NASA-CR-192898

April 21, 1993

G2-591

Interfacing a High Performance Disk Array File Server to a Gigabit LAN

Srinivasan Seshan and Randy H. Katz

IN-62-CR
158663
p. 19

(NASA-CR-192898) INTERFACING A HIGH PERFORMANCE DISK ARRAY FILE SERVER TO A GIGABIT LAN (California Univ.) 19 p

N93-25458

Unclas

G3/62 0158663

Our previous prototype, RAID-I, identified several bottlenecks in typical file server architectures [Chervenak91]. The most important bottleneck was the lack of a high-bandwidth path between disk, memory and the network. Workstation servers, such as the Sun-4/280, have very slow access to peripherals on busses far from the CPU. For the RAID-II system, we addressed this problem by designing a crossbar interconnect, Xbus board, that provides a 40MB/s path between disk, memory and the network interfaces. However, this interconnect does not provide the system CPU with low latency access to control the various interfaces. To provide a high data rate to clients on the network, we were forced to carefully and efficiently design the network software. A block diagram of the system hardware architecture is shown in Figure 1. In the following subsections, we describe pieces of the RAID-II file server hardware that had a significant impact on the design of the network interface. Other papers, [Lee92, Katz93], describe the architecture and implementation of the RAID-II server in greater detail.

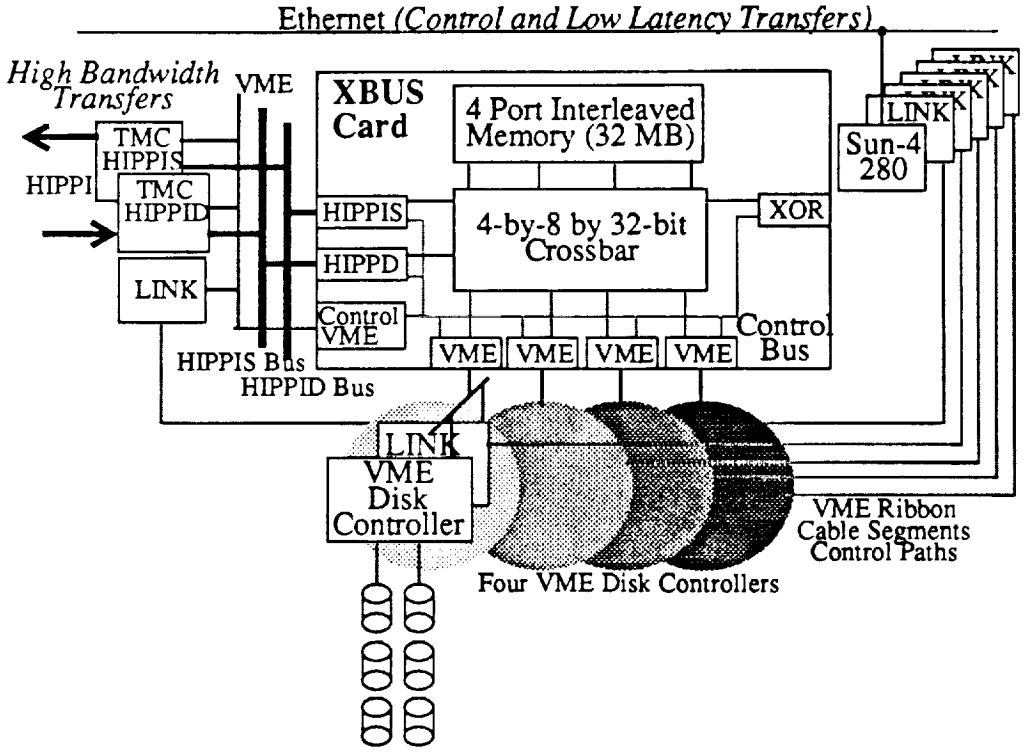


FIGURE 1. **: RAID-II Organization.** A high-bandwidth crossbar interconnect ties the network interface (HIPPI), the disk controllers, a multiported memory system, and a parity computation engine. An internal control bus provides access to the crossbar ports, while external point-to-point VME links provide control paths to the surrounding SCSI and HIPPI interface boards. Up to two VME disk controllers can be attached to each of the four VME interfaces. The design originally had 8 memory ports and 128 MB of memory; however, we built a four memory port version to reduce manufacturing time.

2.1 VME Link Boards

The remote VME links that connect the host CPU to the other sections of the system are extremely slow: about 2Mbytes/second for most applications. Single word transfers across the link take 2us each. In a few select applications, the link board can DMA data at up to 20Mbytes/second. To meet our performance goal for data transfer to the network, very little data can be transferred between the host CPU and the other parts of the system.

2.2 TMC I/O Backplane

The TMC I/O backplane consists of two unidirectional busses, HIPPIS bus and HIPPID bus, that move data between the TMC HIPPI boards and the XBUS board. Both busses are addressless and only the TMC boards may be the bus master. We used a simple bus protocol that allows the TMC board to select a target or source board for the transfer and to do flow control. Since the bus is addressless, any source or target device (for example the XBUS board) must be set up before the transfer is initiated.

2.3 Host

The host CPU in the RAID-II server is a Sun-4/280 single board computer. It has 32MB of VME memory and runs the Sprite operating system. The CPU performance of this machine is approximately 8 SPECMarks. The host is responsible for running most of the code that controls the RAID-II server. It is responsible for running the file system code and controlling the drive interfaces, XBUS board and HIPPI boards. The host is slow by today's standards and is expected to be heavily loaded with file system and control tasks. It is important that the network interface not place a significant additional load on the CPU.

2.4 XBUS Board

The XBUS card implements a 4-by-8 32-bit wide crossbar bus. This board provides a high bandwidth path between the disk controllers, memory and the network interface. Two of the crossbar ports provide connections to the TMC I/O backplane. Since the TMC I/O backplane busses are addressless, the Xbus board must be set up for any transfers across the backplane in advance. These ports can sustain 40Mbytes/second of transfer to/from the TMC HIPPI boards. A control VME connection uses another single port. The Xbus board is controlled by a set of registers present on this VME interface. These registers can be written to by the TMC HIPPI boards or the Sun-4 CPU. Other ports provide connections to a 32MB memory, a hardware XOR compute engine and four disk controller boards.

2.5 SCSI controllers

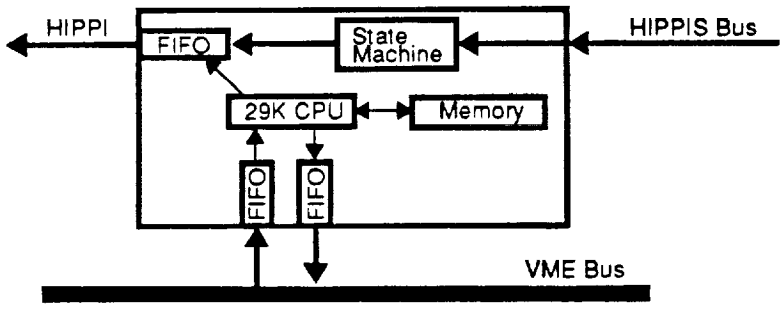
The XBUS board is connected to a set of four VME busses. Each of these VME busses currently contains an Interphase Cougar SCSI controller. Each board is capable of handling approximately 7 Mbytes/second of data traffic from two independent SCSI strings. Physical packaging limits each string to 3 disks.

These boards limit the RAID-II system to a maximum of 28Mbytes/second of disk bandwidth and 24 disks. Software and other bottlenecks may limit the performance further. Future SCSI boards will allow the system to use 72 disk drives and to provide up to 32Mbytes/second per XBUS board.

2.6 TMC HIPPI Boards

The HIPPI interface for RAID is implemented using a two board set built by Thinking Machines Corporation (TMC). The architecture of the boards is shown in Figure 2, "TMC HIPPI Board Block Diagram," on page 4. Each board contains an interface to a single direction of the HIPPI channel, a unidirectional backplane bus and a control VME bus. Each board also contains a AMD 29000 (29K) processor and some local memory. Programs and data for the 29K processor can be downloaded from the VME control bus. The 29K processor can be used to set up transfers and to run any general purpose code (protocol code). Some significant differences exist between the two boards. The individual boards are described in more detail in the next few subsections.

- Source Board:



- Destination Board:

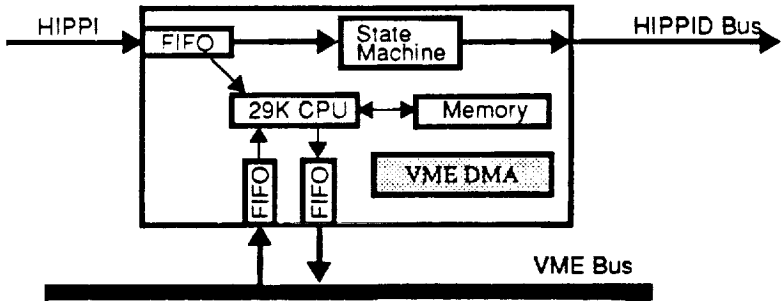


FIGURE 2. TMC HIPPI Board Block Diagram

2.6.1 HIPPI Source Board

The HIPPI source board interfaces to the VME bus through a set of five registers. The functions of these registers are summarized in Table 1. The input and output FIFO are the most important of these registers since they provide the only general purpose communication interface between code running on the 29K processor and the host CPU. Since the source board has no VME bus mastering capability, data must be copied into the input FIFO and from the output FIFO. This VME interface has two important consequences.

TABLE 1. HIPPI Source Board VME registers

VME Register	Description
Configuration	Sets up VME functionality of board - enable interrupts, set VME address modifier, etc.
Input FIFO	Receives data from VME into FIFO read by 29K.
Output FIFO	Stores data written by 29K into a FIFO that can be read from the VME.
Status	Stores current status of board
Reset	Controls reset of various sections of board

First, the Xbus board must be set up for transfers to the source board by some other part of the system (e.g host CPU). Second, since there is no mechanism to lock access to the FIFOs, the Sun-4 CPU and the destination board cannot both communicate with the source board. Therefore, to prevent mixing of data from two sources, we limit access to only the Sun-4 CPU.

The source board's interface to the HIPPI output channel and the input backplane bus is controlled by a set of on-board registers. These registers are only accessible by the 29K processor. A single FIFO is used to store data to be sent out on the HIPPI channel. A simple state machine fills this FIFO with data from the TMC I/O backplane. The 29K initiates this transfer by writing various registers. It must know the total length of the transfer in advance. By not involving the 29K in copying data from the backplane, the system can achieve the full HIPPI bandwidth (100Mbytes/second).

2.6.2 HIPPI Destination Board

The destination board includes several more features than the source board. The VME interface has several new registers that provide general purpose communication with the host. The destination board VME registers are summarized in Table 2. In addition to the new registers, the destination board supports VME bus mastering. The board can write data from the output FIFO to any VME location. Similarly, it can read VME locations to the input FIFO. As a result, the HIPPI destination board can set up the XBus board for transfers.

The destination board's interface to the HIPPI input channel is composed of a set of registers accessible by the 29K. Data from the HIPPI channel is automatically placed in a FIFO, which can be copied by a state machine to the backplane. The 29K must set up the transfer to the backplane by writing the length of the transfer to an on-board register. The 29K must then poll the status of the state machine to identify the end of the transfer.

TABLE 2. HIPPI Destination Board VME Registers

Register	Description
Configuration	Sets up VME functionality of board - enable interrupts, set VME address modifier, etc.
Input FIFO	Receives data from VME into FIFO, can be read by 29K.
Output FIFO	Stores data written by 29K into a FIFO that can be read from the VME.
Status	Stores current status of board
Command	Stores data written from VME, can be read by 29K.
Response	Stores data written by 29K, can be read from VME.
Reset	Controls reset of various sections of board

2.7 Ultranet

The UltraNetwork is a hub-based store and forward network capable of transmission rates up to 1Gbit/second. Figure 3 shows our Ultranet topology. The hubs create a high speed switching interconnection by routing incoming packets to the proper destination.

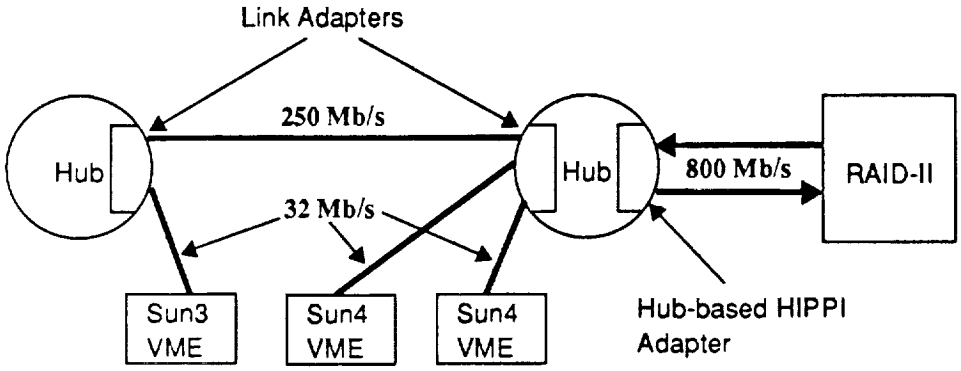


FIGURE 3. UC Berkeley UltraNetwork Topology

Hubs are physically connected by serial links capable of transmission rates of 250Mbits/second. Up to 4 links can be used between a pair of hubs. Data is striped across these links to achieve Gbit/second speed. These links terminate in link adapters in the hubs. Link adapters are also used to connect to machines with Ultranet host adapters. Host adapters are available for machines with industry standard backplanes (e.g. VME). Each host adapter contains an on-board microprocessor and can perform DMA to the host's memory. The on-board microprocessor does all the protocol processing necessary to communicate across the UltraNetwork to remote clients. Computers without standard backplanes, typically mainframes and super-computers, can connect to the UltraNetwork using standard channel interfaces (for example HIPPI, HSX) to a hub-based adapter. This essentially moves the network interface into the hub itself. Much of the UltraNetwork protocol is handled by the processor on the hub-based adapter. However, software must run on channel

connected hosts to handle communication to the hub-based adapter. This software is described in more detail below.

VME Ultranet host adapters in a Sun system provide a maximum of about 4Mbytes/second to the network. On the basis of the RAID-II performance goal of 40Mbytes/second, we decided that a HIPPI attachment to the drive array was necessary.

Each transfer between the UltraNetwork hub and the hub-adapter attached host is composed of a DMA word followed by either a request block or data. The maximum size of the data segment of each transfer is limited to 32KB by the Ultranet adapter. The DMA word accompanying each transfer describes the contents of the transfers. Analyzing the DMA word provides sufficient information to identify the correct memory destination for the transfer. Request blocks are commands that pass between the hub-based adapter and the host. Each request block roughly has an analogue in BSD 4.2 network socket calls. This made it easy to provide the file system with a socket interface to the network. Several of the most important request blocks are summarized in Table 3. Only a few standard data formats are used to transmit the various request blocks. As a result each request block requires sending significantly more data than is necessary.

TABLE 3. UltraNetwork Request Blocks

Request Block	BSD Equivalent
OPEN	socket()
ADAPTER LISTEN	combination of bind(), listen() and accept()
CONNECT	connect()
CLOSE	close()
SEND	send()
RECEIVE	recv()

3.0 Software Architecture/ Implementation

Both TMC and Ultranet provided software to support the original uses of their systems. After examining the provided code, we decided that completely new software was needed for several reasons. First, the RAID-II file server runs the Sprite Operating System. Both the TMC and Ultranet software were developed for Sun-OS and needed a significant amount of work to port to Sprite. Second, the software was developed to support the more standard machine interconnection. As a result it could not provide the high performance we needed on the RAID system. In this section, we describe the organization of the networking software we developed for the RAID-II file server. We examine the decisions made during the software implementation and the reasoning behind these decisions.

3.1 Architecture

The interface provided to the file system code and the division of code between the 29K and Sun-4 CPU were two basic issues of the software architecture. Based on the Ultranet request block format, we decided to provide the file system code with a socket interface to the network, making both the networking and file system code easier to implement. Also, we decided to implement most of the software in the 29K for a variety of reasons. First, it had been estimated that the Sun-4 CPU would be heavily loaded by running the file system software and controlling the hardware of the RAID-II system. Second, the connection of the Sun-4 CPU to the rest of the system is through slow VME link boards. The involvement of the Sun-4 CPU in data transfers would reduce the bandwidth of the RAID-II server significantly. To support a high bandwidth between the network and memory, the 29K CPUs must control as much of the data transfer as possible. The 29K CPUs were programmed to understand the Ultranet request block interface and handle incoming data transfers. However, since access to the source board cannot be shared, the Sun-4 CPU must set up the outgoing data transfers. The software architecture is shown in Figure 4. An example transfer is described in the next section to clarify the software architecture.

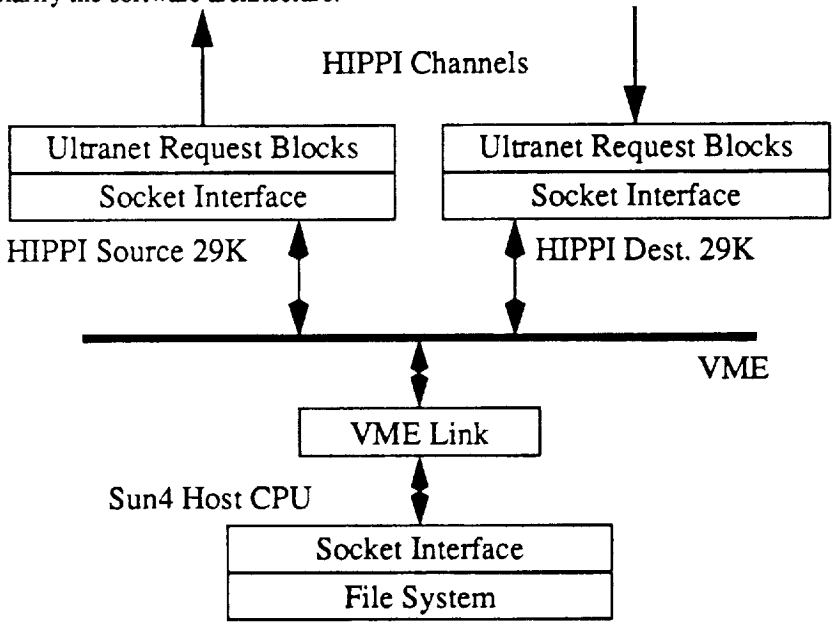


FIGURE 4. Software Architecture Division

3.1.1 Sample Transaction

This section describes a sample network transaction that may occur between the RAID server and a client on the Ultranet. In this example, the client creates a connection to the server, sends some data and receives a reply. This communication is graphically shown in Figure 5.

1. The file server will start by issuing an `open()` of a socket. This will result in the HIPPI source board sending out an OPEN request block. The HIPPI destination board will receive the completed OPEN request block from the Ultranet hub. The destination 29K interprets the request block and returns a new socket id to the file server and the `open()` call completes.

2. The file server issues a `listen()` on the socket id. This is accomplished by the source board sending an ADAPTER LISTEN request block to the Ultranet hub. `listen()` is a combination of BSD bind(), listen() and accept(). When a client creates a connection to the file server the completed ADAPTER LISTEN request block returns to the destination board. The destination board sends information about the established connection to the file server and the `listen()` call completes.

3. The file server does a `recv()` on the connected socket id. A pointer to an empty host buffer and a tag that uniquely identifies the transfer are passed to the destination board. The source board sends a RECEIVE request block to the Ultranet hub. The Ultranet matches up the request block with a client's send of data and transfers the data to the HIPPI destination board. The destination board uses the unique tag to identify the transfer. The destination board then sets the XBus board up for the transfer and begins the transfer to the backplane. A completed RECEIVE request block is sent by the Ultranet hub after the transfer completes. The destination board sends the request block status to the file server and the `recv()` completes.

4. The host will issue a `send()` of the reply data on the socket id. A pointer to the host data buffer and a tag that uniquely identifies the transfer are passed to the destination board. The source board sends a SEND request block to the Ultranet hub. The Ultranet matches up the request block with a client's receive of data. The Ultranet hub sends a request to the HIPPI destination board to begin transfer of the data. The destination board uses the unique tag to identify the transfer request and determine the data to be sent. The destination board requests the Sun-4 to set up the Xbus board and source board to transfer the desired data to the Ultranet hub. A completed SEND request block is sent to the destination board by the Ultranet hub after the transfer completes. The destination board sends the request block status to the file server and the `send()` completes.

3.2 Source Board Code

From the example transfer, it should be evident that the HIPPI source board must send both data and request blocks to the Ultranet hub. The commands to perform these actions are summarized in Table 4. These commands are executed by the Sun-4 CPU writing to the source boards VME FIFO. In general, the Ultranet request block formats contain many data fields that can be eliminated. Commands between the 29K and Sun-4 CPU contain only the essential fields of the associated request blocks. The effectiveness of this "compression" is discussed in Section 4.2, "Reduction of VME Link Traffic".

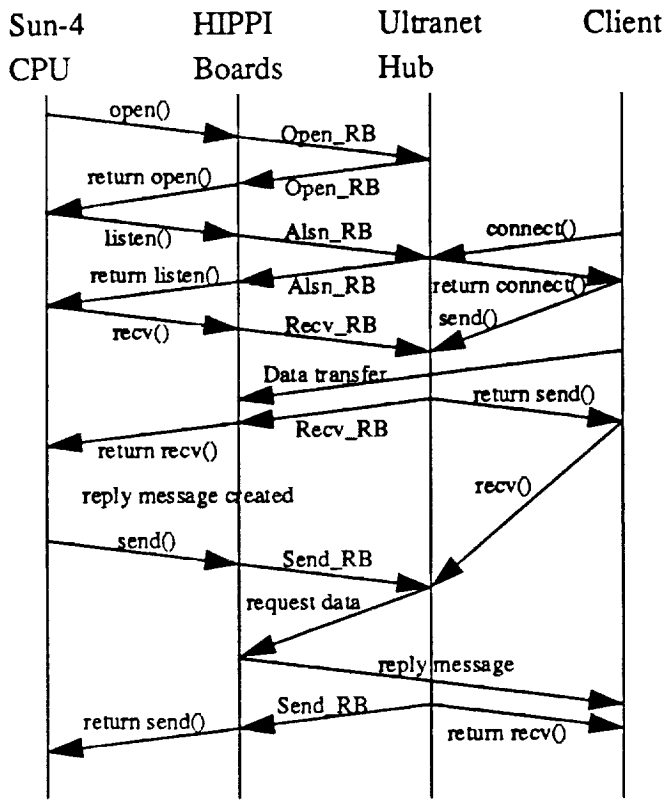


FIGURE 5. Communication between Sun-4 CPU, TMC HIPPI Boards, Ultranet Hub

TABLE 4. Commands between Source 29K and host CPU

Command	Description
`UltraOpen()`	Sends request to UltraNetwork to open a socket.
`UltraListen()`	Sends request to bind Socket ID to a port. Then listens for a connection and accepts it.
`UltraClose()`	Sends request to close connection active on a socket ID.
`UltraSend()`	Sends request to send data on the connection associated with a socket ID. Each send request block is given a unique 8 bit tag that identifies it.
`UltraRecv()`	Sends request to receive data on the connection associated with a Socket ID. Each receive request block is given a unique 8 bit tag that identifies it.
`SendData()`	Sends a requested number of bytes from the Sun-4 and the XBUS on the HIPPI channel.

3.3 Destination Board Code

To support the example transfer, the destination board needs to interpret the incoming Ultranet request blocks and scatter-gather Ultranet data requests. The Sun-4 uses the command described in Table 4 to notify the destination board of buffers allocated for both incoming and outgoing transfers.

TABLE 5. Commands Between Host CPU and Destination 29K

Command	Description
`ScatterGather()`	Allocates buffers in both Sun-4 and Xbus Memory for a transfer associated with a specific `tag`

The destination board must also read and interpret all transfers from the Ultranet. Every HIPPI transfers sent from the Ultranet hub to the destination board starts with the following DMA word structure.

31. . . .24	23. . . .16	15. . . .8	7. 0
Content Description			
Transfer Offset			
Tag			
Transfer Length			

`Content Description` identifies if the transfer contains a request block, a data request or data. If the current transfer is part of a larger multipart data transfer (larger than 32KByte transfer), `Transfer Offset` provides the byte offset of the data being sent into the entire transfer. `Tag` is the unique identifier for every send or receive of data. `Transfer Length` is the byte length of the current transfer. **Due to buffering limitations in the Ultranet hub, `Transfer Length` is never more than 32KBytes.**

3.3.1 Completed Request Blocks

The destination board 29K must notify the Sun-4 of any completed request blocks it receives. When a request block arrives at the destination board, the VME DMA engine is used to copy the essential fields (same fields that are used by the source board to send a request block) of the request block into the Sun-4 CPU's main memory. Next, the destination board interrupts the Sun-4 CPU to notify it of the completion of a Ultranet request. The host CPU may then examine the completed request block for either status or returned values

3.3.2 Incoming Data

When incoming data arrives at the destination board, the 29K processor uses the `tag`, `transfer offset` and `transfer length` fields of the DMA word and previously processed ScatterGather() commands to determine the destination of the data. If the destination address of the data is in host memory, the 29K removes the data from the HIPPI channel and DMA copies the data to the proper VME location. How-

ever, if the data should be placed in XBUS board memory, the destination board sets the XBUS board up for the transfer by writing to the XBUS VME registers. Next, the 29K enables the state machine to copy data from the HIPPI channel to the XBUS board. The data transfer is complete when the state machine finishes.

3.3.3 Outgoing Data

When a Ultranet request for data arrives at the destination board, the 29K processor uses the `tag`, `transfer offset` and `transfer length` fields of the DMA word and previously received Scatter-Gather() commands to determine the source of the data. The destination board cannot use its VME DMA engine to set up the transfer for several reasons. First, access to the source board VME FIFO cannot be shared by the host and the destination board. Second, the destination board's VME DMA engine reads data into the destination board's VME input FIFO. However, the host CPU must also access this input VME FIFO. Access to this FIFO cannot be shared. As a result the host CPU must set up the transfer of data. The destination board copies the length and source address of the transfer to its VME output FIFO and interrupts the host CPU. The host CPU uses the length and source address to set up the XBUS and HIPPI source board for the transfer. This is done by writing to the XBUS control registers and issuing the SendData() command to the HIPPI source board.

3.4 Implementation

The approximate size of code running on the TMC HIPPI boards is summarized in Table 6.

TABLE 6. HIPPI Board Code Statistics

Section	Lines of Code	Estimated Man Hours
`Destination Board C Code`	3500	900
`Source Board C Code`	3500	
`Shared TMC Boards C Code`	1500	
`Shared TMC Boards Assembly`	700	

Almost 7000 additional lines of code were written for the Sun-4 host CPU to support the UltraNetwork and HIPPI boards. Much of the code and time can be attributed to the lack of documentation for the Ultranet and the poor match between the architecture and the Ultranet protocol.

4.0 Performance Measurements

In this section, we examine the end-to-end network performance of the RAID-II file server. We analyze measurements of network bandwidth, CPU load of the RAID-II system and system hardware bandwidths to identify the bottlenecks that limit the network performance of the system.

4.1 RAID-II Hardware Performance

The RAID-II system was designed to support a 40MBytes/second data path between disk, memory and network. The performance of the system is carefully analyzed in [Chen93]. Measurements show that transfers between the XBus board memory and the TMC HIPPI boards have a latency of 1.1ms and a maximum throughput of 38.5MBytes/second. The majority of this latency is attributed to the configuring of the XBus board and the handling of the HIPPI channel by software on the TMC board. These measurements were taken on a system with minimal software on the host CPU and on the 29K CPUs. They indicate the maximum achievable performance from the RAID-II hardware.

4.2 Reduction of VME Link Traffic

To improve network performance of the RAID-II system, we include only the essential fields of Ultranet request blocks in the messages between the Sun-4 and the HIPPI boards. This was done to reduce the utilization of the slow VME link between the Sun-4 and HIPPI boards. This link is capable of handling 2MBytes/second. The link must carry messages between the Sun-4 and HIPPI boards and file system metadata. The "compression" achieved is summarized in Table 7. On the average, messages are reduced in size by 50%. Unfortunately, the utilization of the link is not easily measurable. As a result, we cannot identify if

TABLE 7. Ultranet Request Block Size in Bytes

Request Block	Normal Size	Compressed Size
OPEN	44	20
LISTEN	92	56
CLOSE	92	20
SEND	44	24
RECEIVE	44	24

it is near saturation.

4.3 Network Performance

The UltraNetwork currently installed at UC Berkeley supports three Sun VME workstations. Each Sun workstation can produce or consume approximately 3.5MBytes/second [Clinger89]. This provides a maximum aggregate bandwidth of 10.5MBytes/second. RAID-II is capable of completely satisfying this network load (and more). Under the current maximum load, all clients receive data at their full desired bandwidth. Therefore, bandwidth limitations of the RAID-II network interface can currently only be estimated from scaling arguments. The performance numbers reported are based on a thousand packets of a fixed size sent over a single connection between the Xbus memory in RAID-II and a client machine on the Ultranet. The

time to complete these transfers was used to obtain both average bandwidth and latency measurements for various packet sizes.

Figure 6 shows the bandwidth of data for different sized packets being sent between RAID-II and individual clients. The bandwidth of a Cray supercomputer communicating with a single Sun-3 client is shown for comparison. SunOS 3.5 operates approximately 10-15% faster than SunOS4.1. The maximum bandwidth for the Sun-3 clients is 3.5MBytes/second reading data from RAID-II and 3.7MBytes/second writing data to RAID-II. The maximum bandwidth for the Sun-4 clients is 3.0MBytes/second reading data from RAID-II and 3.8MBytes/second writing data to RAID-II. This large performance gap reading and writing data from a Sun-4 is due to cache conflicts in the Sun-4 memory system. When data is being written to the

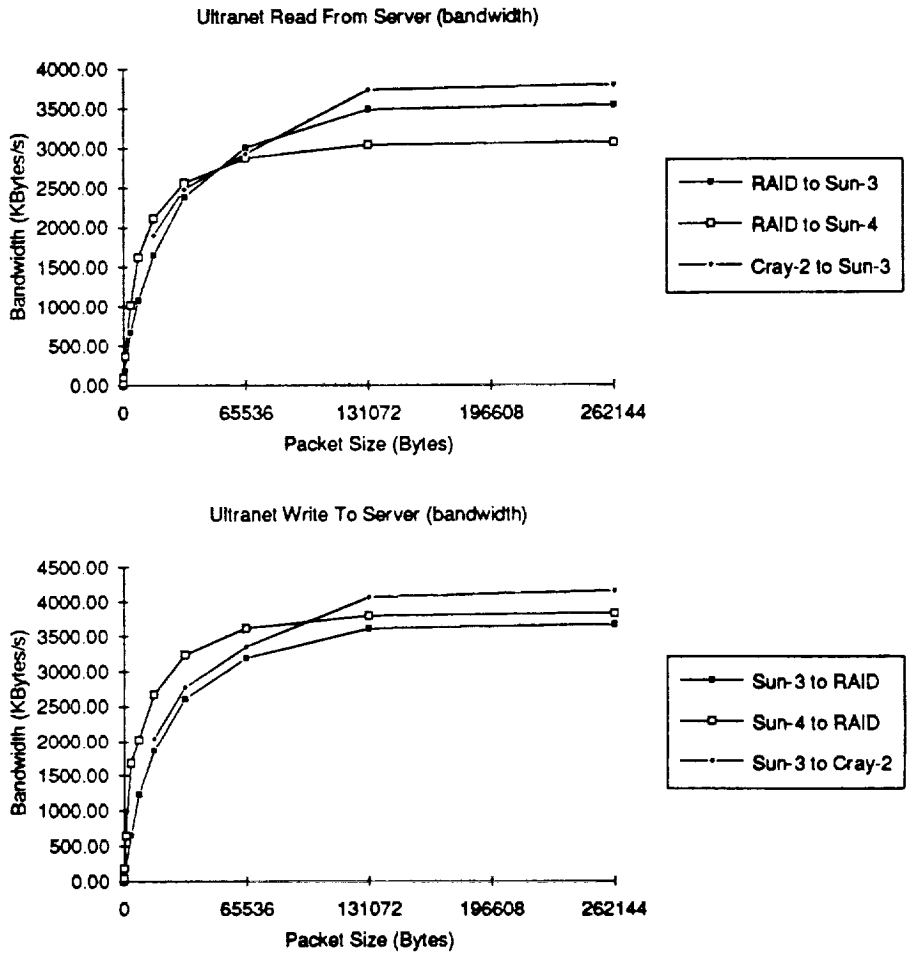


FIGURE 6. Bandwidth vs. Packet Size for transfers between RAID-II and a single client

Sun-4 memory from the network, the virtually addressed cache in the Sun-4 must be updated. This results in a lower bandwidth writing to the Sun-4 memory.

Figure 7 shows the latency to send different sized packets between RAID-II and individual clients. The performance of a Cray supercomputer communicating with a single Sun-3 client is shown for comparison. The minimum latency of packets for a Sun-3 is 6.0ms reading from RAID-II and 4.8ms writing to RAID-II.The minimum latency of packets for a Sun-4 is 2.2ms reading from RAID-II and 1.3ms writing to RAID-II. Measurements of the RAID-II hardware [Chen93] indicate that approximately 1.1ms of this latency is due to delays in the file server. These numbers indicate that it is the processing speed of the clients that limits the end-to-end latency of communication.

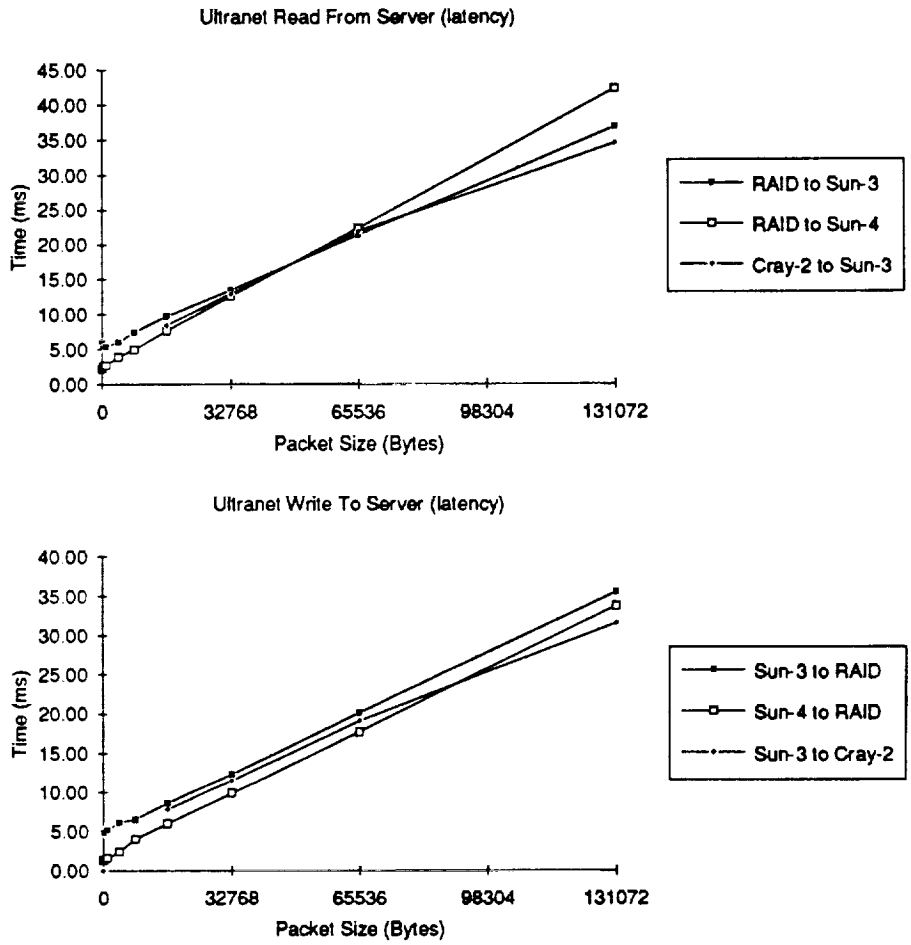


FIGURE 7. Latency vs. Packet Size for transfers between RAID-II and a single

Applications on the clients are unable to consume the 3.5Mbytes/second of data delivered. For example, video stored on the RAID-II file server can be played back on the Sun-4 clients at a rate of 5 frames/second. This corresponds to a transfer rate of 1.5Mbytes/second. The video data is copied across the clients VME backplane twice, first from the network interface to memory and then from memory to the frame buffer. This contention for the VME backplane reduces the available bandwidth in half.

4.4 CPU Utilization

The network software for RAID-II splits the workload of network communication across three processors, the Sun-4 host CPU and the two AMD 29K CPUs on the HIPPI boards. In this section, we examine the CPU utilization of these processors during transfers.

The utilization of the Sun-4 CPU is highly dependant on the packet size of the transfers occurring. Figure 8 shows the utilization of the Sun-4 CPU when all three clients transferring data. The three clients consume/create approximately 10.5MBytes/second of data traffic. When the clients are writing data to RAID-II the host CPU must take a single interrupt per packet. As a result the load on the host CPU is inversely proportional to the packet size. When clients are reading data from RAID-II, the host CPU must be interrupted for every outgoing data fragment transfer requested by the Ultranet hub. All packets are fragmented into 32Kbyte transfers across the HIPPI channel. As a result, the host CPU utilization has a minimum of 48%. CPU utilization remains almost constant for packets larger than 32Kbytes.

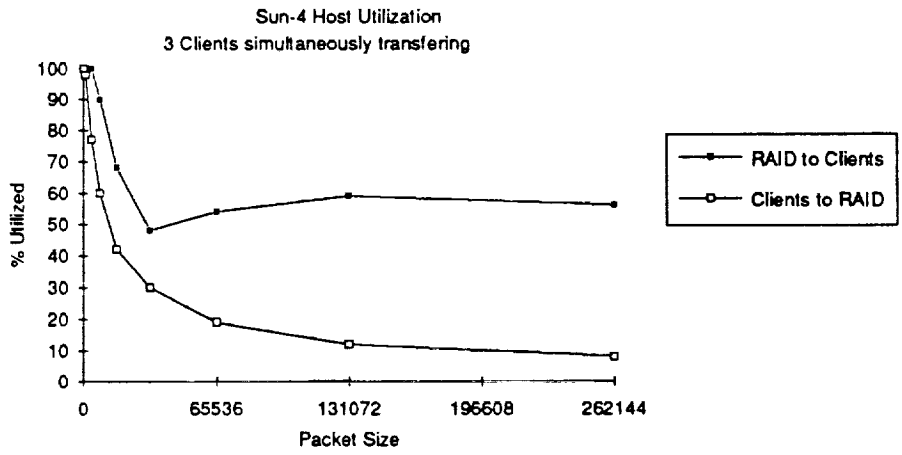


FIGURE 8. Sun-4 Host CPU utilization vs. Packet Size for 3 clients communicating with RAID-II

The host CPU utilization limits the network performance of clients reading from RAID-II to 21MBytes/second, about twice the currently available performance. Since packets on the UltraNetwork can be several megabytes, the host utilization places no limits on the bandwidth of clients writing data to the RAID-II system

The utilization of the 29K CPUs on the HIPPI boards depends mostly on the bandwidth of data being transferred. This is due to the fact that the 29K processors have a fixed computation overhead per 32KByte fragment transferred on the HIPPI channel. Their utilization is, therefore, not dependant on packet size but only on the actual bandwidth of data. Table 8 shows the utilization of the 29K CPUs for different bandwidths of data. When writing data to RAID, the destination board is highly utilized since it must set up and perform

the data transfers. For the same transfers, the source board only processes outgoing request blocks. During reads from the RAID system, the source board must perform the overhead of transferring the data. The destination must still set up the transfers of data.

TABLE 8. Utilization of 29K Processors during Network Transfers

Bandwidth (Mbytes/ second)	Read From RAID		Write To RAID	
	Source 29K	Destination 29K	Source 29K	Destination 29K
3.5	21%	7%	18%	14%
7.0	N/A	18%	16%	23%
10.5	35%	20%	18%	27%

These numbers indicate that the 29K CPUs would limit the network to approximately 32Mbytes/second for both reads and writes to RAID-II.

5.0 Conclusions

The two basic goals of the RAID-II network software were to provide high bandwidth to clients on the UltraNetwork and reduce the load on the host CPU. [Chen93] measurements indicate that the RAID-II system hardware can support a raw bandwidth of 38.5MBytes/second between memory and the network. Based on our scaling estimates, the RAID-II server can source approximately 21Mbytes/second to the Ultranet (limited by the host CPU) and sink 32MBytes/second (limited by the destination board 29K CPU) from the network. Upgrading the host CPU to more modern hardware would allow the RAID-II system to source 32MBytes/second to the network. This bandwidth is significantly higher than that of Ethernet-based file servers in our environment. For comparison, our Sprite OS file server supports a bandwidth of about 1MByte/second to the network [Welch90]. These results show that the RAID-II network interface was effective at providing a high bandwidth to clients on the Ultranet. Although the software design did reduce the load on the host CPU by effectively using the 29K CPUs, we could not prevent the host CPU from being a critical resource for sourcing data. We feel that the network performance of the RAID-II server with Ultranet clients cannot be improved significantly.

With some minor hardware changes, there are a number of mechanisms to improve the performance of the system to the maximum 38.5Mbytes/second First, the limiting CPU utilizations could be reduced by sharing access to the HIPPI source board by the host CPU and the HIPPI destination board. The sharing would make it unnecessary to interrupt the Sun-4 host every 32Kbytes. However, this sharing is impossible to achieve efficiently without an improved VME interface on the HIPPI boards. Another possibility would be using larger packets to communicate to/from the TMC HIPPI boards. The Ultranet hub architecture cur-

rently limits us to 32Kbyte transfers. The utilization of both the 29K CPUs and the Sun-4 CPU would greatly be reduced by the use of larger packets. This would allow us to scale to much higher bandwidths. To increase the packet size, we plan on replacing the Ultranet with a HIPPI switch network. Using the HIPPI switch network we hope to support transfers at over 70Mbytes/second to a pair of XBus boards.

6.0 References

[AnonA] *Network Operations Manual,* Ultra Network Technologies, Part Number 06-0001-001, Revision A, (1990). Chapter 2: UltraNet Architecture; Chapter 3: UltraNet Hardware.

[AnonB] *HPPI Destination Module (HPPID) Hardware Specification.* Thinking Machine Corp. October 1990.

[AnonC] *HIPPI Source Interface Hardware Register Specification.* Thinking Machine Corp. September 1990.

[ANSI91] *High-Performance Parallel Interface - Framing Protocol (HIPPI-FP),* American National Standard for Information Systems X3T9.3/89-013 Rev 4.2. June 1991.

[Chen93] Peter M. Chen, Edward K. Lee, Ann L. Drapeau, Ken Lutz, Ethan L. Miller, Srinivasan Seshan, Ken Shirriff, David A. Patterson, Randy H. Katz. Performance and Design Evaluation of the RAID-II Storage Server. to appear in *International Parallel Processing Symposium 1993 Workshop on I/O.*

[Chervenak91] Ann L. Chervenak and Randy H. Katz. Performance of a Disk Array Prototype. *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* volume 19, pages 188-197, May 1991.

[Clinger89] Marke Clinger. Very High Speed Network Prototype Development; Task 2.1: Measurement of Effective Transfer Rates. Ultra Network Technologies. October 1989.

[Katz91] Randy H. Katz. High Performance Network and Channel-Based Storage. *Proceedings of the IEEE, Vol 80, No. 8.* pages 1238-1260. August 1992.

[Katz93] Randy H. Katz, Peter M. Chen, Ann L. Drapeau, Edward K. Lee, Ken Lutz, Ethan L. Miller, Srinivasan Seshan, and David A. Patterson. RAID-II: Design and Implementation of a Large Scale Disk Array Controller. *1993 Symposium on Integrated Systems,* 1993. University of California at Berkeley UCB/CSD 92/705.

[Lee92] Edward K. Lee, Peter M. Chen, John H. Hartman, Ann L. Chervenak Drapeau, Ethan L. Miller, Randy H. Katz, Garth A. Gibson, and David A. Patterson. RAID-II: A Scalable Storage Architecture for High-Bandwidth Network File Service. Technical Report UCB/CSD 92/672, University of California at Berkeley, February 1992.

[Patterson88] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *International Conference on Management of Data (SIGMOD),* pages 109-116, June 1988.

[Welch90] Brent B. Welch. Naming, State Management, and User-Level Extensions in the Sprite Distributed File System. University of California at Berkeley UCB/CSD 90/567. April 1993