Swift: Using Distributed Disk Striping to Provide High I/O Data Rates

Luis-Felipe Cabrera
Computer Science Department
IBM Almaden Research Center

Darrell D. E. Long*
Computer & Information Sciences
University of California, Santa Cruz

Computing Systems, vol. 4, no. 4 (1991) pp. 405–436.

Abstract

We present an I/O architecture, called Swift, that addresses the problem of data rate mismatches between the requirements of an application, storage devices, and the interconnection medium. The goal of Swift is to support high data rates in general purpose distributed systems.

Swift uses a high-speed interconnection medium to provide high data rate transfers by using multiple slower storage devices in parallel. It scales well when using multiple storage devices and interconnections, and can use any appropriate storage technology, including high-performance devices such as disk arrays. To address the problem of partial failures, Swift stores data redundantly.

Using the UNIX operating system, we have constructed a simplified prototype of the Swift architecture. The prototype provides data rates that are significantly faster than access to the local SCSI disk, limited by the capacity of a single Ethernet segment, or in the case of multiple Ethernet segments by the ability of the client to drive them.

We have constructed a simulation model to demonstrate how the Swift architecture can exploit advances in processor, communication and storage technology. We consider the effects of processor speed, interconnection capacity, and multiple storage agents on the utilization of the components and the data rate of the system. We show that the data rates scale well in the number of storage devices, and that by replacing the most highly stressed components by more powerful ones the data rates of the entire system increase significantly.

Keywords: high-performance storage systems, distributed file systems, distributed disk striping, high-speed networks, high-speed I/O, client-server model, video server, multimedia, data resiliency.

1 Introduction

The goal of our I/O architecture is to support high data rates in a general purpose distributed system. This architecture, called Swift, addresses the problem of data rate mismatches between the requirements of an application, the maximum data rate of the storage devices, and of the interconnection medium. Swift accomplishes this goal by using a high-speed interconnection medium to provide high data rate transfers by using multiple slower storage devices in parallel. Swift has the flexibility to use any appropriate storage technology, including high-performance devices such as disk arrays. It can adapt to technological advances to provide for ever increasing I/O demands.

The current generation of distributed computing systems do not support I/O-intensive applications well. In particular, they are incapable of integrating high-quality video with other data in a general purpose environment. For example, multimedia applications require this level of service and include scientific visualization, image processing, and recording and play-back of color video. The data rates required by some of these applications range from 1.2 megabytes/second for DVI compressed video and 1.4 megabits/second for CD-quality audio [16], and up to 90 megabytes/second for uncompressed full-frame color video.

*Supported in part by the National Science Foundation under Grant NSF CCR-9111220, by the Institute for Scientific Computing Research at Lawrence Livermore National Laboratory and by faculty research funds from the University of California, Santa Cruz.

Advances in VLSI, data compression, processors, communication networks, and storage capacity mean that systems capable of integrating continuous multimedia will soon emerge. In particular, the emerging ANSI fiber channel standard will provide data rates in excess of 1 gigabit/second over a switched network. In contrast to these advances, neither the positioning time (seek-time and rotational latency) nor the transfer rate of magnetic disks have kept pace.

The architecture we present solves the problem of storing and retrieving very large data objects from slow secondary storage at very high data rates. Its goal is to support high I/O data rates in a general purpose distributed storage system. It stripes data over several disks [24], much like RAID [20], driving the disks in parallel to provide high data rates. Swift, unlike RAID, was designed as a distributed storage system. It provides the advantages of easy expansion and load sharing, and also provides better resource utilization since it will use only those resources that are necessary to satisfy a given request. In addition, Swift has the flexibility to use any appropriate storage technology, including a disk array, or other high-performance storage devices such as an array of tapes.

We have conducted two studies to validate the Swift architecture. The first was a proof-of-concept prototype of a simplified version of Swift implemented on an Ethernet using the UNIX[1] operating system. This prototype provides a UNIX-like file system interface, that includes **open**, **close**, **read**, **write** and **seek**. It uses distributed disk striping over multiple servers to achieve high data rates. In the case of synchronous writes, the prototype on a single Ethernet segment with three servers achieves data rates that are more than double that provided by access to the local SCSI disk. In the cases of reads and asynchronous writes, the data rates achieved by the prototype scale approximately linearly in the number of storage agents up to the saturation of the Ethernet segment.

The second study is a discrete-event simulation of a simplified local-area instance of the Swift architecture. It was constructed to evaluate the effects of technological advances on the scalability of the architecture. The simulation model shows how Swift can exploit a high-speed (gigabit/second) local-area network and faster processors than those currently available. The simulation is also used to locate the components that will limit I/O performance. Our simulation includes processor utilization, taking into account not only the data transmission but also the cost of computing parity blocks for the data.

The remainder of this paper is organized as follows: the Swift architecture is described in §2 and our Ethernet-based local-area prototype in §3. Measurements of the prototype are presented in §4. Our simulation model is then presented in §5. In §6 we consider related work and present our conclusions in §7.

2 Description of the Swift Architecture

Swift builds on the basic notion of striping data over multiple storage agents and driving them in parallel. The principle behind our architecture is simple: use a high-speed interconnection medium to aggregate arbitrarily many (slow) storage devices into a faster logical storage service, making all applications unaware of this aggregation. Several concurrent I/O architectures, such as Imprimis ArrayMaster [14], DataVault [26], CFS [21, 22], RADD [25] and RAID [20, 18], are based on this observation. Mainframes [13, 7] and super computers [15] have also exploited this approach.

Swift is a client/server distributed architecture made up of independently replaceable components. The advantage of this modular approach is that any component that limits the performance can either be *replaced* by a faster component when it becomes available or can be *replicated* and used in parallel. We first describe the architecture in general terms and then present a more detailed description of each of its components.

Since Swift is a distributed architecture with many independent components, partial failures are an important concern. If no precautions are taken, then the failure of a single component, in particular a storage agent, could hinder the operation of the entire system. For example, any object which has data in a failed storage agent would become unavailable, and any object that has data being written into the failed storage agent could be damaged. The accepted solution for this problem is to use redundant data, including *multiple copy* [9] and *computed copy* (erasure-correcting codes) [20]. While either choice is compatible with our architecture, we plan to use computed copy redundancy (in the form of parity) in a future version of the prototype. This simple approach will provide

[1] UNIX is a trademark of AT&T Bell Laboratories

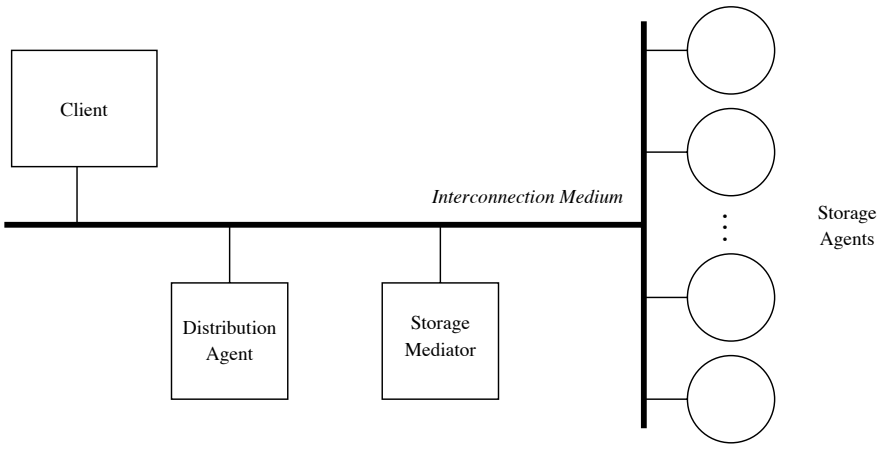

Figure 1: Components of the Swift Architecture

resiliency in the presence of a single failure (per parity group) at a low cost in terms of storage but at the expense of some additional computation.

Swift assumes that objects are created and used by clients and that they are managed by the several components of Swift. In particular, objects are stored by *storage agents*. An implementation of Swift operates as follows: when a client issues a request to store or retrieve an object, a *storage mediator* reserves resources from all the necessary storage agents and the communication subsystem in a session-oriented manner. The storage mediator then presents a *distribution agent* with a *transfer plan*. Swift assumes that sufficient storage and data transmission capacity has been reserved and thus will be available, and that negotiations between the client (that can behave as a *data producer* or a *data consumer*) and the storage mediator will allow the preallocation of these resources. Resource preallocation implies that storage mediators will reject any request with requirements it is unable to satisfy. The request can be delayed and reissued later when more resources are available. To then transmit the object to or from the client, the distribution agent stores or retrieves the data at the storage agents following the transfer plan with no further intervention by the storage mediator. The components of the Swift architecture are depicted in figure 1.

In Swift, the storage mediator selects the striping unit (the amount of data allocated to each storage agent per stripe) according to the data rate requirements of the client. If the required transfer rate is low, then the striping unit can be large and the data can be spread over only a few storage agents. If the required data rate is high, then the striping unit must be chosen small enough to spread the data over sufficient storage agents to exploit all the parallelism needed to satisfy the request. By allowing variable striping units the Swift architecture achieves better resource utilization by using only the resources that it needs to satisfy a request.

In the following subsections the distribution agent, storage mediator, and storage agent are presented in more detail.

2.1 Distribution Agent

The distribution agent acts on behalf of its clients, the data producer and the data consumer, in the storage and retrieval of all data. Although not strictly required, we expect that in practice both the data producer and the data consumer will be co-resident with the distribution agent serving them.

The distribution agent interacts with the storage mediator to obtain directory service, access rights to objects, encryption keys, and transfer plans. In addition, all computed transformations of the data, such as encryption and erasure correcting codes, are done by the distribution agents. Authentication is accomplished through a secure exchange of keys with the storage mediator to obtain a trusted communication channel.

The primary task of the distribution agent is to implement striping of the data over several storage agents. When retrieving an object from storage the distribution agent assembles the object from the incoming data streams

according to the transfer plan. When storing an object, the distribution agent distributes the object among the several storage agents. In both cases the distribution agent performs all necessary redundancy computations to provide fault tolerance. As we shall see in §5.2, these computations can put a significant burden on the processor. Fortunately, this will be mitigated by the availability of increasingly powerful processors.

2.2 Storage Mediator

The storage mediator is central to establishing and administering the storage and communication resources of the system. First, it determines the size of the *transfer unit* for the request on hand. When storing an object, the transfer unit is the size of the data block that is used for computing parity blocks and for transmission over the interconnection medium. Second, the storage mediator negotiates with the storage agents to reserve sufficient space and communication capacity. Third, the storage mediator determines how to best meet the resiliency requirements and returns this as part of the transfer plan.

The transfer plan contains all the information necessary to store or retrieve an object administered by the system. In particular, it contains the transfer unit for the request, a list of storage agents to hold the data, a list of storage agents to act as checks on the data, and the internal handles to be used when transferring data to and from these storage agents.

The storage mediator is the sole repository for encryption keys. Encryption is the mechanism that will be used to provide authentication, access control, and security of the data. The storage mediator will use a secure key exchange protocol to authenticate the distribution agents.

In order to achieve high performance, a pessimistic storage allocation strategy is used. Since all resources are preallocated, requests that would exceed current storage or communication capacity will be denied. These requests can be reissued at a later time when more resources are available.

The storage mediator will use a call-back mechanism to provide cache coherency. When a distribution agent requests access to an object that still may exist in the cache of some other client, the storage mediator will cause that cache to be flushed as part of the resource allocation protocol. Resource preallocation allows Swift to effectively support sequential write sharing, as alternative access to the same data will always find the data in only one cache, making invalidation simple.

The storage mediator must be available and the metadata it maintains be fault tolerant. For example, each directory entry contains the name of the object, its protection status, a list of data segments and storage agents that hold the object. The loss of directory information implies the inaccessability of all objects referenced by that directory. The integrity of the storage mediator's data can be insured in several ways. Our preferred method is to let Swift administer the metadata and specify a high degree of redundancy. Another approach would be to use standard data base techniques such as a write-ahead log [12, 6].

2.3 Storage Agents

The storage agents administer all aspects of secondary storage media, including data layout optimization and off-line data alignment. Each storage agent may administer many storage devices. These storage devices can be disks or other high speed devices including arrays of disks or tapes.

Since the Swift architecture is intended for objects much larger than any cache, we believe that caches will be used most often for staging data into transfer units than for storing complete objects. For small objects, we expect that caches will be as beneficial as in other systems [17].

Swift achieves reliability through the appropriate use of redundancy. For example, object descriptors store redundant information that allows the reconstruction of all objects by scavenging the data in the storage agents, should a catastrophic failure, or a software error, render the storage mediator inoperative. By using the error detecting capabilities of the disks, a single parity disk is sufficient to tolerate a single failure [10, 20]. In this way, if a disk fails it can be reconstructed using the information on the other disks. Higher level erasure-correcting codes can be used if more than one failure is to be tolerated [11].

3 Ethernet-based Prototype of Swift

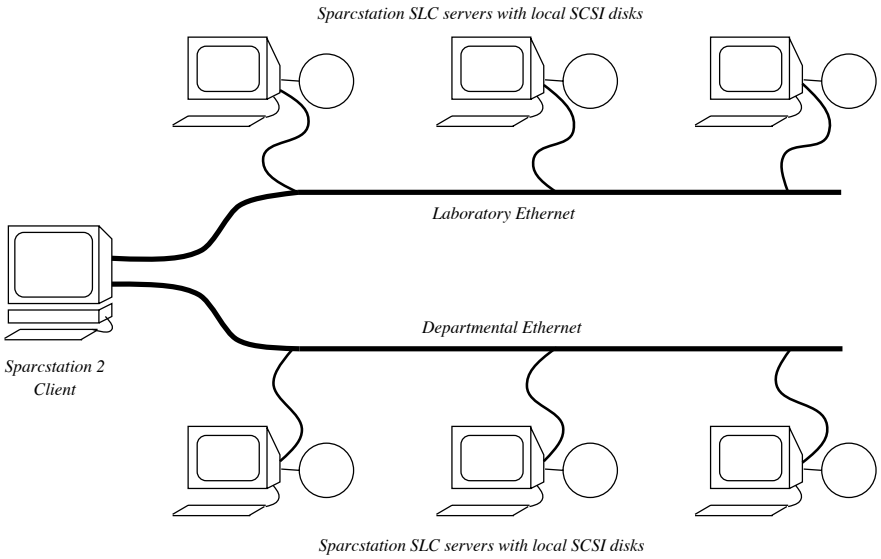

Figure 2: An Ethernet-based implementation of the Swift architecture.

A simplified prototype of the Swift architecture has been built as a set of libraries that use the standard filing and interprocess communication facilities of the UNIX operating system. We have used the UNIX file system facilities to name and store objects, making the storage mediators unnecessary. There is no need to explicitly build transfer plans since the library interleaves data uniformly among the set of identical storage agents that are used to service a request.

Objects administered by the prototype are striped over several servers, each of which has its own local SCSI disk and act as a storage agent on an Ethernet-based local-area network. Clients are provided with a UNIX-like file system interface that includes **open**, **close**, **read**, **write** and **seek** operations.

The structure of the prototype is depicted in figure 2. The Swift distribution agent is embedded in the libraries that are invoked by the client. The storage agents are represented by UNIX processes on servers that use the standard UNIX file system.

When an I/O request is made, the client communicates with each of the storage agents involved in the request so that they can simultaneously perform the I/O operation on the striped file. Since a file may be striped over any number of storage agents, the most important performance-limiting factors are the rate at which the client and its servers can send and receive packets, and the maximum transfer rate of the Ethernet.

The current prototype has allowed us to confirm that a high aggregate data rate can be achieved with the Swift architecture. The data rates of an earlier prototype using a data transfer protocol built on the TCP [8] network protocol had proved unacceptable.

In our first prototype a TCP connection was established between the client and each server. These connections were multiplexed using **select**. Since TCP delivers data in a stream with no message boundaries, a significant amount of data copying was necessary. The data rates achieved were never more than 45% of the capacity of the Ethernet. Initially, **select** seemed to be the point of congestion. A closer inspection revealed that using TCP was not appropriate since buffer-management problems prevented the prototype from achieving high data rates.

The current prototype has been built using a light-weight data transfer protocol on top of the UDP [8] datagram protocol. To avoid as much unnecessary data copying as possible, scatter-gather I/O was used to have the kernel deposit the message directly into the user buffer.

In the current prototype the client is a Sun 4/75 (SPARCstation 2) with 64 megabytes of memory and 904 megabytes of local SCSI disk (unused in our experiments). It has a list of the hosts that act as storage agents. All storage agents

were placed on Sun 4/20s (SLC) each with 16 megabytes of memory and identical local SCSI disks each with a capacity of 104 megabytes. Both the client and the storage agents use dedicated UDP ports to transfer data and have a dedicated server process to handle the user requests.

3.1 The Data Transfer Protocol

The Swift client uses a unique UDP port for each connection that it makes. This was done in an effort to allocate as much buffer space as possible to the client. The client services an **open** request by contacting a storage agent at its advertised UDP port address.

Each Swift storage agent waits for **open** requests on a well-known UDP [8] port address. When an **open** request is received, a new (secondary) thread of control is established along with a private port for further communication regarding that file with the client. This thread remains active and the communications channel remains open until a **close** request is received from the client; the primary thread always continues to await new **open** requests.

When a secondary thread receives a **read** or **write** request it also receives additional information about the type and size of the request that is being made. Using this additional information the thread can calculate which packets are expected to be sent or received.

In the case of a **read** request, the client sends requests for the first several blocks to the client. This work-ahead allows several blocks to be in various states of transmission to the client and has resulted in significant data-rate benefits. The client and its servers implement a simple sliding window protocol to assure that packets are not lost and are assembled in the correct order at the client.

With a **write** request, the client sends out the data to be written one block at a time, and receiving an explicit acknowledgment for each. Using explicit acknowledgments was necessary to prevent the client from flooding the servers with packets while they were performing synchronous writes. In our experiments, the extra acknowledgments did not have a significant impact on the measured data rate, although in a faster network a more sophisticated protocol should be used. On receipt of a **close** request, the client expires the file handle and the storage agents release the ports and extinguish the threads dedicated to handling requests on that file.

4 Measurements of the Swift Prototype

To measure the performance of the Swift prototype, one, four, eight, and sixteen megabytes were both read from and written to Swift objects. In order to calculate confidence intervals, ten samples of each measurement were taken. Analogous tests were also performed using the local SCSI disk and the NFS file service. For all data rate measurements in this section, kilobytes is used to denote thousands of bytes per second.

In order to maintain cold caches, the file was mapped into the virtual address space of a flushing process. All pages of the file were then invalidated, which requires them to be refetched from disk when the file is next accessed. Finally, the mappings were removed to delete all references to the pages of the file. Other methods, such as dismounting the disk partition containing the file, were also tried and yielded similar results.

To calibrate the performance of the prototype we first measured the performance of the local SCSI disk on each Sun 4/20 that would act as a server. The measurements were similar for each host, and the results from one of these hosts is presented in table 1.

The results of the local SCSI measurements indicate that a Sun 4/20 with a Quantum 104S local disk is capable of reading data at approximately 425 kilobytes/second. The measured values decrease slightly as the amount of data to be read increases, with a precipitous drop when the file is 8 megabytes in length. The experiments with synchronous write measure the rate at which the local SCSI disk can write data to the actual device in 8 kilobyte blocks. This rate is approximately 70 kilobytes/second, and decreases with the length of the file. The most probable reason for this is the increased complexity in accessing data blocks as the length of the file increases. The experiments with asynchronous write measure the rate at which writes can be made by the file system to memory and then written to disk when the file system can do it most efficiently. This is the performance a user perceives when writing to the local disk. The data is not guaranteed to be written to disk even after the file has been closed. These measurements reveal that the rate at which the local file system of a Sun 4/20 will asynchronously write data is approximately 420 kilobytes/second for large files. There is a significant drop in performance from a relatively

small file (one megabyte) at over 550 kilobytes/second to a large file (sixteen megabytes) at 416 kilobytes/second. The reason for this drop in performance is that buffer space is exhausted and so the file system is forced to begin writing data to disk.

Table 1: Measured performance of local SCSI disk in kilobytes/second.

	Read			Synch-write			Asynch-write		
Request		95% Confidence			95% Confidence			95% Confidence	
megabytes	$\bar{x}$	low	high	$\bar{x}$	low	high	$\bar{x}$	low	high
1	452	403	501	80.9	79.8	82.1	559	555	562
4	428	426	430	72.5	72.1	72.8	471	463	480
8	397	393	402	68.1	67.9	68.3	422	416	428
16	424	409	439	67.0	66.6	67.4	416	414	418

The performance of a single Swift server is presented in table 2. The prototype performs nearly as well as accessing the local SCSI disk, with the prototype being slightly slower due to data transmission and protocol processing delays. The prototype sends several requests (in this case two), which allows the server to work-ahead on the file, processing further accesses to the local disk while the data is being transmitted. This enables the prototype to read approximately 350 kilobytes/second using a single Sun 4/20 as a server. When synchronous writes are considered, the prototype performs almost identically to the local file system, with the prototype being only slightly slower due transmission delays. When asynchronous writes are considered, the performance of the prototype with a single server is comparable with the local file system. The prototype is able to write approximately 400 kilobytes/second for large files. This is only slightly slower than the local file system, and can be attributed to the transmission delay introduced by the Ethernet.

Table 2: Swift performance with a single server in kilobytes/second.

	Read			Synch-write			Asynch-write		
Request		95% Confidence			95% Confidence			95% Confidence	
megabytes	$\bar{x}$	low	high	$\bar{x}$	low	high	$\bar{x}$	low	high
1	362	310	414	75.0	74.8	75.3	470	449	491
4	345	344	347	67.4	67.2	67.6	462	457	468
8	327	322	333	65.2	65.1	65.2	429	424	433
16	352	342	362	63.2	63.1	63.4	385	378	391

The performance of the prototype with two servers on a single Ethernet segment is summarized in table 3. In all cases, the prototype with two servers performs approximately twice as well as the prototype with a single server. When reads are considered, the prototype performs slightly less than twice as well with two servers as it did with a single server. The primary reason for this disparity, which as we will see becomes increasingly an issue when more servers are employed, is the ability of the SPARCstation 2 to receive data over the Ethernet. In the case of synchronous writes, the data rate of the prototype with two servers almost exactly doubles its data rate with a single server. There is no noticeable degradation since the Ethernet is only lightly loaded by the approximately 130 kilobytes/second being transferred. In the case of asynchronous writes, the prototype with two servers approximately doubles the data rate of the prototype with a single server.

The results for the experiments with the prototype using three servers on a single Ethernet segment are presented in table 4. In the case of reads, the data rate is approximately three times that of a single server. The data rate increase is not strictly linear since the Ethernet is becoming increasingly saturated, and the ability of the SPARCstation to receive data quickly over the Ethernet is limited. When synchronous writes are considered, the

Table 3: Swift performance with two servers on one Ethernet in kilobytes/second.

Request	Read			Synch-write			Asynch-write		
		95% Confidence			95% Confidence			95% Confidence	
megabytes	$\bar{x}$	low	high	$\bar{x}$	low	high	$\bar{x}$	low	high
1	689	622	756	158	157	159	868	845	892
4	668	664	671	144	143	144	908	897	919
8	664	656	672	135	135	135	908	894	923
16	653	651	655	128	128	129	861	852	870

data rate continues to scale approximately linearly in the number of servers. For asynchronous writes, the increase in data rate is also muted by the Ethernet. Even so, the performance is better than that of reads since transmitting data over the Ethernet requires fewer context-switches and less data copying than it does in receiving it, which puts less load on the SPARCstation 2.

Table 4: Swift performance with three servers on one Ethernet in kilobytes/second.

Request	Read			Synch-write			Asynch-write		
		95% Confidence			95% Confidence			95% Confidence	
megabytes	$\bar{x}$	low	high	$\bar{x}$	low	high	$\bar{x}$	low	high
1	943	928	959	247	245	248	917	884	950
4	956	943	969	213	213	213	1033	1024	1043
8	960	955	965	207	207	208	1046	1034	1058
16	955	949	960	196	196	196	1055	1045	1064

For the purposes of comparison, measurements were also made of the data rates provided by a high performance NFS file server. This server was a Sun 4/490 with 64 megabytes of memory and 5 gigabytes of disk space on IPI drives. This server is connected to the same SPARCstation 2 client that was used in the prototype experiments by a second network interface. The network is a shared departmental Ethernet, so measurements were conducted late at night to minimize interference from other tasks.

The results of these experiments are presented in table 5. In the case of reads, the high performance NFS file server performs approximately 73% better than the prototype with a single server. This is not surprising since the Sun 4/490 is designed to act as a file server, and the IPI disks are many times faster than the local SCSI disk used by the prototype. When synchronous writes are considered, the NFS file server performs approximately 46% better than the prototype with a single server. But when multiple Swift servers are used, the prototype provides significantly better performance than the NFS file server. When asynchronous writes are considered, the Swift server also provides significantly better performance when even a single server is employed. We must be careful when comparing Swift to NFS, since Swift is a prototype and does not provide all of the features that NFS must support. For example, NFS must be stateless and provide guarantees even on asynchronous writes that are much stronger than those provided by Swift.

The measurements of the prototype on a single Ethernet segment demonstrate that the Swift architecture can achieve high data rates on a local-area network by aggregating data rates from slower data servers. The prototype also validates the concept of distributed disk striping in a local-area network. This is demonstrated by the prototype providing data rates higher than both the local SCSI disk and the NFS file server.

Table 5: NFS performance from Sun 4/75 client to Sun 4/490 server.

Request	Read			Synch-write			Asynch-write		
		95% Confidence			95% Confidence			95% Confidence	
megabytes	$\bar{x}$	low	high	$\bar{x}$	low	high	$\bar{x}$	low	high
1	641	511	771	103	102	104	127	124	130
4	637	560	713	99.2	98.2	100	118	116	120
8	631	570	693	96.9	96.7	97.2	115	114	115
16	612	576	648	92.7	92.2	93.3	107	105	109

4.1 Measurements with a Second Ethernet Segment

To determine the effect of doubling the data rate capacity of the interconnection, we added a second Ethernet segment to the client and added additional storage agents. In this case the storage agents were Sun 4/20s (SLC) in faculty offices which have only 8 megabytes of memory, but have the same local SCSI disks as the hosts in the laboratory. This second Ethernet segment is shared by several groups in the department. Measurements were performed late at night, and during this period the load on the departmental Ethernet segment was seldom more than 5% of its capacity.

The interface for the second network segment was placed on the S-bus of the client. As the S-bus interface is known to achieve lower data rates than the on-board interface, we did not expect to obtain data rates twice as great as those using only the dedicated laboratory network. We also expected to see the network subsystem of the client to be highly stressed.

The results for two storage agents on two separate Ethernet segments are presented in table 6. When compared with the results for two storage agents on a single Ethernet segment, it is apparent that in the case of reads, having storage agents on two networks is slightly *worse* than having them on a single network. There are several reasons for this anomaly. First, since there are two network interfaces more interrupts must be fielded by the client. Second, the second network is shared by the entire department and so there was a slight load even late at night. Finally, the hosts on the departmental Ethernet segment have less memory than those in the laboratory. The results for synchronous writes are comparable with those obtained with a single Ethernet segment and the data rate scales linearly. The measurements of asynchronous writes are also comparable with those obtained with a single Ethernet segment. The reason for the difference in scaling is that a write puts less stress on the SPARCstation 2 that a read does. A write requires fewer context-switches and less data must be copied. A write also does not require an interrupt to signal its completion.

The results for four storage agents on two separate Ethernet segments is presented in table 7. When reads are considered, it is apparent that the data rate does not scale linearly since it is comparable with that of three storage agents on a single Ethernet segment. This is again due to the complexity of using two Ethernet interfaces. In the case of synchronous writes, the data rate continues to scale linearly, performing twice as well with four storage agents as it did with two. Asynchronous writes also perform very well, but has ceased to scale linearly since the client is approaching its capacity to transmit data over the two Ethernet interfaces.

In table 8, the effect of adding another pair of storage agents is considered. The data rate for read improves, although not in proportion to the number of storage agents. The client is now highly stressed in fielding interrupts and copying data. In the case of synchronous writes, the data rate continues to scale linearly. This is not unexpected since the low base data rates of the local SCSI disks do not stress the network. In the case of asynchronous writes, there is essentially no change from four storage agents since the client has reached the limits of ability to transmit data.

The measurements made of the prototype validate the concept of distributed disk striping and demonstrate that it scales well until a component becomes saturated. When faster components, whether processors or interconnection media the Swift architecture can will be able to make immediate use of it and its data rates will scale accordingly.

Table 6: Swift performance with two servers on two Ethernets in kilobytes/second.

Request	Read			Synch-write			Asynch-write		
		95% Confidence			95% Confidence			95% Confidence	
megabytes	$\bar{x}$	low	high	$\bar{x}$	low	high	$\bar{x}$	low	high
1	498	458	537	158	157	159	975	960	991
4	480	477	482	142	141	143	909	881	937
8	476	475	478	134	134	135	886	868	904
16	476	475	477	130	130	130	852	842	862

Table 7: Swift performance with four servers on two Ethernets in kilobytes/second.

Request	Read			Synch-write			Asynch-write		
		95% Confidence			95% Confidence			95% Confidence	
megabytes	$\bar{x}$	low	high	$\bar{x}$	low	high	$\bar{x}$	low	high
1	921	908	933	344	341	346	1409	1387	1430
4	927	923	930	288	287	290	1629	1582	1677
8	931	928	935	271	269	272	1634	1582	1686
16	932	925	939	262	261	262	1647	1588	1707

5 Simulation-based Performance Study

We modeled a hypothetical high-speed local-area token-ring implementation of the Swift architecture. Our primary goal of the simulation was to show how our architecture could exploit network and processor advances. Our second goal was to demonstrate that distributed disk striping is a viable technique that can provide the data rates required by I/O-intensive applications. Our third goal was to confirm the scaling properties of Swift.

Since we did not have the necessary network technology available to us, a simulation was the most appropriate exploration vehicle. The token-ring local-area network was assumed to have a transfer rate of 1 gigabit/second. All clients were modeled as disk-less hosts with a single network interface connected to the token-ring. The storage agents were modeled as hosts with a single disk device and a single network interface. To evaluate the possible effect of processor bottlenecks we simulated two processor types: 100 and 200 million instructions/second processors.

Table 8: Swift performance with six servers on two Ethernets in kilobytes/second.

Request	Read			Synch-write			Asynch-write		
		95% Confidence			95% Confidence			95% Confidence	
megabytes	$\bar{x}$	low	high	$\bar{x}$	low	high	$\bar{x}$	low	high
1	1100	1076	1124	521	508	534	1328	1245	1411
4	1274	1262	1286	443	441	445	1565	1513	1617
8	1204	1126	1281	414	412	417	1572	1543	1601
16	1245	1224	1265	394	393	396	1636	1624	1648

5.1 Structure of the Simulator

The simulator implemented a simplified version of the Swift architecture designed to address the goals stated above. It modeled neither caching, nor preallocation of resources, nor the providing of performance guarantees. These would have required appropriate traces of file system activity and such traces were unavailable to us. In addition, the simulator did not model the storage mediator. Since the storage mediator is not in the path of the data transmitted to and from clients but consulted only at the start of an I/O session, it is not part of the critical path in data rate evaluation.

The system is modeled by client requests that drive storage agent processes. A generator process creates client requests using an exponential distribution to govern request interarrival times. The client requests are differentiated according to pure read, pure write, and a conservative read-to-write ratio of 4:1 [3]. There is no modeling of overlapping execution, instead requests are modeled serially: only after a request is complete is the next issued.

In our simulation of Swift, for a read operation, a small request packet is multicast to the storage agents. The client then waits for the data to be transmitted by the storage agents. For a write operation the client transmits the data to each of the storage agents. Once the blocks from a write request have been transmitted, the client awaits an acknowledgment from the storage agents that the data have been written to disk.

The simulator models parity computations. Computing the data parity is an important factor in processor utilization. The simulator charges 5 instructions to compute the parity of each byte of data. We consider this to be a processor-expensive way of computing the parity that errs on the conservative side. Less expensive implementations would require less processor power to achieve the results we have simulated.

The disk devices are modeled as a shared resource. Multiblock requests are allowed to complete before the resource is relinquished. The time to transfer a block consists of the seek time, the rotational delay and the time to transfer the data from disk. The seek time and rotational latency are assumed to be independent uniform random variables, a pessimistic assumption when advanced layout policies are used [23]. Once a block has been read from disk it is scheduled for transmission over the network.

Our model of the disk access time is conservative in that advanced layout policies are not considered, no attempt was made to order requests to schedule the disk arm, and caches were not modeled. Staging data in the cache and sequential preallocation of storage would greatly reduce the number of seeks and significantly improve performance. As it is, our model provides a lower bound on the data rates that could be achieved.

Transmitting a message on the network requires protocol processing, time to acquire the token, and transmission time. The protocol cost for all packets has been estimated at 1,500 instructions [5] plus one instruction per byte in the packet. The time to transmit the packet is based on the network transfer rate. These estimated costs remain adequate as newer hardware technology has yet to decrease the total software overhead of accessing the network.

5.2 Simulation Results

The simulator gave us the ability to determine what data rates were possible given a configuration of processors, interconnection medium and storage devices. The modeling parameters varied were the processor speed of the intervening computing nodes, the number of disk devices representing storage agents and the size of the transfer unit.

The clear conclusion for obtainable data rate is that when sufficient interconnection capacity is available, the data rate is almost linearly related to both the number of storage agents and to the size of the transfer unit. Even though the cost of computing parity is non-negligible, the processor becomes a clear bottleneck only for write requests directed to large numbers of storage agents. When the 200 million instructions/second processor is used the utilization at the storage agents increases some 50% at high loads, as the processor in the client is less of a bottleneck.

For our simulation model, we have chosen disk parameters that are typical for a high-performance disk like those that are commonly used in the file servers. The disk is assumed to spin at 3600 revolutions/minute, yielding an average rotational latency of 8.3 milliseconds. Unless specific knowledge of the file system structure is available, the rotational latency can be accurately modeled by a uniform distribution. The average seek time is also assumed

to be uniformly distributed with a mean of 12 milliseconds. This assumption simplifies the seek time distribution, which in actual disk drives has an acceleration and deceleration phase not modeled by the uniform distribution.

The reason the transfer unit has such a large impact on the data rates achieved by the system is that seek time and rotational latency are enormous when compared to the speed of the processors and the network transfer rate. This also shows the value of careful data placement and indicates that resource preallocation may be very beneficial to performance.

As small transfer units require many seeks in order to transfer the data, large transfer units have a significantly positive effect on the data rates achieved. For small numbers of disks, seek time dominated to the extent that its effect on performance was almost as significant as the number of disks.

The per-message network data transfer processing costs are also an important factor in the effect of the transfer unit. For example, it was assumed that protocol processing required 1500 instructions plus 1 instruction per byte in the packet. As the size of the packet increases, the protocol cost decreases proportionally to the packet size. The cost of 1 instruction per byte in the packet is for the most part unavoidable, since it reflects necessary data copying.

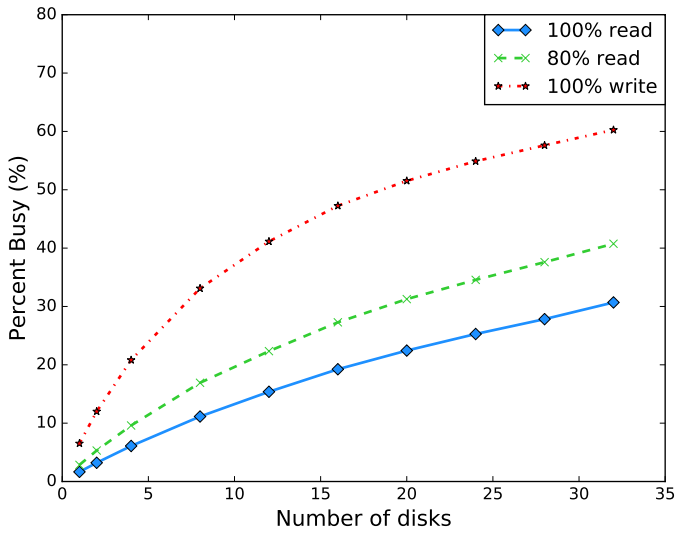

Figure 3: Fraction of processor used with 16384-byte blocks.

In figure 3 we see that the demands on the processor are significant even in the case of pure reads. The read costs are due to the cost of protocol processing and the necessary cost of copying of data from the system to the user address space. By having a hardware network interface that could merge the striped data directly into the user address space a significant amount of copying could be saved (in our simulation one instruction per byte). In the case of writes the processor can be seen to be a significant performance limiting factor. This is due to the cost of computing the parity code (exclusive-or). We have conservatively estimated that is would cost 5 instructions per byte to compute the parity operation. In the case of a highly optimized implementation, we believe that this could be brought down to approximately one instruction per byte on the average.

In figure 4 we see that the utilization of the disks decreases as more disks are used. This is due to the saturation of the processor, especially for writes, and the increased load on the interconnection network. Notice the correspondence to figure 3 processor utilization: the processor utilization for writes is high, while the corresponding utilization of the disks is low.

The fraction of the network capacity used is presented in figure 5. While the network capacity is not at all close to saturation it does have a significant load when a large number of disks are used. This high load has an effect on the data rate of the system (and on the utilization of the disks).

The read-to-write ratio is considered in figure 6. As with processor and disk utilization, the ratio of reads to writes has a significant effect on the data rate of the system. This is because of the increased load on the processor to compute the redundancy code when writing data. Since we are only considering large requests, writes are not

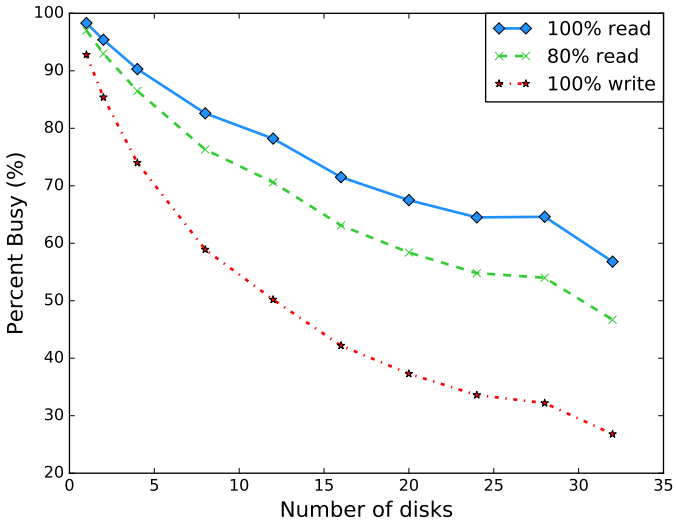

Figure 4: Fraction of disk used with 16384-byte blocks.

penalized by the multiple access costs associated with small writes. To preserve the parity function with small writes it is necessary to read the parity block even if only one block has been modified, and then to write both that block and the parity block.

In figure 7 we illustrate the effect of block size on the data rate. As one expects, the larger the block size the higher the data rate. The data rate will continue to increase until a cylinder boundary is reached forcing a disk seek per logical request, or until the transfer rate capacity of the device is reached. This illustrates that both the rotational latency and the seek time are significant sources of delay in the system.

5.2.1 The Effect of Doubling the Processor Speed

We modeled the effect of doubling the processor speed, to evaluate its effect on the processor bottleneck. In figure 8 we see that for the highest number of disks, 32, the percent busy on the processor decreased from 30% to 18% for a pure read load, and from 60% to 43% for a pure write load.

The simulation also showed that one client can now achieve pure read data rates on the order of 23 megabytes/second for 32 disks, in contrast to the 20 megabytes/second depicted in figure 6. For pure writes the system can achieve data rates of 13 megabytes/second for 32 disks, versus the 9 megabytes/second depicted in figure 6. As for the disk themselves, when 32 of them are being used simultaneously they are now utilized to 68% of their capacity for pure reads and to 39% of their capacity for pure writes. The corresponding utilization percentages, in figure 4 were 58 and 27, respectively.

The substantial increase of the data rate for pure writes highlights the effect of the cost of computing error-correcting parity codes in software. With the faster processor the bottleneck seems to have shifted to the serial nature of the data transmission protocol modeled. This explains why the pure read data rates did not increase as much as the write data rates did, even though both the network interconnection medium and the disk storage subsystem had spare capacity.

6 Related Research

The notion of *disk striping* was formally introduced by Salem and Garcia-Molina [24]. The technique, however, has been in use for many years in the I/O subsystems of super computers [15] and high-performance mainframe systems [13]. Disk striping has also been used in some versions of the UNIX operating system as a means of

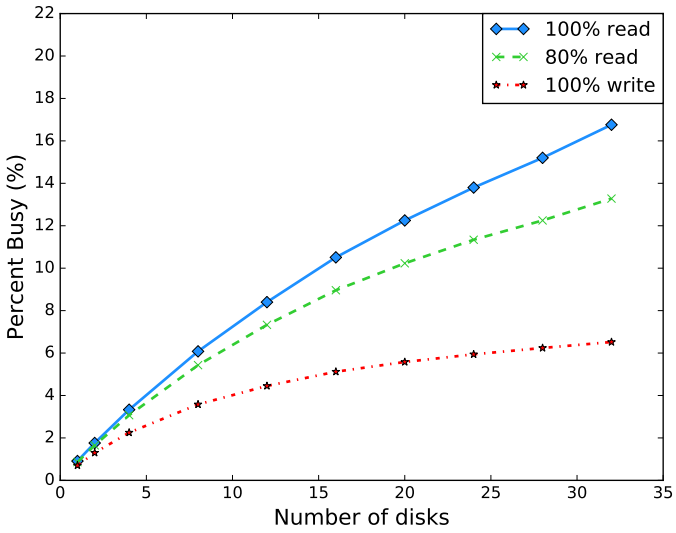

Figure 5: Fraction of network capacity used with 16384-byte blocks.

improving swapping performance [24]. To our knowledge, Swift is the first to use disk striping in a distributed environment, striping files over multiple servers.

Examples of some commercial systems that utilize disk striping include super computers [15], DataVault for the CM-2 [26], the airline reservation system TPF [13], the IBM AS/400 [7], CFS from Intel [21, 22], and the Imprimis ArrayMaster [14]. Hewlett-Packard is developing a system called DataMesh that uses an array of storage processors connected by a high-speed network [27]. For all of these the maximum data rate is limited by the interconnection medium which is an I/O channel. Higher data rates can be achieved by using multiple I/O channels.

The aggregation of data rates proposed in the Swift architecture generalizes that proposed by the RAID disk array system [20, 18, 19] in its ability to support data rates beyond that of the single disk array controller. In fact, Swift can concurrently drive a collection of RAIDs as high speed devices. Due to the distributed nature of Swift, it has the further advantage over RAID of having no single point of failure, such as the disk array controller or the power supply.

Swift improves on traditional disk striping systems in two important areas: scaling and reliability. By interconnecting several communication networks, Swift is more scalable than centralized systems. When higher performance is required additional storage agents can be added to the Swift system increasing its performance proportionally. By selectively hardening each of the system components, Swift can achieve arbitrarily high reliability of its data, metadata, and communication media. In CFS, for example, there is no mechanism present to make its metadata tolerant of storage failures. In CFS, if the repository on which the descriptor of a multi-repository object fails, the entire object becomes unavailable.

A third difference from traditional disk striping systems is that Swift has the advantages of sharing and of decentralized control of a distributed environment. Several independent storage mediators may control a common set of storage agents. The distributed nature of Swift allows better resource allocation and sharing than a centralized system. Only those resources that are required to satisfy the request need to be allocated.

Swift incorporates data management techniques long present in centralized computing systems into a distributed environment. In particular, it can be viewed as a generalization to distributed systems of I/O channel architectures found in mainframe computers [4].

6.1 Future Work

There are two areas that we intend to address in the future: enhancing our current prototype and simulator, and extending the architecture.

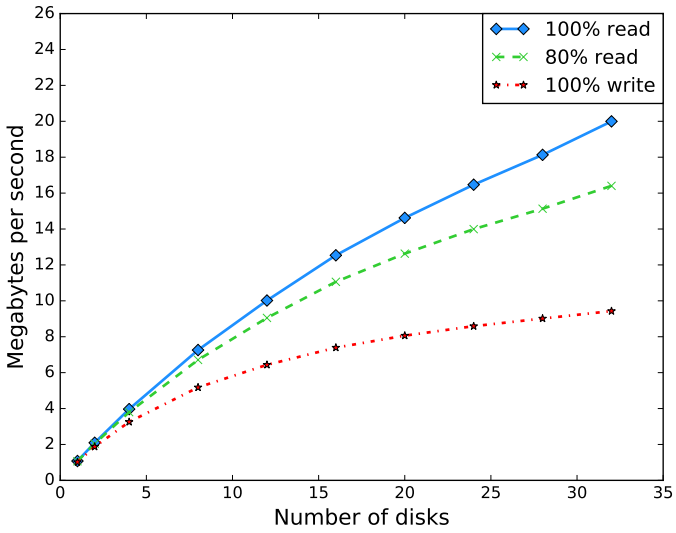

Figure 6: Data-rate with 16384-byte blocks varying read-to-write ratio.

6.1.1 Enhancements to the Prototype

The current prototype needs to implement data redundancy. The prototype will be enhanced with code that computes the check data, and both the **read** and **write** operations will have to be modified accordingly. With this enhancement in place, we plan to measure the effect that computing the check data has on data rates.

We also plan to incorporate mechanisms to do resource preallocation and to build transfer plans. With these mechanisms in place, we plan to study different resource allocation policies, with the goal of understanding how to handle variable loads.

6.1.2 Enhancements to the Architecture

The goal of this research is to develop a high-speed distributed storage system that provides a general purpose file system with integrated storage and retrieval of large data objects, such as digital audio and video, at guaranteed data rates. Applications of such a system range from visualization of scientific computations to real-time recording, editing and play-back of color video.

Support for integrated access to continuous media such as digital audio and video is difficult for current computing systems. They lack the necessary capacity to provide data at a sufficient rate, and do not support the necessary end-to-end performance guarantees. The interest in fields such as scientific visualization require that this be addressed.

We intend to extend the architecture with techniques for providing data rate guarantees for magnetic disk devices. While the problem of real-time processor scheduling has been extensively studied [28, 29], and the problem of providing guaranteed communication capacity is also an area of active research [2], the problem of scheduling real-time disk transfers has received considerably less attention.

A second area of extensions is in the co-scheduling of services. In the past, only analog storage and transmission techniques have been able to meet the stringent demands of multimedia audio and video applications. To support integrated continuous multimedia, resources such as the central processor, peripheral processors (audio, video), and communication network capacity must be allocated and scheduled together to provide the necessary data-rate guarantees. This *meta-scheduling* has been studied by Anderson [1].

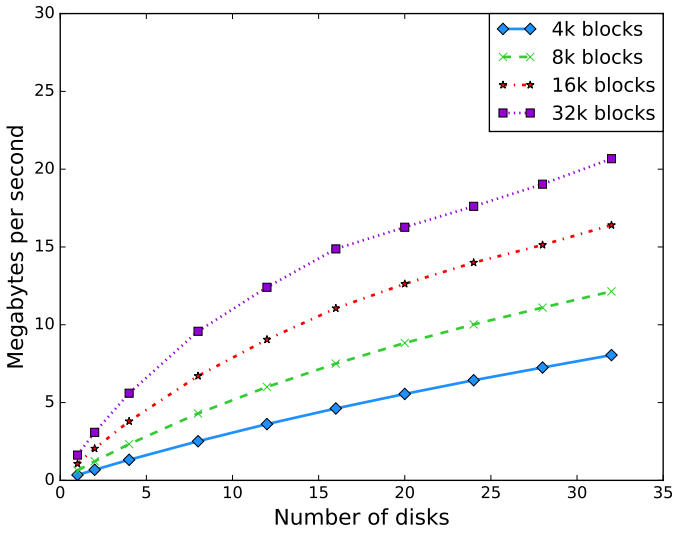

Figure 7: Effect of block size on throughput.

7 Conclusions

This paper presents two studies conducted to validate Swift, a scalable distributed I/O architecture that achieves high data rates by striping data across several storage agents and driving them concurrently. The prototype validates the concept of distributed disk striping in a local-area network.

A prototype of the Swift architecture was built using UNIX and an Ethernet-based local-area network. It demonstrated that the prototype of the architecture can achieve high data rates on a local-area network by aggregating data rates from slower data servers. Using three servers on a single Ethernet segment, the prototype achieved more than double the data rates than were provided by access to the local SCSI disk, and it achieved then times the NFS data rate for asynchronous writes, double the NFS data rate for synchronous writes, and almost twice the NFS data rate for reads. The performance of our local-area network prototype was limited by the speed of the Ethernet-based local-area network.

When a second Ethernet path was added between the client and the storage agents, the data rates measured demonstrated that the Swift architecture can make immediate use of a faster interconnection medium. The data rates for writes almost doubled. For reads, the improvements were less pronounced because the client could not absorb the increased network load.

Our simulations of the architecture show how Swift can exploit more powerful components in the future, and which components limit I/O performance. The simulations show that data rates under Swift scale proportionally to the size of the transfer unit and the number of storage agents when sufficient interconnection and processor capacity are available.

Even though Swift was designed with very large objects in mind, it can also handle small objects, such as those encountered in normal file systems. The penalties incurred are one round trip time for a short network message, and the cost of computing the parity code. Swift is also well suited as a swapping device for high performance work stations if no data redundancy is used.

The distributed nature of Swift leads us to believe that it will be able to exploit all the current hardware trends well into the future: increases in processor speed and network capacity, decreases in volatile memory cost, and secondary storage becoming very inexpensive but not correspondingly faster. The Swift architecture also has the flexibility to use alternative data storage technologies, such as arrays of digital audio tapes.

Lastly, a system like our prototype can be installed easily into an existing operating system without needing to modify the underlying networking hardware or file specific software. It can then be used to exploit the emerging high-speed networks using the large installed base of current file servers.

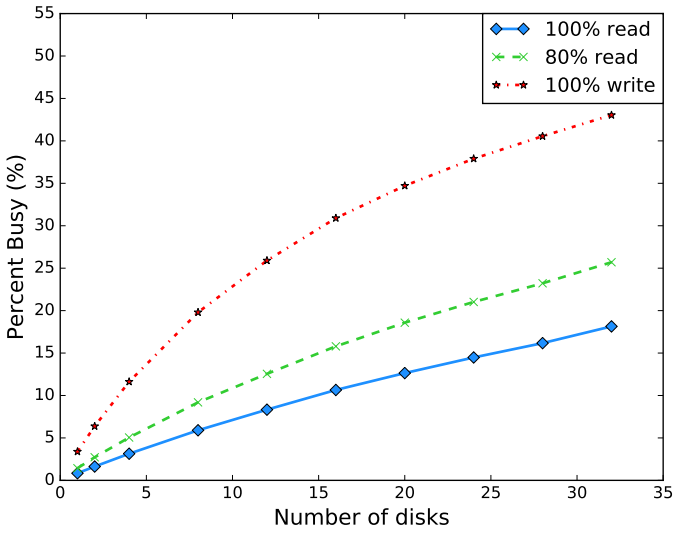

Figure 8: 200 MIPS: Fraction of processor used with 16384-byte blocks.

Acknowledgments

We are grateful to those that contributed to this research including Aaron Emigh and Dean Long for their work with the prototype, Laura Haas and Mary Long for their thoughtful comments on the manuscript, and John Wilkes for stimulating discussions on distributed file systems. Simulation results were obtained with the aid of SIMSCRIPT, a simulation language developed and supported by CACI Products Company of La Jolla, CA.

References

[1] D. Anderson. Meta-Scheduling for Distributed Continuous Media. Technical Report UCB/CSD 90/599, University of California, Berkeley, Oct. 1990.

[2] D. P. Anderson, R. G. Herrtwich, and C. Schaefer. SRP: A Resource Reservation Protocol for Guaranteed Performance Communication in the Internet. Technical Report UCB/CSD 90/596, University of California, Berkeley, Sept. 1990.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium of Operating Systems Principles*, pp. 198–212, 1991.

[4] J. Buzen and A. Shum. I/O architecture in MVS/370 and MVS/XA. *CMG Transactions*, 54:19–26, 1986.

[5] L.-F. Cabrera, E. Hunter, M. J. Karels, and D. Hosher. User-Process Communication Performance in Networks of Computers. *IEEE Transactions on Software Engineering*, 14(1):38–53, 1988.

[6] L.-F. Cabrera and J. Wyllie. QuickSilver Distributed File Services: An Architecture for Horizontal Growth. In *Proceedings of the Second IEEE Conference on Computer Workstations*, pp. 23–37, Santa Clara, California, Mar. 1988.

[7] B. E. Clark and M. J. Corrigan. Application System/400 performance characteristics. *IBM Systems Journal*, 28(3):407–423, 1989.

[8] D. Comer. Internetworking with TCP/IP: Principles, Protocols, and Architecture, 1988.

[9] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *Computing Surveys*, 17(3):341–370, Sept. 1985.

[10] H. Garcia-Molina and K. Salem. The Impact of Disk Striping on Reliability. *IEEE Database Engineering Bulletin*, 11(1):26–39, Mar. 1988.

[11] G. A. Gibson, L. Hellerstein, R. M. Karp, and D. A. Patterson. Failure Correction Techniques for Large Disk Arrays. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 123–132. ACM, 1989.

[12] J. N. Gray. *Notes on database operating systems*, pp. 393–481. Springer-Verlag, 1979.

[13] IBM Corporation. *TPF-3 Concepts and Structure Manual*.

[14] Imprimis Technology. *ArrayMaster 9058 Controller*, 1989.

[15] O. G. Johnson. Three-dimensional wave equation computations on vector computers. *Proceedings of the IEEE*, 72(1):90–95, Jan. 1984.

[16] A. C. Luther. *Digital Video in the PC Environment*. McGraw-Hill, 1989.

[17] M. Nelson, B. Welch, and J. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[18] S. Ng. Pitfalls in designing disk arrays. In *Proceedings of the IEEE COMPCON Conference*, Feb. 1989.

[19] S. Ng. Some design issues of disk arrays. In *Digest of Papers of the Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*, pp. 137–142. IEEE, 1989.

[20] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pp. 109–116. ACM, 1988.

[21] P. Pierce. A concurrent file system for a highly parallel mass storage system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pp. 155–160, 1989.

[22] T. W. Pratt, J. C. French, P. M. Dickens, and S. A. Janet. A Comparison of the Architecture and Performance of Two Parallel File Systems. In *Proceedings of the Fourth Conference on Hypercubes*, Monterey, Mar. 1989.

[23] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. pp. 1–15, Oct. 1991.

[24] K. Salem and H. Garcia-Molina. Disk Striping. In *Proceedings of the Second International Conference on Data Engineering*, pp. 336–342. IEEE, 1986.

[25] M. Stonebraker and G. A. Schloss. Distributed RAID—a new multiple copy algorithm. In *Proceedings of the Sixth International Conference on Data Engineering (ICDE '90)*, pp. 430–437, Feb. 1990.

[26] Thinking Machines, Incorporated. *Connection Machine Model CM-2 Technical Summary*, May 1989.

[27] J. Wilkes. DataMesh-Project Definition Document. Technical Report HPL-CSP-90-1, Hewlett-Packard Laboratories, 1990.

[28] W. Zhao. *A heuristic approach to scheduling hard real-time tasks with resource requirements in distributed systems*. Ph. D. dissertation, University of Massachussetts, Amherst, 1986.

[29] W. Zhao, K. Ramamritham, and J. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, 36(8):949–960, 1987.