RAID-II: Design and Implementation of a Large Scale Disk Array Controller[1]

R. H. Katz, P. M. Chen, A. L. Drapeau, E. K. Lee, K. Lutz, E. L. Miller, S. Seshan, D. A. Patterson
Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

Abstract: We describe the implementation of a large scale disk array controller and subsystem incorporating over 100 high performance 3.5" disk drives. It is designed to provide 40 MB/s sustained performance and 40 GB capacity in three 19" racks. The array controller forms an integral part of a file server that attaches to a Gb/s local area network. The controller implements a high bandwidth interconnect between an interleaved memory, an XOR calculation engine, the network interface (HIPPI), and the disk interfaces (SCSI). The system is now functionally operational, and we are tuning its performance. We review the design decisions, history, and lessons learned from this three year university implementation effort to construct a truly *large scale system assembly.*

1. Introduction

Over the past dozen years, the University VLSI research community has undergone a dramatic evolution. A VLSI system was originally defined as a “complete system on a single integrated circuit chip.” Given the level of circuit complexity at the time, few truly interesting systems could actually be reduced to a single chip. They needed to be surrounded with considerable support circuitry for memory and interfaces to subsystems like a workstation backplane. Initially, this system context was provided by placing the custom VLSI on wire-wrap boards. As the community’s design experience grew, the VLSI “systems” became complex enough to span multiple chips, placed and interconnected on printed circuit boards. Now the systems we design are significantly more complicated, requiring more complex packaging. They span modules, printed circuit boards, and racks. We call such systems *large scale system assemblies*.

VLSI systems building at Berkeley provides an illustration of the community’s evolution. Between 1978 and 1981, Patterson, Séquin, and their students developed the Berkeley RISC chips. These were true single chip systems, with just enough of a surrounding context to run some benchmark test programs. SOAR came next, developed by Patterson and his students between 1981 and 1984. They expanded the system from a single chip to a complete microcomputer wire-wrap board and a run-time Smalltalk system.

The SPUR multiprocessor project, from 1984 to 1987, took the system concept one step further. The processing element was a CMOS chip set: a CPU, an FPU co-processor, and an integrated cache controller/memory manager. In conjunction with the chips, a large printed circuit board was designed for the cache memory and bus interfaces. Substantial amounts of software were also developed: a distributed multiprocessor operating system, Sprite, a C compiler, and a parallel LISP system. Patterson and Katz led the hardware development, Hodges supervised the integrated circuit designs, Ousterhout directed the operating system development, and Hilfinger oversaw the compiler and LISP system implementations.

Our most recent project is RAID-II, a large scale system assembly that implements a high performance disk array and file server. The design and construction of RAID-II began in 1989 under Katz’s direction, and is now reaching its completion. Like much of the research community, we have replaced custom VLSI design with commercially available high speed programmable logic devices. We’ve undertaken significant mechanical design, to support the construction of an assembly able to hold more than 100 small formfactor disk drives and numerous printed circuit boards for disk controllers, network interfaces,

[1]. Research supported by DARPA/NASA Contract NAG2-591 and the California MICRO Program in conjunction with Array Technology Corp., DEC, HP, IBM, Intel, Seagate, StorageTek, Sun, and Thinking Machines.

and their interconnect. Our hardware design effort has focused on building a large, double-sided printed circuit board in surface mount technology that integrates several commercially available subsystems. The distributed file services are being developed within Sprite [Lee 92].

This paper describes the design context, goals, and constraints for RAID-II, a disk array controller that interfaces a file server with a Gb/s local area network. The rest of the paper is organized as follows. In Section 2, we describe the design context for RAID-II, including disk arrays and gigabit networks. Section 3 covers the RAID-II architecture. Section 4 describes the implementation details, including the board design and fabrication of the disk array racks. Section 5 describes the software elements of the controller. The lessons we have learned so far and the project's design history are reviewed in Section 6. Our summary, conclusions, and future plans are given in Section 7.

2. Network-Attached Storage

In this section, we briefly review the context that influenced the design of RAID-I. We describe redundant arrays of inexpensive disks (RAID) and the major interfaces supported by the controller: the SCSI storage device interface and the HIPPI high bandwidth channel interface. We also discuss the Ultranet Gb/s local area network and lessons we learned from our first prototype. More details on the underlying technologies can be found in [Katz 92].

2.1. Redundant Arrays of Inexpensive Disks

RAIDs (*Redundant Arrays of Inexpensive Disks*) are disk system organizations that store redundant data to achieve high availability. Methods that improve availability through data redundancy always sacrifice some storage capacity and write bandwidth. Alternative RAID organizations tradeoff between availability, I/O performance, and the redundancy overhead [Patterson 88]. The organization best suited for high availability and high I/O operation and data rate is the parity array (RAID Level 5). We describe it and the SCSI device interface next.

RAID Level 5: Interleaved Parity

The array is partitioned into independent recovery groups of N data disks each. Parity is computed bit-wise horizontally across the data disks, and is stored on an N+1st parity disk. If a single disk fails, a given bit on the failed drive can be reconstituted from the bits on the surviving disks, by maintaining the appropriate sense of the parity. The redundancy overhead is only 1/(N+1) bit for each data bit.

For each written data bit, the parity bit must also be updated. The old data bits are read, the difference between them and the new bits is computed, the old parity is read, and parity bits associated with changed data bits are complemented. A logical write becomes four physical I/Os. Thus, the array write rate is reduced to 25% of a conventional disk system. The Log Structured File System developed at Berkeley circumvents this problem by treating disk as an append-only medium to which large segments are written. Parity can be computed in advance for these large, stripe-oriented writes [Rosenblum 91].

Small Computer System Interface

SCSI is the storage interface supported by small formfactor disk drives. It views a disk drive as a linear byte stream; its detailed structure in terms of sectors, tracks, and cylinders is not visible. The interface defines a high-level message-based protocol for communications among initiators (masters) and targets (slaves). Initiators manage multiple simultaneous operations. Targets explicitly notify the initiator when they are ready to transmit data or when they need to throttle transfers.

2.2. Gigabit Networks and Diskless Supercomputers

It is now becoming possible to extend the workstation client-server model to higher performance environments, integrating supercomputer, workstations, and storage services on a very high performance network. A key goal of RAID-II is to demonstrate the feasibility of *diskless supercomputers*. The network is rapidly emerging as the "backplane" of high performance systems.

High Performance Parallel Interface

The High Performance Parallel Interface, HIPPI, is a high speed unidirectional point-to-point interface. Two-way communications requires two HIPPI channels, one for commands and write data (the *write channel*) and one for status and read data (the *read channel*). Data is transmitted at a nominal rate of 800 Mb/s.

HIPPI implements a simple data transfer protocol. The source asserts a request signal to gain access to the channel. A connection signal grants it the channel. The source begins the transfer when the destination asserts ready, providing simple flow control.

The minimum unit of data transfer is the *burst* of 1 to 256 words. The destination must be able to accept a full burst if it asserts ready. The burst is sent as a continuous stream of words, one per clock period, and continues as long as the channel burst signal is asserted. When this is unasserted, the sender transmits a CRC checksum for the whole burst.

UltraNetwork

The UltraNetwork is a hub-based Gb/s network. The hubs provide the high speed interconnect for packet routing. They are connected by unidirectional *serial links*, used in pairs. If optical fiber is chosen for the links, data can be transmitted at rates of up to 250 Mb/s and distances to 4 Km. Higher speed is achieved by interleaving transmissions over multiple serial links.

Computers are connected to the network in two different ways: through *host adapters* and *hub-resident adapters*. A host-based adapter resides within a workstation backplane. The adapter contains an on-board microprocessor and performs direct memory accesses, just like any other peripheral controller.

A different approach is needed for supercomputers and the RAID-II controller. These connect to the network through HIPPI interfaces, not standard backplanes. The hub-resident adapters place the network interface to the Ultranet within the hub itself.

2.3. RAID-I Prototype and Experiences

RAID-I, the first RAID prototype, consisted of a SUN 4/280 file server with 128 MB of memory, four Interphase dual SCSI channel host bus adapters, and 32 340 MB 5.25" Wren-IV disk drives (4 drives on each of 8 SCSI channels).

While we had hoped the design would be disk limited, [Chervenak 91] discovered numerous performance bottlenecks. The most serious is the server's memory system, which limited application throughput to only 2.3 MB/s. I/O operations caused excessive memory-to-memory copies and cache flushes. The array did better on small random reads, achieving nearly 300 per second before the server becomes CPU-limited. Other bottlenecks include the VME backplane (about 15 MB/s), bandwidth on the disk controller (4 MB/s), and overheads associated with the SCSI protocol. Nevertheless, RAID-I provided an indispensable test-bed for experimenting with algorithms for data striping, reconstruction, and hot spare management.

3. RAID-II Architecture

3.1. Design Goals and Constraints

RAID-II is a HIPPI-attached high performance file server that interfaces a SCSI-based disk array to the Ultranet. It packages over 100 IBM 3.5" disk drives (320 MB capacity — 1.2 GB drives are now available in an identical formfactor) in the space of three 19" racks, providing approximately 40 GB of storage. While the Ultranet provides high bandwidth, its latency is actually worse than slower speed networks. RAID-II also supports Ethernet for small transfers where network latency dominates service time.

To minimize the design effort, we used commercially available components when possible. Thinking Machines (TMC) provided a board set for the HIPPI channel interface. Array Technology (ATC) provided a VME-based multiple SCSI string board. We designed the controller to provide very high bandwidth between the network (HIPPI), disk (SCSI), and memory interfaces.

The major hardware design is the *XBUS card*, a crossbar that connects the HIPPI boards, multiple VME busses, and an interleaved, multiported semiconductor memory. It provides the high bandwidth data-

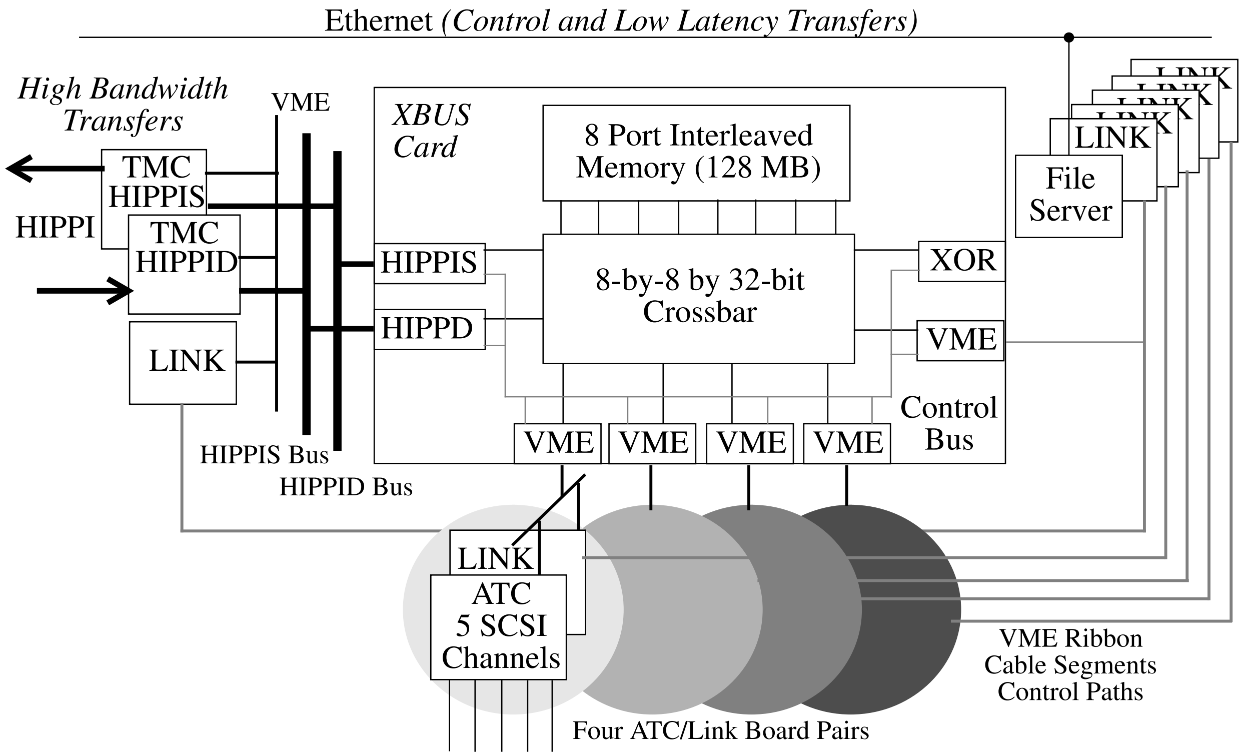

Figure 1: *RAID-II Organization as Originally Designed*
A high bandwidth crossbar interconnection ties the network interface (HIPPI) to the disk controllers (ATC) via a multiported memory system. Hardware to perform the parity calculation is associated with the memory system. An internal control bus provides access to the crossbar ports, while external point-to-point VME links provide control paths to the surrounding SCSI and HIPPI interface boards. Due to printed circuit board manufacturing difficulties, the actual prototype was scaled down to a 4-by-8 crossbar with 32 MB of memory.

path between the network and the disks. It is controlled by an external file server through a memory-mapped control register interface. A block diagram for the controller is shown in Figure 1.

3.2. Controller Architecture

Controller Overview

The XBUS card implements an 8-by-8 32-bit wide crossbar bus. All crossbar transfers involve the on-board memory as either the source or the destination of the transfer. The ports are designed to burst transfer at 50 MB/s, and sustain transfers of 40 MB/s. The crossbar provides an aggregate bandwidth of 400 MB/s.

The controller memory is allocated eight of the crossbar ports. Data is interleaved across the eight banks in 32-word interleave units. Although the crossbar is designed to move large blocks to/from memory and the network and disk interfaces, it is still possible to access a single word when necessary. The external file server can access the on-board memory through the XBUS card's VME control interface.

Two of the remaining eight ports are dedicated as interfaces to the TMC I/O bus (HIPPIS/HIPPID busses). The HIPPI board set also interfaces to this bus. Since these XBUS ports are unidirectional, the controller is limited to a sustained transfer to/from the network of 40 MB/s.

Four additional ports are used to connect the XBUS board to four single-board multi-string disk controllers via the industry standard VME bus. Because of the physical packaging of the array, 15 to 30 disks can be attached to each disk controller, in five rows of three to six disks each. Thus, 60 to 120 disk drives can be connected to each XBUS card.

Of the remaining two ports, one is dedicated for special hardware to compute the parity for the disk array. The last port links the XBUS board to the external file server. It provides access to the on-board memory as well as the board's control registers (through the board's control bus). This makes it possible for file server software, running off of the controller, to access network headers and file meta-data in the controller cache.

XBUS Crossbar

The XBUS is a synchronous multiplexed (address/data) crossbar-based interconnect that uses a centralized strict priority-based arbitration scheme. All paths to memory can be reconfigured on a cycle-by-cycle basis. Each of the eight 32-bit XBUS ports operates at a cycle time of 80 ns.

The XBUS supports reads and write transactions. During each XBUS transaction, 1 to 16 words are transferred over the interconnect. Each transaction consists of an arbitration phase, an address phase, and a data phase. If there is no contention for memory, the arbitration and address phases each take a single cycle; data is then transferred at the rate of one word per cycle. The memory may arbitrarily insert wait cycles during the address and data cycles to compensate for DRAM access latencies and refreshes. The shortest XBUS transaction is a single word write, which takes three cycles, one each for the arbitration, address, and data phases.

While contention for memory modules is a concern, actual contention is infrequent. Most XBUS ports perform large sequential accesses. When they do conflict, the loser of the arbitration deterministically follows the winner around the memory modules, avoiding further conflicts.Each XBUS port buffers 1 KB of data to/from the XBUS to even out fluctuations.

Before deciding upon the XBUS, we carefully considered using a single wide bus-based interconnect. Its main attraction was conceptual simplicity. However, because we decided to limit the design to TTL/CMOS technology, we were limited to a bus cycle time of approximately 80 ns. To sustain 40 MB/s over each of the two HIPPI ports (40 MB/s of reads and 40 MB/s of writes), we need a bus bandwidth of at least 200 MB/s: 80 MB/s for the reads (40 MB/s into and 40 MB/s out of memory) and 120 MB/s for writes (same as for reads plus 40 MB/s to compute the parity). Since we cannot realistically expect to achieve a bus utilization greater than 70-80 percent, this implied that the bus would have to be 256-bits wide with a peak bandwidth of 400 MB/s.

Unfortunately, the number of FIFO and transceiver chips required to implement such a wide bus port is huge. While we could have used a small number of time-multiplexed 256-bit ports, interfaced to narrower 32-bit and 64-bit busses, the result would have been a more complex system. Eventually, the 256-bit bus was abandoned in favor of a crossbar-based interconnection scheme, the XBUS.

Processing

It may seem strange that there is no processor within the XBUS card. Actually, the configuration of Figure 1 contains no less than seven microprocessors: one in each of the HIPPI interface boards, one in each of the ATC boards, and one in the file server. The processors within the HIPPI boards handle some of the network processing normally performed within the server. The processors within the disk interfaces manage the low level details of managing the SCSI interfaces. The file server CPU must do most of the conventional file system processing. Since it is executing file server code, the file server needs access only to the file system meta-data, not user data. This makes its possible to locate the file server cache within the XBUS card, close to the network and disk interfaces.

Performance and Scaling Strategy

Since a single XBUS card is limited to 40 MByte/second, our strategy for scaling is to interleave data transfers across multiple XBUS card. These can share a common HIPPI interface through the TMC I/O busses. Two XBUS boards can sustain 80 MB/s, more fully utilizing the available bandwidth of the server's HIPPI interface.

This architecture performs well for large data transfers requiring high bandwidth. But it is not the best for small transfers, where latency dominates performance more than transfer bandwidth.Thus, the Sprite group has organized the server software to accept remote file system requests over a conventional network, like Ethernet, as well as the Ultranet.

4. RAID-II Implementation

4.1. Disk Array Subsystem

Our goal was to create an experimental prototype that could be built within the project schedule, yet prove serviceable for the duration of the research project and beyond. This led us to several critical packaging decisions. Two disk packaging orientations commonly found in disk arrays are the “wall of disks,” in which the drives are mounted in a single vertical plane within a 19" chassis, and the horizontal “shelves of disks.” We chose the latter scheme, illustrated in Figure 2.

Our main motivation was to take make effective use of the depth of a standard 19" rack. A wall of disks does this only to the extent that the drive housing has depth in its longest dimension. As drive form-factors shrink to 3.5" and below, this dimension also shrinks. Less of the rack depth is used in a wall of disks. For our design, the shelf approach give us a high capacity per square foot. Based on the highest currently available capacity 3.5" drives, our shelf design yields a capacity of 72 1.2 GB drives in a 5 ft. 19" chassis. The total storage is 86.4 GB, an efficiency of 2.34 GB/ft^3.

We accomplished power distribution within the RAID-II chassis with embedded power supply shelves (see the disk racks in Figure 3). Each power supply shelf serves four drive shelves, two above and two below. This symmetric arrangement minimizes the need to cable high current, low voltage power from the supply to the drive. To provide a measure of fault isolation, each drive shelf is powered by an independent AC-DC power converter, so a power supply shelf holds four power supplies. Each of these supplies is independently fused. If one should fail, it cannot affect its neighbors via the AC distribution system. Similarly, a power supply drives four strings on a disk shelf. Each of these is independently fused and switched, to minimize fault propagation in the DC distribution system. Finally each drive is individually fused, further isolating power faults, thus protecting the system from the failure of a single drive.

A secondary goal of the subsystem is to support the servicing of the disk array when a drive fails. To allow users to insert and remove drives from the array, without tools or special knowledge, we made the disk drive the basic field replacable unit. We package the drives in a housing that allows simple insertion and removal from the array. The SCSI bus connection, power connection, SCSI address, and a spindle sync signal are connected from the drive to two connectors mounted on our drive housing assembly. This sub-

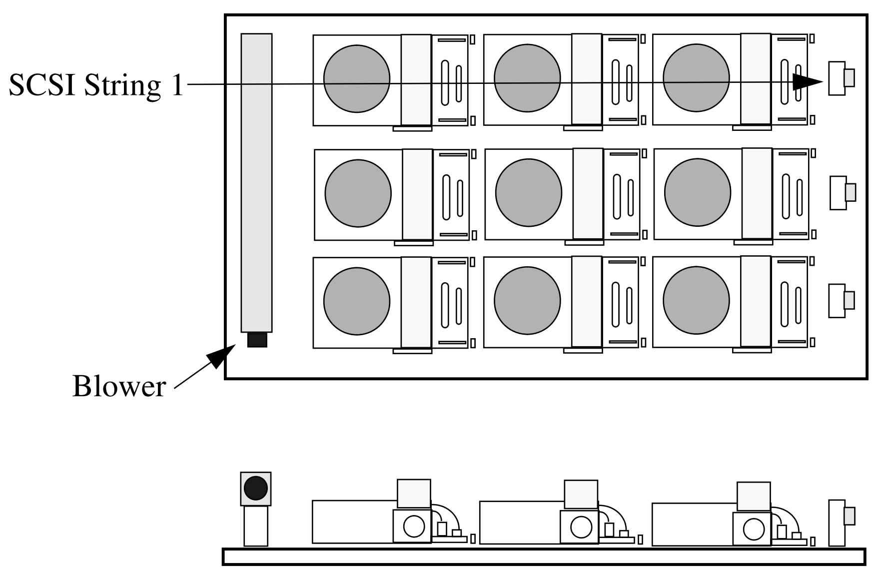

Figure 2: *RAID-II Disk Shelves*
The figure show the disk shelves used in the RAID-II prototype. Drives are placed flat in a 3-by-3 orientation on the shelf. This is in contrast to the “wall of disks,” in which drives are placed on one face of the rack. For 3.5" disk drives, shelves yield a higher storage capacity per cubic foot than the disk wall.

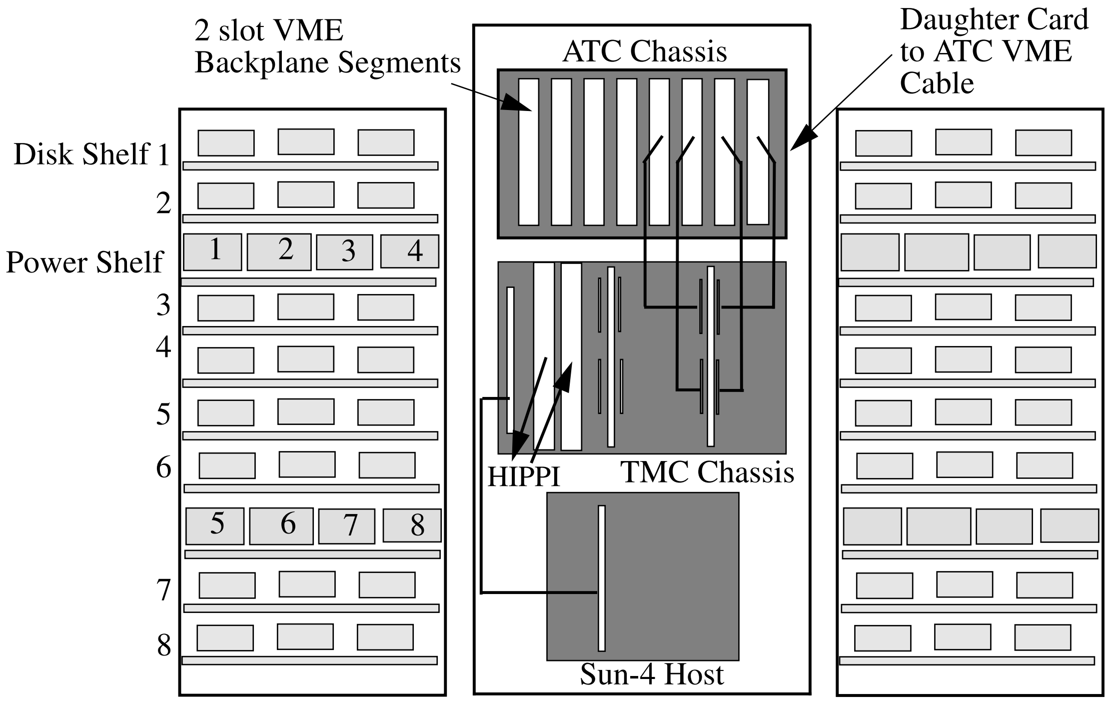

Figure 3: *RAID-II Hardware Racks*

The RAID-II hardware rack contains a SUN-4/280 host at the bottom, a TMC chassis in the middle, and a VME backplane at the top. The TMC chassis holds the HIPPI interface board set, RAID-II XBUS card(s), and a VME-VME link board to allow the host to control the boards in the chassis. The VME backplane is partitioned into independent 2-slot segments for interfacing the disk drives.

plate is designed to float to ensure smooth engagement with the mating connectors on the shelf SCSI backplane. The backplane provides geographic addressing for the drive

4.2. XBUS Card

Physical Design

The XBUS card is the single most complex printed circuit board every designed at Berkeley. It makes extensive use of surface mount technology. We chose surface mount to achieve high component density and to minimize interconnect lengths, especially in the crossbar. To further increase density, we partitioned the design into a main board and four (identical) daughter cards (see Figure 4). The daughter cards interface between the crossbar and the VME interfaces necessary to interface to the ATC string boards. This has the advantage of using the vertical space above and below the main board, In addition, it simplifies the layout task, since the daughter card physical design is done once and used four times.

We designed the boards using RACAL's printed circuit board software. While the design of the daughter card proceeded fairly smoothly using 8 mil lines and spaces, the complexity of the XBUS card eventually exceeded RACAL's ability to handle it. We were forced to wait one month for beta releases to fix the software limitations.

Table 1 summarizes some critical statistics of the two boards we designed, the XBUS and the daughter cards, and compares these with our previous "most complicated board," the SPUR CPU board. In addition, we show the statistics for the original 8-Port version of the XBUS card, which could not be successfully constructed by our board fabricator because of poor yields on blind vias. Blind vias make connections between internal layers without taking space on the top and bottom routing layers. Without them, we were forced to reduce the complexity of the board, to reduce the pins/in^2. Nevertheless, the 4-port board still has dramatically increased pin densities and number of holes on the board. This is remains a good measure of the evolving complexity of the board-level systems the community is now able to design.

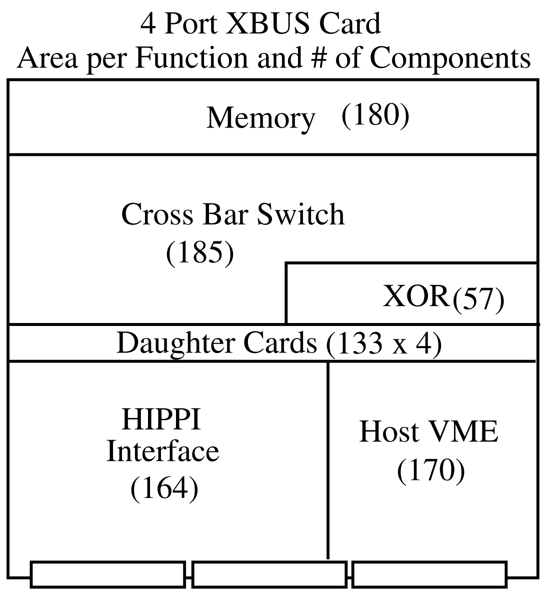

Figure 4: *XBUS Card Layout*
The major subsystems on the board are the interleaved memory system, the crossbar switch, the XOR engine, the daughter card interfaces, the HIPPI source and destination interfaces, and the host VME control interface.

Table 1: IOC PRINTED CIRCUIT BOARDS

Board	XBUS (4-Port)	XBUS (8-Port)	Daughter Card	SPUR CPU Board
Technology	Double Sided SMT/Thru Hole	Double Sided SMT/Blind Vias	SMT/Thru Hole	All Thru Hole
Size, inches	18.4 x 18.3	18.4 x 18.3	6.5 x 8.2	15.7 x 14.44
Layer Count	18 layers, 14 routing	20 layers, 18 routing	10 layers, 8 routing	10 layers, 8 routing
Line/Space (mils)	6/6	6/6	8/8	8/8
Pin Density pins/sq in	54.7	74.5	47	45
# Nets/Connections	1789/13,203	2147/19,171	162/1211	1472/7041
Total Holes	16,429	23,116	2556	12,702
Via Size	0.016	0.016	0.016	0.025

XBUS Crossbar Implementation

Initially, we considered implementing the XBUS with either commercially available crossbar components, Xilinx programmable gate-arrays, or custom VLSI. We ruled out commercial crossbars because of their long configuration latencies. These chips were designed to support message passing in multiprocessor systems rather than much lower latency memory accesses.

While well-suited for implementing complex control units, the Xilinx gate arrays are inappropriate for highly interconnected datapaths with guaranteed propagation delays. The high speed, high pin-out Xilinx components also very expensive.

While we can fabricate custom VLSI chips at nominal cost through MOSIS, we decided against a custom design. This would have increased our risk, because of the long design time and the difficulty of making design changes late in the project.

The crossbar-based memory system meets our requirement for high aggregate memory bandwidth while using relatively inexpensive 32-bit ports. Note, however, that a single port cannot utilize the full 400 MB/s of memory bandwidth; it is limited to only 40 MB/s. This is not a serious restriction because only the HIPPI ports can sustain more than 40 MB/s.

While the XBUS ports themselves are inexpensive, the crossbar itself uses a large number of highly connected components, which complicates the printed circuit board implementation. The XBUS datapath is implemented using 196 16-bit transceivers, while the control unit uses eight 3-to-8 decoders, eight 20R8 PAL's and eight 8-bit registers. By using surface mount technology, we implemented the crossbar in 120 square inches or approximately 20% of the RAID controller's total board area.

Memory Subsystem

The memory system of RAID-II must support high bandwidth and large capacity for buffering multiple HIPPI requests. To accomplish this, we chose to use 4 Mb DRAMs supplied by IBM (16 Mb DRAMs have only recently become available) and to interleave across eight memory banks. Each memory bank is independently accessed via the XBUS. Memory is interleaved on a fine-grain basis across these banks, 16 words at a time. We chose fine-grain interleaving to smooth contention between banks. In the ideal case, multiple large memory accesses will fall in lock-step, with all accesses orderly proceeding from one bank to the next.

We performed memory refresh using "CAS before RAS," where an internal row address is used on each memory chip. Refresh is only performed between XBUS accesses. Normal read and write operations determine the memory's power budget because all memory banks can be active simultaneously. Thus, refreshing all memory banks at the same time does not raise the maximum power requirement.

VME Interfaces

The XBUS card implements five VME ports: a server control port and four ATC VME ports. The server uses the VME control port to read and write the XBUS card's memory and control registers. The other VME ports provide connectivity between the ATC disk controllers and the XBUS memory. The five ports are functionally identical, except that the server port contains additional logic to read and write control registers on the XBUS card. These include (1) a readable, writable board reset register, (2) a read-only status register, and (3) registers on each VME port for recording the VME address and data during parity errors.

Each port has one interface to the VME bus and another to the XBUS memory. We use three different clock strategies. First, the interface to the XBUS memory runs at the same clock rate as the other crossbar ports. Second, the VME interface logic runs at twice the XBUS clock rate, to allow for efficient handshaking of data across the VME bus. Lastly, the VME bus itself is asynchronous. To interact with the bus interface logic, several VME control signals must be synchronized.

Synchronous FIFOs with independent read and write clocks form the interface between the normal speed and double speed clock regions in the VME ports. Communication from the VME bus interface to the XBUS interface uses two control FIFOs and one address/data FIFO. The opposite direction uses one data FIFO.

Our VME interface implementation uses a commercially available PAL chip set that provides most of the logic needed to implement a VME slave module and to generate VME interrupts. We also used single chip parity transceivers to generate and check parity between the XBUS memory and the VME bus.

HIPPID, HIPPIS Interfaces

All DMA engines are controlled from the file server through control register interfaces. These are implemented using synchronous FIFOs. The FIFO allows the server to prepare multiple buffers before data actually needs to go to/from the HIPPI. Unfortunately, the FIFO also prevents random access to the control registers. This ability would have proven useful for the HIPPID. For example, if the server sets up a DMA buffer before the HIPPI packet actually arrives (to shorten latency), the server would not know a priori the size of the HIPPI packet. As a result, a DMA buffer would likely be left only partially full. We added a special bit in the reset register to flush the last partial buffer.

We also used FIFOs to interface between the clock worlds of the XBUS and the HIPPI boards. These also proved useful as speed-matching buffers. Each XBUS port sustains a maximum of 40 MB/s, but a HIPPI board can burst at 100 MB/s. We can generate or receive a 32 KB series of HIPPI packets at full HIPPI speeds by using 32 KB of FIFOs.

Interfacing to the TMC I/O busses posed little design difficulty. Both buses are without addresses, and transfer a data word every 80 ns. However, control signals are driven at 40 ns intervals. We decided to use very fast (7 ns) PALs clocked on the same edge as our control signal assert clock. With 7 ns PALs and fast bus drivers, the propagation delay was an acceptable 11 ns. Had the PALs been slower, we would have needed more logic to pipeline the control signals, further complicating the design.

XOR Engine

The XOR DMA engine is also implemented with FIFOs. Multiple XOR operations can be queued up on the XBUS board. The XOR DMA engine can XOR up to 4095 source buffers and move it into one result buffer. As degenerate cases, the XOR makes it possible to zero memory quickly or to perform quick memory copies.

Hardware Complexity

The complexity of the various state machines within the different portions of the XBUS card are described in Table 2. We made extensive use of conventional programmable logic. In general, the state machines are

Table 2: State Machine Summary

Subsystem		# of States	# of PALs	Components Used
Memory Module		9	2	22V10
HIPPD XBUS Port		9	2	22V10
HIPPIS XBUS Port		10	2	22V10
HIPPID I/O Bus		5	1	16R6
HIPPIS I/O Bus		10	1	16R8
XOR XBUS Port	DMA	10	2	22V10
	XOR	12	2	22V10
VME XBUS Port	Parity	4	1	22V10
	VME Control	9	2	22V10
	XBUS Control	15	3	22V10
VME Control Port	Reg. Decode	5	1	22V10

fairly simple, and easily fit within a small number of high speed PALs. However, the VME interfaces proved to be the most complex state machines. Recall that the six PAL implementation must be replicated for each of the five VME interfaces on the board.

A summary of the hardware implementation level of effort is given in Table 3. We have broken down the subsystems into the memory, HIPPI interfaces, XOR engine, VME interface, XBUS crossbar, system design, and mechanical/RAID chassis design. System design consists of time spent on high level design decisions, like partitioning and clocking of the major subsystems, and thus has no schematic, simulation, and debug elements. The mechanical and chassis design effort includes all activities related to the design and construction of the physical design of the RAID system. This involves the electrical design of

Table 3: Hardware System Complexity

Section	Design	Schematic Capture	Simulation and Debug	Hardware Debug	Total
Architectural Design	960				960
Mechanical/Chassis Design	1120	907	160		2187
Memory	100	60	80	160	400
HIPPI Destination	40	50	50	60	200
HIPPI Source	40	50	70	60	220
XOR	30	20	20	10	80
VME Interface	400	150	300	280	1130
XBUS	80	20	20	10	130
Total	1810	1257	700	580	5307

the disk shelves, including a SCSI backplane and power isolation logic, and the mechanical design of the disk shelves. In addition, we have lumped the implementation of the daughter card into this category.

Of the individual hardware subsystems, the most complex by far was the VME interfaces. Its complexity comes from the need to implement efficient interfaces to the VME and XBUS busses. Our implementation supports full VME functionality while optimizing single word and block transfers. The XBUS memory interface is particularly complex. The basic problem is that we cannot know in advance how much data will be accessed in a VME block mode operation. Because the XBUS protocol requires that the bus request line must be deasserted once cycle before the end of a transaction, we are forced to read ahead in order to predict the end of a transfer.

The table does not include approximately 1000 hours of design time invested in the wide bus interconnection strategy that we later abandoned, 1800 CPU hours of logic simulation, and 2000 CPU hours of printed circuit board routing time. The latter two figures were run in a distributed manner over several workstations, and do not represent elapsed time. It is interesting to note that the routing time of our scaled-down 4-port version of the XBUS card was only 60 hours of CPU time in comparison.

5. RAID Controller Software

5.1. Software Overview

Because of the high communications latency between the server CPU and the network interfaces, the software for RAID-II is partitioned between the server and the processors embedded in the HIPPI interfaces. The critical performance issue is interrupt handling, especially since low latency for HIPPI response is essential. Moving software functionality to the HIPPI boards allows the server to field fewer interrupts per packet.

We partitioned the driver software according to the piece of hardware being controlled or functionality it provided, as shown in Figure 5. The RAID driver maps logical blocks within a multi-disk RAID into logical blocks on each individual disk, using the XOR engine (via the XBUS driver) to do parity calculations and reconstruction. The XBUS driver manages the XOR hardware directly, hiding the hardware interface details from the RAID driver. Similarly, the ATC driver provides the RAID driver with a device-independent view of the disks. The VME link driver performs two functions. First, it initializes the VME link boards, and provides access to individual memory locations on the VME for debugging purposes. Second, it runs the DMA engine on the link boards to give higher-level software a relatively fast (8 MB/s) data

Figure 5: *Interconnections Between RAID-II Software Modules*
This diagram shows the communications paths between the various software modules. Several of these do not run on the file server host; their control and data goes over the VME backplane and the VME link boards.

path between the XBUS board and the server. However, the link driver does not orchestrate single-word transfers or interrupts over the VME links; these are done transparently in hardware. The HIPPI driver works with the HIPPI-D and HIPPI-S board code to provide high-level network software with Ultranet virtual circuits over the HIPPI interface.

This separation of functionality has several advantages. First, the device drivers could be debugged as the hardware became available, allowing us to bring up the software in parallel with the hardware. Second, the hardware interface was isolated from the high-level code, which managed network protocols and the file system. Thus, the high-level code was tested even before the hardware was ready, using lower-performance driver stubs to test functionality. For example, we tested the high-level networking code using the low-performance VME backplane rather than the high-bandwidth XBUS connection. Since the software interface to the network was the same for both, the high-level functionality was debugged before the XBUS board was completed.

5.2. The RAID Device Driver

The RAID device driver implements the logical abstraction of RAID Level 5 disk arrays through a Sprite block device driver. In a sense, the RAID driver is a “meta” device driver that maps logical I/O requests to physical I/O requests that are processed by lower level device drivers it manipulates.

The RAID driver stripes I/O request over multiple disks. This improves performance by increasing data transfer rates and providing automatic load balancing. It also maintains a parity checksum for the data stored on the disks so that in the event of a disk failure, data is not lost. In the event of a disk failure, it reconstructs a failed disk to a spare. By using the parity checksum, the data on the failed disk is made available while the data is being reconstructed to a spare disk.

Each parity block in a RAID Level 5 disk array is associated with several logical blocks of storage. A system crash during the update of a given block can make inconsistent the parity for other blocks which share the same parity block. The consistency of parity blocks must always be maintained.

After a system crash,it is difficult to know the state of the array. Either all parity must be regenerated or a logging mechanism must be used to identify just those parity blocks that may be inconsistent at the time of the crash. It is dangerous to operate a RAID Level 5 disk array with a failed disk, since a system crash can make the parity inconsistent and make it impossible to reconstruct the data on the failed disk.

For the above reasons, most implementations of RAID Level 5 disk arrays perform some form of status logging to stable storage. In our case, we use an extra disk as this log. Careful tuning has made the logging overhead negligible. In designing and developing the RAID device driver, we found that properly dealing with the various failure modes of disk arrays is by far the most difficult task.

Figure 6: *Read Processing*
1. Read request arrives at the HIPPID board.
2. HIPPID board copies the message into the XBUS board and interrupts the Sun-4
3. HIPPI driver calls back the high-level code to service the request.
4. High-level code calls the RAID driver to read blocks from disk into XBUS memory.
5. RAID driver maps calls ATC driver (possibly multiple times) to read the desired blocks from disk into XBUS board.
6. RAID driver calls back high-level code when done.
7-8. Reply header is passed down to the HIPPIS board.
9. HIPPIS board sends out header followed by data from XBUS board.

Figure 7: *Write Processing*
For the write diagram, here are the steps:
1. Write request arrives at the HIPPID board. Data and headers are put into preallocated buffers in XBUS board memory.
2. HIPPI-D board notifies both the HIPPI driver and the HIPPIS board.
3a. HIPPI-S board acknowledges receipt of the write.
3. HIPPI driver calls back high-level code.
4. High-level code tells RAID driver to write certain buffers in the XBUS memory.
5. RAID driver coordinates ATC driver and XBUS driver to write data and parity. Parity is calculated on the XBUS board using the XOR engine.
6. RAID driver notifies high-level software when the write is complete so the buffers may be recycled.

5.3. Software Control Flow

All network information, whether read or written, goes over an Ultranet virtual circuit. The first step in any file operation is to establish such a circuit (if one does not already exist for that file). Most of this setup is handled by Ultranet hardware and the HIPPI boards; higher-level software receives a circuit identifier when the setup completes. Next, the client sends the file name to the server over the virtual circuit. The connection is now ready to process file reads and writes. Data can move in two directions — disk to network (client read) or network to disk (client write), as shown in Figure 6 and Figure 7 respectively. To be brief, we have included the detailed operational steps in the figure captions.

5.4. Software Complexity

Table 4 summarizes the number of lines of code and estimated level of effort associated with each major software component of the design. Most of the driver software was rather straightforward. However, two elements were very complex: the RAID driver and the software associated with controlling the HIPPI-based network interfaces. The HIPPI driver communicates with the TMC HIPPI boards over a VME-link. Unlike the other drivers, its interface to higher level software is based on network-oriented sockets rather than the standard device driver interface. The embedded code on the HIPPIS and HIPPID manages connections over the Ultranet. A remarkably small amount of this code is written in assembly language.

6. Lessons Learned and Design History

6.1. Lessons Learned

We still have much to learn about the effectiveness of our design and implementation for RAID-II that will only be discovered through long term use. Nevertheless, we have learned several important lessons,.

First, logic simulation tools are very mature; they work and they work well. We were able to write reasonably complete VHDL models for complex OEM VLSI chips, such as our FIFOs and the VME PAL

Table 4: System Software Complexity

System Component	Subsystem	Lines of Code	Estimated Person Hours
VME Link Driver		1916	200
XBUS Board Driver		1560	100
HIPPI Drivers		6634	400
RAID Driver		7086	500
ATC Driver		2576	200
Embedded HIPPIS and HIPPID	29000 Assembly Code	700	500
	High Level Code	8500	
Totals		28,972	1900

set, and to use these effectively within our simulation environment. We had very few logic bugs in the actual implementation.

Second, despite our long history of system building, we continue to underestimate the time it takes to complete logic simulation. A good rule of thumb is that we spend as much time in simulation for verification and debugging as we do for logic design and capture. In the case of this project, it was about six months of logic design and six months of simulation.

Third, physical design tools, even industry standard tools, are inadequate for complex, two-sided, surface mount boards. The tools we used provided no coupling between the logic design and physical board placement. For example, Viewplace performs an initial placement within our schematic environment that is then lost when exporting the netlist to RACAL.

Part of our CAD problem was that commercial autorouters, originally designed for thru hole boards, do not perform well on very dense surface mount designs, A greater emphasis must be place on getting every routed signal from surface mount pads to a vias from which they can be routed. RACAL did not make the correct tradeoff between routing a signal (thereby creating blockages for later nets not yet routed) versus getting the pads connected to vias before routing. This forced us to create many hand routes from pads to vias, often using diagonal routes. It is clear that routers need new algorithms specifically designed for dense surface mount and to exploit blind vias.

Fourth, many aspects of system building can be performed by subcontractors. This is critical for the success of university system building projects, because few groups can do it all themselves. We used different local subcontractors to design our sheet metal, to fabricate our shelves and disk holders, and assemble our surface mount printed circuit boards.

Fifth, we were effective at leveraging our industrial contacts to obtain major subsystems of our design, namely the TMC HIPPI boards and the ATC SCSI string boards. However, such interactions have their drawbacks. Much of our design complexity comes from interfacing to boards for which we have no schematics or access to internal firmware. For example, ATC would not give us source to their firmware, and only one member of the design team, Ken Lutz, had restricted access to their schematics. While not the ideal situation, ATC did dedicate a software engineer, Tony Andrews, to assist us with our software problems.

Sixth, we used a locally written operating system, Sprite, as the basis of our server software architecture. This decision has its pluses and minuses. On the positive side, we had considerable local expertise, rapid turnaround on support (most of the time), and ready access to the source code. The negatives were that we could not leverage existing device drivers written for industry standard operating systems, and that the long-term future of our underlying software base remains unclear.

6.2. Design History

The RAID group began its research on high performance I/O subsystems in early 1987, with a variety of discussions on the tradeoffs between availability and performance in disk systems. The paper originally defining the RAID taxonomy was written in Fall 1987. NSF sponsored the implementation of the first "proof of concept" RAID-I between 1987 and 1990. That design effort focused on choosing the appropriate commercially available technology for disk drives, disk interfaces, and I/O controllers, and the development of the fundamental algorithms for data striping and reconstruction after disk failures. We began putting the prototype together in late 1988. It became operational by mid-1989 and was stable by year end.

A DARPA-sponsored project to scale up the design for HIPPI interconnects and substantially larger numbers of disk drives, RAID-II, started in mid-1989. During much of 1990, we gained operational experience with RAID-I while designing RAID-II. Ultimately, RAID-I became a main file server our research groups.

Initially we had planned to design our own HIPPI and SCSI interfaces. As we learned more about the complexity of these subsystems, we sought industrial partners to help us implement them. In early 1990. TMC agreed to give us with their two-board HIPPI interface, while ATC provided us with their VME five-string SCSI boards at their manufacturing cost.

We knew that RAID-II would require much greater bandwidth between the HIPPI interfaces and the disk drives than what would be possible with a conventional file server backplane. As described above, our original plan was to use a single wide, high bandwidth bus to interconnect the HIPPI interfaces, the SCSI disk interfaces, and the memory system. This is similar in style to the organization of the strategy HIPPI RAID-3 product described in [Katz 92]. We made considerable progress on this design, but we were forced to abandon it in mid-1990 when we learned that the OS designers could not force network headers to be aligned on eight byte boundaries. Ed Lee proposed the crossbar interconnection network, and sold it to the rest of the group in several stormy design reviews. This became the basis of the final design.

The logic design began in Fall 1990 and was essentially complete by September 1991. We used Viewlogic for schematic capture and simulation. While a year may appear long, it was to be expected. The hardware design team consisted of only four students. This was their first major logic design. The VME state machines where quite complex, requiring frequent revision as pieces of the design came together. And finally, there was the usual turnover among the designers, leading to lost time as someone new came up to speed. In retrospect, the design effort could have used more personnel.

Ken Lutz began the physical design in the Fall of 1991, and it quickly became clear that the logic would not fit on a single board. The logic was partitioned into an XBUS card and four daughter cards containing the VME interfaces for the ATC cards. The latter was fabricated by January 1992, and provided an early learning experience with surface mount technology.

Even after partitioning, the XBUS card remained too complex to be routed by RACAL. While waiting for new beta software, we took the opportunity to perform more extensive system simulations of the board. This was worthwhile, because several obscure timing problems were uncovered and fixed.

We spent the first three months of 1992 fighting RACAL to complete the XBUS physical design. Iterative routing passes typically took hundreds of hours, and still required considerable hand completion. To minimize the risk, we decided to overlap the development of the original 8-by-8 port board and a simpler 8-by-4 port board. The latter greatly reducing the interconnect complexity of the crossbar. In fact, the original board, though finally routed, could not be fabricated by the PCB manufacturer, in part due to poor yields on blind vias. Our operational prototype is based on the second design.

The boards were submitted for fabrication in April 1992. The 4-port board was fabricated, assembled, and ready for testing by the end of May. It came up very rapidly. Only a small number of logic problems were encountered, attesting to the completeness of our simulation strategy. Most of the time was spent improving its electrical performance and on repairing the usual interfacing problems.

Figure 8 summarizes the bugs found per week, between 8/4/92 and 10/5/92. Figure 9 shows their sources. Many bugs are found initially, but then trail off in a long tail. The major source of errors was due to the manufacturing and assembly processes. These included bent pins or the mounting of the wrong com-

Figure 8: *Bugs Found Per Week*
The number of bugs found are shown as a function of the week of debugging. The shape of this curve is very familiar to system designers: a rapidly rising number of bugs with a long tail of falling numbers of bugs per week.

Figure 9: *Sources of Bugs*
The largest single source of bugs is from the manufacturing and assembly process, such as a bent pin or a misplaced component. Design flaws are mainly interface protocol errors while logic flaws are mistakes in the schematics. Electrical flaws are noise problems that require us to run at slower clock rates. Software problems are self-explanatory.

ponent on the printed circuit board. The logic and software errors have been modest compared with our previous designs.

At this writing (early October 1992), all of the major subsystems are operational and data can be read and written through the HIPPI interfaces and the XBUS memory to disk. Obviously, a good deal of software tuning remains to be done to achieve the original performance goals.

7. Summary, Conclusions, Future Plans

Figure 10 gives a photograph of the RAID-II disk array storage server. It represents a natural evolution in system building projects within the universities. We have moved away from designing custom LSI chips towards constructing large scale assemblies of mechanical and electrical components. In these kinds of

Figure 10: *RAID-II Prototype*
Ann Drapeau is replacing a failed disk drive in the RAID-II storage server. The main equipment rack, containing the HIPPI boards, ATC disk controllers, VME links, and SUN host, is at the center. On either side are the racks of disk shelves. This configuration can hold up to 144 disk drives, 72 in each of the disk racks.

projects, we must worry as much about sheet metal and mechanical design as we used to worry about circuit design. And we are working with more complex software interfaces as well. RAID-II is network-attached hardware, adding another level of software complexity beyond the operating system.

In 1987, we developed the original RAID taxonomy. Yet even today, there are few commercially available RAID Level 5 systems, and none that would give us the open environment for the kinds of experimentation with network-attached storage we plan to pursue next. For example, our prototype contains over 100 disk drives and supports a HIPPI/network interface to the outside world. It has been a worthwhile effort to build our prototype. This shows that universities can still operate at the forefront of technology, given generous help from industry. We offer our special thanks to the company's that supported us most generously with their technology: IBM for disk drives and memories, ATC for disk controllers, TMC for their HIPPI interfaces, IDT for their high speed FIFOs, and UltraNetwork Technologies for their networking equipment.

We have several major projects underway that plan to make use of RAID-II as a storage server on a high speed network. We are developing application software to use RAID-II as a high capacity video server. The prototype will be used as the destination for the real-time data capture of high resolution video microscope images over a HIPPI-based network between campus and the Lawrence Berkeley Laboratory, In addition, we plan to use RAID-II as the disk frontend for a distributed mass storage system spanning multiple robotic storage systems, including optical jukeboxes and 8 mm and VHS tape robots.

Despite many successful system building projects at Berkeley, we still have many lessons to learn. Better project management is always needed. The project must strike the right balance between building real systems and writing enough papers for the team members to get degrees and jobs. Commercial tools will still break with cutting edge projects. Unfortunately, unlike our projects in the past that depended on university tools, we have little leverage with companies to get the software fixed. And finally, change management and version control remain primitive, despite the extensive research in this area. Our 4-port XBUS card has half the memory we thought it would have because of an error in our hand-maintained version control.

8. References

[Chervenak 91] A. L. Chervenak, R. H. Katz, "Performance Measurements of the First RAID Prototype," Proceedings ACM SIGMETRICS Conference, San Diego, CA, (May 1991).

[Katz 92] R. H. Katz, "High Performance Network and Channel-Based Storage," *Proceedings of the IEEE*, V. 80, N. 8, (August 1992).

[Lee 92] E. K. Lee, P. M. Chen, J. H. Hartman, A. L. Drapeau, E. L. Miller, R. H. Katz, G. A. Gibson, D. A. Patterson, "RAID-II: A Scalable Storage Architecture for High-Bandwidth Network File Service," Technical Report UCB/CSD 92/672, (February 1992).

[Patterson 88] D. A. Patterson, G. A. Gibson, R. H. Katz, "The Case for RAID: Redundant Arrays of Inexpensive Disks," Proceedings ACM SIGMOD Conference, Chicago, IL, (May 1988), pp. 106–113.

[Rosenblum 91] M. Rosenblum, J. Ousterhout, "The Design and Implementation of a Log-structured File System," Proc. ACM Symp. on Operating Systems Principles, (October 1991).