Using Data Striping in a Local Area Network

Luis-Felipe Cabrera
IBM Almaden Research Center
Computer Science Department

Internet: cabrera@almaden.ibm.com

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

Internet: darrell@cis.ucsc.edu

Abstract

We use the technique of storing the data of a single object across several storage servers, called data striping, to achieve high transfer data rates in a local area network. Using parallel paths to data allows a client to transfer data to and from storage at a higher rate than that supported by a single storage server. We have implemented a network data service, called Swift, that uses data striping. Swift exhibits the expected scaling property in the number of storage servers connected to a network and in the number of interconnection networks present in the system.

We have also simulated a version of Swift to explore the limits of possible future configurations. We observe that the system can evolve to support very high speed interconnection networks as well as large numbers of storage servers. Since Swift is a distributed system made up of independently replaceable components, 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. This should allow the system to incorporate and exploit emerging storage and networking technologies.

1 Introduction

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 that require this level of service include scientific visualization, image processing, and recording and playback 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 [1], and up to 90 megabytes/second for uncompressed full-frame color video.

Using data striping [2], a technique that distributes the data of an individual object over several storage servers, we have built a system, 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 in general purpose distributed systems the high data rates necessary for multimedia applications.

Swift was designed as a distributed storage system. This 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. However, much like RAID [3], Swift drives storage servers in parallel to provide high data rates. In [4] we have described the underlying I/O architecture of this system.

Swift was build using the UNIX [1] operating system. The prototype provides a UNIX-like file system interface, that includes **open**, **close**, **read**, **write** and **seek**. Swift 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. In the case of synchronous writes, the prototype on a single Ethernet segment with three servers achieves data rates that are more than double the data rate 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 servers up to the saturation of the Ethernet segment.

Our measurements show that Swift scales well when using multiple storage servers and interconnections, and can use any appropriate storage technology, including high-performance devices such as disk arrays.

We have also constructed a simulation model to demonstrate how systems like Swift can exploit advances in processor, communication and storage technology. We consider the effects of processor speed, interconnection capacity, and multiple storage servers on the utilization

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

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. 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: a description of Swift is found in §2. Measurements of the prototype are presented in §3. A simulation model is then presented in §4. In §5 we consider related work and present our conclusions in §6.

2 Description of Swift

Swift builds on the notion of distributing data across multiple storage servers and driving them in parallel. The principle behind Swift is simple: aggregate many (slow) storage devices into a faster logical storage service, making all applications unaware of this aggregation. In Swift, sets of storage servers work concurrently to satisfy the requests made by clients. Several concurrent I/O architectures, such as Imprimis ArrayMaster [4], DataVault [5], CFS [6], RADD [7] and RAID [3, 8], are based on this observation. Mainframes [9, 10] and super computers [11] have also exploited this approach.

Swift is a client/server system made up of independently replaceable components. Clients are connected to sets of storage servers through an interconnection medium. Both the storage servers and the interconnection medium can be upgraded independently. 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. When experimenting with our prototype, for example, we used two different interconnection configurations: one based on a single Ethernet and a second based on two Ethernets.

Partial failures are an important concern in Swift. If no precautions are taken, then the failure of a single component, in particular a storage server, could hinder the operation of the entire system. For example, any object which has data in a failed storage server would become unavailable, and any object that has data being written into the failed storage server

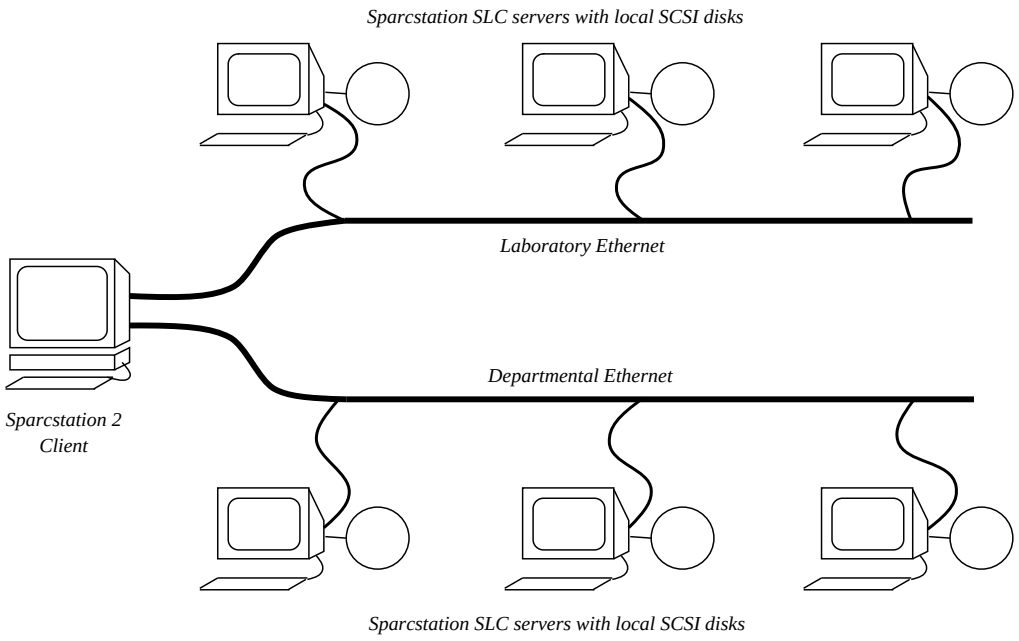

Figure 1: A two-Ethernet configuration of Swift with only one client.

could be damaged. The accepted solution for this problem is to use redundant data, including *multiple copy* [12] and *computed copy* (erasure-correcting codes) [3]. While either choice is compatible with our system, the prototype will use computed copy redundancy (in the form of parity). This simple approach provides 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 has been built as a set of libraries that use the standard filing and interprocess communication facilities of UNIX. The library assumes that all storage servers are homogeneous and interleaves data uniformly among the set of storage servers that are used to service a request. The library translates the file names of the user requests into Swift file names using the underlying file system naming.

A two-Ethernet configuration of Swift is depicted in figure 1. When an I/O request is made by the client, the Swift library communicates with each of the storage servers involved in the request so that they can simultaneously perform the I/O operation on the striped file. The client is unaware of the actual placement of the data. Since a file may be striped over any number of storage servers, 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.

Our prototype has allowed us to confirm that a high aggregate data rate can be achieved. The data transfer protocol of Swift has been built using a light-weight data transfer protocol on top of the UDP [13] datagram protocol. To avoid as much unnecessary data copying as possible, scatter-gather I/O was used to have the kernel deposit the data coming over the network directly into the buffer in the client address space.

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 servers. All storage servers 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 servers use dedicated UDP ports to transfer data and have a dedicated server process to handle the user requests.

2.1 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 server at its advertised UDP port address.

Each Swift storage server waits for **open** requests on a well-known UDP [13] 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 client **write** request, the Swift library sends out the data to be written one block at a time and receives an explicit acknowledgment for each. The client's **write** request returns only when all the corresponding acknowledgements have been received. 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 servers release the ports and extinguish the threads dedicated to handling requests on that file. For read operations we did not need explicit acknowledgements for the data as the client could always stay ahead of the servers.

3 Measurements of Swift

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 one kilobyte is used to denote 1000 bytes.

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.

The experiments with the local SCSI disk used 8 kilobyte blocks. They measure the rates the system can access the actual device. 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 424 ± 15 kilobytes/second for a sixteen megabyte file. The measured values decrease slightly, from a high of 452 ± 49 kilobytes/second for a one megabyte file, as the amount of data to be read increases. The synchronous write rate obtained for a sixteen megabyte file is 67 ± 0.4 kilobytes/second, and decreases some for longer files. 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 writes 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.

The data rates for asynchronous writes are much higher that those for synchronous write. The measurements reveal that the rate at which the local file system of a Sun 4/20 will asynchronously write data is 416 ± 2 kilobytes/second for a sixteen megabyte file, and 559 ± 3.5 kilobytes/second for a one megabyte file. This significant drop in performance is due to the fact that for larger files, the file system buffer space is exhausted, thus the file system needs to write data to disk and to pause for disk I/O completions.

The performance of the Swift prototype is presented in figure 2. When used with one storage server, 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 a 16 megabyte file at a rate of 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.

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 limited 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

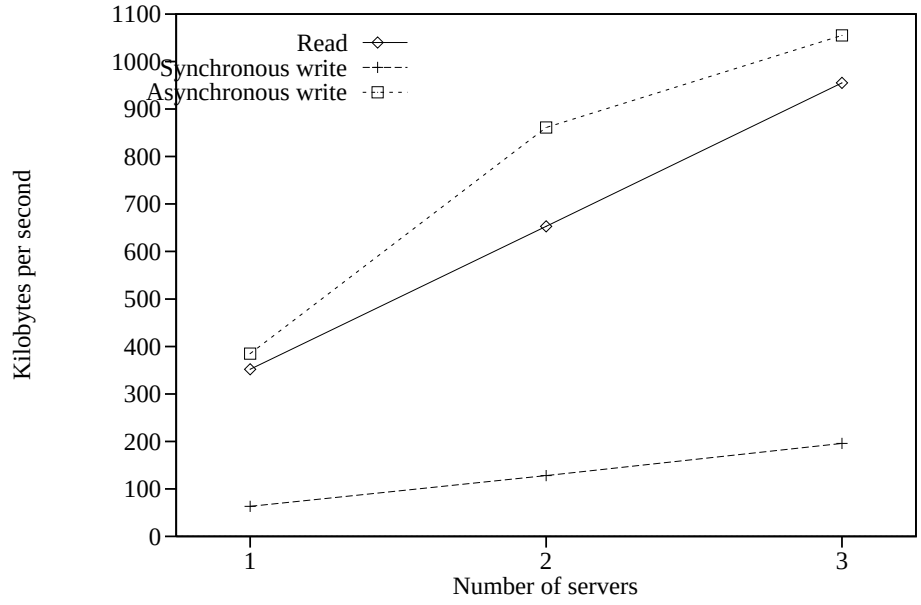

Figure 2: Data rate of the Swift prototype on one Ethernets.

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.

For three servers, the data rate is approximately three times that of a single server in the cases of reads and synchronous writes. For asynchronous writes, however, the increase in data rate is muted by the Ethernet as it is becoming increasingly saturated. Even so, the performance is better than that of reads since transmitting data over the Ethernet requires fewer process context-switches, less data copying, and no interrupt signal at completion.

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.

In the case of reads, the high performance NFS file server provides a data rate of 612±36 kilobytes/second for a sixteen megabyte file which is 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 faster than the local SCSI disk used by Swift. When synchronous writes are considered, the NFS file server provides a data rate of 92.7±0.55 kilobytes/second for a sixteen megabyte file which is 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 NFS file server provides a data rate of 107±2 kilobytes/second for a sixteen megabyte file. In this case the Swift server 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 Swift 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.

3.1 Effect of Adding a Second Ethernet

To determine the effect of doubling the data rate capacity of the interconnection, we added a second Ethernet segment to the client and added additional storage servers. In this case the storage servers 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 (the standard bus on Sun SPARCstations). 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 large 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 the Swift prototype with two storage servers on two separate Ethernet segments are presented in figure 3. When compared with the results for two storage servers on a single Ethernet segment, it is apparent that in the case of reads, having storage servers 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.

When four storage servers are used, placed two on each Ethernet, it is apparent that the data rate for reads does not scale linearly since it is comparable with that of three storage servers 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 servers as it did with two. Asynchronous writes have ceased to scale linearly since the client is approaching its capacity to transmit data over the two Ethernet interfaces.

When six storage servers are used, placed three on each Ethernet, the data rate for read improves, although not in proportion to the number of storage servers. 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 servers as the client is saturated.

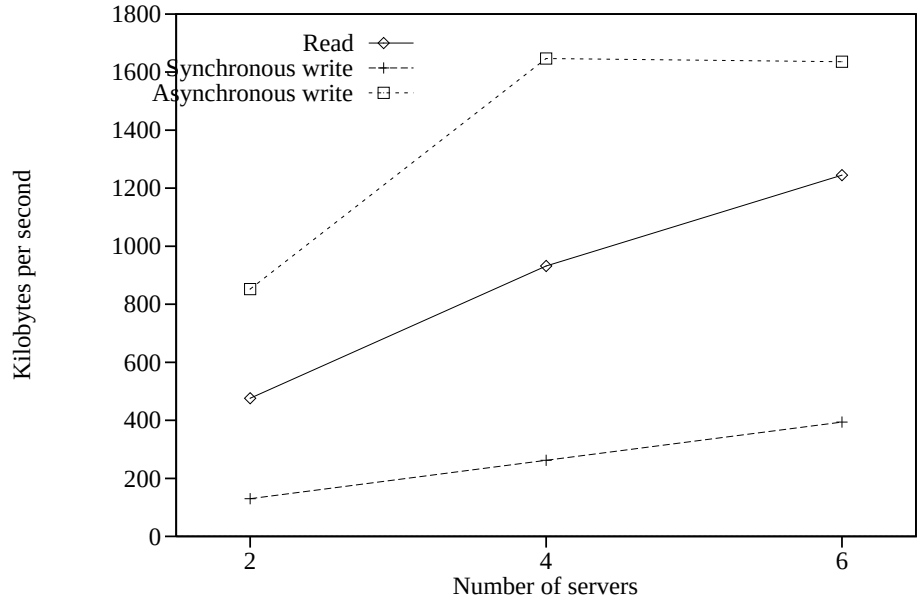

Figure 3: Data rate of the Swift prototype on two Ethernets.

4 A Simulation-based Performance Study

We modeled a hypothetical high-speed local-area token-ring implementation of Swift. Our primary goal of the simulation was to show how a system like Swift 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 servers 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.

4.1 Structure of the Simulator

The system is modeled by client requests that drive storage server 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 [4]. There is no modeling of overlapping execution, instead requests are modeled serially: only after a request has completed is the next issued.

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

The simulator models parity computations. When write requests are made, the simulator calculates the cost of producing the parity blocks for the data being written. 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.

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 [4]. Once a block has been read from disk it is scheduled for transmission over the network.

4.2 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 servers and the size of the transfer unit. Using the simulator we could observe how system bottlenecks moved from the storage servers to the processor power

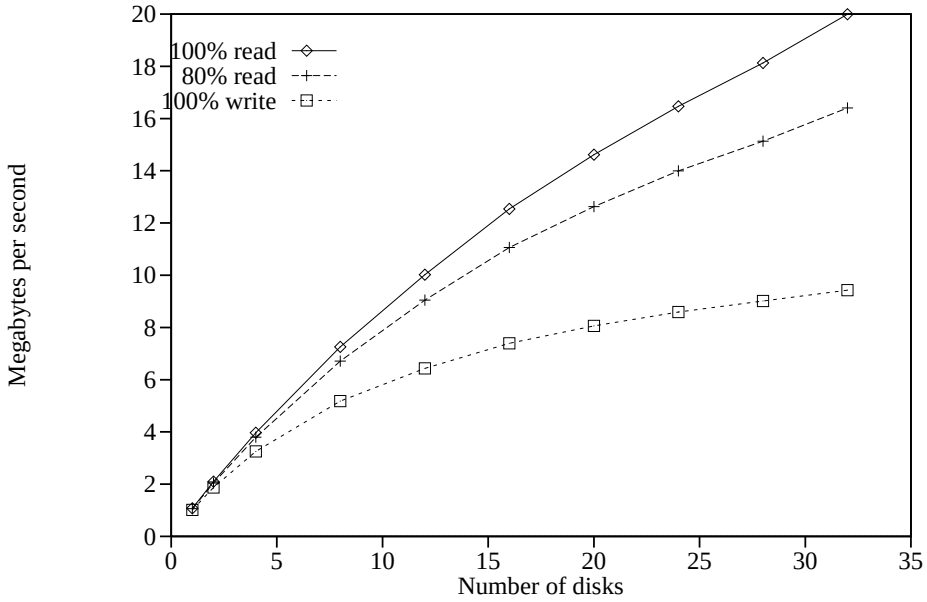

Figure 4: Data rate with 16 kilobyte blocks varying read-to-write ratio.

of the client depending on the specific configuration.

When simulating we chose disk parameters typical of those that are commonly used in current file servers. The disk were assumed to spin at 3600 revolutions/minute, yielding an average rotational latency of 8.3 milliseconds. The rotational latency was modeled by a uniform distribution. The average seek time was 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 read-to-write ratio is considered in figure 4. 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 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 in the parity set has been modified, and then to write both that block and the parity block.

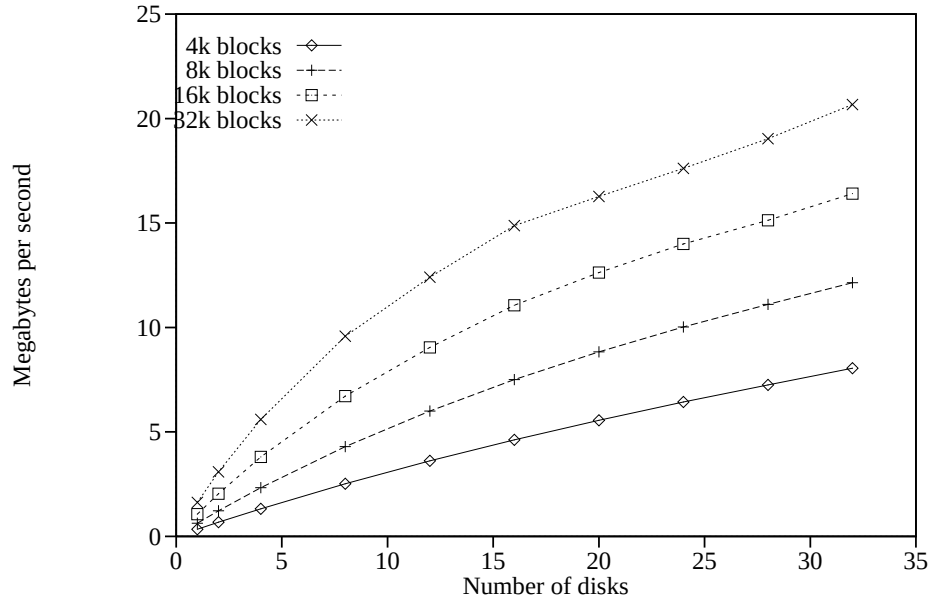

Figure 5: Effect of block size on throughput.

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 servers and to the size of the transfer unit. Even though the cost of computing parity is non-negligible, the processor becomes a bottleneck only for write requests directed to large numbers of storage servers. When the 200 million instructions/second processor is used the utilization at the storage servers increases some 50% at high loads, as the processor in the client is less of a bottleneck.

In figure 5 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.

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

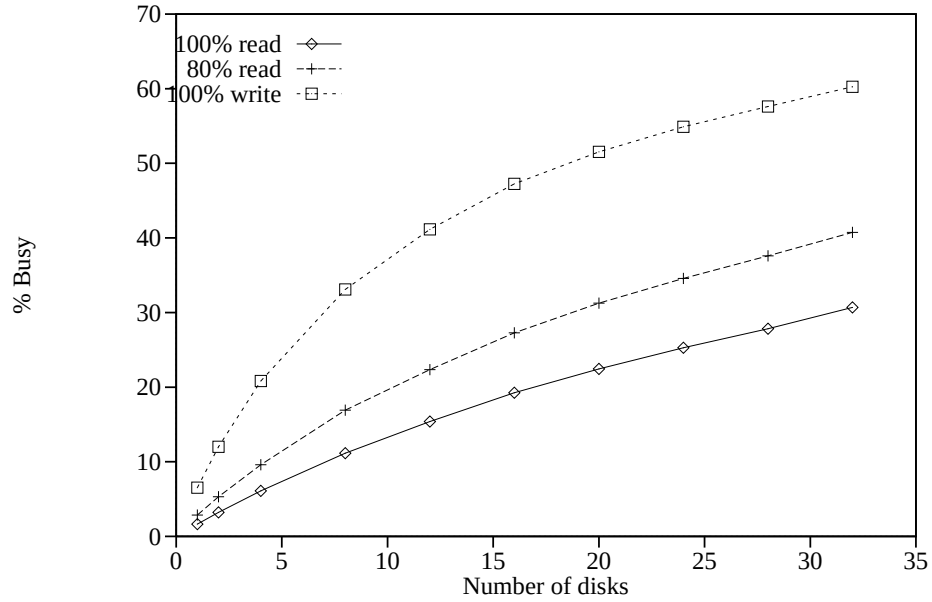

Figure 6: Fraction of processor used with 16 kilobyte blocks.

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 [14]. 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.

In figure 6 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

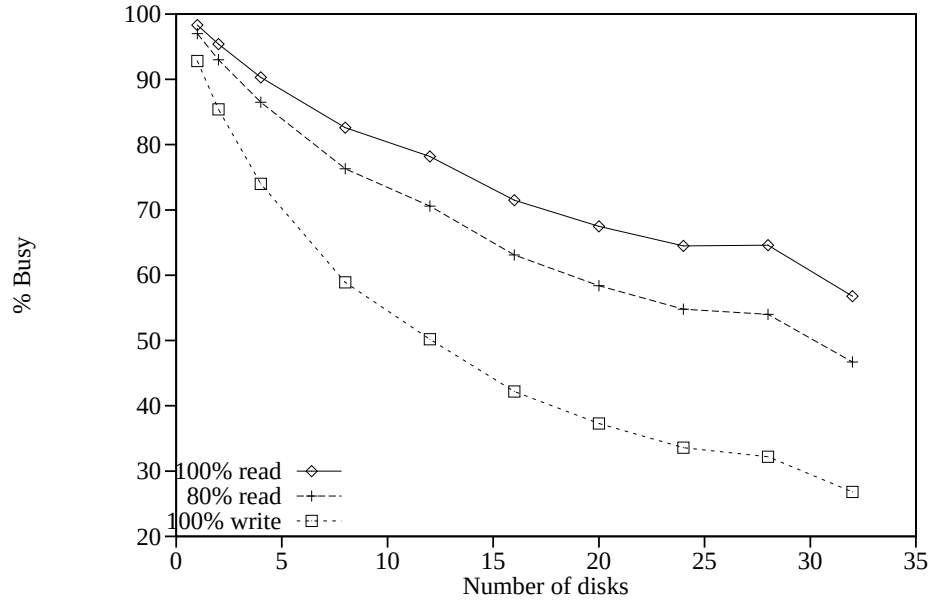

Figure 7: Fraction of disk used with 16 kilobyte blocks.

writes the processor can be seen to be a significant performance limiting factor. This is due to our assumed cost of computing the parity code (exclusive-or).

In figure 7 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 6 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 8. 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).

4.2.1 The Effect of Doubling the Processor Speed

We modeled the effect of doubling the processor speed, to evaluate its effect on the processor bottleneck. Comparing figures 9 and 6 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%

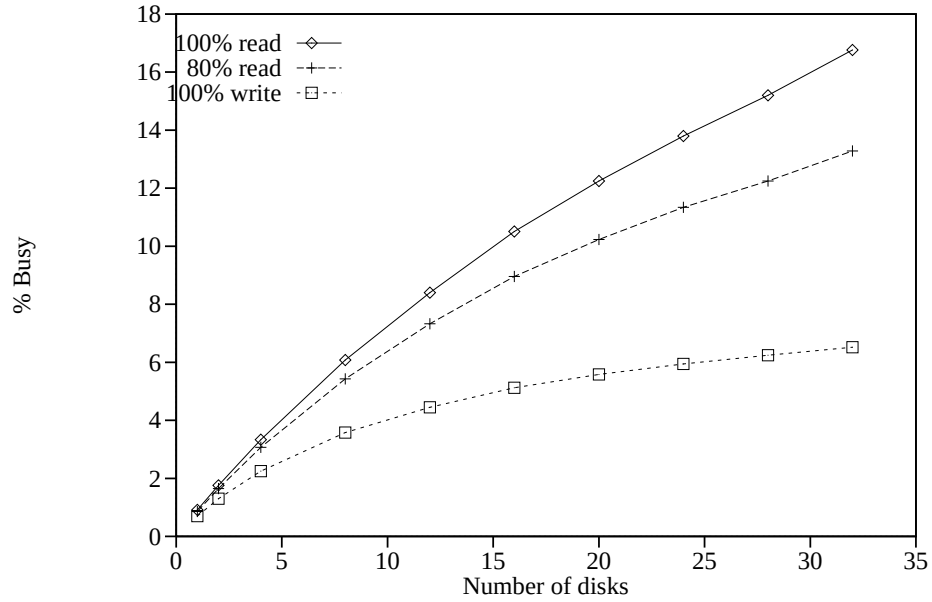

Figure 8: Fraction of network capacity used with 16 kilobyte blocks.

to 43% for a pure write load.

In figure 10 we see that the simulation shows 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 4. For pure writes the system can achieve data rates of 13 megabytes/second for 32 disks, versus the 9 megabytes/second depicted in figure 4. 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 7 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 has shifted to the serial nature of the data transmission protocol. 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.

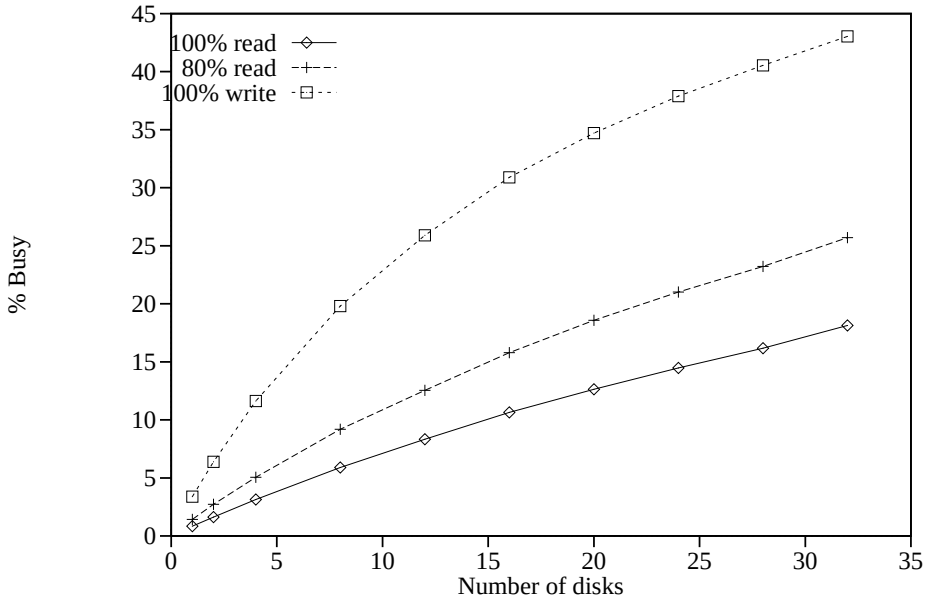

Figure 9: 200 MIPS: Fraction of processor used with 16 kilobyte blocks.

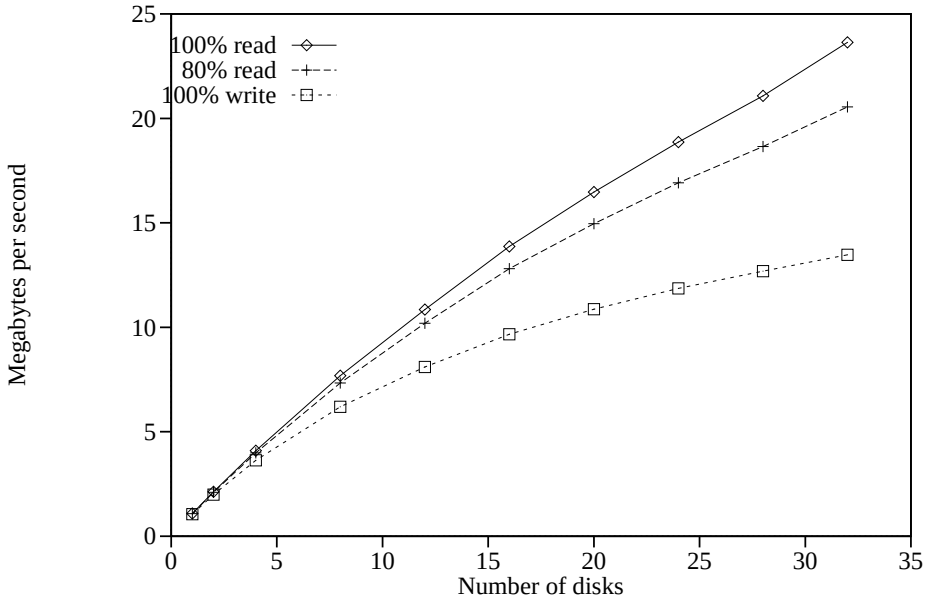

Figure 10: Data rate with 16 kilobyte blocks varying read-to-write ratio.

5 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 in a local-area network.

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], 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 [4]. 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 used in Swift generalizes that proposed by the RAID disk array system [3, 8] 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 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 servers 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. Clients may access several independent

sets of storage servers.

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 [15].

6 Conclusions

This paper presents Swift, a scalable distributed I/O system that achieves high data rates by striping data across several storage servers and driving them concurrently. The system validates the concept of distributed disk striping in a local-area network. Swift presents a flexible, powerful way of doing I/O in a local-area network by using aggregation of servers to satisfy the requests of a client. With Swift sets of available storage servers can be viewed as one larger, more powerful, storage server.

Swift was built using UNIX and an Ethernet-based local-area network. Swift demonstrates that one 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 twice the data rates than were provided by access to the local SCSI disk, and it achieved ten times the NFS data rate for asynchronous writes, twice the NFS data rate for synchronous writes, and almost twice the NFS data rate for reads. The performance of Swift 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 servers, the data rates measured demonstrated that Swift 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 a system like Swift show how it 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 servers when sufficient interconnection and processor capacity are available.

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. Swift 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.

Acknowledgements. 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] A. C. Luther, *Digital Video in the PC Environment*. McGraw-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] L.-F. Cabrera and D. D. E. Long, "Swift: Using distributed disk striping to provide high i/o data rates," *Computing Systems*, vol. 4, pp. 407–438, Dec. 1991.

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

[6] 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.

[7] M. Stonebraker and G. A. Schloss, "Distributed RAID – a new multiple copy algorithm," in *Proceedings of the* 6th *International Conference on Data Engineering*, (Los Angeles), pp. 430–437, IEEE Computer Society, Feb. 1990.

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

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

[14] 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.

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