An Efficient Network Interface for the RAID-II File Server

Srinivasan Seshan

A master's report

1.0 Abstract

Distributed systems in use today depend heavily on network communications between clients and servers. In this report, we describe the design and implementation of the network architecture (hardware, software and protocols) of the RAID-II system. RAID-II is a high speed network file server connected to an UltraNetwork. In order to support high bandwidth network transfers with the RAID-II server, we partitioned the networking software among the various processors in the system. Measurements of the system show that the RAID-II server can sustain 21MB/s of data bandwidth to the Ultranet.

2.0 Introduction

2.1 Problem Description

Over the past decade, we have experienced a major shift from centralized computing using mainframes to a distributed model of computing using workstations connected via high-performance networks. In the traditional mainframe-centered view of computer systems, storage devices are tightly coupled to the computation system. In the newer workstation model of computing, storage is attached to file servers distributed throughout a network. The workstation clients make file requests

to these servers through a message based protocol over a high speed network.

This centralized file storage has several advantages and disadvantages. Users can have access to the same file system on different client machines. Also, a centralized file system simplifies administration. Typical client/server environments have approximately many workstations per file server. Therefore, the speed of the entire system is highly dependant upon the efficiency of the communication between the server and its clients. In this report we explore the design of a network architecture (hardware, software, and protocols) for RAID-II, a high-speed network file server.

2.2 RAID-II Hardware

There are three major sources of overhead in a client/server system: the I/O subsystem of the server, the operating system overhead in both the client and server, and the network communication overhead. [Chen93, Welch90] explore operating system and I/O hardware performance of the RAID-II system. This paper is primarily concerned with network overhead in the file server.

RAID-II ("RAID the Second") is a high-performance network file server developed at the University of California at Berkeley. The system was designed to support both high bandwidth (scientific, engineering, multi-media) applications and low latency applications (UNIX, database). The hardware currently allows a maximum of 144 3.5" disk drives, providing 45GB of storage (150GB+ with state-of-the-art disk drives). RAID-II currently supports clients on an Ultranet network, and in the future will support HIPPI networks and Ethernet.

A significant portion of the RAID-II system is made from commercially available components. Thinking Machines (TMC) provided a board set for the HIPPI channel interface. Interphase provided a single board multiple string VME SCSI controller. Custom hardware, the XBUS board, was developed by the RAID research group to provide an interconnect between multiple VME busses, the HIPPI boards, and an interleaved, multiported memory. A Sun-4 CPU is used to control the various systems and run the file system software. A block diagram of the interconnections is shown in Figure 1.

There are a number of programmable CPU's present in the RAID-II system. The Sun-4 CPU runs the Sprite Operating System. It is respon-

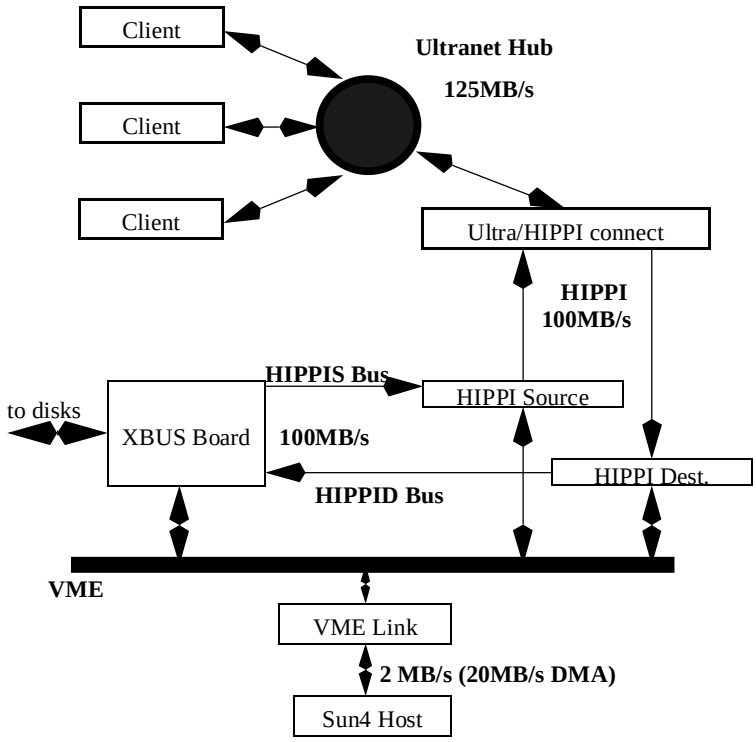


FIGURE 1. RAID-II Block Diagram

sible for controlling the XBUS board, the disk controllers, the HIPPI boards and running the file system code. In addition to the Sun-4, there is a AMD 29000 processor on each TMC HIPPI board. These processors are user programmable and are used to implement a significant portion of the network protocol.

2.3 Report Organization

This report describes goals, constraints and design of the RAID network interface. The report is organized as follows. Section 3.0 describes the RAID-II hardware, target application and performance goals. Section 4.0 provides a detailed description of the network interfaces involved in the system. Both the Ultranet network and the TMC HIPPI board set are covered. Section 5.0 explains the network interface architecture we chose. Section 6.0 presents some network performance measurements taken of the RAID-II system. We present our summary and conclusions in Section 7.0 .

3.0 Raid - II

In this section, we describe sections of the RAID-II file server hardware that had a significant impact on the design of the network interface. We describe the initial performance and functionality goals for the server. We also discuss the interconnection and the programmability of resources in the system. Details about the system are available in [Lee93].

3.1 Goals

The RAID-II file server was designed to support applications typical to high-speed workstations of the future. These applications are composed of a mixture of high bandwidth scientific, engineering and multimedia data and low latency, high transaction rate UNIX-like I/O patterns. The main performance goal for the server was to provide 40MB/s to high bandwidth applications. The file server uses both RAID and LFS technologies to meet our goals for both large and small transfers. Since the full data rate must be made available to clients on the network, an efficient network interface was critical to the success of the project.

3.2 Overall architecture

The RAID-II server is composed of four major parts: the host CPU, the XBUS board, the disk interfaces and the HIPPI interfaces. These components and interconnections are described in greater detail in the following sections. The interconnection of these interfaces is shown in Figure 1.

3.3 VME Link Boards

The remote VME links that connect the host CPU to the other sections of the system are extremely slow. The remote links are implemented using Bit-3 VME link boards. Each link supports about 2MB/s for most applications. In a few select applications, the link board can DMA data at up to 20MB/s. To meet our performance goal of 40MB/s of data transfer, it is necessary that very little data be transferred to the host CPU across the VME link boards.

3.4 HIPPIS Bus and HIPPID Bus

The two HIPPI backplane busses are unidirectional busses that move data between the TMC HIPPI boards and the XBUS board. Both busses are addressless and only the TMC boards can be the bus master. The simple bus protocol used 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 for the transfer before being selected.

Due to physical limitations only 2 XBUS boards may reside on the backplane at one time. The backplane can theoretically support 100MBytes/second, however, each XBUS board supports only 40MBytes/second.

3.5 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 SPEC-Marks. The host is responsible for running most of the code that coordinates the RAID-II server. It is responsible for running the file system code and controlling the drive interfaces, XBUS board and HIPPI boards.

3.6 XBUS Board

The XBUS card implements a 4-by-8 32-bit wide crossbar bus. All transfers use a four port memory as either the source or the destination. Each memory port can sustain transfers of 40MB/s. Two of the remaining eight interfaces provide connections to the HIPPI backplane busses. Since these busses are addressless, the XBUS board must be set up for any transfers across this bus in advance. These ports can sustain 40MB/s of transfer to/from the TMC HIPPI boards. A single port is used for a hardware XOR compute engine. This XOR unit is used to compute parity for the disk array. A VME connection to the host CPU uses another single port. The XBUS board is controlled by a set of registers present on this VME interface. The remaining four ports provide connections to VME busses. These interfaces connect to standard VME SCSI controllers. Each VME interface can sustain 8MB/s. This gives an aggregate bandwidth of 32MB/s to the SCSI disk controllers.

3.7 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. A single Cougar board is capable of handling approximately 7 MB/s of data traffic from two independent SCSI strings. Physical packaging limits each string to 3 disks. The RAID-II system is limited by the current SCSI boards to 28MB/s of disk bandwidth and 24 disks. Future SCSI boards will allow the system to use 72 disk drives and to provide up to 32MB/s per XBUS board.

3.8 HIPPI interface

The HIPPI interface is described in detail in the Section 4.0 .

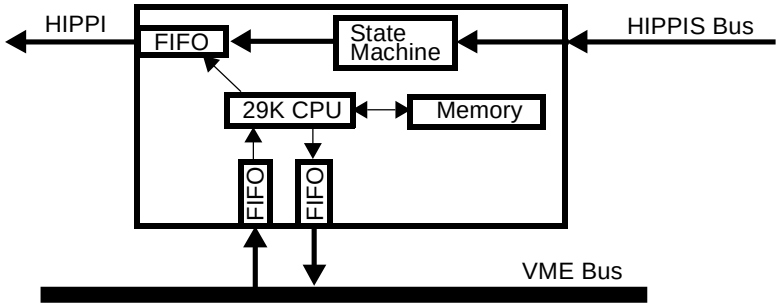
4.0 Network Interfaces

In this section, we describe the network environment of the RAID-II file server. The operation of the TMC HIPPI boards, which provide the network interface for RAID-II, is explained. The UltraNetwork, which provides connectivity to the client machines, is also described in this section.

4.1 TMC HIPPI Boards

The HIPPI interface for RAID was implemented using a two board set built by Thinking Machines Corporation (TMC). The architecture of the boards is shown in Figure 2. Each board contains an interface to a single direction of the HIPPI channel, a unidirectional backplane bus, and a control VME bus. The two backplane busses are addressless and only the HIPPI boards can be bus masters. This requires that the opposite end of any transfer be set up in advance to send or receive data. Each board also contains an AMD 29000 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 as well as run any general purpose code (protocol code). There are some significant differences between the two boards. The individual boards are described in more detail in the next few sections.

- Source Board:



- Destination Board:

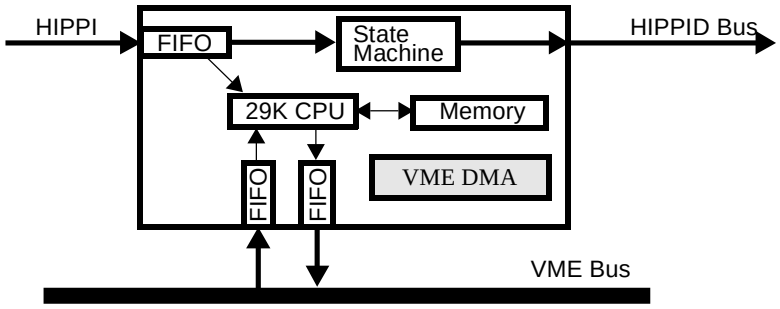


FIGURE 2. TMC HIPPI Board Block Diagram

4.1.1 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, the host CPU must copy data into the input FIFO and from the output FIFO.

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

The Source board's interface to the HIPPI output channel and the input backplane bus is controlled by seven registers. Their function is summarized in Table 2. These registers are only accessible by the 29k processor. The HIPPI data FIFO contains the data to be sent out on the HIPPI channel. This FIFO can be filled by data from the 29k or from the backplane bus. When the source of the data transferred is the 29k, the 29k must force the data to be sent from the data FIFO by writing HIPPI events to the HIPPI event FIFO. This is an extremely slow process and limits the transfer rate of data. When data is being sent from a different board on the backplane, the data is copied by a state machine on the Source board from the backplane to the HIPPI channel. The 29k must initiate this transfer by writing the length of the transfer to the total counter register. The transfer is started by the 29k writing to the HIPPI command register. The completion of the transfer is signaled by a change in the status register. By not involving the 29k in transfers from the backplane, the system can achieve the full link bandwidth (100MB/s).

TABLE 2. HIPPI Source Board HIPPI registers

Register	Description
Data FIFO	Stores data to be sent on HIPPI channel.
Total Counter	Stores number of 64-bit words to receive from backplane.
Event FIFO	Used to perform HIPPI transfers such as sending bursts of data and starting/ending packets.
HIPPI Command	Used to enable and control status of HIPPI channel and backplane
HIPPI Status	Stores current status of HIPPI channel and backplane.
Backplane Target RAM	Stores slot number of boards in backplane to receive data from.
Backplane Counter	Stores number of 64-bit words to receive from each board in the backplane. Used to stripe data from multiple boards.

4.1.2 HIPPI Destination Board

TMC designed the Destination board much after the Source board. Thus, it includes many more features than the Source board. The VME interface includes several new registers and the input and output FIFOs used to communicate with the host. The Destination board VME registers are summarized in Table 3. The Destination board also has VME bus master capabilities. The board can write data from the output

FIFO to any VME location. Similarly, it can read VME locations to the input FIFO.

TABLE 3. 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

The Destination board's interface to the HIPPI in channel and the backplane output is summarized in Table 4. These registers are only accessible by the 29k. Data from the HIPPI channel is automatically placed in the HIPPI data FIFO. Data in the FIFO can either be read out by the 29k or copied by a state machine to the backplane. The reading of data by the 29k occurs at a fraction of the maximum link bandwidth. However, transfers using the state machine can occur at the maximum link bandwidth. The 29k must set up the transfer to the backplane by writing the length of the transfer to the total transfer counter. The transfer is started by writing to the HIPPI command register. The end of the transfer is signaled by a change in the status register.

4.1.3 Original Use

TMC originally built the boards for use with a CM-2. These boards plug into a specially engineered backplane containing three busses. Two of these are simple high speed unidirectional busses, the HPPIS bus and the HPPID bus. The third is a VME bus that is used to control the board set. In a typical TMC CM-2 setup, a single chassis contains the two HIPPI boards, a Sun-4 CPU board (to control the HIPPI boards), and up to 7 IOP controllers that connect to the CM-2. Very little code runs on the 29k processor in the typical setup. Much of the transfer set up and protocol processing is done by the Sun-4 host in the system. The Sun-4 host remains lightly loaded since it has no other computational responsibilities. This allowed TMC to easily debug their network code since it is run-

TABLE 4. **HIPPI Destination Board HIPPI Registers**

Register	Description
Data FIFO	Stores data receive from HIPPI channel.
Total Counter	Stores number of 64-bit words to send to backplane.
Event FIFO	Describes incoming HIPPI transfers.
HIPPI Command	Used to enable and control status of HIPPI channel and backplane
Backplane Status	Stores current status of backplane.
HIPPI Status	Stores current status of HIPPI channel.
Backplane Target RAM	Stores slot number of boards in backplane to send data to.
Backplane Counter	Stores number of 64-bit words to send to each board in the backplane. Used to stripe data across multiple boards.

ning under a standard UNIX system. However, they sacrificed speed to obtain this improved programming environment.

4.2 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.

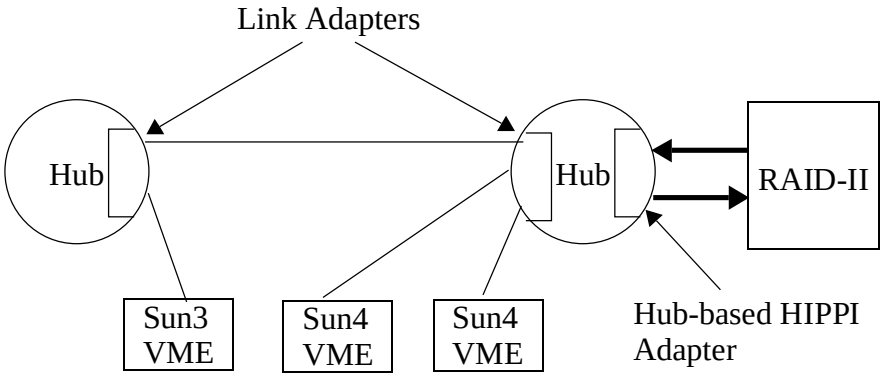


FIGURE 3. **UC Berkeley UltraNetwork Topology**

Hubs are physically connected by serial links capable of transmission rates of 250Mbits/second. Up to 4 links can be used to connect a pair of hubs. Data can be striped across these links to achieve Gbit/second speed. These links terminate in link adapters in the hubs. The link adapt-

ers are also used to connect to machines with Ultranet host adapters. Host adapters are available for machines with industry standard backplanes (for example 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 supercomputers, can connect to the UltraNetwork using standard channel interfaces (for example HIPPI, HSX) attached to a hub-based adapter. This essentially moves the network interface into the hub itself. Much of the UltraNetwork protocol is handled by a 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.

Typical VME Ultranet host adapters provide a maximum of about 8MB/s to the network. On the basis of the RAID-II performance goal of 40MB/s, we decided to use a HIPPI attachment between the drive array and the Ultranet hub.

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 32KB. The DMA word accompanying each transfer describes the contents of the transfers. Analyzing the DMA word provides sufficient information to identify the correct destination in the system 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. Several of the most important request blocks are summarized in Table 5. 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 5. 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()

4.2.1 Original Use

Typical UNIX UltraNetwork configurations use host based adapters. This enables the network interface processor to access the host memory through the use of DMA. This setup allows the host CPU to provide pointers to the necessary data and information to the host adapters onboard protocol processor. The protocol processor then interprets the information using some built-in microcode and sends the appropriate packets to the UltraNetwork hub. This reduces the amount of data copying and computation the host CPU does. Systems that used the hub-based adapters were not similar enough to RAID-II to provide insight for the network software development.

5.0 Software Architecture/ Implementation

In this section, we describe the organization of the networking software of the RAID-II file server. Various aspects of the implementation, such as lines of code and difficulty, are also described.

5.1 Motivation

Before developing the networking code for RAID-II, it was necessary to examine code written by TMC and UltraNetwork for their hardware's original uses.

The first major challenge was that the RAID-II system runs the Sprite Operating System. Sprite was used to improve disk performance through the use of the Log Structured File System (LFS) and other file system improvements unique to the Sprite OS. Both the TMC and Ultranet software were developed for Sun-OS and needed a significant amount of work to port to Sprite.

Thinking Machines had developed their code in a system environment that was very different from RAID-II. In their typical system, a Sun-4 machine was located on the same VME bus as the HIPPI board set. This allowed high bandwidth, low latency communication between the Sun-4 CPU and the 29k CPU's on the HIPPI boards. In addition the Sun-4 CPU was lightly loaded since it had no other computational responsibilities. In the RAID-II system, the Sun-4 CPU was connected to the HIPPI boards through a VME link board. This limited the bandwidth between the Sun-4 VME bus and the TMC VME bus to 2MB/second. The Sun-4 CPU was also responsible for running file system software

and handling the control of the remainder of the RAID-II system (XBUS board, SCSI controllers, etc.). The RAID-II architecture required that the 29k processors on the TMC HIPPI boards run much more sophisticated software This software would have to interpret and control the data transfers without the help of the Sun-4 CPU. This would reduce the communication between the Sun-4 CPU and the remote VME bus. It would also reduce the load on the host CPU.

UltraNetwork's code was written to support a host-based adapter, not a hub-resident adapter. In addition, RAID-II's network interface had some very significant peculiarities. Although accessing host memory was very slow (across a VME link card), the network interface could access memory on the XBUS board via a very fast backplane. Also, unlike most network adapters the TMC HIPPI boards only had very limited DMA capabilities. RAID-II needed additional software that emulates some of the functions present in a host-based adapter. These various factors made it necessary for the host to perform some of the actions normally assigned to the network controller. To keep the load on the host CPU as low as possible, as much of the network header processing as possible was done by the 29k CPU's on the HIPPI board.

Little of the networking code from TMC and UltraNetwork could be used in the RAID-II system. We needed new code that was much more suited to the interconnection and performance of the system.

5.2 Architecture

The RAID-II block diagram [Figure 1] shows that only one path exists between the network and disk that supports the desired bandwidth of 40MB/second. This path passes through the HIPPI boards, XBUS board and the SCSI controllers. Since only the 29k processors lie on this data path, it is necessary that they control the flow of information. For the 29k processors on the HIPPI board to place data in the correct location, they were programmed to understand the UltraNetwork protocol. This division of software is shown in Figure 4. Using the RAID-II architecture and the characteristics of the Ultranet communication protocol, we came up with a example transfer of data that would meet our performance goals. This example is subsection below.

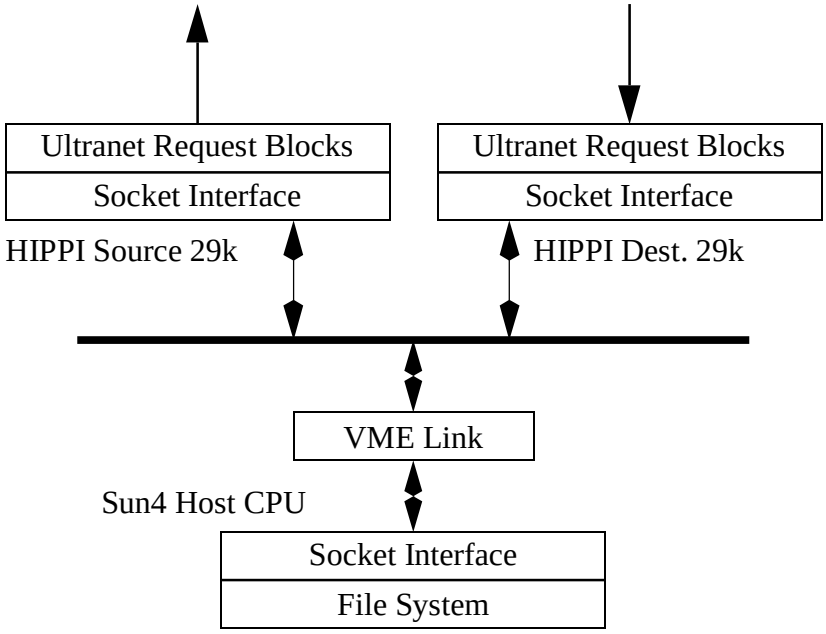


FIGURE 4. Software Architecture Division

5.2.1 Sample Transaction

This section describes a sample transaction that may occur between the RAID server and a client on the Ultranet. In this example the client opens a file, reads 64K bytes and then closes the file. More complicated transfers are possible but this gives a good example of the communications involved and the software needed. The communication needed to open a file is graphically shown in Figure 5.

1. The file server host 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 with a new socket id. This socket id is returned to the file server and the *open()* call completes.
2. The file server host 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 LISTER 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()* for file request information 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 "open file x" and transfers the data to the HIPPI destination board. The destination board uses the unique tag to identify the transfer and DMA's the data into the

correct memory location. 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 open the appropriate file, and create data structures to handle it. The server will construct a message: "Your file open has status x" and will issue a `send()` 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 return status. 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 source board must be set up 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.
5. The file server does a `recv()` for file request information on the connected socket id. This proceeds similarly to the previous `recv()` request. However, the client requests "read 64KBytes at position x".
6. The host will perform the actions necessary to read the appropriate data into an XBUS memory buffer and will issue a `send()` of the data. A pointer to the XBUS buffer and a tag that uniquely identifies the transfer are passed to the destination board. As before, 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 expected data. In this transfer, the Ultranet hub sends a request to the HIPPI destination board to send only the first 32KBytes of the transfer. This is due to buffering limitations in the Ultranet hub. The destination board uses the unique tag to identify the transfer request and determine the portion of the data to be sent. The source board must be set up to transfer the desired data to the Ultranet hub. Theses 32KByte transfers are repeated until the full transfer completes. A completed SEND request block is then sent to the destination board by the Ultranet hub. The destination board sends the request block status to the file server and the `send()` completes.
7. The file server does a `recv()` for file request information on the connected socket id. This proceeds similarly to the previous `recv()` request. However, the client requests "close file x".
8. The host will close the file then issue a `close()` on the socket id. This is accomplished by having the source board send a CLOSE request block to the Ultranet hub. When the hub frees up resources allocated to the connection it returns the completed CLOSE request block to the destination board. The destination board forwards the CLOSE status to the host and the `close()` completes.

5.3 Source Board Code

From the example transfer, it is evident that the HIPPI source board has to be able to send both data and request blocks to the Ultranet hub. The commands to perform these actions are summarized in the following sections. These commands are written into the HIPPI source

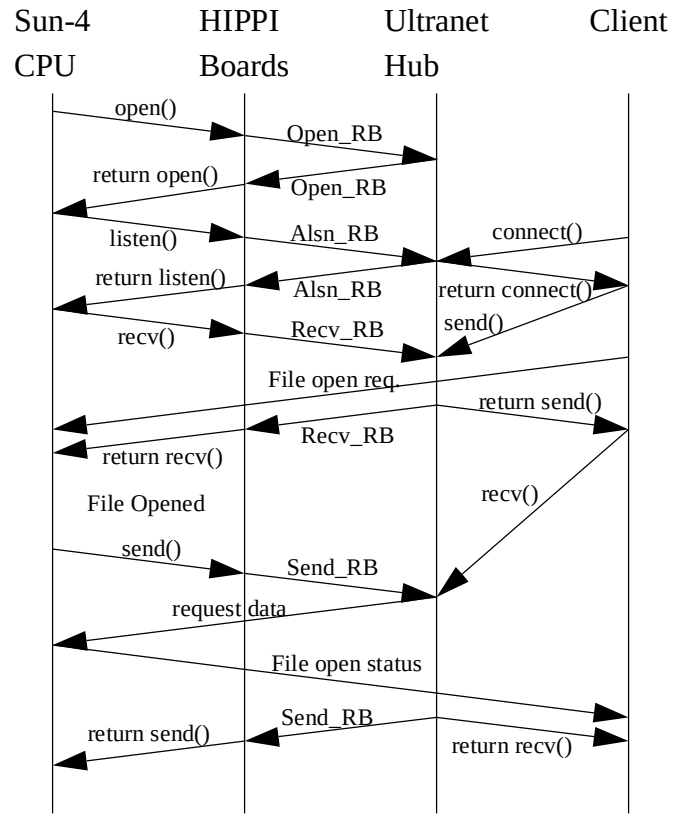


FIGURE 5. Communication between Sun-4 CPU, TMC HIPPI Boards, Ultranet Hub and Ultranet client to open a file

board's VME FIFO. Every command to the source board starts with the following command header:

31. . . .24	23. . . .16	15. . . .8	7. 0
Source Magic		Serial	Opcode

Source magic is a single unique number used to help identify corrupted commands. *Serial* is used to trace the execution of a single command. *Opcode* identifies which command type follows. All parameters for the command follow the command header.

5.3.1 Sending Data

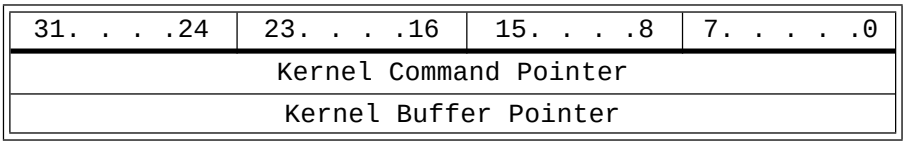
```
SendData(
    int VMEDataSize;
    int XBUSDataSize;
    int VMEData[VMEDataSize];)
```

The above command requests the HIPPI source board to send *VMEData* and *XBUSDataSize* words from the XBUS on the HIPPI channel.

5.3.2 Ultranet Request Blocks

Most of the commands to the Source board send specific UltraNetwork request blocks. Each command copies only the essential parameters of the associated request block to the 29k processor. This reduces the traffic between the Sun-4 and the source board. Each command is summarized in Table 6.

In addition to the needed request block fields the following must also be sent for each Ultranet specific command:



These fields are necessary to identify the return values of the command to the kernel.

TABLE 6. Command Format between 29k and host CPU

Command	Description
`UltraOpen()` short Socket ID; char status;	Sends request to UltraNetwork to open a socket. Socket ID and status are empty parameters.
`UltraListen()` short Socket ID; char status; int Connection Parameters[4]; char remote address[7]; char local address[7]; char port[4];	Sends request to bind Socket ID to port. Then listens for a connection and accepts it. Connection Parameters specify UltraNetwork specific parameters for each connection.
`UltraClose()` short Socket ID; char status;	Sends request to close connection active on socket ID.
`UltraSend()` short Socket ID; char status; int size; char tag;	Sends request to send size bytes on connection associated with Socket ID. Each send request block is given a unique 8 bit tag that identifies it.
`UltraRecv()` short Socket ID; char status; int size; char tag;	Sends request to receive size bytes on connection associated with Socket ID. Each receive request block is given a unique 8 bit tag that identifies it.

5.4 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 VME commands and HIPPI transfers that the destination board must understand are summarized in the following sections.

Every VME command is written to the destination board VME FIFO. These commands must starts with the following command header:

31. . . .24	23. . . .16	15. . . .8	7. 0
Destination Magic		Serial	Opcode

Destination magic is a single unique number used to help identify corrupted commands. *Serial* is used to trace the execution of a single command. *Opcode* identifies which command type follows. All parameters for the command follow the command header.

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 the type of request block or data being sent. 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.

5.4.1 Allocating Buffers

```
ScatterGather(
     short numberElements;
     char tag;
     struct ScatterGatherElements {
           int size;
           int address;
     } elements[numberElements];)
```

This command, sent by the host CPU to the destination board 29k, allocates buffers for the transfer labeled with *tag.*

5.4.2 Ultranet Request Blocks

When the HIPPI destination board receives a completed request block from the Ultranet hub, it must notify the Sun-4 host. The destination board uses the same message format to communicate with the Sun-4 as is used to send request block from the source board. These data formats are summarized in Table 6. Since only the most necessary fields of each request block are sent, traffic between the 29k processors and Sun-4 is reduced. The destination board uses its VME DMA engine to copy the message into the host CPU's main memory. Next, the destination board interrupts the Sun-4 CPU to notify it of the completion of an Ultranet request. The host CPU may then examine the completed request for either returned values or status.

In addition to the request block fields listed in Table 5, the following is also copied to Sun-4 memory for each Ultranet request block:

31. . . .24	23. . . .16	15. . . .8	7. 0
Kernel Command Pointer			
Kernel Buffer Pointer			

These fields are necessary to identify the return values of the command to the kernel.

5.4.3 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. However, 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.

5.4.4 Outgoing 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 received ScatterGather() 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 and its access 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.

5.5 Implementation

The approximate complexity of code running on the TMC HIPPI boards is summarized in Table 7.

TABLE 7. 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 by Ethan Miller for the Sun-4 host CPU to support the UltraNetwork and HIPPI boards.

Table 7 describes how long the different network interface development activities took. No documentation existed for developing code to interface to a hub-based adapter. This interface had to be reverse engineered from the Ultranet UNIX device driver code and experimentation with the Ultranet hub. This proved to be the most time consuming task. Much of the testing of the HIPPI board code was done independent of the Sun-4 CPU code. We defined the interface between the Sun-4 CPU and

TABLE 8. Implementation Time

Activity	% of total time
Reverse engineering Ultranet and Ultranet specific debugging	45%
Writing HIPPI Board Code	25%
Interfacing with Sun-4 OS device driver code	5%
Miscellaneous debugging	25%
TOTALS	100%

the HIPPI boards early in the development process. This early definition of the interface helped reduce the time taken to combine the Sun-4 CPU code and the HIPPI board code. A number of tasks compose the Miscellaneous debugging entry. These tasks include: debugging the HIPPI channel communication code and identifying hardware problems in the system.

6.0 Performance Measurements

The two basic goals of the RAID-II network software were to provide high bandwidth to clients on the UltraNetwork and to make the load on the Sun-4 host CPU as light as possible. In this section, we present measurements of network bandwidth and CPU load of the RAID-II system.

6.1 Reduction of VME Link Traffic

To improve network performance of the RAID-II system, we included only the essential fields of Ultranet request blocks in the messages between the Sun-4 and the HIPPI boards. The “compression” achieved is summarized in Table 9. On the average, messages were reduced in size by 50%. This reduction had two effects. It reduced the load on the server CPU by reducing the number of memory copies. It also lowered the utilization of the slow VME link connecting the server CPU and the TMC HIPPI boards.

6.2 Bandwidth

The UltraNetwork currently installed at UC Berkeley supports three Sun VME workstations. Each Sun (Sun-3 or Sun-4) workstation

TABLE 9. Ultranet Request Block Size

Request Block	Normal Size	Compressed Size
OPEN	11	5
LISTEN	23	14
CLOSE	23	5
SEND	11	6
RECEIVE	11	6

supports a 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. 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. A thousand packets of a fixed size were sent between RAID-II and a client machine. The time to complete these transfers was used to obtain average bandwidth and latency measurements for various 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. The SunOS version used in the measurements with the Cray, SunOS 3.5, performs network transfers approximately 10-15% faster than the version used in measurements with the RAID-II system, SunOS 4.1.2. 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 Sun-4 memory system. When data is being written to 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

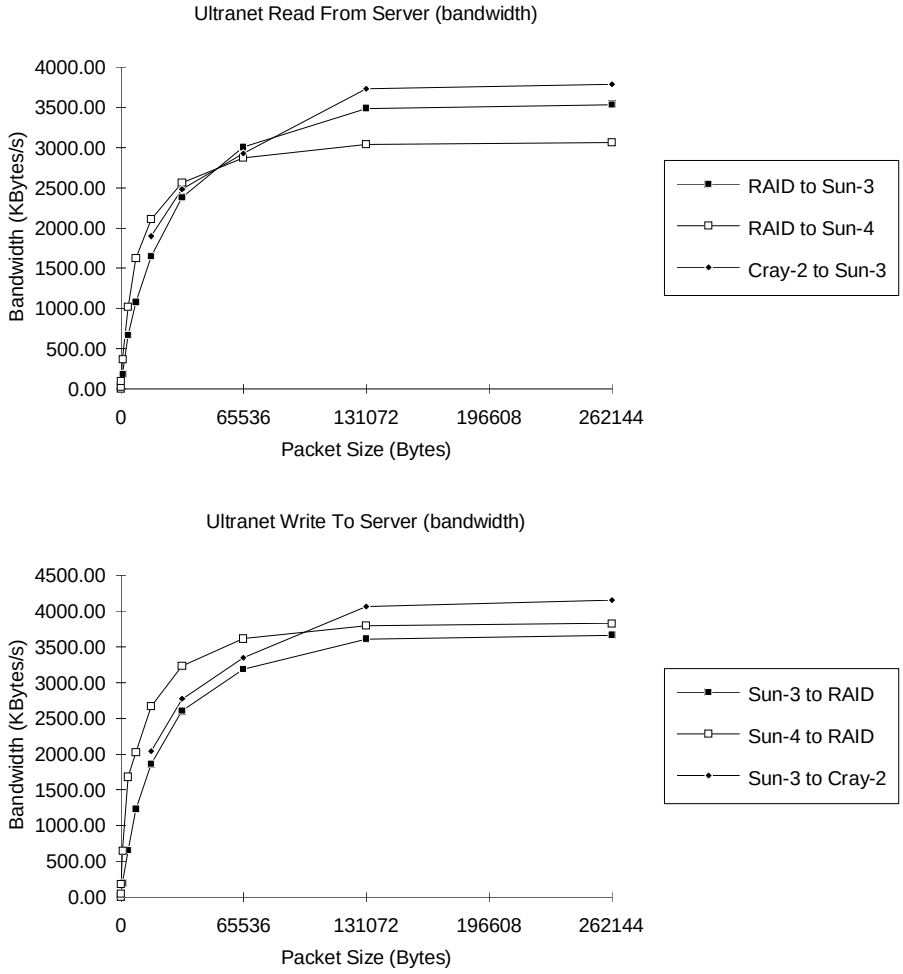


FIGURE 6. Bandwidth vs. Packet Size for transfers between RAID-II and a single client

to RAID-II. These numbers indicate that it is the processing speed of the clients that limits the end-to-end latency of communication.

6.3 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 CPU's on the HIPPI boards.

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

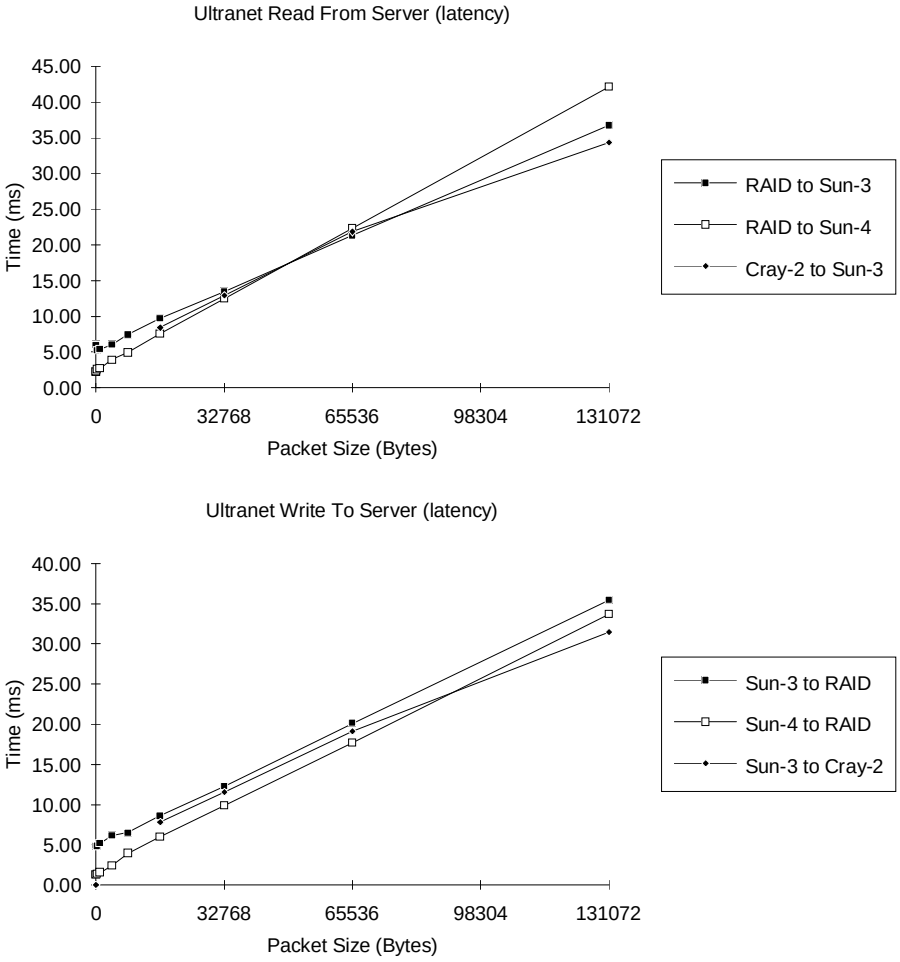


FIGURE 7. Latency vs. Packet Size for transfers between RAID-II and a single client.

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. A packet size of 32Kbytes reduces the host CPU utilization to a minimum of 48%. CPU utilization stays roughly constant for packets larger than 32Kbytes.

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

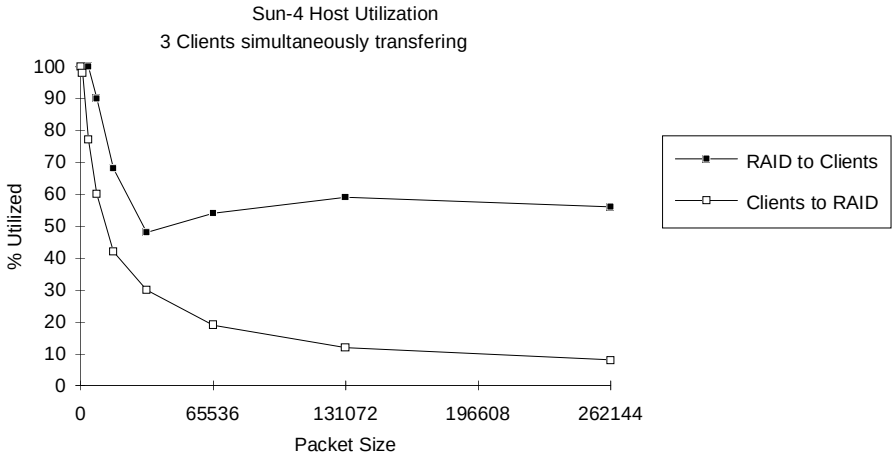


FIGURE 8. Sun-4 Host CPU utilization vs. Packet Size for three clients communicating with RAID-II

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 CPU's on the HIPPI boards depends on the actual bandwidth of the communication. 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 number of bytes transferred. Table 10 shows the utilization of the 29k CPU's with 1,2 and 3 clients transferring. 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 10. Utilization of 29k Processors during Network Transfers

	Read From RAID		Write To RAID	
Bandwidth (MB/s)	**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 CPU's would limit the network to approximately 32Mbytes/second for both reads and writes to RAID-II.

7.0 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/sec 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/sec to the network [Welch90]. These results show that the RAID-II network interface is effective at providing a high bandwidth to clients on the Ultranet. However, although the software design did reduce the load on the host CPU by effectively using the 29K CPU's, 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 up 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 currently limits us to 32Kbyte transfers. The utilization of both the 29K CPU's 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/sec to a pair of XBUS boards.

8.0 Acknowledgments

I would like to thank Professor Randy Katz and the RAID group for their help and support.

9.0 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