Exploiting Multiple I/O Streams to Provide High Data-Rates

Luis-Felipe Cabrera
IBM Almaden Research Center
Computer Science Department

Internet: cabrera@ibm.com

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

Internet: darrell@sequoia.ucsc.edu

Abstract

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

In installations with a high-speed interconnection medium, Swift will provide high data-rate transfers by using multiple slower storage devices in parallel. The data-rates obtained with this approach scale well when using multiple storage devices and multiple interconnections. Swift has the flexibility to use any appropriate storage technology, including disk arrays. The ability to adapt to technological advances will allow Swift to provide for ever increasing I/O demands. 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. Using a single Ethernet-based local-area network and three servers, the prototype provides data-rates that are almost three times as fast as access to the local SCSI disk in the case of writes. When compared with NFS, the Swift prototype provides double the data-rate for reads and eight times the data-rate for writes. The data-rate of our prototype scales almost linearly in the number of servers and the number of network segments. Its performance is shown to be limited by the speed of the Ethernet-based local-area network.

We also constructed a simulation model to show how the Swift architecture can exploit storage, communication, and processor advances, and to locate the components that will limit I/O performance. In a simulated gigabit/second token ring local-area network the data-rates are seen to scale proportionally to the size of the transfer unit and to the number of storage agents.

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

1 Introduction

The current generation of distributed computing systems are incapable of integrating high-quality video with other data in a general purpose environment. Multimedia applications that require this level of service include scientific visualization, image processing, and recording and play-back of color video. The data-rates required by some of these applications vary from 1.2 megabytes/second for DVI compressed video and 1.4 megabits/second for CD-quality audio [1], to more than 20 megabytes/second for full-frame color video.

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, called Swift, solves the problem of storing and retrieving very large data objects from slow secondary storage at very high data-rates. The goal of Swift is to support integrated continuous multimedia in a general purpose distributed storage system. Swift uses disk striping [2], much like RAID [3], driving the disks in parallel to provide the required data-rate. Since Swift, unlike RAID, was designed for distributed systems it provides the advantages of easy expansion and load sharing. Swift also provides better resource utilization since it will use only those resources that are necessary to satisfy the request. In addition, Swift has the flexibility to use any appropriate storage technology, including a RAID or an array of digital audio tapes.

Two studies have been conducted to validate the Swift architecture. The first was a proof-of-concept prototype of a simplified version of Swift implemented on an Ethernet-based local-area network using the UNIX[1] operating system. This prototype provides a file system with UNIX semantics. It uses distributed disk striping over multiple servers to achieve high data-rates. On a single Ethernet-based local-area network, the prototype achieves data-rates up to three times as fast as the data-rate to access the local SCSI disk in the case of writes. When compared to a high-performance NFS file server, the Swift prototype exceeds the NFS data-rate for writes by eight times, and provides almost double the NFS data-rate for reads.

The second study is a discrete-event simulation of a simplified local-area instance of the Swift architecture. This 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, and is used to locate the components that will limit I/O performance.

The remainder of this paper is organized as follows: a brief description of 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 idea of striping data over multiple storage devices and driving them in parallel. The principle behind our architecture is simple: aggregate arbitrarily many (slow) storage devices into a faster logical service, making all applications unaware of this aggregation. Several concurrent I/O architectures, such as Imprimis ArrayMaster [4], DataVault [5], CFS [6, 7] and RAID [3, 8], are based on this observation. Mainframes [9, 10] and super computers [11] have also exploited this approach.

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

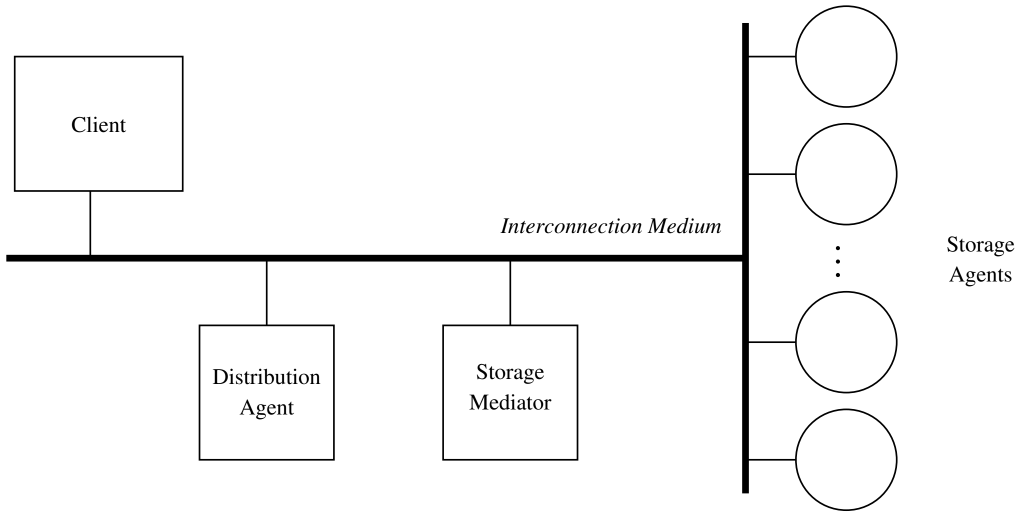

Figure 1: Components of the Swift Architecture

Swift is a 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. In this paper we concentrate on a prototype implementation and a simulation of the Swift architecture. A more detailed description of the architecture and its rationale can be found elsewhere [12, 13].

Swift assumes that objects are managed by *storage agents*. The system 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 from 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 will be available, and that negotiations among 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. 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. Figure 1 depicts the components of the Swift architecture.

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 Swift can spread the data over only a few storage agents. If the required data-rate is high, then the striping unit will be chosen small enough to exploit all the parallelism needed to satisfy the request.

Partial failures are an important concern in a distributed architecture such as Swift. 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 the *multiple copy* [14] and *computed copy*

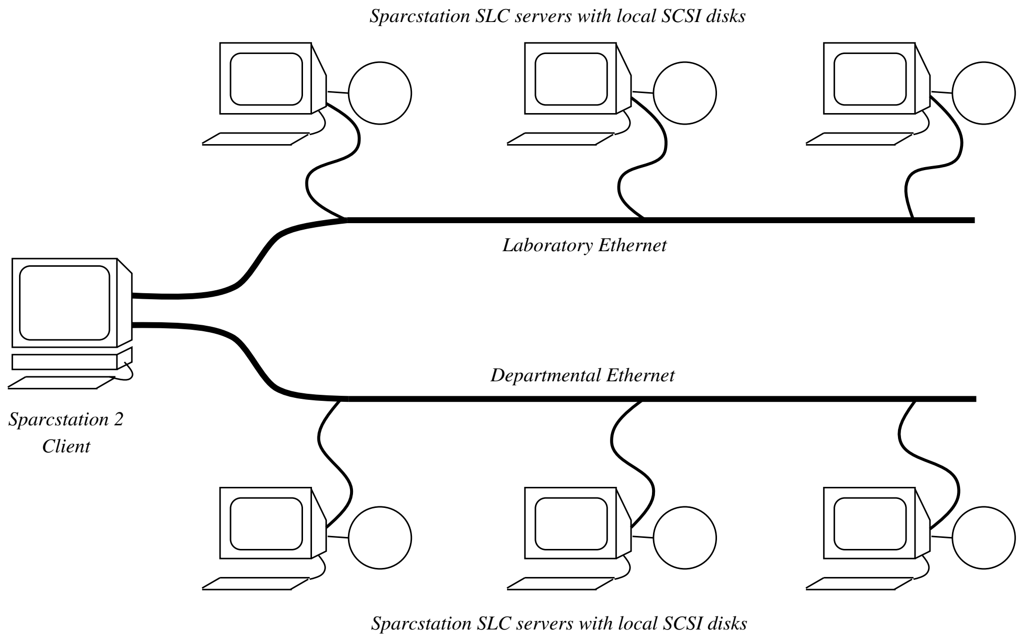
Sparcstation SLC servers with local SCSI disks
Laboratory Ethernet
Departmental Ethernet
Sparcstation 2 Client
Sparcstation SLC servers with local SCSI disks

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

[3] approaches. In the Swift prototype we propose to use computed copy redundancy since this approach provides resiliency in the presence of a single failure (per group) at a low cost in terms of storage but at the expense of some additional computation.

3 Ethernet-based Implementation of Swift

A simplified prototype of the Swift architecture has been built as a set of libraries using the standard filing and networking facilities of the Unix operating system. We have used file system facilities to name and store objects which makes the storage mediators unnecessary. Transfer plans do not need to be built explicitly as the library interleaves data uniformly among the set of files used to service a request.

Objects administered by Swift are striped over several servers, each of which has its own local SCSI disk and act as storage agents on an Ethernet-based local-area network. Clients are provided with **open**, **close**, **read**, **write** and **seek** operations that have Unix file system semantics.

The Swift distribution agent is embedded in the libraries and is represented by the client. The storage agents are represented by Unix processes on servers which 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 might be striped over any number of storage agents, the most important performance-limiting factors are the transmission speed of the communication network and the data-rate at which the client and the servers can send and receive packets over the network.

The current prototype has allowed us to confirm that a high aggregate data-rate can be achieved with Swift. The data-rates of an earlier prototype using a data transfer protocol built on the TCP [15] network protocol proved to be unacceptable. The current prototype was built using a light-weight data transfer protocol on top of the UDP [15] network protocol. To avoid unnecessary data copying, scatter-gather I/O was used to have the kernel deposit the message directly into the user buffer.

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-based local-area network. At first **select** seemed to be the performance limiting factor. A closer inspection revealed that using TCP was not appropriate as buffer management problems prevented the prototype from achieving high data-rates.

In the current prototype the client is a Sun 4/75 (SPARCstation 2). It has a list of the hosts that act as storage agents. All storage agents were placed on Sun 4/20s (SLC). 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

Each Swift storage agent waits for **open** requests on a well-known IP [15] port. When an **open** request is received, a new (secondary) thread of control is established along with a private port for further communication about that file with the client. This thread remains active and the communications channel remains open until the file is **closed** by 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 handling of a **read** operation, packet loss rates caused by lack of buffer space in the SunOS kernel necessitated that the client maintain only one outstanding packet request per storage agent, while the storage agents fulfilled the packet requests as soon as they were received. As the measurements will show, this had a negative effect on the performance of the prototype. No acknowledgments are necessary with **read**, since the client keeps sufficient state to determine what packets have been received and thus can resubmit requests when packets are lost.

With a **write** operation, the client sends out the data to be written as fast as it can. Each storage agent checks the packets it receives against the packets it was expecting and either acknowledges receipt of all packets or sends requests for packets lost. The client requires explicit acknowledgements from the storage agents to determine that a **write** operation is successful. On receipt of a **close** operation, the client expires the file handle and the storage agents release the ports and extinguish the threads dedicated to handling requests on that file.

Several problems were encountered by our current prototype with SunOS. For both **read** and **write**, we often ran out of buffer space on the client. When writing at full speed, the kernel would drop packets and claim that they had been sent. Because of this, the prototype could not write as fast as possible; we had to incorporate a small wait loop between **write** operations.

4 Measurements of the Swift Implementation

To measure the performance of the Swift prototype, three, six, and nine megabytes were read from and written to a Swift object. In order to calculate confidence intervals, eight samples of each measurement were taken. Analogous tests were performed using the local SCSI disk and the NFS file service. All data-rate measurements in this paper are given in kilobytes per second. Maintaining cold caches was achieved by using **/etc/umount** to flush the caches as a side effect. Other methods such as creating a large virtual address space to reclaim pages were also tried with similar results.

Table 1: Swift **read** and **write** data-rates on a single Ethernet in kilobytes/second.

Operation	$\bar{x}$	σ	min	max	90% Confidence	
					Low	High
Read 3 MB	893	18.6	847	904	880	905
Read 6 MB	897	3.4	891	900	894	899
Read 9 MB	876	16.6	848	892	865	887
Write 3 MB	860	44.6	767	890	830	890
Write 6 MB	882	5.00	875	889	879	885
Write 9 MB	881	1.01	857	889	874	888

The measurement data for **read** and **write** with Swift were obtained using a single client and three storage agents. The client was a Sun 4/75 (SPARCstation 2) with 16 megabytes of memory and a local 207 megabyte local SCSI disk under SunOS[2] 4.1.1. The three storage agents were Sun 4/20s with 16 megabytes of memory and a local 104 megabyte local SCSI disk also under SunOS 4.1.1. These hosts were placed on a 10 megabit/second dedicated Ethernet-based local-area network. Aside from the standard system processes, each of the servers was dedicated to run exclusively the Swift storage agent software. The results are summarized in Table 1.

Measurements of synchronous **write** operations with the Swift prototype have not been obtained at this time. We encountered a problem with SunOS that would not allow us to have the storage agents write synchronously to disk due to insufficient buffer space.

For Swift, the limiting performance factor was the Ethernet-based local-area network. Using three Swift storage agents, the utilization of the network ranged from 77% to 80% of its measured maximum capacity of 1.12 megabytes/second. Including a fourth storage agent would only saturate the network while not significantly increasing performance.

The measurements for a local SCSI disk connected to a Sun 4/20 (SLC) with 16 megabytes of memory under SunOS 4.1.1 are given in Table 2. All measurements were taken with a cold cache. The effect of the synchronous mode SCSI are most apparent for the **read** data-rates which are twice as fast as those that we obtained with version 4.1 of SunOS, which provided only asynchronous SCSI mode. All **write** operations to the SCSI disk were done synchronously.

The NFS measurements made using a Sun 4/390 with 32 megabytes of memory and

[2]The version of the operating system is significant since SunOS 4.1.1 allowed the use of synchronous mode on the SCSI drives. This doubled the read data-rate.

IPI disk drives under SunOS 4.1 as a server, and a Sun 4/75 (SPARCstation 2) as the client are summarized in Table 3. The NFS measurements were run over a lightly-loaded shared departmental Ethernet-based local-area network, not over the dedicated laboratory network. The traffic present in this shared network when the measurements were made was less than 5% of its capacity. This level of network traffic load should not significantly affect the measured data-rates.

Table 2: SCSI **read** and **write** data-rates in kilobytes/second.

Operation	$\bar{x}$	σ	min	max	90% Confidence	
					Low	High
Read 3 MB	654	10.3	641	668	647	661
Read 6 MB	671	6.4	662	682	666	674
Read 9 MB	682	2.4	679	685	680	683
Write 3 MB	314	1.3	312	316	313	315
Write 6 MB	316	0.6	315	316	315	316
Write 9 MB	315	2.1	310	316	313	316

Table 3: NFS **read** and **write** data-rates in kilobytes/second.

Operation	$\bar{x}$	σ	min	max	90% Confidence	
					Low	High
Read 3 MB	462	56.0	375	531	424	491
Read 6 MB	456	30.4	406	490	435	476
Read 9 MB	488	22.1	444	516	473	502
Write 3 MB	112	4.1	107	117	109	114
Write 6 MB	109	5.2	98	114	105	112
Write 9 MB	111	1.9	108	114	109	112

The measurements shown in Table 1 through Table 3 indicate that the Swift prototype achieves significantly higher data-rates than either the local SCSI disk or the NFS file system, even when NFS uses a high-performance server with the best IPI disk drives Sun had available[3] at the time. The Swift data-rates were up to three times better than those achieved accessing the local SCSI disk. The difference was even more accentuated a few months ago when only asynchronous SCSI mode was available. In the case of **read**, the data-rates of the Swift prototype were nearly double those of NFS, while in the case of **write** the data-rates were more than eight times those of NFS. As mentioned before, the comparison of Swift **write** data-rates with NFS **write** data-rates is not completely straightforward since NFS does synchronous writes to the disk.

When compared with the local SCSI disk performance, the Swift prototype only performs between 29% and 36% better. This contrasts sharply with previous measurements

[3]These drives were purchased Autumn 1990.

taken under SunOS 4.1 where the Swift prototype performed about 250% better than local SCSI read access. This change is attributable to the availability of synchronous SCSI mode under SunOS 4.1.1 and it supports the assertion that the performance of the Swift prototype is limited primarily by the Ethernet-based local-area network.

In contrast to its read performance, when writes are considered, the Swift prototype shows between a 274% and a 280% increase over that of the local SCSI disk. The ideal performance improvement would have been 300% if the interconnection medium were not limiting performance. Since the performance of the Swift prototype is less than 300% of the local SCSI performance, this again supports the assertion that factor most limiting the performance of the Swift prototype is the Ethernet-based local-area network.

When the Swift prototype is compared with the high-performance NFS file server, its performance is between 180% and 197% better in the case of reads. This shows that Swift can successfully provide increased I/O performance by aggregating several low-speed storage agents and driving them in parallel.

In the case of writes, the Swift prototype performs between 767% and 809% better than the high-performance NFS file server. When interpreting the measurements one should also keep in mind that the **write** data-rate measurements in NFS reflect the write-through policy of the server. This makes data-rates for **write** somewhat difficult to compare with those of Swift.

While the Swift **write** performance was measured using asynchronous writes, the values obtained are not unfair to the NFS server for three reasons. First, the local SCSI measurements were obtained using synchronous writes and are clearly more than one third the speed of the Swift performance numbers. Second, the Ethernet-based local-area network is clearly the limiting factor in the performance of the prototype. Third, the speed of the IPI disk drives (rated at more than 3 megabytes/second) should not so severely impact the performance when compared with the SCSI drives. Thus, the way in which writes are done in the Swift prototype is not the dominant performance factor.

The Swift prototype demonstrates 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 Swift prototype providing data-rates higher than both the local SCSI disk and the NFS file server.

4.1 Effect of Adding a Second Ethernet

To determine the effect of doubling the data-rate capacity of the interconnection, we added a second Ethernet-based local-area network segment between the client and additional storage agents. This second network segment is shared by several groups in the department. During the measurement period its load was seldom more than 5% of its capacity.

The interface for the second network sement 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. To our surprise, our measurements show that for **write** operations the Swift prototype almost doubled its data-rate.

In the case of reads, the increase in performance of the Swift prototype is less pronounced. This can be attributed to several factors including the increased load on the client and a lack of buffer space. It can also be attributed to the increased complexity of the read protocol which requires many more packets to be sent than does the write protocol.

The measurements for the Swift prototype using both the dedicated laboratory network and the shared departmental network are summarized in Table 4. These measurements demonstrate that the Swift architecture can make immediate use of a faster interconnection medium and that its data-rates scale accordingly.

Table 4: Swift **read** and **write** data-rates on two Ethernets in kilobytes/second.

Operation	$\bar{x}$	σ	min	max	90% Confidence	
					Low	High
Read 3 MB	1120	36.8	1040	1150	1093	1143
Read 6 MB	1150	8.5	1140	1170	1145	1156
Read 9 MB	1130	11.0	1120	1150	1126	1140
Write 3 MB	1660	10.1	1640	1670	1650	1663
Write 6 MB	1670	3.0	1660	1670	1665	1669
Write 9 MB	1660	14.3	1630	1680	1652	1671

5 Simulation-based Performance Study

To evaluate the scaling properties of the architecture we modeled a hypothetical implementation on a high-speed local-area token-ring network. The main goal of the simulation was to show how the architecture could exploit network and processor advances. A second goal was to demonstrate that distributed disk striping is a viable technique that can provide the data-rates required by applications such as multimedia.

Since we did not have the necessary network technology available to us, a simulation was the appropriate exploration vehicle. The token-ring local-area network was assumed to have a transfer rate of 1 gigabit/second. The clients were modeled as diskless hosts with a 100 million instructions/second processor and a single network interface connected to the token-ring. The storage agents were modeled as hosts with a 100 million instructions/second processor and a single disk device. In our simulation runs no more than 22% of the network capacity was ever used, and so our data-rates were never limited by lack of network capacity.

5.1 Structure of the Simulator

The simulator was used to locate the components that were the limiting factors for a given level of performance. The simulator did not model caching, did not model computing data parity blocks, did not model any preallocation of resources, nor did it attempt to provide performance guarantees. Traces of file system activity would have been required in order to model these effectively and such traces were unavailable to us. In addition, the simulator did not model the storage mediator as it is not in the path of the data transmitted to and from clients, but is consulted only at the start of an I/O session.

Components of the system are modeled by client requests and storage agent processes. A generator process creates client requests using an exponential distribution for request interarrival times. The client requests are differentiated according to a read-to-write ratio. In each of the following figures, this ratio has been conservatively estimated to be 4:1, motivated by the Berkeley study [16] and by our belief that for continuous media read access will strongly dominate write access.

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

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 [17]. 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 [18] plus one instruction per byte in the packet. The time to transmit the packet is based on the network transfer rate.

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 type and number of disk devices, representing storage agents, and the size of the transfer unit.

The clear conclusion 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. The reason the transfer unit impacts so much 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.

In Figure 3, the amount of time required to satisfy a 1-megabyte client request was plotted against the number of requests issued per second. The base storage device considered was a Fujitsu M2372K, which is typical for 1990 file servers.

The load that could be carried depended both on the number of disks used and the block size. The delay was dominated by the disk, with an average seek time of 16 milliseconds, an average rotational delay of 8.3 milliseconds and a transfer rate of 2.5 megabytes per second. The result was that transferring 32 kilobytes required about 37 milliseconds on the average. As the block size was increased, seek time and rotational delay were mitigated and the transfer time became more dependent on the amount of data transferred.

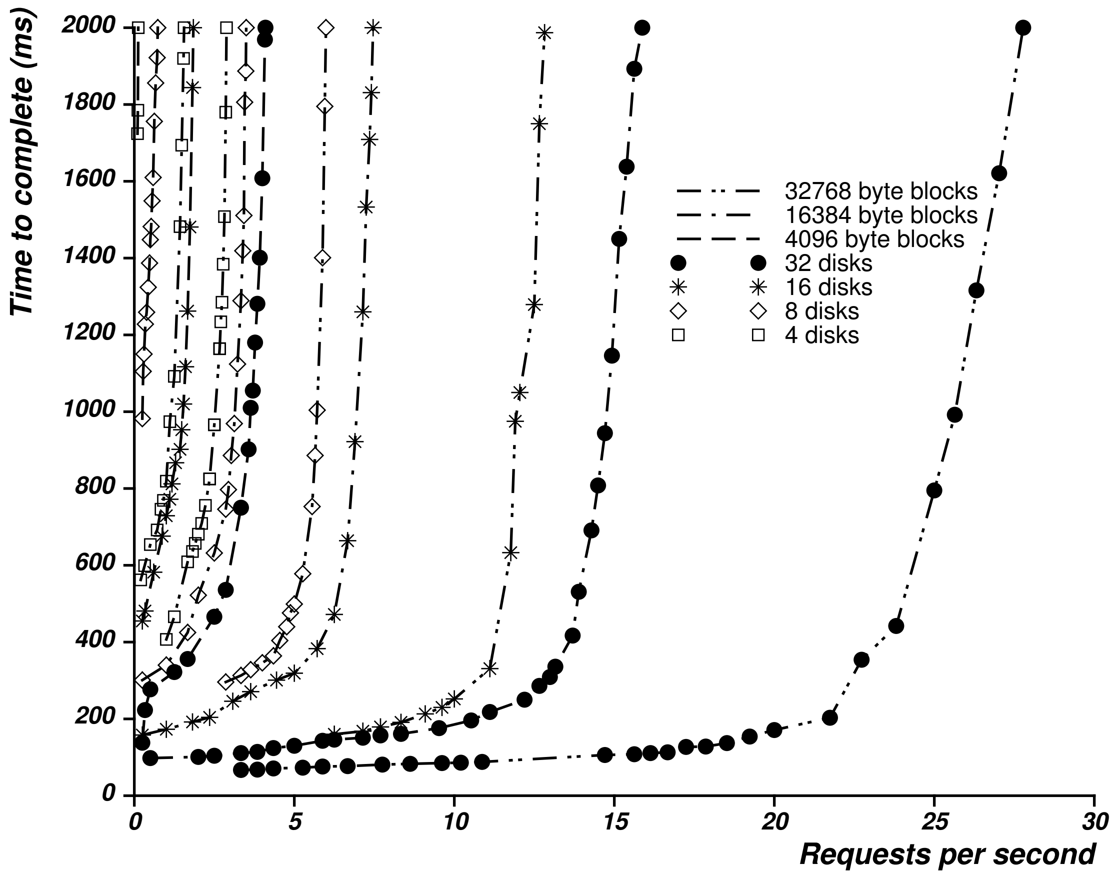

Figure 3: Average time to complete a client request.

Simulation parameters: average seek time = 16 ms, average rotational delay = 8.3 ms, transfer rate = 2.5 megabytes/second, client request = 1 megabyte, disk transfer unit = {4, 16, 32} kilobytes.

As small transfer sizes require many seeks in order to transfer the data, large transfer sizes 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.

When 4 disks were used, the system saturated quickly. This is because fewer disks had to service a larger number of requests. For larger numbers of disks, the response time was almost constant until the knee in the performance curve was reached. For 32 disks, the maximum sustainable load was reached at about 22 requests per second. At this point the disks were 50% utilized on the average. The rate of requests that are serviceable increased almost linearly in the number of disks. Increased rotational delay and a slight loading of the communication medium prevents it from being strictly linear.

The data transfer processing costs also need to be taken into account. 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

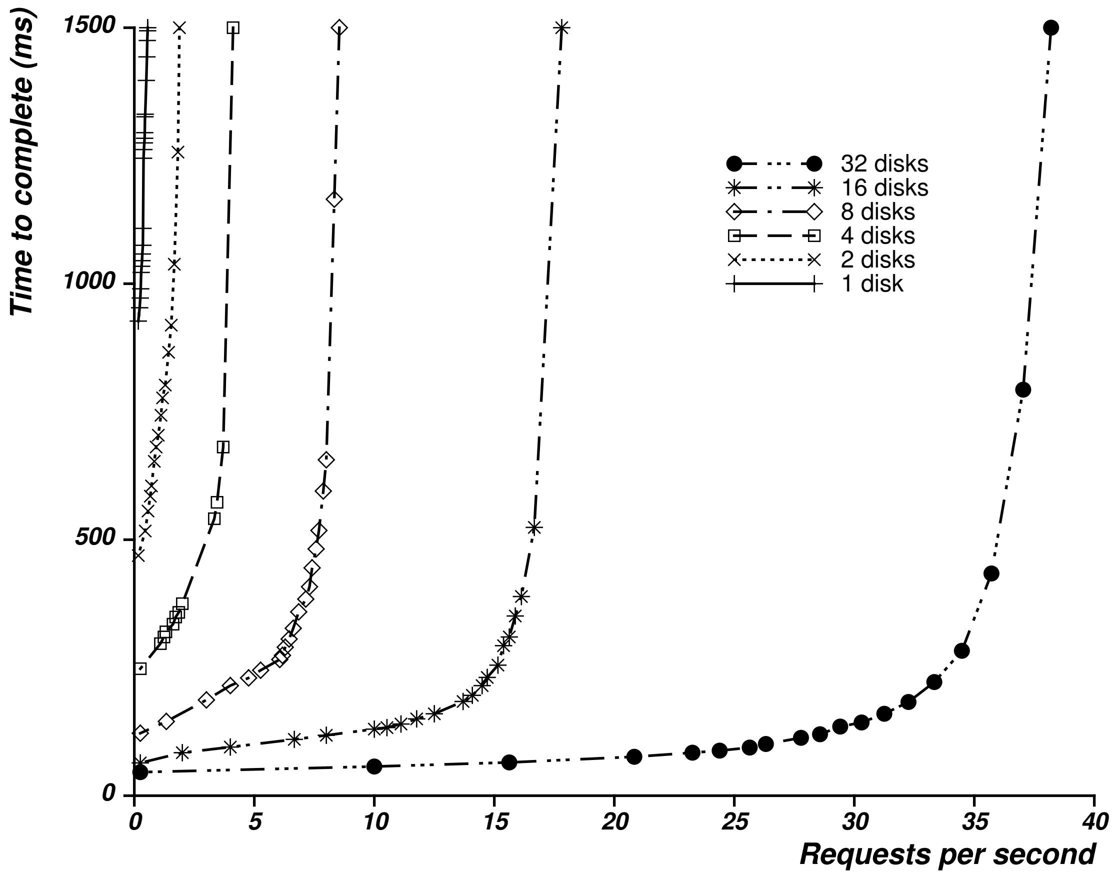

Figure 4: Average time to complete a client request.

Simulation parameters: average seek time = 16 ms, average rotational delay = 8.3 ms, transfer rate = 1.5 megabytes/second, client request = 128 kilobytes, disk transfer unit = 4 kilobytes.

unavoidable, since it is necessary data copying.

In Figure 4 we present the amount of time required to satisfy a 128 kilobyte client request using a slower storage device.

In Figures 5 and 6, the maximum sustainable data-rate is considered. The maximum sustainable data-rate is the data-rate observed by the client when the average time to complete a request is the same as the average time between requests.

The effect of seek time is seen once again. For transfer units of 4 kilobytes, the maximum sustainable data-rate for 32 disks is approximately 2 megabytes per second. When transfer units of 32 kilobytes are used, the maximum sustainable data-rate increases to nearly 12 megabytes per second for the same 32 disks. The increase in effective data-rate is almost linear in the size of the transfer unit.

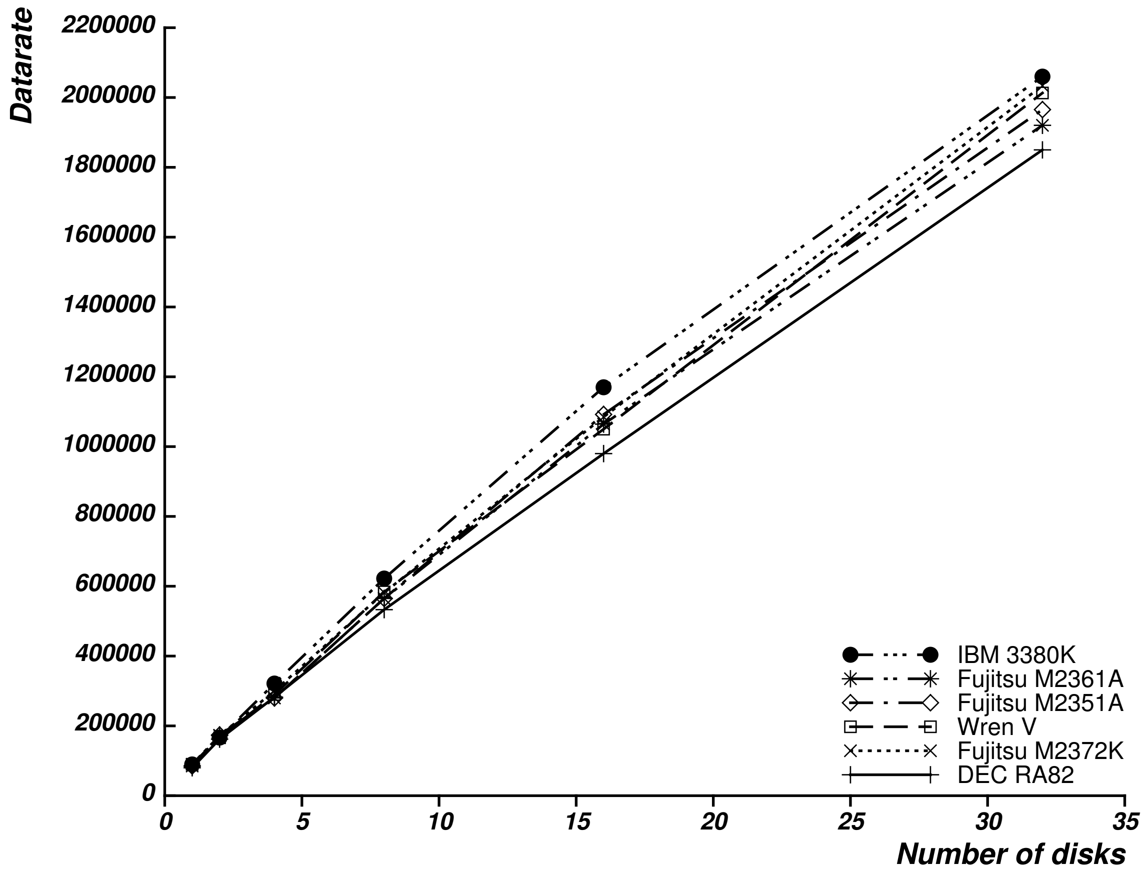

Figure 5: Observed client data-rate at maximum sustainable load.

Simulation parameters: client request = 128 kilobytes, disk transfer unit = 4 kilobytes.

6 Related Research

The notion of *disk striping* was formally introduced by Salem and Garcia-Molina [2]. The technique, however, has been in use for many years in the I/O subsystems of super computers [11] and high-performance mainframe systems [9]. Disk striping has also been used in some versions of the Unix operating system as a means of improving swapping performance [2]. 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 [11], DataVault for the CM-2 [5], the airline reservation system TPF [9], the IBM AS/400 [10], CFS from Intel [6, 7], and the Imprimis ArrayMaster [4]. Hewlett-Packard is developing a system called DataMesh that uses an array of storage processors connected by a high-speed switched network [19]. 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 [3, 8, 20] in its ability to support data-rates beyond

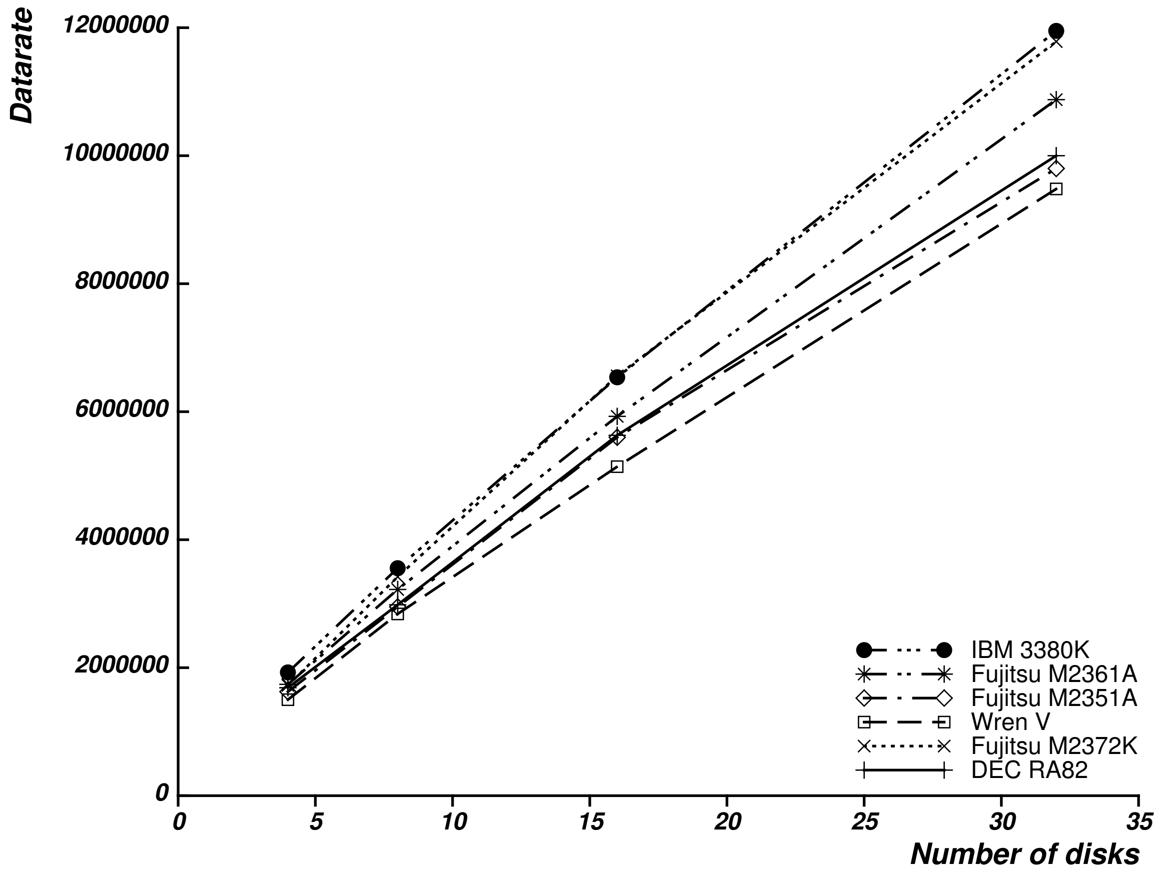

Figure 6: Observed client data-rate at maximum sustainable load.

Simulation parameters: client request = 1 megabyte, disk transfer unit = 32 kilobytes.

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 differs from 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 [21].

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.

6.1.1 Enhancements to the Prototype and the Simulator

Both the current prototype and the simulator need to address 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. The simulator needs additional parameters to incorporate the cost of computing this derived data. With these enhancements in place we plan to study the impact 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

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, and the problem of providing guaranteed communication capacity is also an area of active research, 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.

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 devices and driving them concurrently. The prototype validates the concept of distributed disk striping in a local-area network.

A prototype of Swift was built using Unix and an Ethernet-based local-area network. It demonstrated that the Swift architecture can achieve high data-rates on a local-area network by aggregating data-rates from slower data servers. The prototype achieved up to three times faster data-rates than the data-rate to access the local SCSI disk, and it achieved eight times the NFS data-rate for writes and almost twice the NFS data-rate for reads. The performance of our local-area network Swift 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 **write** almost doubled. For **read**, the improvements were only on the order of 25% because the client could not absorbe the increased network load.

Second, simulations show how Swift can exploit more powerful components in the future, and where components limiting I/O performance will be. 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 capacity is 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 much 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 using much of the existing hardware. It can be used to fully exploit the emerging high-speed networks and the large installed base of file servers.

Acknowledgements. We are grateful to those that contributed to this work including Daniel Edelson for his assistance with the simulator, Aaron Emigh for his work on the prototype and in measuring its performance, Laura Haas, Richard Golding and Mary Long for their thoughtful comments on the manuscript, Dean Long for building the custom kernels necessary for our measurements, and John Wilkes for stimulating discussions on distributed file systems. Darrell Long was supported in part by a University of California Regents Junior Faculty Fellowship.

References

[1] A. C. Luther, *Digital Video in the PC Environment.* M^{c}Graw-Hill, 1989.

[2] K. Salem and H. Garcia-Molina, "Disk striping," in *Proceeding of the* 2nd *International Conference on Data Engineering*, pp. 336–342, IEEE, Feb. 1986.

[3] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the ACM SIGMOD Conference*, (Chicago), pp. 109–116, ACM, June 1988.

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

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

[6] P. Pierce, "A concurrent file system for a highly parallel mass storage subsystem," in *Proceedings of the* 4th *Conference on Hypercubes*, (Monterey), Mar. 1989.

[7] 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* 4th *Conference on Hypercubes*, (Monterey), Mar. 1989.

[8] S. Ng, "Pitfalls in designing disk arrays," in *Proceedings of the IEEE COMPCON Conference*, (San Francisco), Feb. 1989.

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

[10] B. E. Clark and M. J. Corrigan, "Application System/400 performance characteristics," *IBM Systems Journal*, vol. 28, no. 3, pp. 407–423, 1989.

[11] O. G. Johnson, "Three-dimensional wave equation computations on vector computers," *Proceedings of the IEEE*, vol. 72, Jan. 1984.

[12] L.-F. Cabrera and D. D. E. Long, "Swift: A storage architecture for large objects," in *Proceedings of the* 11th *Symposium on Mass Storage Systems*, (Monterey, California), IEEE, Oct. 1991.

[13] L.-F. Cabrera and D. D. E. Long, "Swift: A storage architecture for large objects," Tech. Rep. IBM Almaden Research Center RJ7128, International Business Machines, Oct. 1990.

[14] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in partitioned networks," *Computing Surveys*, vol. 17, pp. 341–370, Sept. 1985.

[15] D. E. Comer, *Internetworking with TCP/IP: Principles, Protocols, and Architecture.* Prentice-Hall, 1988.

[16] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A trace-driven analysis of the UNIX 4.2 BSD file system," in *Proceedings of the* 10th *Symposium on Operating System Principles*, (Orcas Island, Washington), pp. 15–24, ACM, Dec. 1985.

[17] F. Douglis and J. Ousterhout, "Log-structured file systems," in *Proceedings of the IEEE COMPCON Conference*, (San Francisco), Feb. 1989.

[18] L.-F. Cabrera, E. Hunter, M. J. Karels, and D. A. Mosher, "User-process communication performance in networks of computers," *IEEE Transactions on Software Engineering*, vol. 14, pp. 38–53, Jan. 1988.

[19] J. Wilkes, "DataMesh — project definition document," Tech. Rep. HPL-CSP-90-1, Hewlett-Packard Laboratories, Feb. 1990.

[20] S. Ng, "Some design issues of disk arrays," in *Proceedings of the IEEE COMPCON Conference*, (San Francisco), Feb. 1989.

[21] J. Buzen and A. Shum, "I/O architecture in MVS/370 and MVS/XA," *ICMG Transactions*, vol. 54, pp. 19–26, 1986.