# An Analysis of iSCSI for Use in Distributed File System Design

**Mike Brim and George Kola**

**Abstract**

*We evaluate the performance of iSCSI for use in design of a distributed file system. We perform a detailed full system characterization of a client using iSCSI. Using the analysis ,we identify the overheads associated with iSCSI and compare and contrast it with local disk access and NFS file access. Understanding the overheads associated with iSCSI, we attempt to tune the system to optimize iSCSI performance and show the impact of various parameters. Using insights gleaned from the analysis, we design a distributed file system optimized for use with iSCSI. We point out the class of applications which would benefit from such a distributed file system. We also report the status of our implementation.*

## 1. Introduction

Internet SCSI, or iSCSI, is an IETF draft standard [J03] for a protocol to transmit Small Computer Systems Interface (SCSI) commands over existing TCP/IP-based networks. The iSCSI protocol has two halves to it - the ***initiator*** resides on a client computer, and sends commands to the iSCSI ***target***. The target performs the work requested by the initiator, and sends a reply back. All communications take place via TCP/IP. There has been much interest lately in the role of iSCSI [VS01][SV02], as it provides lower overall cost and higher scalability than current Storage Area Networks that rely on Fiber Channel technology and the characteristics of TCP/IP networks are better understood.

In this project, we look at various aspects of iSCSI that are important in the design of a distributed file system. In our distributed file system, we allow clients to directly access the iSCSI device. This is different from normal usage of iSCSI where only a few servers access the iSCSI device and there is no sharing between them. We also discuss the implications of such a design and also point out the advantages of such a design.

We present an in-depth analysis of iSCSI protocol and the overhead associated with it. We also do a full system characterization of an iSCSI initiator. Such a full system characterization would be useful for network card manufacturers, kernel developers and system builders. We also compare and contrast iSCSI access with that of local disk access and remote NFS file access. This comparison would be useful for storage planners who are thinking of using iSCSI. During the analysis of iSCSI, we provide specific insights into key performance characteristics that would affect the performance of a distributed file system built on top of iSCSI. Using these insights, we describe the design and current implementation status of a simple, application-specific distributed file system optimized for use with iSCSI.

The rest of the paper is organized as follows. In Section 2 we provide an overview of work related to our own. Section 3 describes our experimental environment. Section 4 details our performance analysis of iSCSI, including comparisons to local disk and NFS performance. Section 5 provides a description of the distributed file system we designed using insights gained from our analysis of iSCSI, as well as current implementation status. In Section 6, we provide concluding remarks and Section 7 relates our directions for future work.

## 2. Related Work

Our evaluation of the performance implications of using iSCSI in the context of a distributed file system touches on many areas presented in current literature. These areas include iSCSI performance evaluation, various network-based storage architectures, and distributed file system design.

In the area of iSCSI performance evaluation, our work is most similar to the evaluations presented in [AGPW03] and [TMI]. In [AGPW03][1], iSCSI is evaluated as a competitor to Fiber Channel for use in Storage Area Networks (SANs) in four different configurations: two using a commercial grade SAN employing Fiber Channel and real iSCSI target hardware, and two using iSCSI implemented completely in software. Their work looks at network and iSCSI throughput and did not look at the impact on the client operating system and the load on the CPU. We feel that understanding where time is being spent in processing iSCSI at the client side (iSCSI initiator) is very important and would be useful for the research community trying to optimize iSCSI or developing new protocols and we address that. We agree to their statement that using jumbo packets improves performance. We go a step further and find out the impact of jumbo packet on the sender and receiver and explain the reason for the performance gain. The work presented in [TMI] provides a comparison with NFS for various storage access patterns. Their analysis shows that iSCSI outperforms NFS for access patterns including data streaming, transactions, and other small-file accesses. They make a broad statement that iSCSI is useful only for applications that do not require file sharing. We attempt to address that limitation by building a distributed file system on top of iSCSI.

With respect to network-based storage architectures, the goals of our research are most closely related to both Storage Area Networks (SANs) and Network Attached Storage (NAS) architectures. In SANs, networked-storage devices provide a block-level interface to hosts on a Fiber Channel network. The networked-storage device used in our analysis is conventionally targeted towards a SAN environment, and provides a SCSI interface as required for use with iSCSI. NAS environments differ in that the networked-storage devices provide a filesystem abstraction to clients attached to a standard network such as Ethernet. Network-Attached Secure Disks [G97], or NASD, is a research NAS architecture where the networked-storage devices have special processing abilities in order to provide access authentication on file objects. Clients in a NASD environment must obtain capabilities and file data object location information for specific files from a filesystem meta-data manager host before directly requesting the objects from the storage devices. The storage devices then assert that the capabilities provided are valid before granting any file data object requests. However, such NASD devices are not currently available and show no signs of appearing in the near future. We do not assume an object-based interface for our storage devices, instead focusing on the devices widely-available for SAN environments.

Distributed file system design is a broad area of research due to the various operational environment targets and performance requirements, thus we focus here on work that is specifically related to ours in some aspect. Data Reservoir [KIT02] is an environment which uses iSCSI for large-file transfers for sharing between scientific applications in a wide-area Gigabit network. Although we do focus on large-file sharing, our analysis is strictly confined to a local-area investigation. Slice [ACV00] is a distributed file system for use with network-attached

[1] We would like to mention that this work was released only a few days ago and was not available before our work started.

storage that provides an object-based interface similar to that used in the NASD work. Slice uses a proxy embedded in the network stack of clients to transform and encode standard NFS requests into object requests for the network-attached storage devices. We believe that a distributed file system using iSCSI should be tailored to its specific behavior, and thus do not attempt to virtualize NFS.

## 3. Experimental Environment

The research presented herein was performed in its entirety in a live environment, the Wisconsin Advanced Internet Laboratory (WAIL) at the University of Wisconsin-Madison. WAIL is a self-contained networking laboratory that enables recreation of all portions of the Internet, including end hosts, last-hop hardware, enterprise and backbone routers, enterprise-class networked-storage, and various other hardware too numerous to mention. Given that the focus of our research was performance of iSCSI in the context of a local-area network, we obviously did not make full use of the breadth of equipment available. Figure 1 shows our experimental hardware setup. As shown in the figure, the environment includes multiple end hosts to serve as clients and a server, two Ethernet switches, an iSCSI gateway device, and an enterprise-class networked-storage device. Each end host runs Linux kernel version 2.4.20 and the iSCSI driver version 2.1.2.9 from the linux-iscsi SourceForge project. Each host also has the same hardware configuration: a 2.0 GHz Intel Pentium4 processor, 1 GB RAM, an Intel Pro/1000T Gigabit Ethernet adapter, a 3Com 3C905-TX-M Ethernet adapter, and a 40 GB IDE hard disk. The two Ethernet switches are a 100BaseT Cisco Catalyst 2950 used for standard communication between the end hosts and a 1000BaseT Cisco Catalyst 6500 for communication with the iSCSI gateway. The iSCSI gateway is a Cisco SN5428 Storage Router, and is connected using Fiber Channel-Arbitrated Loop (FC-AL) to the EMC Symmetrix 3830 networked-storage device.

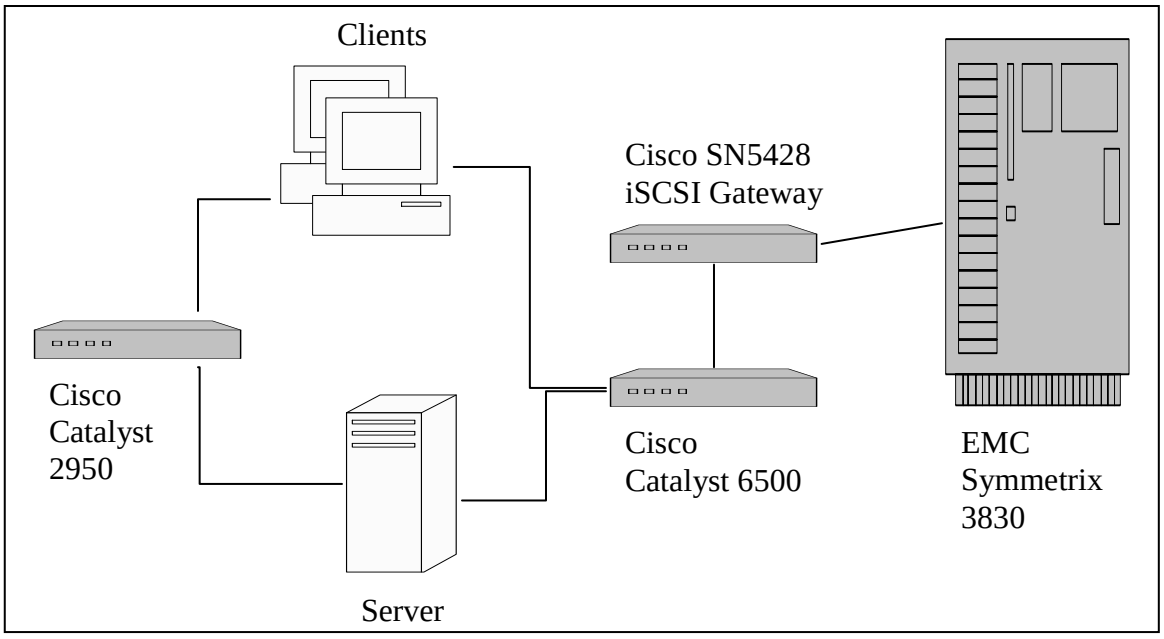



**Figure 1 – Experimental Environment**

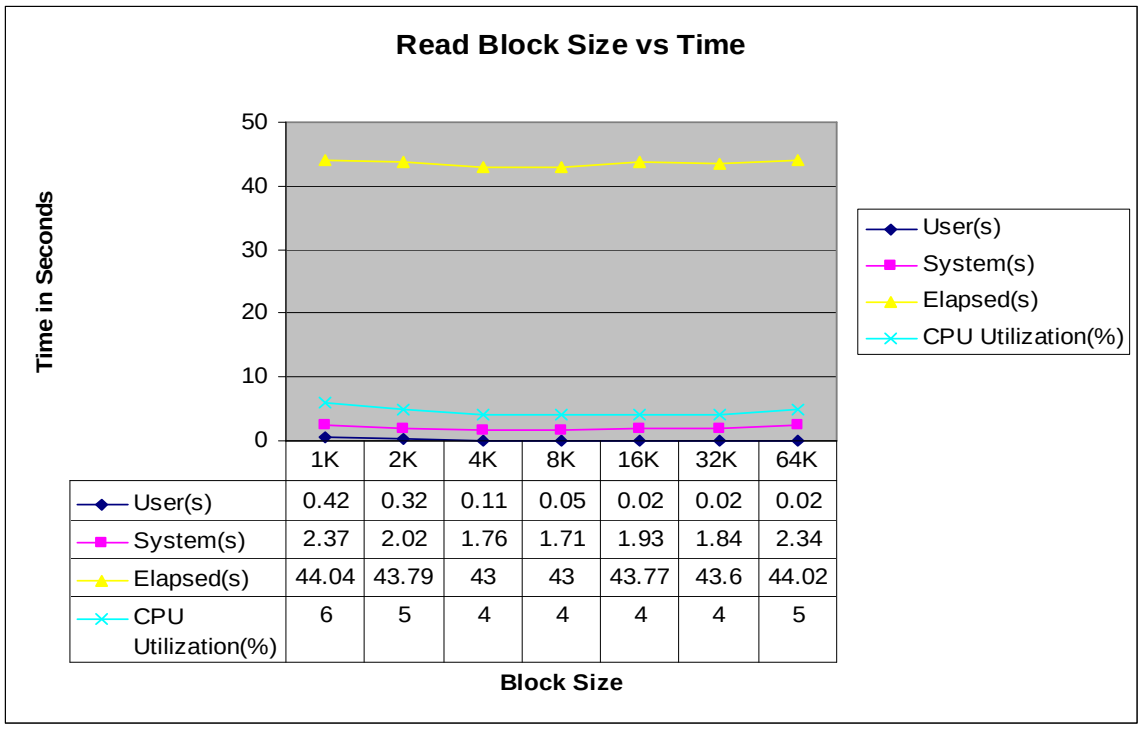
## 4. iSCSI Performance Evaluation

### 4.1 Profiling iSCSI

In order to get a good understanding of iSCSI as a protocol, we wanted to be able to answer the following questions:

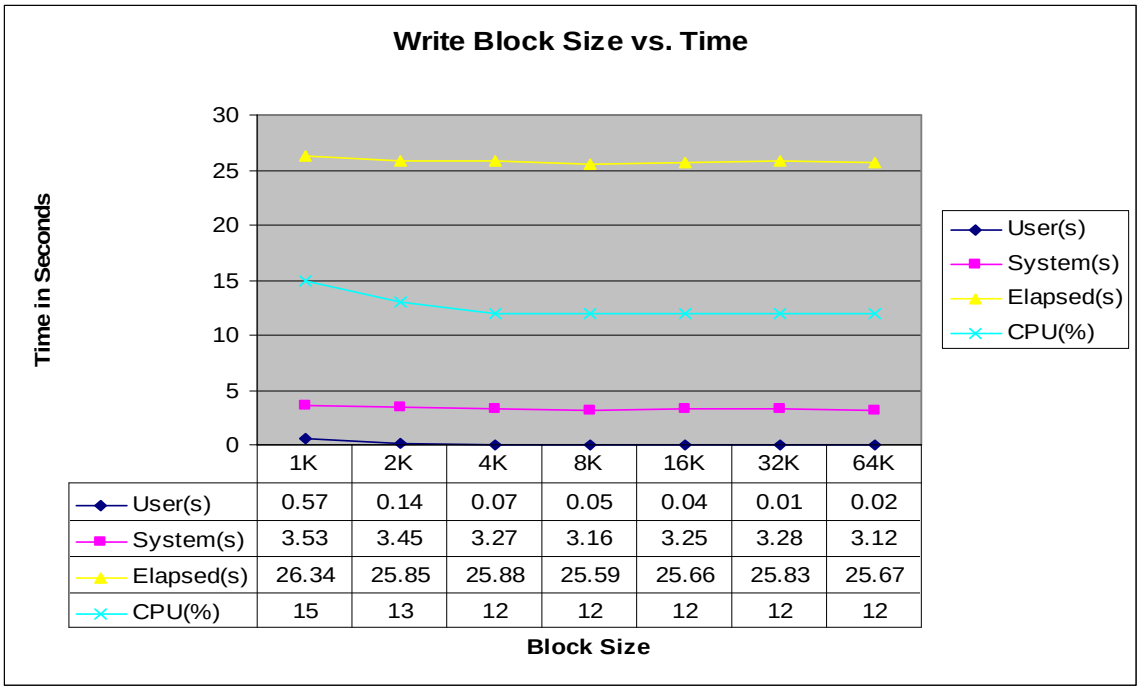
1) What role does block size play on performance?
2) What are the overheads associated with iSCSI protocol?
3) How much of the overhead is because of the Gigabit Ethernet interconnect?
4) What parts of the kernel get stressed out when using iSCSI?
5) How do we tune a system employing iSCSI?
6) What effects o the different tuning parameters have?

To find the effect of block size on performance, we found the time taken to read and write a 1 GB file. The choice of file size was motivated by the idea that a distributed file system built on top of iSCSI would be suitable candidate for the Condor Checkpoint server, and typical checkpoint sizes for large applications are close to 1 GB. Similar file sizes are also seen in multimedia encoding applications. The results for various block sizes are shown in Figures 2 and 3.




| | 1K | 2K | 4K | 8K | 16K | 32K | 64K |
|---|---|---|---|---|---|---|---|
| User(s) | 0.42 | 0.32 | 0.11 | 0.05 | 0.02 | 0.02 | 0.02 |
| System(s) | 2.37 | 2.02 | 1.76 | 1.71 | 1.93 | 1.84 | 2.34 |
| Elapsed(s) | 44.04 | 43.79 | 43 | 43 | 43.77 | 43.6 | 44.02 |
| CPU Utilization(%) | 6 | 5 | 4 | 4 | 4 | 4 | 5 |

**Figure 2: Effect on Block Size on iSCSI Read Performance (1GB Read)**




| | 1K | 2K | 4K | 8K | 16K | 32K | 64K |
|---|---|---|---|---|---|---|---|
| User(s) | 0.57 | 0.14 | 0.07 | 0.05 | 0.04 | 0.01 | 0.02 |
| System(s) | 3.53 | 3.45 | 3.27 | 3.16 | 3.25 | 3.28 | 3.12 |
| Elapsed(s) | 26.34 | 25.85 | 25.88 | 25.59 | 25.66 | 25.83 | 25.67 |
| CPU(%) | 15 | 13 | 12 | 12 | 12 | 12 | 12 |

**Figure 3: Effect of Block Size on iSCSI Write Performance (1GB Written)**

In both these experiments, the MTU was left at the default of 1500B. As shown in the figures, the optimal block size for both read and write is 8KB. The CPU utilization shown here shows the general trend. The actual CPU utilized was actually higher in each case. The reason for this is that we used the Unix time command to get the CPU utilization and the Linux kernel does not account the time spent in iSCSI processing and TCP and network interrupt handling to the process that is making the iSCSI request.

We wanted to get a full system level profile including the time spent in the different functions in the kernel code. However, we did not want to perturb the measurement. Direct tracing of the kernel would slow down the whole system and the numbers obtained would not be realistic. To avoid such perturbation, the profiler we chose to use is *Oprofile*, which is based on the *Digital Continuous Profiling Infrastructure (DCPI)* [A97]. The difference between the two is that *Oprofile* is for the x86 architecture and uses the hardware counters available on the Pentium family of processors, while DCPI uses the hardware counters available on the Alpha architecture.

Before describing the results obtained, we first provide a brief overview on *Oprofile's* operation. On the Pentium4, if we set the event to 'Global Power Events' and set the mask to 'CPU cycles unhalted' then the counter value gets incremented every active clock cycle. When the counter overflows (max counter value is set by us), the counter overflow interrupt occurs. *Oprofile* in the kernel handles the interrupt and records the value of the Instruction Pointer when the interrupt occurred. The Instruction Pointer gives the function which was executing at that point. *Oprofile* counts the number of times the interrupt occurs in each function. Over long time periods this has the effect of finding the relative percentage of time spent in various functions (both kernel and user). Since we did not want the kernel to halt the CPU when idle (we wanted to find the percentage of time the CPU was idle), we passed the idle=poll option to the Linux kernel to use the idle loop instead of halting the CPU. We used a counter value of 996,500 for the results presented here. On the 2 GHz Pentium4 that we used, this has the effect of the counter overflow interrupt occurring every 498 µsec. A lower counter value increases accuracy and overhead. We

tried the experiment with different counter values between 400,000 and 996,500, but the results obtained were the same. Thus, we used the higher counter value to reduce the overhead. The accuracy of the profiling increases if the experiment is performed for long durations. We ran each of the experiments for at least four hours and for different time durations, trying to see the correlation between the readings. During the experiments the machine was idle except for the standard Linux daemons. The results presented here were all verified by multiple-run correlation. The occasional non-conforming reading was found to correspond to situations in which we mistakenly kept using the machine during the experiment, and thus recorded the time spent doing other things.

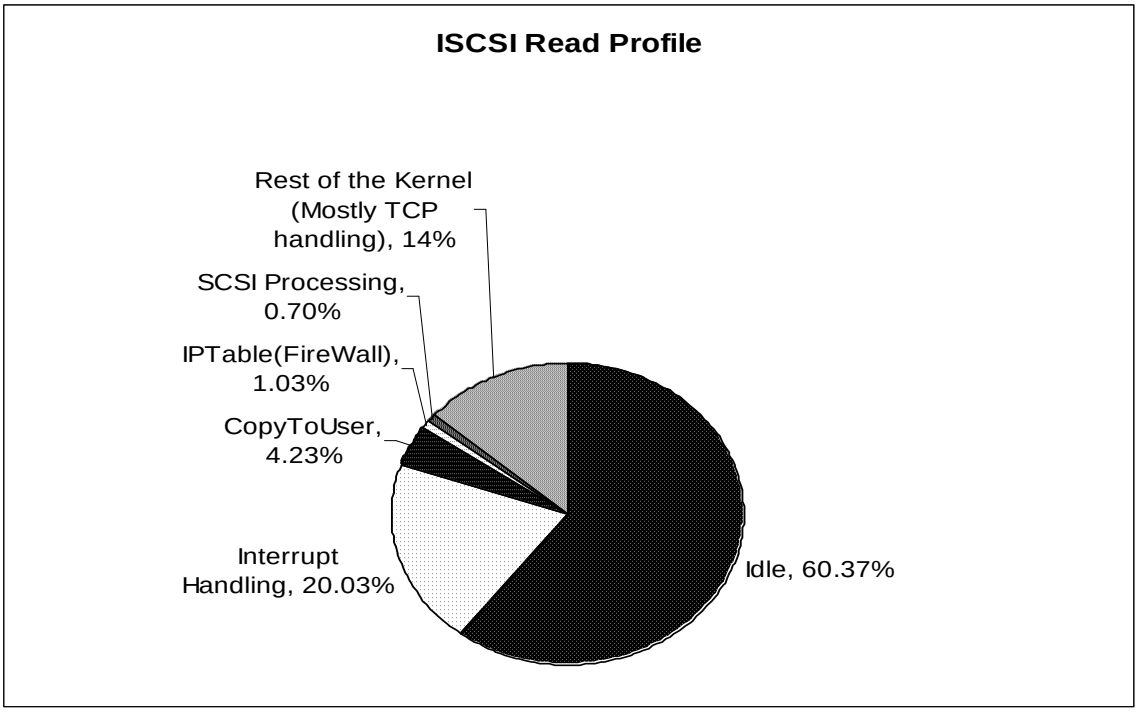



**Figure 4: CPU Utilization for 1 GB Read from an iSCSI Target**

Time taken to read 1GB of data was 43.5 seconds, giving a read throughput of 23.54 MBPS. We find that nearly 40% of the CPU time is used by a single process just reading from the iSCSI device. In addition, 20% of the time is spent handling the interrupts from the Gigabit Ethernet adapter. The adapter has hardware checksum enabled, so checksum verification does not consume any CPU. The SCSI processing was minimal at 0.70%. We also have about 1% of time being used by IPTables, the Linux kernel firewall module. The firewall software could have been turned off, but we find that most servers, even those behind a firewall, leave it on as it allows easy blocking of external-access to certain services. From the results, it appears that the key to improving the performance would lie in optimizing the interrupt handling. In contrast, we performed the same experiment on a local IDE disk. We found that the time taken was 43.6 seconds, yielding a read throughput of 23.48 MBPS. However, the CPU utilization was much lower at only 5.3%. IDE interrupt handling consumes only 3.5% of the CPU. The *file_read* function in the kernel takes 1.4% of the time. The copy to user also consumed less CPU since in the case of local disk, the buffers are aligned.

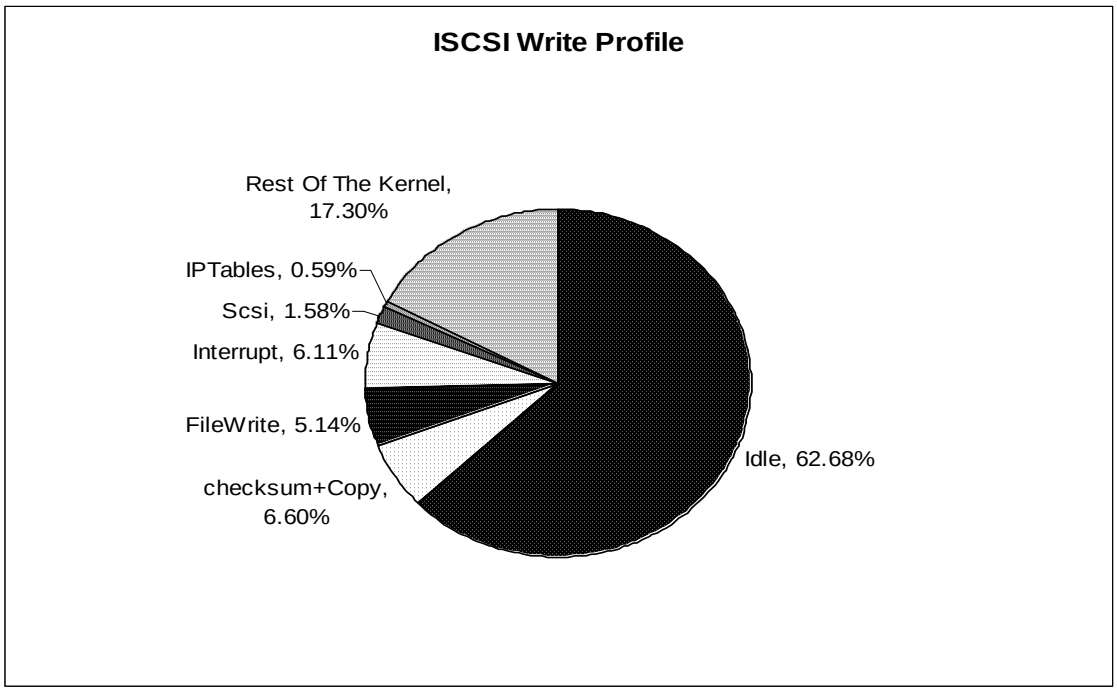



**Figure 5: CPU Utilization for 1 GB Write to an iSCSI Target**

The time taken to write 1 GB of data was 25.6 seconds, giving a write throughput of 39.6 MBPS. We found that write performance was much better than read performance. The main reason for this as seen in the figures is the lower interrupt handling overhead (6.11%). There seems to be a contradiction as write seems to be transferring more data than read but still incurring a lower interrupt cost. The reason for this is that multiple packets are transferred in a single DMA transfer. In the case of read, the Ethernet card tries to coalesce multiple packets and deliver a single interrupt but it is not effective in doing so. Contrasting the write performance with that of the local disk, we found that it took 54.6 seconds to write 1 GB of data giving a transfer rate of 18.75 MBPS. The CPU utilization was at 7.5%. IDE DMA was still taking less than 3.5%, and the kernel *file_write* function was taking 2.8% of the CPU time. In local disks, write is slower than read. We find the reverse to be true using the iSCSI devise. The reason for this is that the EMC box has 8GB of RAM, so all writes just go to memory and the Symmetrix guarantees the data will be written to the disk at some point.

## 4.2 Comparison with NFS

We wanted to compare the performance of iSCSI to that of NFS. It is not a fair comparison since NFS implements a whole file system whereas iSCSI just gives a block level abstraction. A file system needs to be built on top of iSCSI in order to provide a more appropriate comparison. In this study we performed a 1 GB sequential read and write, with the results shown in Figures 6 and 7 respectively. Our motivation in performing this comparison was to find out how NFS handles such a workload, and how should the distributed file system be designed if it is to handle such a workload better than NFS. In the experiments, NFS v3 over UDP was used and the read and write block size was set at 8 KB.

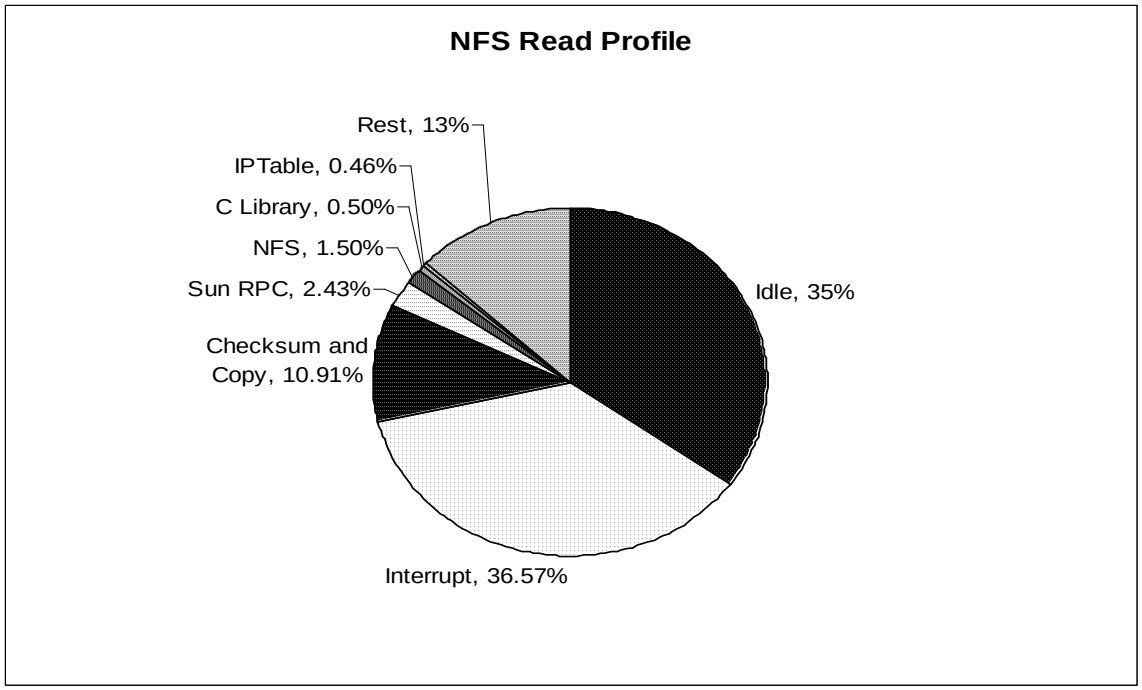



**Figure 6: CPU Utilization for 1 GB Read from NFS server**

Time taken to read a 1 GB file was 23.23 seconds, giving a read throughput of 44 MBPS. We find that 65% of CPU is consumed by a single process doing an NFS read of a 1 GB file. Here again most of the CPU time is being spent in interrupt handling. We find that for a single file read, NFS seems to be doing much better than iSCSI. The NFS server has 1 GB memory, thus most of the 1 GB file seems to fit in memory resulting in good performance.

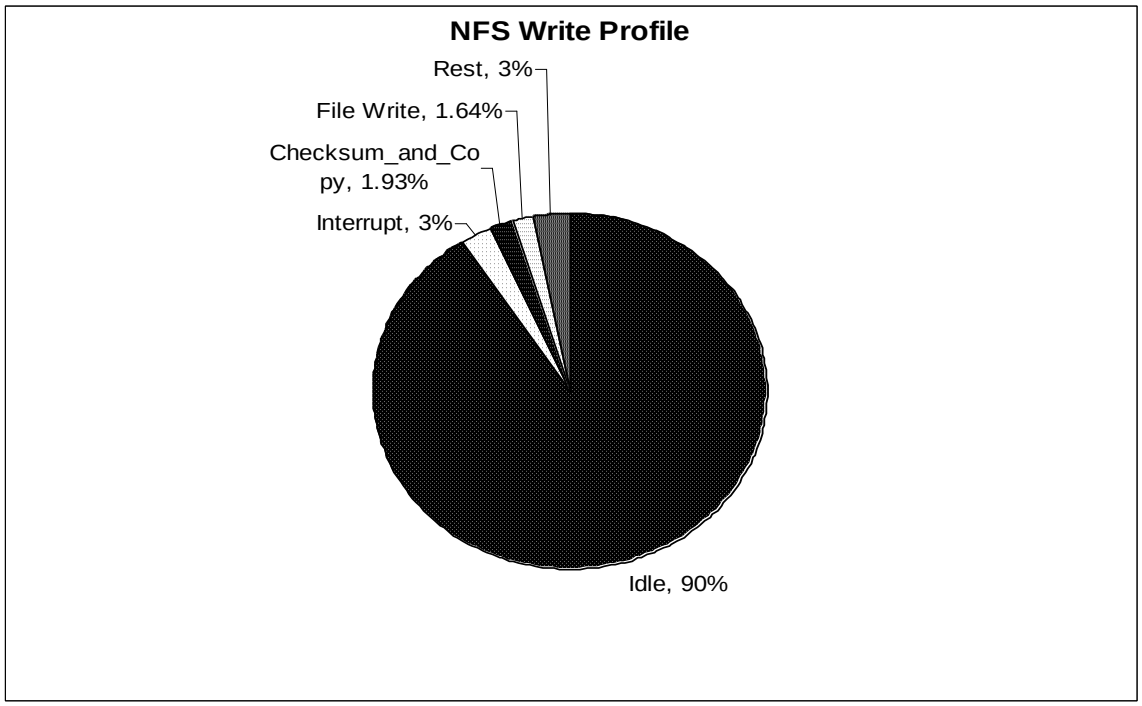



**Figure 7: CPU Utilization for 1 GB Write to NFS server**

Time taken to write a 1 GB file was 1 min 44 seconds, yielding a write throughput of 9.85 MBPS. The reason writes are performing badly in NFS is that before a write to an NFS server returns, the data should have made it to disk. The overall CPU utilization is quite low, around 10%.

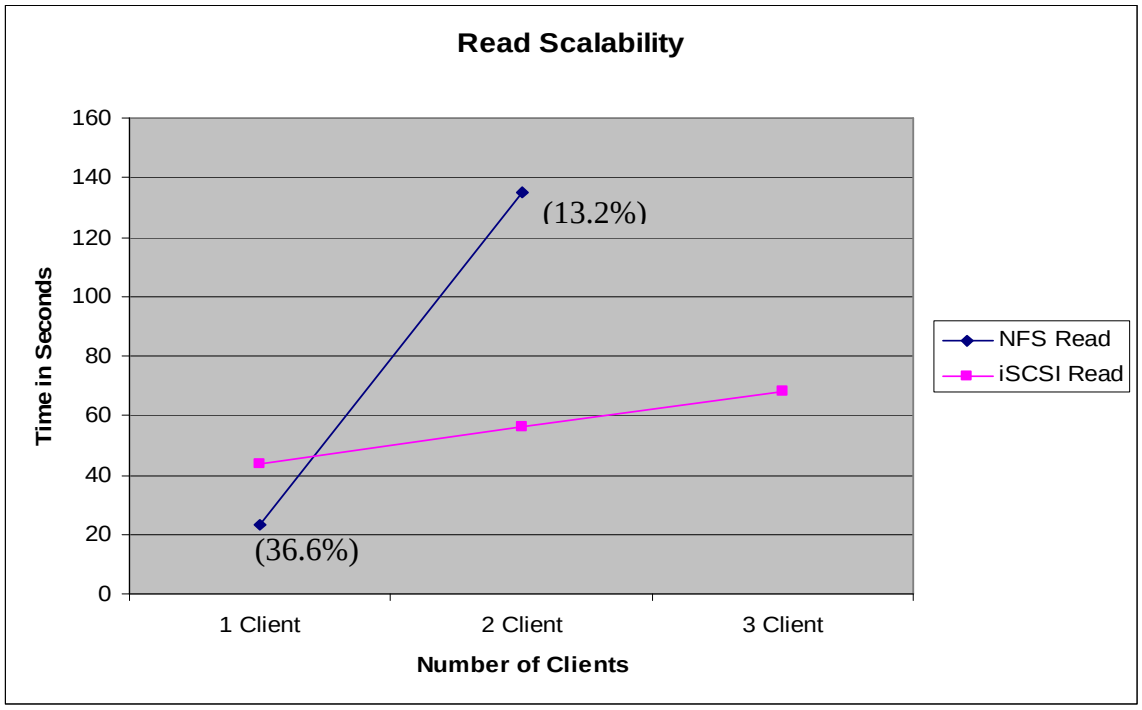



**Figure 8: NFS and iSCSI Read Scalability**

Figure 8 shows the results of our limited analysis of NFS versus iSCSI scalability. Ideally, we would have liked more machines with which to test, but we had a total of 3 machines. For NFS, one of the machines was used as a server leaving only 2 client machines for the scalability study. For iSCSI, all 3 machines participated as clients in the scalability study. For the NFS scalability, the NFS server CPU utilization is shown in parentheses. We find that NFS performance drops as the number of client increases, the reason being that each client is accessing a different file and the nearly 1 GB server side cache is not that effective. Most of the NFS server time is spent waiting for I/O to complete. iSCSI seems more scalable and we think that the trend will continue as the number of clients increases.

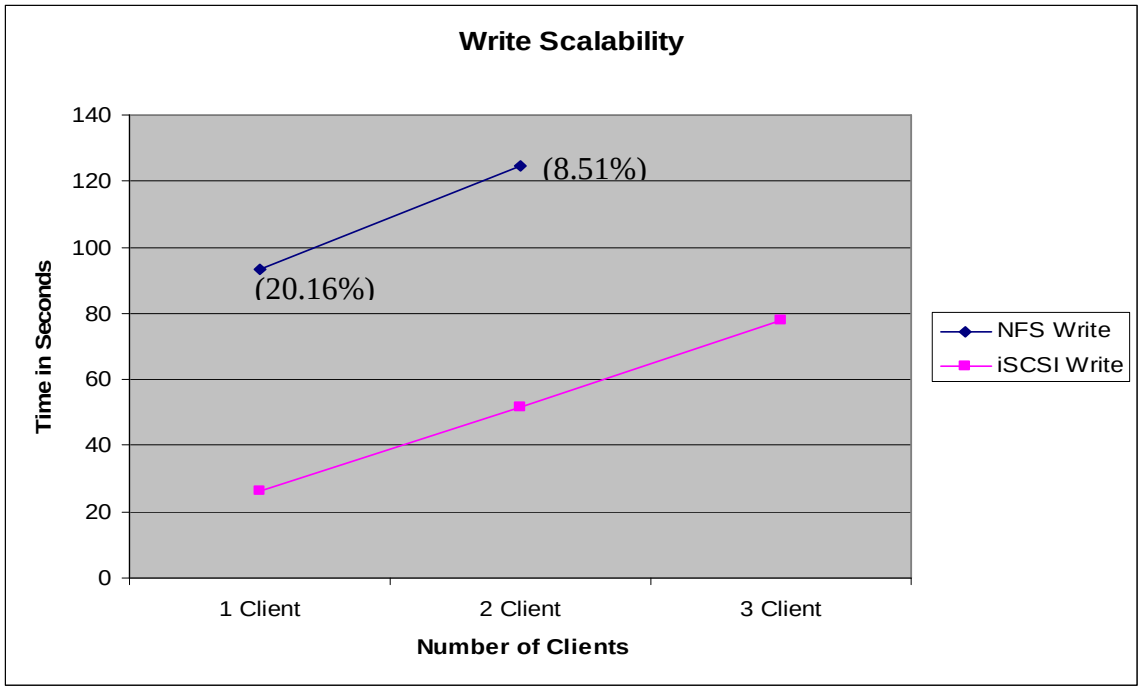



**Figure 9: NFS and iSCSI Write Scalability**

Figure 9 shows the scalability of NFS and iSCSI for writes. As can be seen in the figure, iSCSI performs much better on writes than NFS and appears more scalable.

### 4.3 Tuning iSCSI

Using the information gleaned in the previous analysis, we attempted to tune the performance of iSCSI. We had found that the interrupt handling was the biggest overhead during reads. The first improvement attempted was to use jumbo packets on the Gigabit Ethernet interface. We hoped that a six fold increase in packet size (from 1500 to 9000 bytes) would reduce the number of interrupts on the receiver. We additionally wanted to see what effect it had on NFS performance. Although there was some difficulty enabling the MTU of 9000 on the CISCO SN-5428, the option was finally set. We also had to enable jumbo packets in the Gigabit Ethernet switch. After enabling jumbo packets along the path, the experiments were rerun to find the new profile for iSCSI.

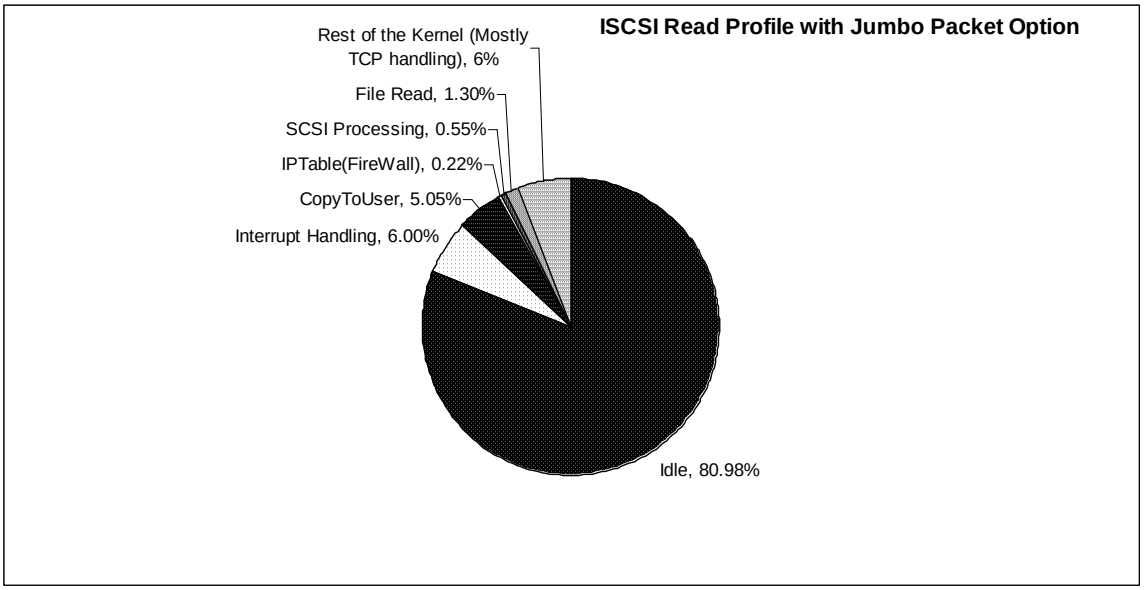



**Figure 10: CPU Utilization for 1 GB Read from an iSCSI target using Jumbo Packet**

By making the iSCSI gateway send jumbo packets (9000 bytes) instead of normal packets (1500 bytes), we find that we have essentially halved the CPU utilization compared to original. Also the read rate improved by 1.25 MBPS. The interrupt processing has become one-third of the original. The CPU usage by other TCP functions also dropped considerably as the number of packets processed dropped to one-sixth of the original. Looking at the packet trace, we found all the incoming data packets were of MTU size.

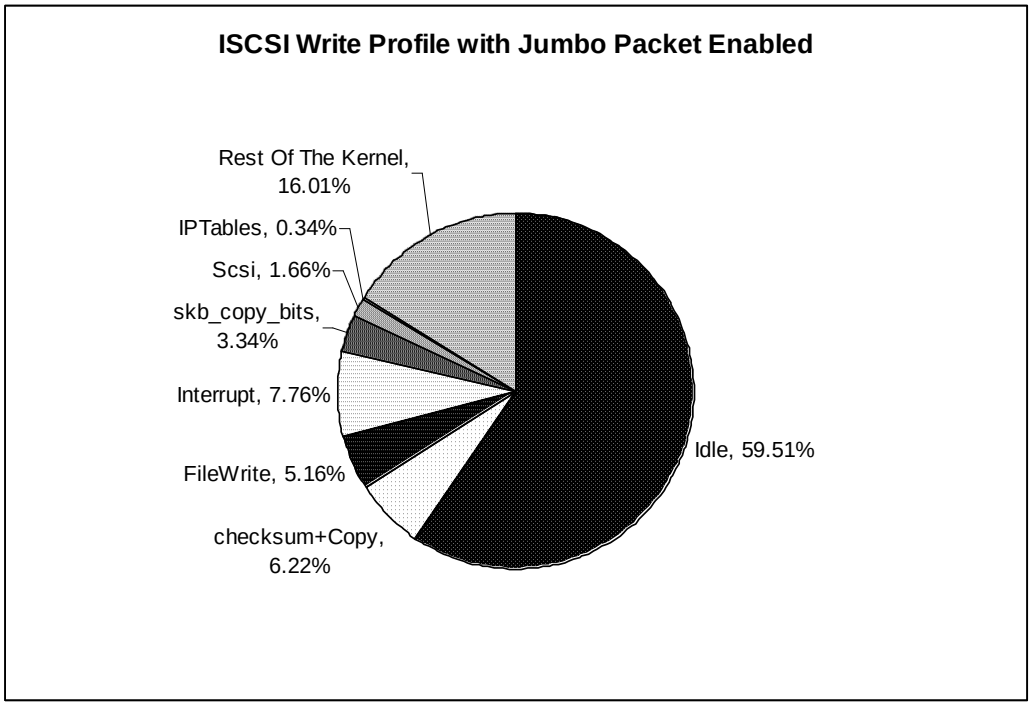



**Figure 11: CPU Utilization for 1 GB Write to an iSCSI target using Jumbo Packet**

Looking at the CPU profile we find that using jumbo packets seems to be increasing the CPU utilization. The reason for this is the *skb_copy_bits* function, which is taking 3.34% of the CPU. Using jumbo packets, the iSCSI driver sends out MTU sized packets. The data for these packets have to be taken from multiple pages requiring a copy for some of the bytes in the packet. We are not sure if the card supports scatter/gather DMA operations.

The iSCSI write performance improved by 1.2 MBPS when sending jumbo packets instead of normal sized packets while the gateway still sent normal packets. However, we found that if the gateway also sent jumbo packets, the performance dropped by 1.36 MBPS. We found the reason to be the sending of ACKs by the gateway. The gateway was advertising a window size of 64 KB, and the sender is forced to stop waiting for ACKs. The Linux TCP implementation sends an ACK every two packets. The gateway does not do that. Observing the flow of packets, it seems like the gateway is following a stop-start protocol. It receives a set of packets and then sends out a set of ACKs. We think that tuning the advertised window on the gateway would improve performance. Increasing the TCP buffer size on the client side did not show any improvement. At sizes over 1 MB, we found performance degradation.

For comparison, we wanted to see the maximum throughput that an application can achieve through the Gigabit Ethernet interface. The maximum throughput was measured using a simple TCP benchmark application of our own devise. The sender wrote data from memory to a socket. The receiver just received the data and discarded it, maintaining only statistics of the amount of data received. Such actions give a realistic measure of the data rate that an application can see as the entire TCP stack is traversed and there is also a copy from/to user space. With normal sized packets we found that the maximum we were able to achieve was 72.3 MB/sec[2]. At this point we found that the receiver CPU was saturated. The sender CPU utilization was at only 35%. Enabling jumbo packets on the interface, we found that we achieved a throughput of 99.4 MB/sec and only 53% receiver CPU utilization. The sender CPU utilization also increased to 54.5%, with 12.5 % used by *skb_copy_bits*. We feel that using jumbo packets changes the bottleneck from receiver to sender. We also feel that scatter/gather DMA is a good option for gigabit Ethernet cards.

When we increased the MTU for NFS server, the read rate seemed to drop. This seemed to deal with the way requests are sent. NFS v3 over UDP seems to be hand tuned for 100 Mbps local area network. Using it over Gigabit Ethernet seems to the stressing the UDP parameters in the RPC library resulting in lower performance. The CPU utilization on receiver client side however did drop considerably.

## 5. Distributed File System Design and Implementation Status

Using the implications presented in the previous section, we now describe the design for our simple, application-specific distributed file system optimized for use with iSCSI. The key insight of our design is similar to that for NASD, that by allowing clients to access data on networked-storage directly, you can improve the scalability of a file server tremendously since it no longer has to perform the disk access. Confirmation of this assumption is shown in limited detail in the scalability study outlined in the previous section. However, even as in traditional distributed file systems, the overhead of crossing the network on every access is prohibitive. The situation is even worse in the case of using iSCSI, due to the very high overhead of interrupt processing from the Gigabit Ethernet adapter. In order to mitigate such performance overhead, we suggest the use

[2] This is the data rate seen over TCP. Actual bits on the wire are higher -- 52 bytes more for every 9000 bytes.

of caching. Caching in our distributed file system would be performed by both clients and the server. Clients should cache data read from the device, and the server should cache file system meta-data (e.g., superblock and inode information) that was recently requested by clients. The use of caching leads us into issues of consistency when dealing with file data writes. As such, our design includes a form of callbacks similar in spirit to those of AFS [H88], so that whenever a server sees a write request for a file that may be cached by other clients reading the file, the server will notify the clients that they must reread their data from the disk. In order to support callbacks, we have devised a simple request/response protocol that sequences the actions of clients. Before being able to read or write a file, the client must issue a file request to the server, who responds with the information necessary for the client to access the networked-storage device. Finally, we briefly treat the idea of security within the distributed file system. Although the iSCSI specification does allow for multiple security methods, they only apply to the security of communication between iSCSI end points. In a distributed file system, we must have a means for authenticating clients before allowing them access to file data. Thus, there needs to be some mechanism for the server to authenticate clients before allowing them access to the device. One option that seems promising is the use of MAC or IP address filtering on the iSCSI gateway, where the addresses allowed are configured dynamically by the distributed file system server.

Throughout the design process, we focused on eliminating any potential source of overhead that could easily be avoided. One such aspect of our design that follows this principle is the used of kernel-level threads for implementing the client and server portions of our DFS protocol, in order to avoid the obvious overhead of context-switching. Both the client and server kernel threads are implemented by a single Linux module that includes a character device allowing the root user to control starting and stopping of the threads. The initial implementation creates one thread for processing client-side requests and six threads for server-side processing, including one thread devoted to listening for new client connections, and five worker threads that provide client request service. The number of server worker threads is identical to the number of service threads provided by default for NFS, thus allowing fairer comparisons. The current design stipulates that the client connects to the server when its thread is created, and the connection is persistent until the client thread is stopped. Once again, the idea for maintaining a persistent connection is to eliminate the overhead of creating and destroying connections per client request. On the server side, a worker thread is currently assigned to a specific connection for its duration. This decision effectively limits the number of active clients to five, and remedies to this limitation are under consideration.

The communication between client and server threads is based on a simple request/response protocol implemented using TCP. Table 1 shows the types of request and response messages, including any data arguments sent and a short description of the messages purpose. As can be seen in the figure, each type of message is associated with a message tag. The tag is used for demultiplexing the type of message upon receipt in order to determine what argument data should also be received.

| **Message Tag** | **Sent By** | **Data Arguments** | **Message Description** |
|---|---|---|---|
| Read_Superblock_Req | Client | Filesystem_id | Request filesystem superblock |
| Read_Superblock_Rsp | Server | Superblock_info_t | Return filesystem superblock info |
| Write_Superblock_Req | Client | Filesystem_id, Superblock_info_t | Request to overwrite superblock info for filesystem |
| Write_Superblock_Rsp | Server | Success_code | Return success status code |
| Release_Superblock_Req | Client | Filesystem_id | Release client from filesystem users list |
| Release_Superblock_Rsp | Server | Success_code | Return success status code |

| | | | |
|---|---|---|---|
| Read_Inode_Req | Client | Filesystem_id, Inode_no | Request inode info, add client callback |
| Read_Inode_Rsp | Server | Inode_info_t | Return inode info |
| Write_Inode_Req | Client | Filesystem_id, Inode_info_t | Request to overwrite inode info, invoke callbacks of current readers |
| Write_Inode_Rsp | Server | Success_code | Return success status code |
| Delete_Inode_Req | Client | Filesystem_id, Inode_no | Request to delete inode, invoke callbacks of current readers |
| Delete_Inode_Rsp | Server | Success_code | Return success status code |
| Release_Inode_Req | Client | Filesystem_id, Inode_no | Release client from inode users list, breaks callback |
| Release_Inode_Rsp | Server | Success_code | Return success status code |
| Break_Callback_Req | Server | Filesystem_id, Inode_no | Request from server to client to break callback on inode |
| Break_Callback_Rsp | Client | Success_code | Return success status code |

**Table 1: Communication Protocol Message Types**

In order to describe the usage pattern of the various message types, we will discuss the messages in the context of the typical usage of the file system by a client. The first thing a client must do to access the file system is to use the mount command. The file system code for handling the mount then adds a *Read_Superblock_Req* to a queue of requests for the client thread, using a "uniquified" version of the SCSI device name as the *Filesystem_id*, and blocks waiting for the response containing the necessary superblock information. The client next issues a *Read_Inode_Req* for inode 0, which is the root directory for the file system. Once the information for the root directory is returned, the client can now access the storage device to read the contents of the directory. Any files discovered by the client when accessing the device can then be accessed by issuing further *Read_Inode_Req* requests and using the returned information to read the file from the device. At this point, a client may be interested in writing to the file. To do so, the client sends a *Write_Inode_Req* to the server indicating the state of the file as if the write had already been done in the *Inode_info_t* argument. Once the server successfully rewrites the inode information and breaks any callbacks for current readers of the file, a success code is returned to the client who may then proceed to actually write the data to the device. When a client has completed its access to a specific file, it issues a *Release_Inode_Req* to the server to break its callback for the file. Similarly, when a client unmounts a file system, it sends a *Release_Superblock_Req* to the server. The message types presented in the figure but not yet explicitly described are straightforward in their actions.

The current status of the implementation is that the request/response protocol is implemented, but not all the functionality implied in the message description field has been finished. Specifically, we do not currently maintain the client lists for open file systems and files necessary for use with the callback mechanism, which itself is currently incomplete. We also do no form of caching of information in the current system on either clients or server. A rough analysis of the time necessary for a request/response transaction shows our kernel thread based implementation to perform favorably, as the time required roughly matches the round-trip time between client and server as reported by the ping utility.

## 6. Conclusions

We have done an in-depth analysis of iSCSI protocol and a system-level characterization of an iSCSI initiator. The protocol has quite high overhead. For high performance servers, replacing server-attached storage with storage accessed via iSCSI may not be a good option. Most of the

overhead is caused by Gigabit Ethernet interrupt handling. Turning on jumbo packet reduces the read overhead by 50 %. For wide-area use this may not be possible because the path MTU is usually much less than 9000 bytes. The jumbo packet option does not seem to reduce the overhead in case of write. We think a detailed analysis of the Gigabit Ethernet driver and the DMA interface would shed light on why this is the case and what might be done.

We think iSCSI can be used to build a distributed file system in a trusted environment like a cluster. The performance overhead of iSCSI is comparable to that of NFS and iSCSI scales better than NFS. To reduce the overhead of network processing, the file system should cache aggressively and employ a strong consistency model like that of AFS. Such a file system built on top of iSCSI would be suitable for data intensive applications being run on clusters.

## 7. Future Work

Although our goals for the project were lofty and we were unable to meet them in the time allotted, we believe that the work yet to be done is still important. Our top priority for future work is finishing the distributed file system implementation, including adding support for caching in the clients and server and implementing the callback mechanism. Once these mechanisms are in place, we will be able to perform a much fairer comparison with NFS. We would additionally like to further the scalability study by including more clients, to give more accurate insights into large scale behavior of our system versus NFS. One last topic that needs more consideration is security within our distributed file system, which is currently non-existent as all clients that can reach the iSCSI gateway over the network can access the storage device.

## References


[A97] J. Anderson et al., “Continuous Profiling: Where Have All the Cycles Gone?” In Proceedings of the Symposium on Operating System Principles (SOSP97), 1997.

[ACV00] D. Anderson, J. Chase, and A. Vahdat. “Interposed request routing for scalable network storage.” In Fourth Symposium on Operating Systems Design and Implementation, 2000.

[AGPW03] S. Aiken, D. Grunwald, A. Pleszkun, and J. Willeke. “A Performance Analysis of the iSCSI Protocol,” 2003.

[G97] G. Gibson et al. “File server scaling with network-attached secure disks.” In Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97), 1997.

[H88] J. H. Howard et al. “Scale and performance in a distributed file system.” ACM Transactions on Computer Systems, 6(1):51--81, February 1988.

[HIT02] K. Hiraki, M. Inaba, and J. Tamatsukuri, “Data Reservoir: Utilization of Multi-Gigabit Backbone Network for Data-Intensive Research,” In Proceedings of IEEE/ACM SC2002 Conference, Nov. 2002.

[J03] J. Satran et al., “iSCSI,” IETF Internet Draft, January 2003. Available online at http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-20.txt

[SV02] P. Sarkar, K. Voruganti, "IP Storage: The Challenge Ahead," 10th Goddard Conference on Mass Storage Systems and Technologies and 19th IEEE Symposium on Mass storage System, pg. 35-42, 2002.

[TMI] “Performance Comparison of iSCSI and NFS IP Storage Protocols,” TechnoMages, Inc. White Paper.

[VS01] K. Voruganti, and P. Sarkar, "An Analysis of Three Gigabit Networking Protocols for Storage Area Networks.” 20th IEEE International Performance, Computing, and Communications Conference", April 2001.