches. During bursts of work, when queues form, requests should be spread across bricks so as to avoid inappropriate idleness and, ideally, so as to reduce disk positioning costs [5, 25]. Achieving these goals is further complicated when heterogeneous collections of bricks comprise the system. In traditional disk array systems, all of these features can be provided by the central disk array controller.

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

D-SPTF provides the desired load distribution properties. During bursts, all bricks with relevant data will be involved in processing of requests, contributing according to their capabilities. Further, when scheduling its next action, a brick can examine the full set of requests for data it stores, using algorithms like Shortest-Positioning-Time-First (SPTF) [15, 22]. Choosing from a larger set of options significantly increases the effectiveness of these algorithms, decreasing positioning delays and increasing throughput. For example, in a brick-based system keeping three replicas of all data (e.g., as in FAB [9]), D-SPTF increases throughput by 12–27% under heavy loads. The improvement increases with the number of replicas.

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

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

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

2 Brick-based storage systems

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

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

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

Dividing the system into 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 small amount of information for making decisions. We focus here on three issues made more difficult by this lack of global information: head scheduling, cache utilization, and inter-brick load balancing. Existing mechanisms address one or two of these problems at the expense of the others. Each of these problems is compounded when the population of bricks is heterogeneous. The D-SPTF protocol addresses these problems by exploiting the high-speed networks expected in brick-based systems, allowing bricks to loosely coordinate their local decisions.

2.1 Head scheduling

Disk drives are difficult to schedule effectively. If the sequence of operations performed by a drive is not ordered carefully (e.g. when using FIFO scheduling), the drive will spend almost all its time seeking and waiting for the media to rotate into position, resulting in low performance. The importance of scheduling well continues to grow as the density of data on media increases: the time spent transferring a block of data off media decreases, while the disk rotation and head positioning speeds are not increasing as quickly.

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

Brick-based systems tend to distribute work over many bricks, which decreases the average queue length at each brick. This, in turn, gives less opportunity for scheduling the disk head well. Avoiding this requires increasing the queue depths at bricks making scheduling decisions. One way to do this is to direct requests to only a few bricks; this will 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 the 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 response latency, overall system throughput is the same as a single brick.

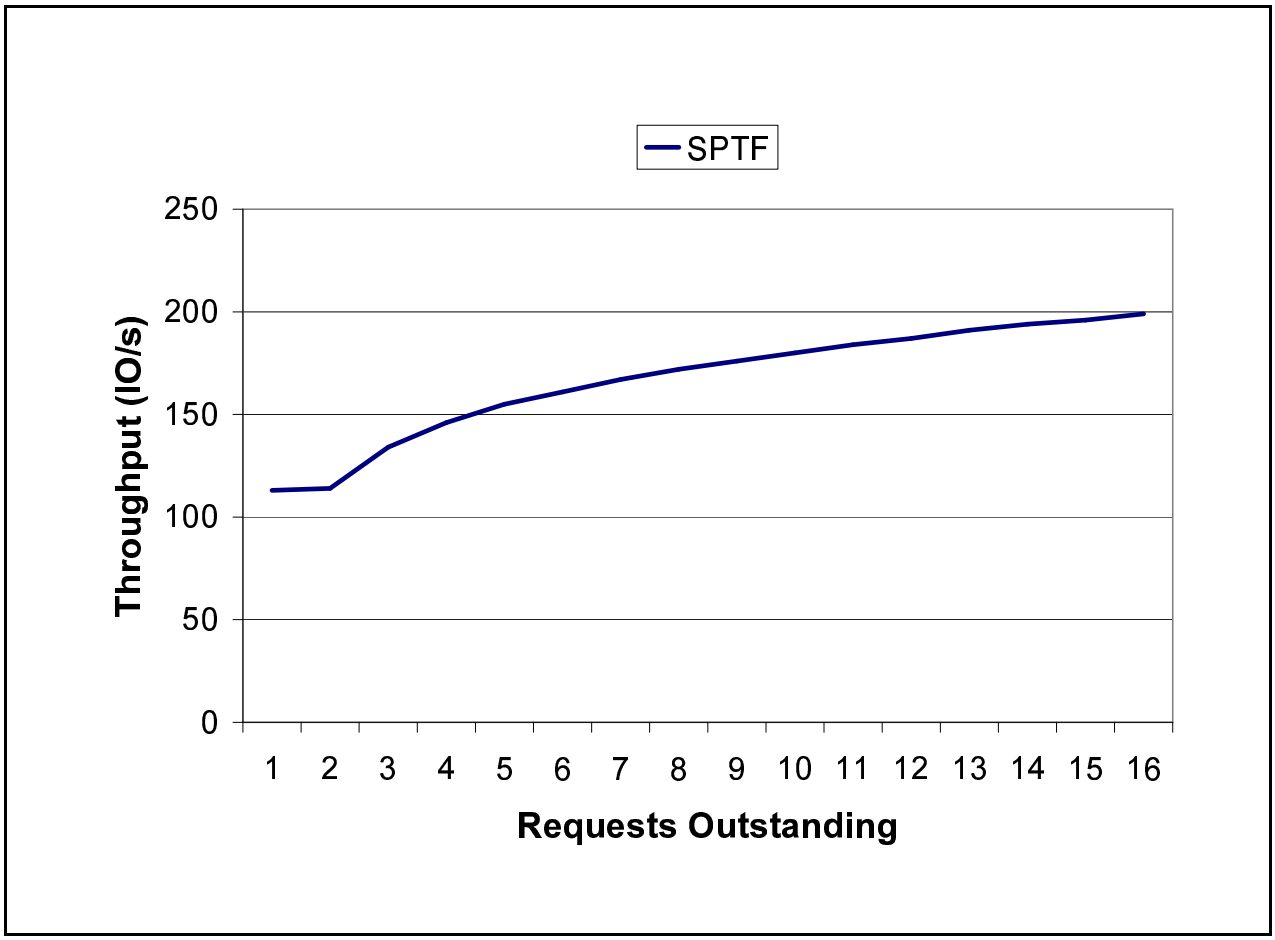

Figure 1: **Disk throughput with SPTF scheduling, as a function of queue depth.** The data shown are for a closed synthetic workload (see Section 5) with a constant number of pending small requests to random locations on a Quantum Atlas 10K disk.

2.2 Load balancing

When there is a choice of where data can be read from, one usually wants to balance load.[1] Centralized systems can do this because they know, or can estimate, the load on each disk. For example, the AutoRAID system directs reads to the disk with the shortest queue [24].

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

However, this is not as good as a centralized system can do: spreading requests gives balance only over the long term, but bursts of traffic can cause transient imbalances. Moreover, different kinds of bricks in the system makes this problem more difficult. With heterogeneous bricks, request distribution algorithms must try to route more requests to faster bricks and fewer to slower ones—where "slower" and "faster," of course, depend on the interaction between the workload, the amount of cache, and the specific disk models that each brick has.

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

2.3 Exclusive caching

Maximizing the cache hit rate is critical to good storage system performance. Hence, the cache resources must be used as efficiently as possible. The cache resources in a brick-based system are divided into many small caches, and replacement decisions are made for each cache independently. This independence can work against hit rates. In particular, the system should generally keep only one copy of any particular data block in cache. 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 cache in the system. Always reading a particular data item from one brick will solve this problem at the cost of dynamic load balancing and dynamic head scheduling.

2.4 Achieving all three at once

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

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

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

D-SPTF also naturally handles heterogeneous brick populations. Under light load, the fast disks will tend to be read from and slow disks will not, while writes are processed everywhere. Under heavy load, when all bricks can have many requests in flight, response time will be determined by the utilization of the brick, and work will be distributed proportional to the speed.

3 The D-SPTF protocol

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

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

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

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

When it comes time to select a request for a disk to service, a brick scans its queue and selects the request that the disk can service with the shortest positioning time (i.e., each brick locally uses SPTF scheduling).

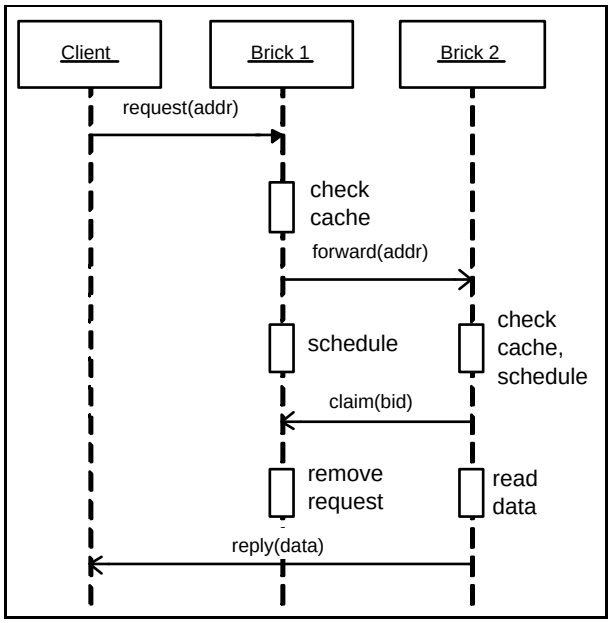

Figure 2: **Read operation in D-SPTF.** The client sends a read request to one brick, which forwards it to others. The brick that can schedule the read first aborts the read at other bricks, and responds to the client.

If the request is a read, the brick then sends a CLAIM message to the other bricks with replicas so that they can remove the request from their queues.

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

Once a request completes at a brick, that brick returns the data to the client. If a brick fails after claiming a read, but before returning the data to the client, the request will be lost. To handle such cases, we assume that clients will timeout and retry.

The write protocol (Figure 3) is different. When a brick receives a write request from a client, it immediately forwards the request to all the other bricks with a replica of the data. When a brick receives a write request, either directly or forwarded, the brick immediately stores the data in its local NVRAM cache. Bricks that receive a forwarded write request send an acknowledgment back to the first brick when the data is safely stored; the first brick waits until it has received acknowledgments from all other replicas, then sends an acknowledgment back to the client. If the original brick does not hear from all bricks quickly enough, some consistency protocol must address the potential brick failure. We believe that the basic D-SPTF protocol can work well with many consistency protocols (e.g., [1, 9, 11]).

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

3.1 Concurrent CLAIM messages

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

D-SPTF avoids this problem by pre-scheduling and waiting for a short period (two times the one-way

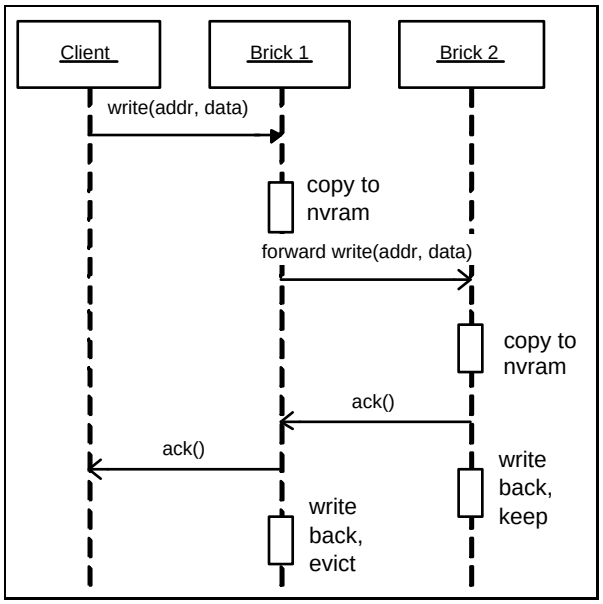

Figure 3: **Write operation in D-SPTF.** The client sends a write request to one brick, which forwards it to others. Each brick determines whether it is the one to keep the data in cache.

network latency) after sending the CLAIM message. Assuming a known bound on network latency, waiting ensures that every brick 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 be chosen. To enable this decision, each CLAIM message includes a *service time bid* (the SPTF-predicted positioning time); with this information, each brick can decide for itself which one will service the request. Our current approach is for the request to be serviced by whichever brick submits the lowest service time bid, ignoring when CLAIM messages were sent. This will work well for high-speed networks, but may induce inefficiency when network latencies are significant fractions of positioning times.

With pre-scheduling, 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 only wait for competing CLAIM messages as long as the expected rotational latency before issuing the request to disk. As illustrated in Figure 4, this does not impact performance, until network latency is a substantial fraction of rotational latency, since the brick is effectively shifting when it waits the rotational latency to before the seek instead of after.

4 Experimental setup

We use simulation to evaluate D-SPTF. The simulation is event driven, and uses the publically-available DiskSim disk models [7] to simulate the disks within bricks. The DiskSim simulator accurately models many disks [3], including the Quantum Atlas 10K assumed in our system model.

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

Each brick contains a request queue. Whenever the disk is about to become idle, the brick scans the request queue and selects the request with the shortest positioning time as the next request for the disk.

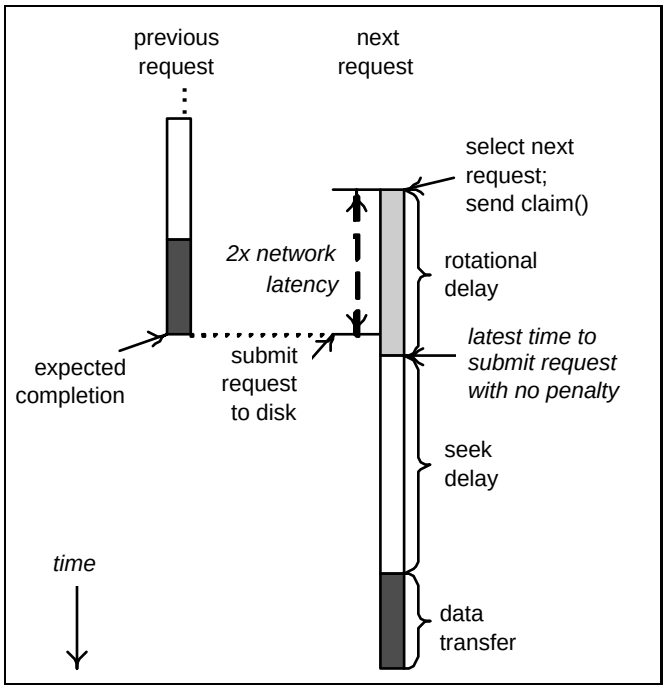

Figure 4: **Overlapping CLAIM communication with rotational latency.** Issuing the request to disk can be delayed up to the time when seek latency must begin to reach the target track before the intended data passes under the read/write head. The rotational latency gives a window for exchanging CLAIM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

The simulation model assumes an outside-the-disk SPTF implementation, which has been demonstrated as feasible [6, 18, 26], in order to allow the abort-from-queue capability needed for D-SPTF.

Each cache uses an LRU replacement policy. Each cache element contains the LBN, size, the associated data, a dirty bit, and a valid bit.

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

The centralized system is designed differently from the decentralized systems. The centralized system contains one single cache with the same aggregate cache space as all the bricks in the decentralized systems. It also contains a single request queue that contains all requests. When some disk is about to complete a request, the system selects the next request for that disk. To present 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 contralized 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.

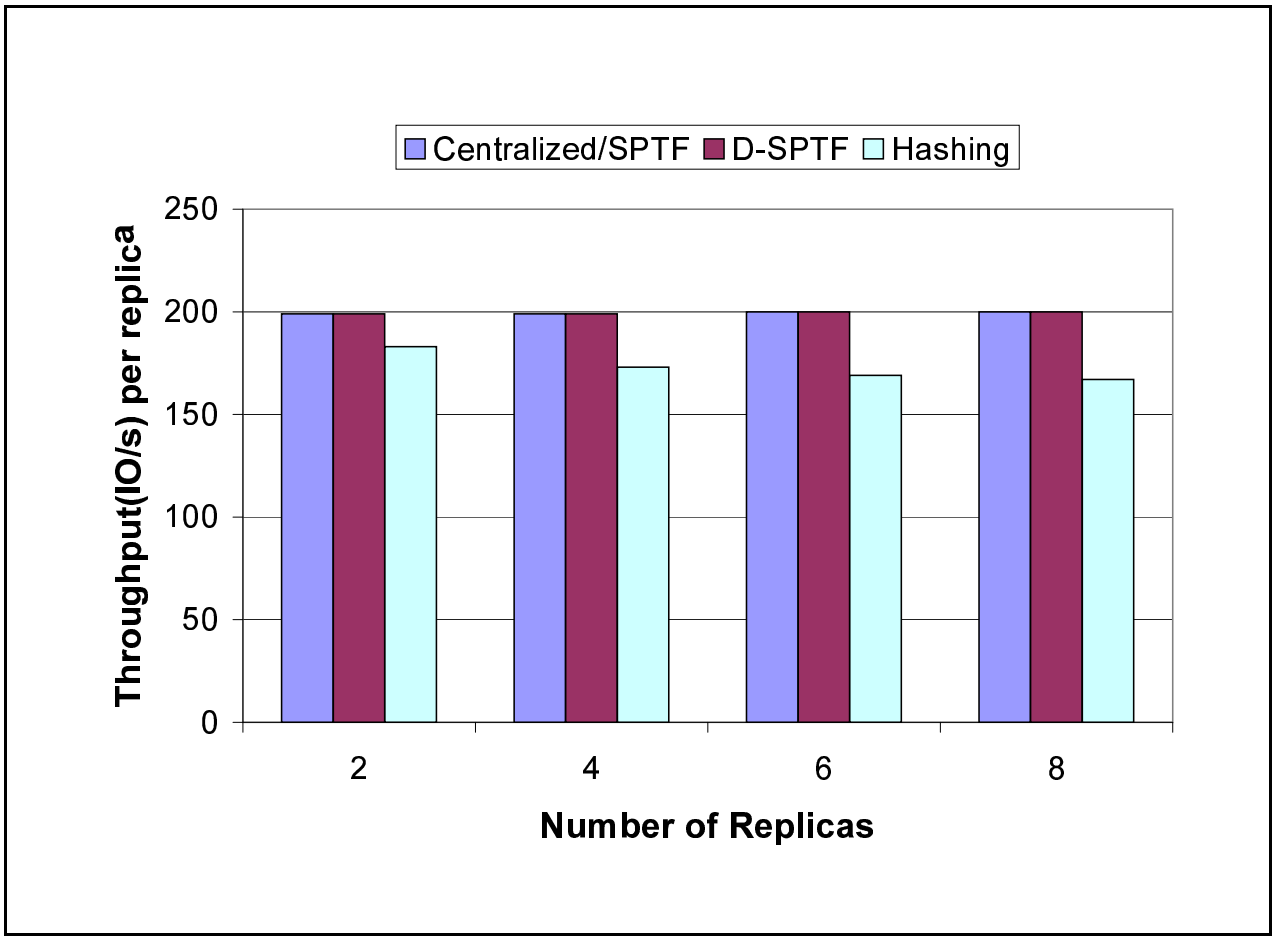

Figure 5: **Throughput comparison of protocols, varying number of replicas.** Throughput is shown per replica (disk or brick).

5 Evaluation

This section presents a number of experiments comparing D-SPTF with the centralized ideal and the decentralized hashing system. We evaluate how effective D-SPTF is with media performance, load balancing and caching behavior. We also evaluate how sensitive D-SPTF is to network performance and how well it adapts to heterogeneous workloads and sets of bricks.

5.1 Media performance

One goal of the D-SPTF protocol is to achieve good performance by ensuring that disk arms can be scheduled well. We expect that D-SPTF will achieve nearly the throughput that an ideal centralized system can achieve, while getting better throughput than a decentralized hashing system.

The experiment used a closed synthetic workload with 16 client threads outstanding and no think time. The LBN of each request was uniformly drawn from the LBN space and the size of each request is drawn from an exponential distribution with mean 4KB. 67% of the requests are read and 33% are writes. System performance is measured as throughput in IO/s.

The systems had 2, 4, 6, or 8 replicas of the data. In the decentralized systems, this meant as many bricks as replicas; in the centralized systems, as many disks as replicas.

Figure 5 shows that D-SPTF outperforms decentralized hashing and is equivalent to the centralized ideal. With basic mirroring (two copies), D-SPTF provides 9% higher throughput than hashing. As the number of

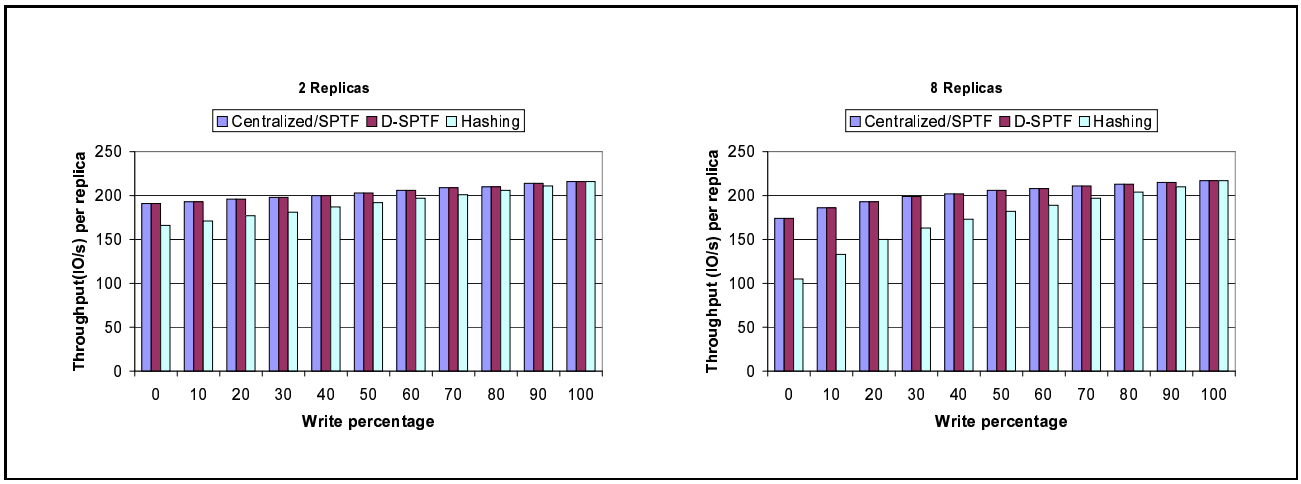

Figure 6: **Effect of write ratio on throughput.** Shows throughput per replica (disk, brick). Write ratio varies from 0% (all reads) to 100% (all writes).

replicas increases, D-SPTF's margin increases. At eight replicas, D-SPTF provides 20% higher throughput than hashing. In every case, the media performance of D-SPTF is equivalent to the centralized ideal.

D-SPTF outperforms hashing because it allows bricks to see greater effective queue depths. Recall that, with SPTF disk scheduling, higher queue depths result in higher overall throughputs. D-SPTF maintains high queue depths by allowing each replica to schedule from all requests that it can service, except for those actively being serviced at another brick.

Decentralized hashing, on the other hand, partitions requests among bricks. For example, if mirroring is used and 16 read requests are outstanding in the system, then brick A and brick B will each receive half the requests on average giving each brick an effective queue depth of 8 read requests. With 8-way replication, 16 outstanding requests will give each brick an average effective queue depth of 2. Writes, on the other hand, go to all replicas, so 16 outstanding write requests will result in a queue of length 16 at all the replicas. In this experiment, the workload was 1/3 writes and 2/3 reads, so for an 8-replica system, each brick will have on average about 5 write requests and 1 or 2 read requests, or about 6 or 7 requests in the queue overall. Figure 1 indicates that a queue of depth 7 provides a throughput of 167 IO/s, which matches the value observed in Figure 5. The results match equally well for other numbers of replicas.

The read/write ratio has a substantial impact on the throughput of the system. Figure 6 shows throughput, as a function of read/write ratio, for both 2- and 8-replica systems, respectively. For two replicas, a workload of all reads results in D-SPTF performing 15% better than hashing. As the write ratio increases, the performance advantage of D-SPTF shrinks until, at 100% writes, D-SPTF and hashing have equivalent performance. For an 8-replica system, a similar trend exists: with a read-only workload, D-SPTF provides 65% higher throughput than hashing. Part of the reason that hashing performs so poorly with 100% reads is that the effective queue depths are so small (i.e., two requests) that occasionally the hashing results in one disk being idle. Like with the 2-replica case, as the write percentage increases, the performance advantage of D-SPTF shrinks.

Interestingly, for low write percentages, each replica of the 2-replica configuration provides more throughput than each replica of the 8-replica setup. For hashing, this effect is caused simply by partitioning the requests. For D-SPTF and the centralized ideal, the effect is less-pronounced, and it is caused by the fact that no disk is idle—each replica subtracts one request from the common queue, reducing the number of requests considered by SPTF. Of course, the extra bricks/replicas result in more total throughput.

5.2 Latency sensitivity

The D-SPTF protocol uses network communication to enable better disk head scheduling. Even with our approach of overlapping communication with disk head positioning time, the communication must be fast

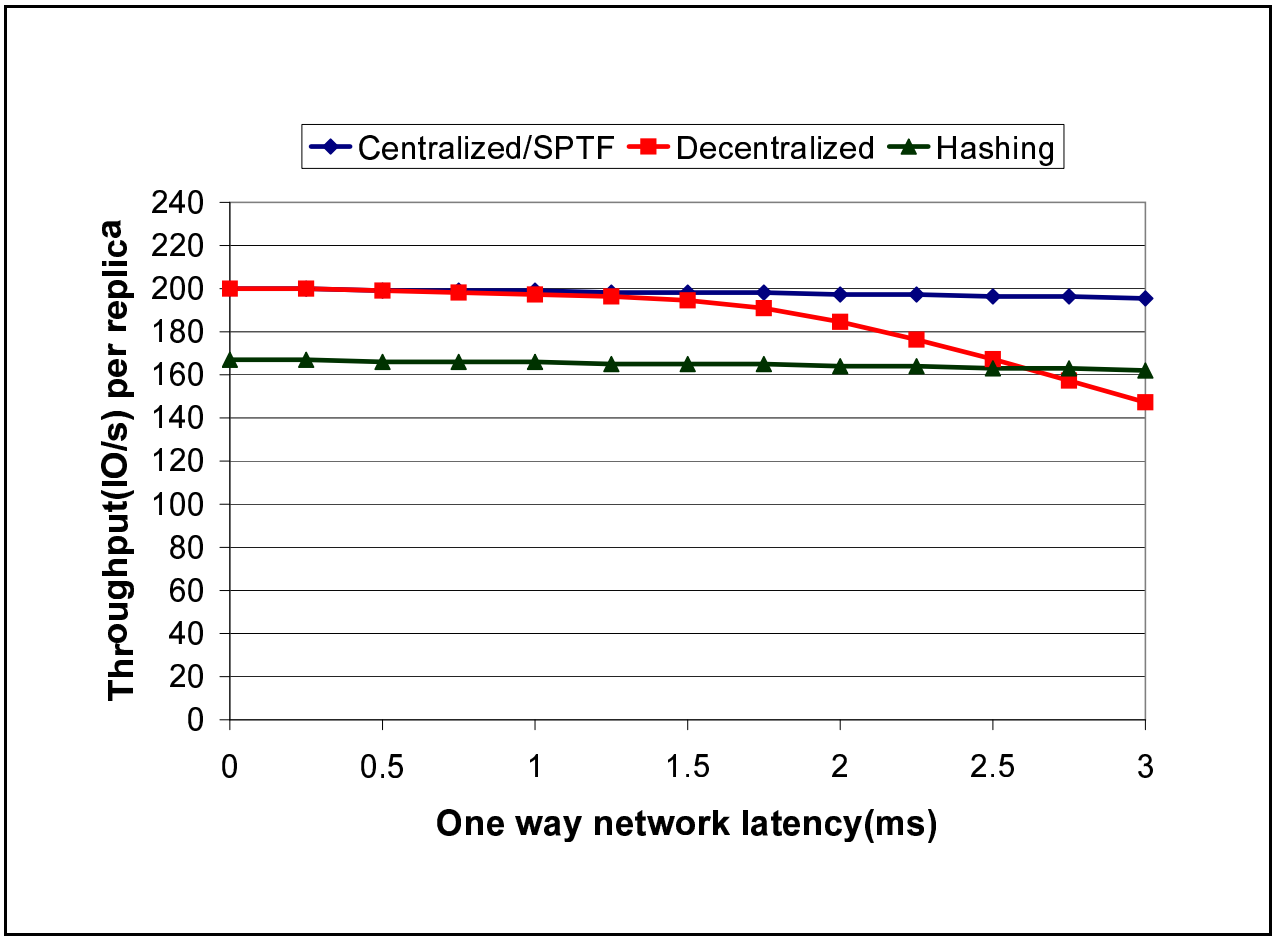

Figure 7: **Effect of network latency on D-SPTF throughput.** Throughput is shown per replica. The network latency varies from 0 ms—instantaneous communication—to 3 ms.

enough that a brick can actually make head scheduling decisions without them getting out of date.

Figure 7 evaluates how serious this effect is. This experiment uses the 8-replica system and workload from the media performance experiments in the previous section, but varies the network communication latency from 0 ms to 3 ms. The results show that the D-SPTF protocol has effectively the same performance as the centralized ideal up to about 1 ms one-way network latency. Even at 1.75 ms network latency, D-SPTF shows less than 5% decrease in throughput. For context, FibreChannel networks have 2–140μs latencies [21], depending on load, and Ethernet-based solutions can provide similar latencies.

D-SPTF performance drops off 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 and more requests complete in less than two network latencies, the disks are forced to wait and performance drops.

5.3 Load balancing homogeneous bricks

To evaluate how effective each scheme is at load balancing requests, we ran the same random synthetic workload on an 8-replica system. We measured both throughput and disk head utilization of each replica. The disk head utilization was measured by counting the total disk media transfer time and dividing it by the simulation time.

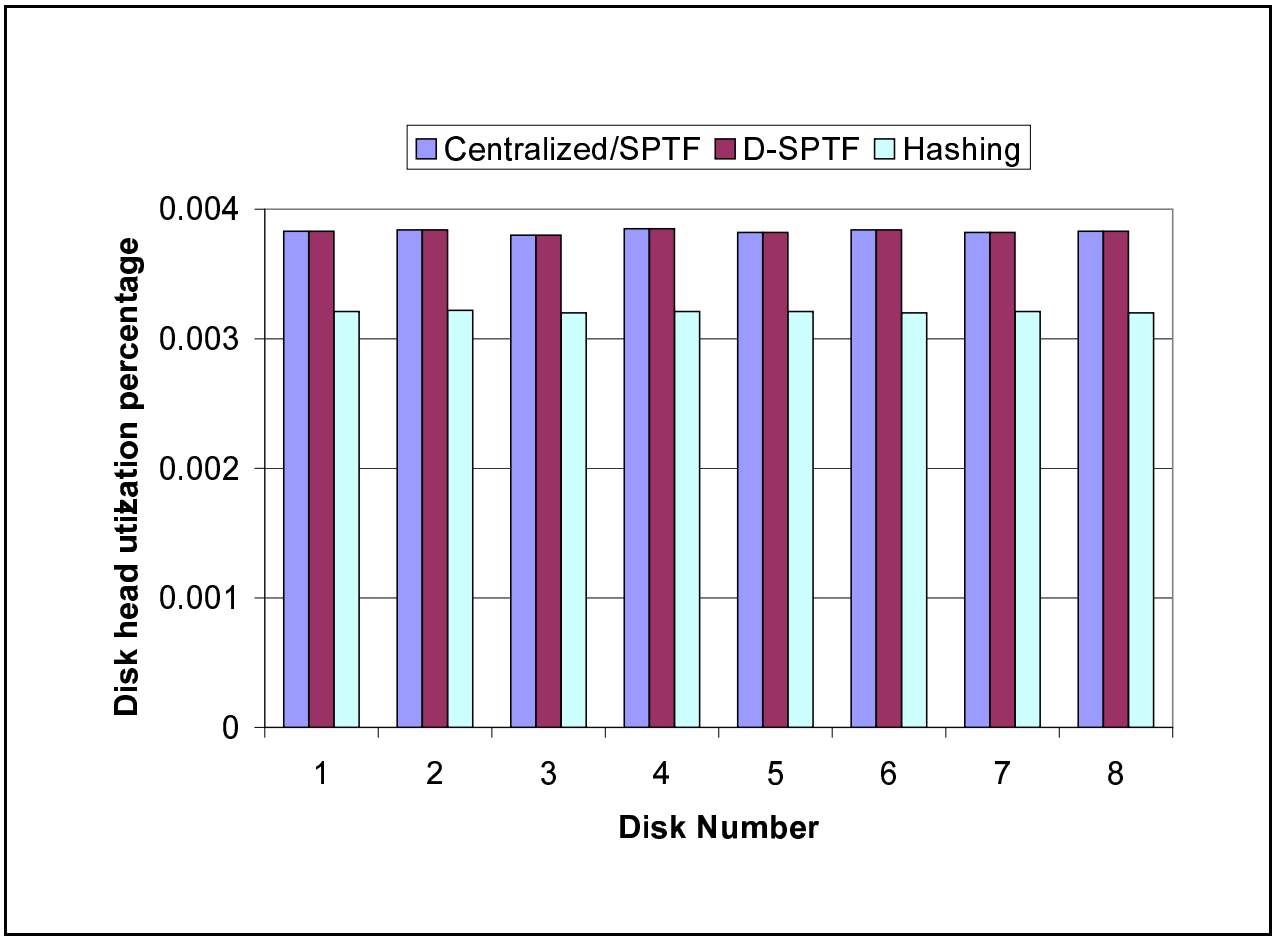

Figure 8: **Disk head utilization at each disk in a homogeneous 8-replica system.** All three request distribution protocols maintain balance across all replicas.

All three schemes balanced load. Of course, this is a random workload, but subsequent sections consider mixed workloads as well. Figure 8 shows the disk head utilization of each of brick or disk. (The disk head utilizations of the three schemes are different because of the differences in media performance, as we saw in Section 5.1.)

5.4 Load balancing heterogeneous bricks

Brick-based distributed storage systems that have been in use for some time are likely to have bricks of different models, with different capacity and performance. In addition, some bricks will periodically have to perform housekeeping activities that will change their performance for a while. These situations make it important for a request distribution protocol to be able to adapt to bricks with heterogeneous performance.

We ran an experiment with two bricks—one "fast" and one "slow"—and varied the performance difference between the two. We varied the rotation speed and seek time of the disk in the slow brick as 100%, 83%, and 67% of that of the fast disk. We used the same synthetic random workload that we used in the media performance (Section 5.1) and load balancing (Section 5.3) experiments. Once again, we measured the throughput and disk head utilization at each brick.

Figure 9 shows the throughput results. Both D-SPTF and the centralized ideal have the desired load balancing property: as the slow disk gets slower, the throughput of fast brick remains nearly unaffected. Random hashing, however, does not do so well: as the slow disk gets slower, it drags down the performance of the fast disk to match.

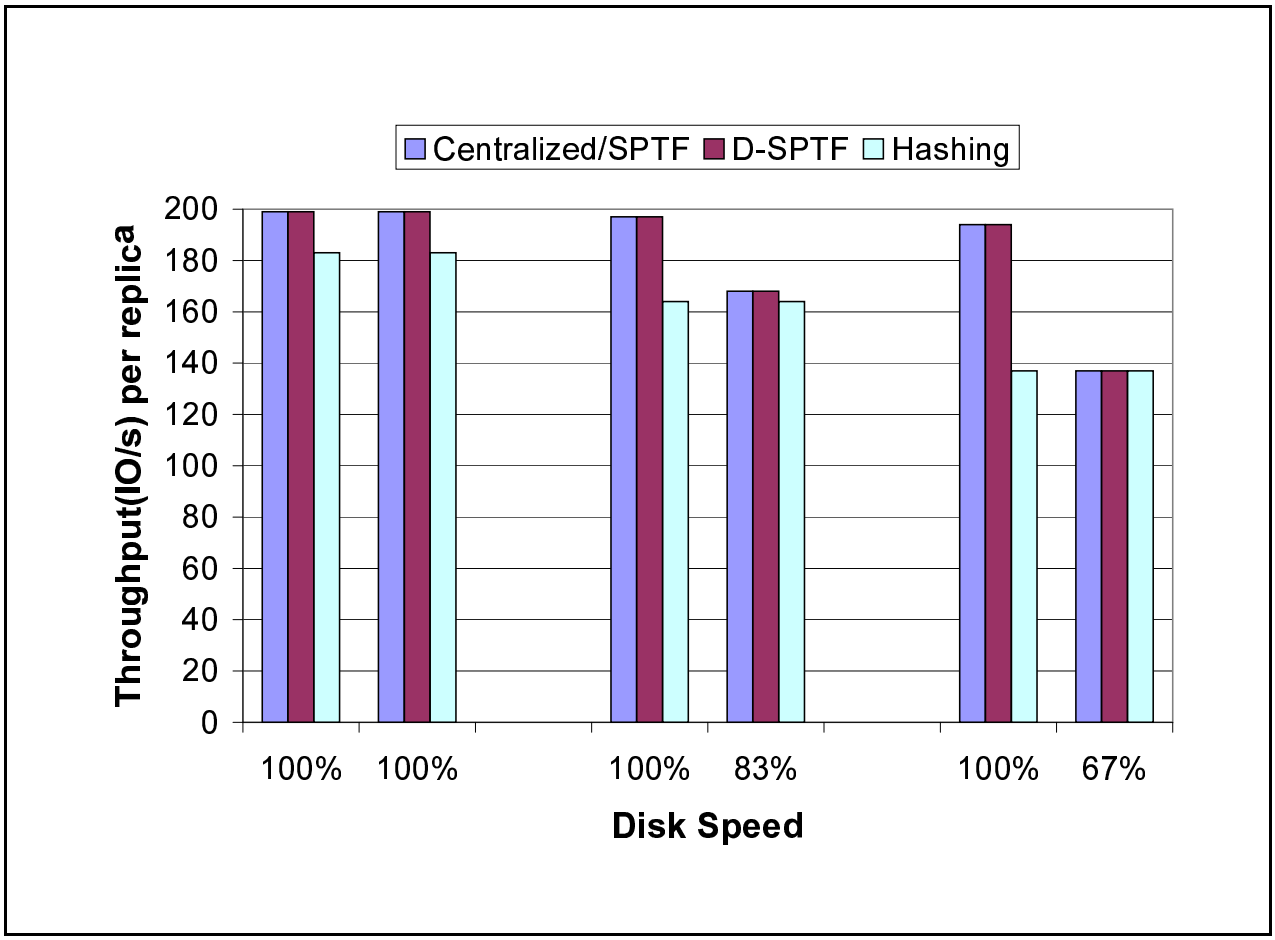

Figure 9: **Throughput balance between a fast and a slow replica.** The disk in the slow replica is the same performance, or 83% or 67% the performance of the disk in the fast replica. D-SPTF and a centralized system keep up the performance of the fast disk, while random hashing paces the fast disk to match the slow disk.

Figure 10 shows the corresponding disk head utilizations. D-SPTF and centralized maintain the utilization of the fast disk constant as the slow disk's performance varies, but the utilization of the slow disk increases somewhat. This is because of writes: since they must go to both bricks, both disks must service the writes. However, the slow disk services the writes more slowly than the fast disk. This results in the slow disk having more writes in its queue than the fast disk, which increases SPTF efficiency. This results in higher utilization of the disk head but not higher throughput, since the disk is slower. Since the speed at which writes complete is impacted by the speed of the slow disk, the fast disk does have a slightly lower throughput as the slow disk gets slower. Decentralized hashing fares the worst of the three protocols: as the slow disk becomes slower, the load between the two disks becomes more and more unbalanced because hashing assumes that both devices are of equivalent speed and thus sends half the requests to each disk. As a result, the rate at which requests are sent to the fast disk is governed by the rate at which the slow disk can service its requests, and the performance advantage of the fast disk is wasted.

5.5 Mixed workloads

Storage systems must handle mixed workloads. This section compares how well each of the request distribution protocols handles a mixture of random and sequential operations. Such mixtures can happen, for example, when a database processes both transaction processing and decision support queries.

The experiment uses a workload that consists of 14 random streams and one sequential stream that keeps

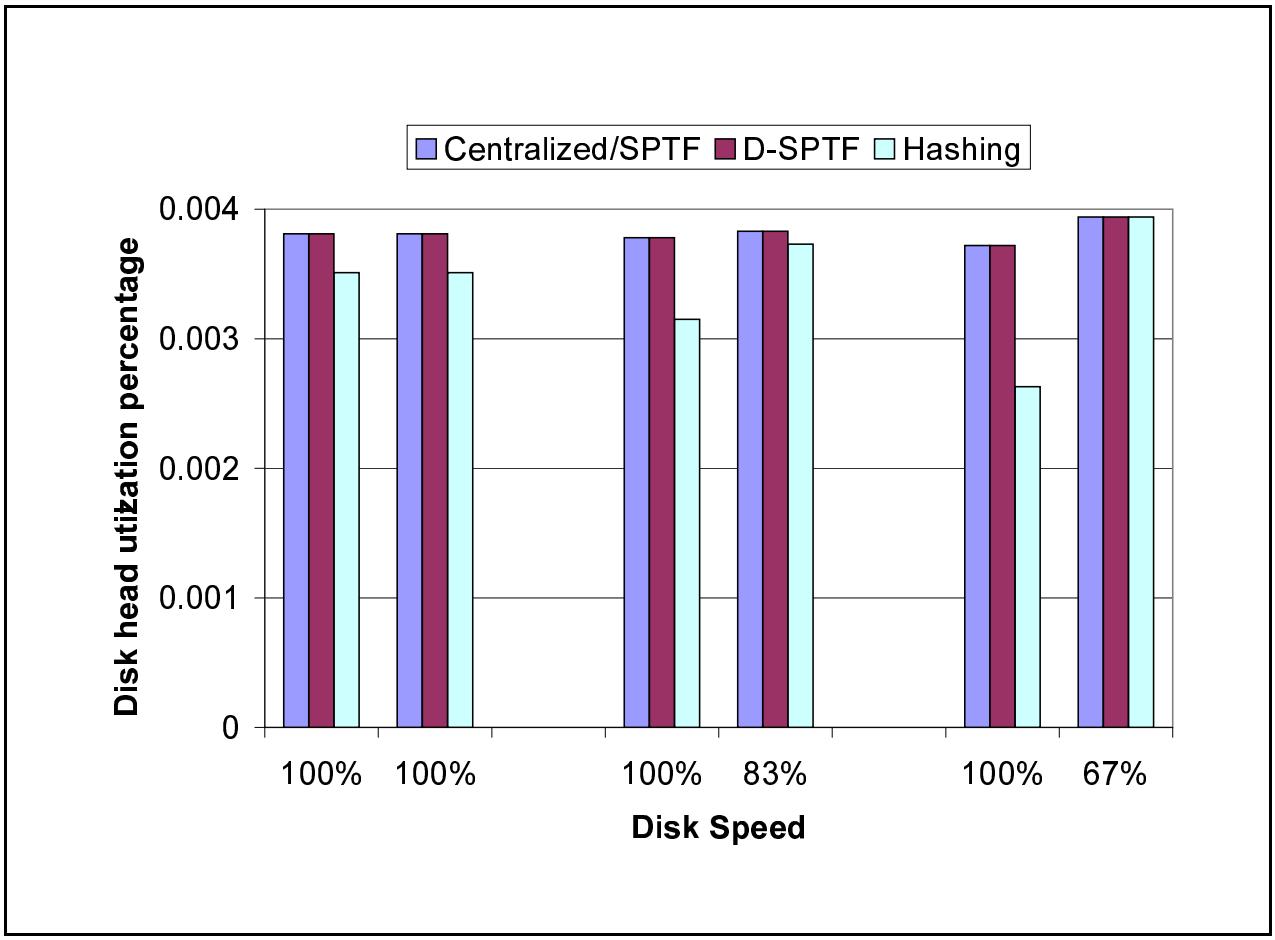

Figure 10: **Utilization balance between a fast and a slow replica.** Same systems as in Figure 9.

two requests outstanding at a time (for efficient prefetching). Each random stream is a closed workload issuing 8KB random requests of which 66% are reads. The sequential stream is also a closed workload issuing 160 KB sequential reads against 512 MB files. There is no think time for the workloads, and the start point (in LBN space) of each sequential stream is randomly selected.

We ran this workload mix against a two-brick mirrored system, using the D-SPTF and hashing request distribution protocols. When reading, the hashing scheme selected alternate replicas every 10 MB.

Figure 11 shows the results. D-SPTF achieves 7% greater throughput than hashing for the sequential portion of the workload, and 2.7x greater throughput than hashing for the random operations. D-SPTF gives both workload classes approximately the same throughput, while hashing gives the sequential workload significantly more throughput than the random.

D-SPTF does better because of how the bricks treat the sequential workload. Recall that bricks use an SPTF disk head scheduling discipline. The next request in a sequential workload will always require the shortest positioning time—zero, or a track or head switch. The result is that a sequential workload ends up "owning" a disk, starving any other requests. The D-SPTF protocol ensures that the random workload requests are considered at both bricks, so that they end up being serviced by the drive that is not currently handling the sequential stream. In this experiment, one disk ended up dedicated to the sequential stream and the other to the random workload.

The hashing protocol always selects a single brick to process a read request, and tries to send an equal fraction of requests to each brick. When one brick is captured by a sequential stream, however, all the random requests that are sent to that brick will be starved until the sequential stream reaches a 10 MB

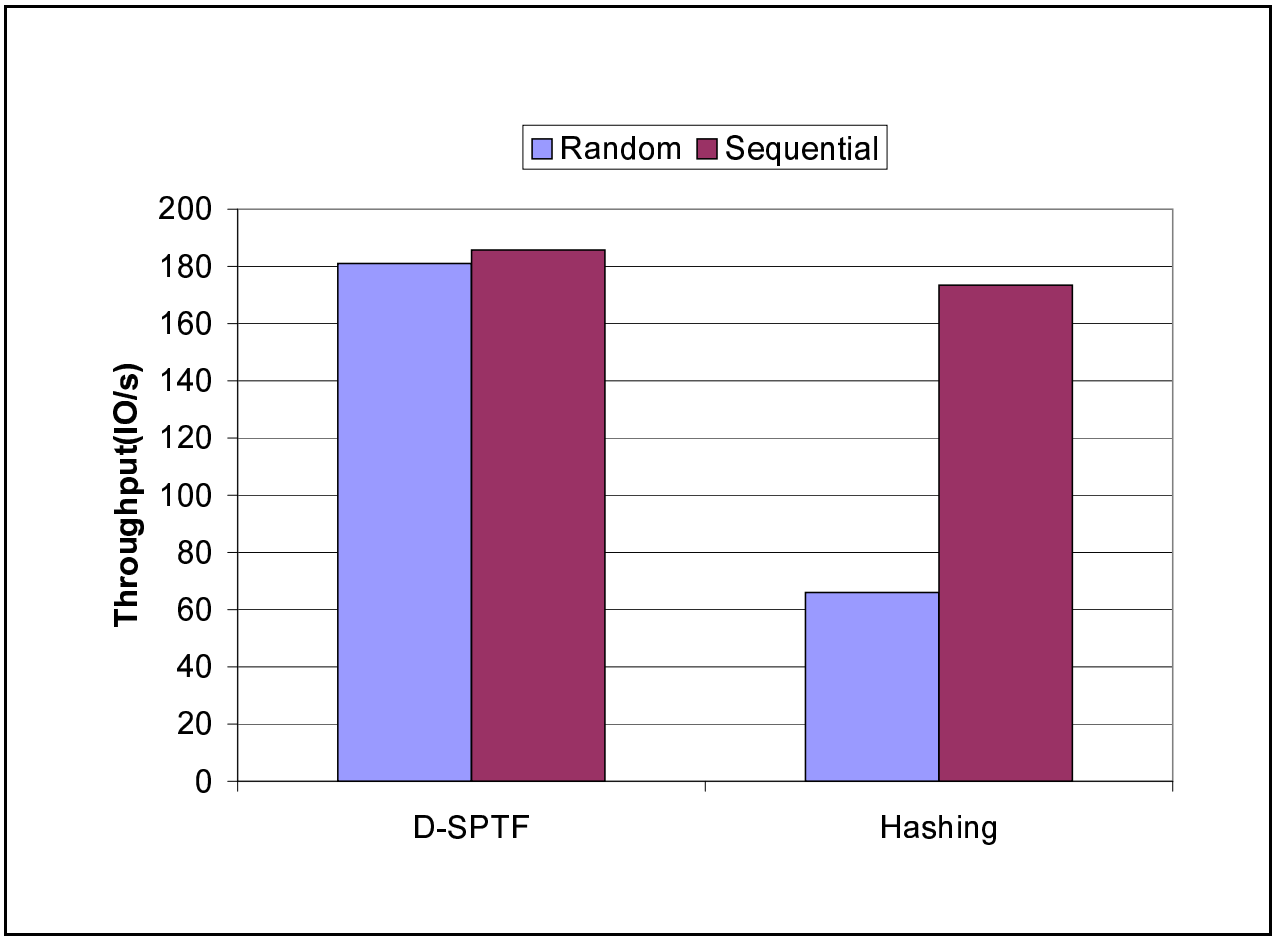

Figure 11: **Throughput comparison of mixed workloads under D-SPTF and hashing.**

boundary and switches to the other brick. As each random request completes, its successor has a 50% chance of being sent to the brick processing the sequential stream. So, before long, all random requests are blocked on that brick. The other brick stays idle until the sequential workload shifts over.

With the hashing protocol, how long the sequential workload occupies a brick is a function of the "stripe size"—the boundaries when the hash function will switch reads from one brick to the other. Figure 12 shows that, as the stripe size increases, the sequential throughput goes up and the random workload's throughput goes down. At small stripe sizes, the sequential workload changes bricks often, and the random workload has more opportunity to make progress. But, each time the sequential workload switches disks, it may have to wait behind some random requests before it gets serviced, decreasing the sequential throughput. As the stripe size increases, switching occurs less often, random operations are blocked longer, and the sequential stream sees less interruption. Such "stripe size" selection is a complexity of hash-based systems not faced by D-SPTF.

5.6 Cache performance

In addition to providing load balancing and good media performance, a decentralized storage system must also provide good cache performance. The cache resources in a brick-based system are split into a number of independently-managed caches, and these may not be able to provide the hit rate that a centrally-managed cache could provide. For example, if one brick's cache is being used more heavily than another's, data will be evicted from the heavily-used cache earlier than it would have been in a centralized system that saw the global block reference stream.

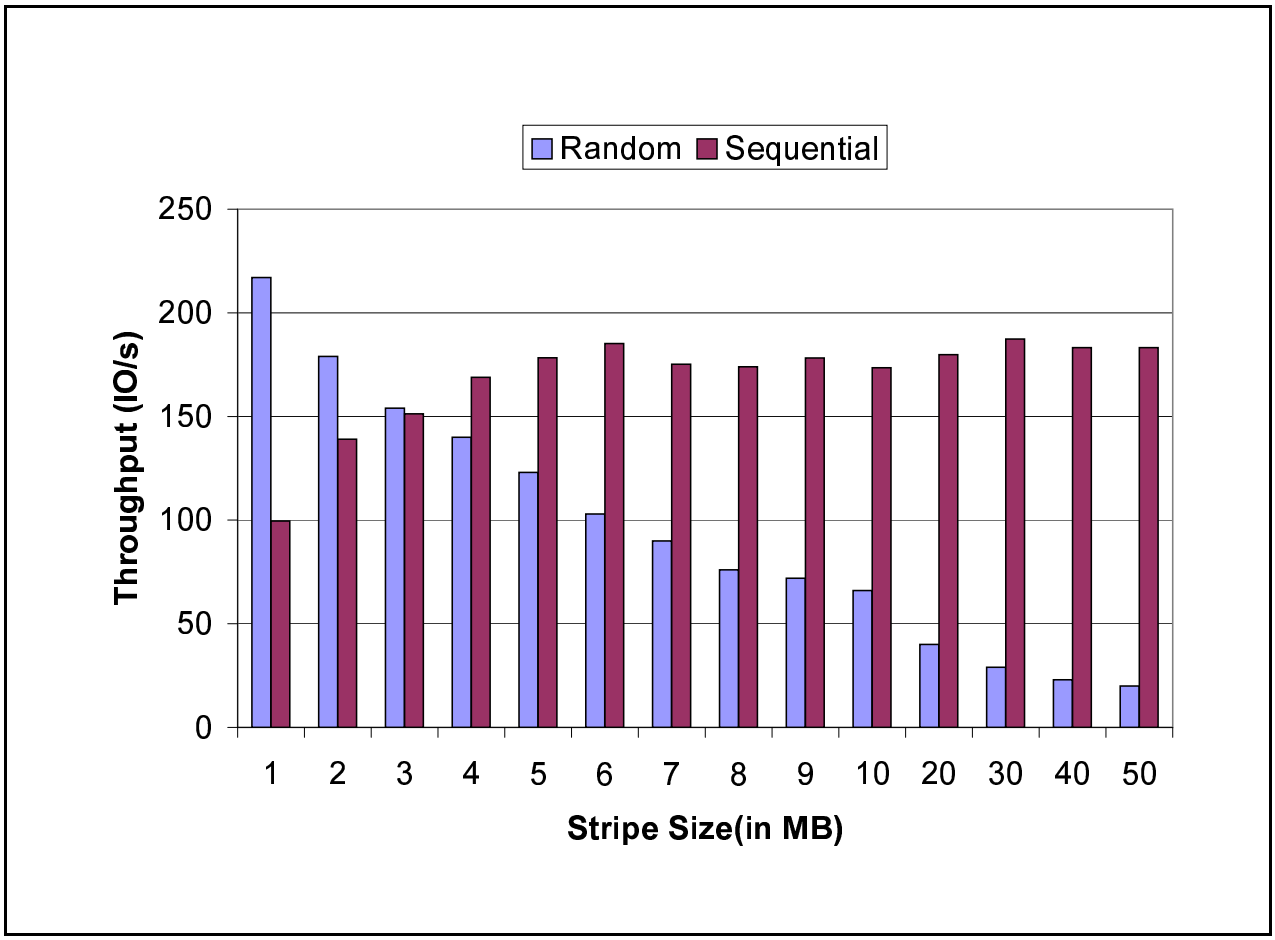

Figure 12: **Effect of stripe size on sequential and random workloads under hashing.** The sequential workload issued 120 KB IOs; the random workload issued 4 KB IOs.

We compared systems with decentralized caches using hashing and D-SPTF request distribution to a centralized cache with the same aggregate cache space. For this experiment, we used 8 bricks containing 2MB each for a total aggregate of 16MB. Both brick caches and the centralized cache used the LRU cache replacement policy. Four one-hour trace segments of the 1999 HPL cello server trace, which captures the activity of a server used for software development and research. Each trace is from a different weekday and a different time of day, resulting in the differing workload characteristics shown in Table 1. The traces were replayed at normal speed.

Figure 13 shows the cache hit percentages: centralized caching always achieves the best hit percentage, and D-SPTF and hashing have just 1–2% fewer hits across the four traces.

To explain why this is, we compared the cache eviction decisions that the centralized cache made with those that the decentralized protocols made. The decentralized system simulator also maintained a model of the state of the centralized cache that would have occurred if the centralized cache saw the same reference pattern as all the brick caches put together. Each time a brick cache chose to evict a block, we measured how deep that block would be in the centralized cache's LRU list. If the brick caches evicted blocks in exactly the same order as the centralized cache would have, then the depth of each evicted block should be 32000 —the size of the centralized cache, given the 16 KB cache block size. We found that over 70% of cache evictions for both the D-SPTF and hashing occured at a stack depth of 32000. 90% of the cache evictions were within stack depth of 28000, and 99% of the cache evictions occurred within a stack depth of 19000. Thus, most of the evictions in the decentralized case were of blocks towards the LRU end of the global access ordering.

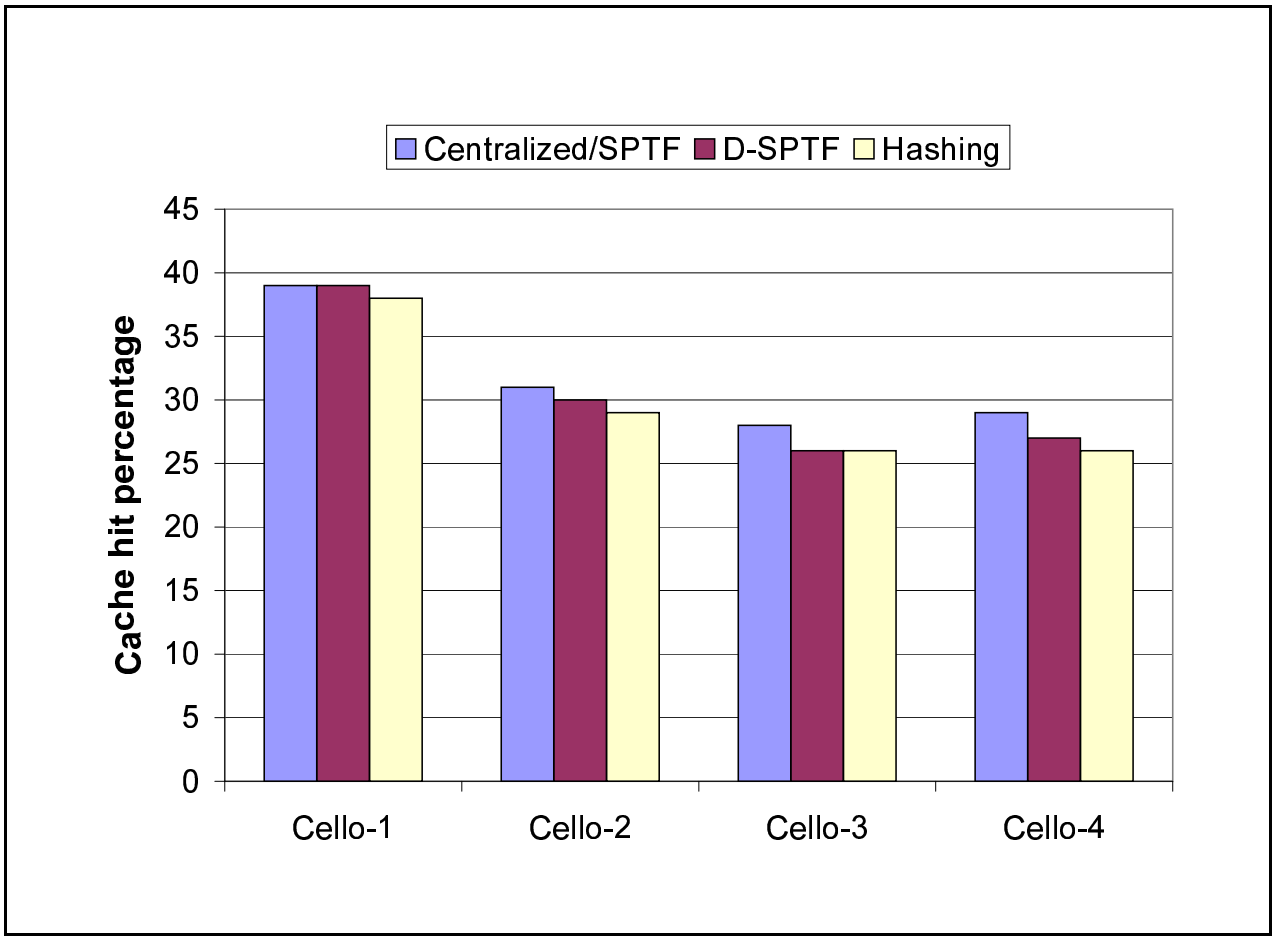

Figure 13: **Comparision of the cache hit percentages of centralized caching, decentralized hashing and D-SPTF.**

6 Additional related work

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

Several groups have explored the centralized multi-replica SPTF approach, labelled "Centralized/SPTF" in our evaluations, to which we compare D-SPTF. For example, Lo [17] proposed and explored Ivy, a system for exploiting replicas by routing requests to the disk whose head is closest to a desired replica. Wilkes et al. [24] explored a similar approach, but routing requests based on predicted positioning time. Most recently, Yu et al. [26] described an approach similar in spirit to D-SPTF across mirrored local disks, for use in evaluating their SR-Array system. D-SPTF builds on this prior work by bringing its benefits to a decentralized context and simultaneously achieving effective exclusive caching and load balancing.

Several groups have explored explicitly cooperative caching among decentralized systems [4, 23]. These systems introduce substantial bookkeeping and communication that are not necessary if requests are restricted to being serviced by their data's homes. However, these techniques could be used to enhance load balancing and memory usage beyond the confines of a scheme like D-SPTF, which focuses on the assigned replica sites for each data block.

Striping and hashing are popular techniques for load balancing. More dynamic schemes that migrate or rebalance load based on feedback are popular for activities like process executions. With a front-end distributing requests across a set of storage servers, feedback-based load distribution works well [2, 13, 16]. Clusters of web servers often use a load-balancing front-end to distribute client requests across the back-end workers. For example, LARD [19] provides such load balancing while maintaining locality.

	Cello-1	Cello-2	Cello-3	Cello-4
Date	03/09/1999	05/11/1999	08/19/1999	10/20/1999
Time	10:00	12:00	11:00	13:00
Read ratio	82%	68%	83%	60%
Avg. size	10KB	10.5KB	11.5KB	10KB
No. Requests	312782	574919	1029252	468557

Table 1: **Trace characteristics.**

7 Conclusions

D-SPTF distributes requests across heterogeneous storage bricks, with no central point of control, so as to provide good disk head scheduling, cache utilization, and dynamic load balancing. It does so by exploiting high-speed communication to loosely coordinate local decisions towards good global behavior. Specifically, D-SPTF provides all replicas with all possible read requests and allows each replica to schedule locally. Limited communication is used to prevent duplication of work. Overall, given reasonable communication latencies (e.g., < 1 ms roundtrip), D-SPTF matches the performance of an idealized centralized system (assuming equivalent aggregate resources) and exceeds the performance of a decentralized hash-based system.

References

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

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

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

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

[5] 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.

[6] Zoran Dimitrijevic, 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.

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

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

[9] Svend Frolund, 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.

[10] 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, August 2003.

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

[12] 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.

[13] Hui-IHsiao and David J. DeWitt. A performance study of three high availability data replication strategies. *Parallel and Distributed Information Systems International Conference* (Miami Beach, FL), pages 18–28, 4–6 December 1991.

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

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

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

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

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

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

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

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

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

[23] 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.

[24] 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.

[25] 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.

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