The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers

Emmanuel A. Arnould, François J. Bitz, Eric C. Cooper, H. T. Kung, Robert D. Sansom, Peter A. Steenkiste

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Nectar is a "network backplane" for use in heterogeneous multicomputers. The initial system consists of a star-shaped fiber-optic network with an aggregate bandwidth of 1.6 gigabits/second and a switching latency of 700 nanoseconds. The system can be scaled up by connecting hundreds of these networks together.

The Nectar architecture provides a flexible way to handle heterogeneity and task-level parallelism. A wide variety of machines can be connected as Nectar nodes and the Nectar system software allows applications to communicate at a high level. Protocol processing is off-loaded to powerful communication processors so that nodes do not have to support a suite of network protocols.

We have designed and built a prototype Nectar system that has been operational since November 1988. This paper presents the motivation and goals for Nectar and describes its hardware and software. The presentation emphasizes how the goals influenced the design decisions and led to the novel aspects of Nectar.

This research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533.

Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III), Boston, Massachusetts, April 3–6, 1989.

1 Introduction

Parallel processing is widely accepted as the most promising way to reach the next level of computer system performance. Currently, most parallel machines provide efficient support only for homogeneous, fine-grained parallel applications. There are three problems with such machines.

First, there is a limit to how far the performance of these machines can be scaled up. When that limit is reached, it becomes desirable to exploit coarse-grained, or task-level, parallelism by connecting together several such machines as nodes in a multicomputer. Second, there is a limit to the usefulness of homogeneous systems; as we will argue below, heterogeneity (at hardware and software levels) is inherent in a whole class of important applications. Third, parallel machines are typically built from custom-made processor boards, although they sometimes use standard microprocessor components. These machines cannot readily take advantage of rapid advances in commercially-available sequential processors.

Current local area networks (LANs) can be used to connect together existing machines, but this approach is unsatisfactory for a heterogeneous multicomputer with both general-purpose and specialized, high-performance machines. It is often not possible to implement efficiently the required communication protocols on special-purpose machines, and typical applications for such systems require higher bandwidth and lower latency than current LANs can provide.

The Nectar (*ne*twork *c*ompu*t*er *ar*chitecture) project attacks the problem of heterogeneous, coarse-grained parallelism on several fronts, from the underlying hardware, through the communication protocols and node operating system support, to the application interface to communication. The solution embodied in the Nectar architecture is a two-level structure, with fine-grained parallelism within tasks at individual nodes, and coarse-grained parallelism among tasks on different nodes. This system-level approach is influenced by our experience with previous projects such as the Warp[1] systolic array machine [1] and the Mach multiprocessor operating system [12].

The Nectar architecture provides a general and systematic way to handle heterogeneity and task-level parallelism. A variety of existing systems can be plugged into a flexible, extensible network backplane. The Nectar system software allows applications to communicate at a high level, without requiring each node to support a suite of network protocols; instead, protocol processing is off-loaded to powerful network interface processors.

We have designed and built a prototype Nectar system that has been operational since November 1988. This paper discusses the motivation and goals of the Nectar project, the hardware and software architectures, and our design decisions for the prototype system. We will evaluate the system with real applications in the coming year.

Section 2 of this paper summarizes the goals for Nectar. An overview of the Nectar system and the prototype implementation is given in Section 3. The two major functional units of the system, the HUB and CAB, are described in Sections 4 and 5. Section 6 describes the Nectar software. Some of the applications that Nectar will support are discussed in Section 7. The paper concludes with Section 8.

2 Nectar Goals

There are three major technical goals for the Nectar system: *heterogeneity*, *scalability*, and *low-latency, high-bandwidth communication*. These goals follow directly from the desire to support an emerging class of large-grained parallel programs whose characteristics are described below.

2.1 Heterogeneity

One of the characteristics of these applications is the need to process information at multiple, qualitatively different levels. For example, a computer vision system may require image processing on its raw input at the lowest level, and scene recognition using a knowledge base at the highest level. A speech understanding system has a similar structure, with low-level signal processing and high-level natural language parsing. The processing required by an autonomous robot might range from handling sensor inputs to high-level planning.

At the lowest levels, these applications deal with simple data structures and highly regular number-crunching algorithms. The large amount of data at high rates often requires specialized hardware. At the highest level, these applications may use complicated symbolic data structures and data-dependent flow of control. Specialized inference engines or database machines might be appropriate for these tasks. The very nature of these applications dictates a heterogeneous hardware environment, with varied instruction sets, data representations, and performance.

Software heterogeneity is equally significant. The most natural programming language for each task ranges from Fortran and C to query languages and production systems. As a result, the system must handle differences in programming languages, operating systems, and data representations.

2.2 Scalability

Often the most cost-effective way of extending a system to support new applications is to add hardware rather than replacing the entire system. Also, by including a variety of processors, the system can take advantage of performance improvements in commercially available computers. In the Nectar system, it must therefore be possible to add or replace nodes without disruption: the bandwidth and latency between existing tasks should not be affected significantly, and it should not be necessary to change existing system software. Using the same hardware design, Nectar should scale up to a network of hundreds of supercomputer-class machines.

2.3 Low-Latency, High-Bandwidth Communication

The structure of these parallel applications requires communication among different tasks, both “horizontally” (among tasks operating at the same level of representation) and “vertically” (between levels). The lower levels in particular require a high data rate (megabyte images at video rates, for example). Moreover, applications often have response-time requirements that can only be satisfied by low-latency communication; two examples are continuous speech recognition and the control of autonomous vehicles.

In general, by providing low-latency, high-bandwidth communication the system can rapidly distribute computations to multiple processors. This allows the efficient parallel implementation of many applications.

Lowering latency is much more challenging than increasing bandwidth, since the latter can always be achieved by using pipelined architectures with wide data paths and high-bandwidth communication media such as fiber-optic lines. Latency can be particularly difficult to

[1] Warp is a service mark of Carnegie Mellon University.

minimize in a large system where multi-hop communication is necessary. Nectar has the following performance goals for communication latency: excluding the transmission delays of the optical fibers, the latency for a message sent between processes on two CABs should be under 30 microseconds; the corresponding latency for processes residing in nodes should be under 100 microseconds; and the latency to establish a connection through a single HUB should be under 1 microsecond.

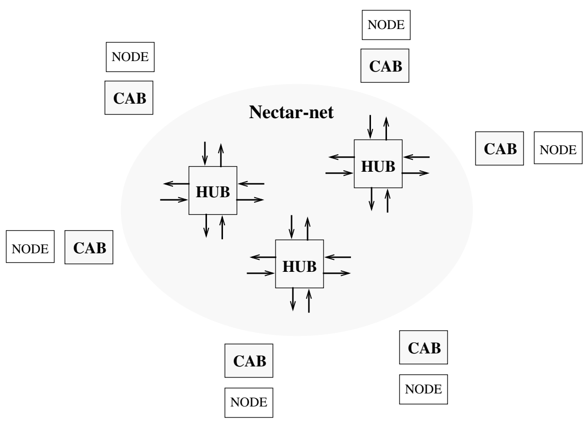

Figure 1: Nectar system overview

3 The Nectar System

3.1 System Overview

The Nectar system consists of a *Nectar-net* and a set of CABs (*communication accelerator boards*), as illustrated in Figure 1. It connects a number of existing systems called *nodes*. The Nectar-net is built from fiber-optic lines and one or more HUBs. A HUB is a crossbar switch with a flexible datalink protocol implemented in hardware. A CAB is a RISC-based processor board serving three functions: it implements higher-level network protocols; it provides the interface between the Nectar-net and the nodes; and it off-loads application tasks from nodes whenever appropriate. Every CAB is connected to a HUB via a pair of fiber lines carrying signals in opposite directions. A HUB together with its directly connected CABs forms a *HUB cluster*.

In a system with a single HUB, all the CABs are connected to the same HUB (Figure 2). The number of CABs in the system is therefore limited by the number of I/O ports of the HUB.

To build larger systems, multiple HUBs are needed. In such systems, some of the I/O ports on each HUB are used for inter-HUB fiber connections, as shown

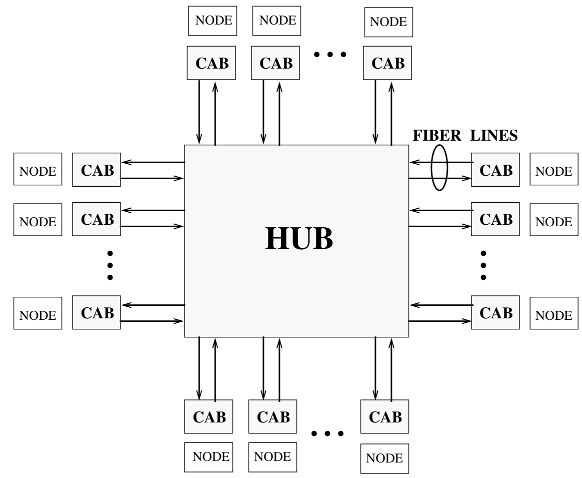

Figure 2: A single-HUB system

in Figure 3. The HUB clusters may be connected in any topology appropriate to the application environment. Since the I/O ports used for HUB-HUB and for CAB-HUB connections are identical, there is no *a priori* restriction on how many links can be used for inter-HUB connections. Figure 4 depicts a multi-HUB system using a 2-dimensional mesh to connect its clusters.

The Nectar-net offers at least an order of magnitude improvement in bandwidth and latency over current LANs. Moreover, the use of crossbar switches substantially reduces network contention. Moderate-size, high-speed crossbars (with setup latency under one microsecond) are now practical; 8-bit wide 32×32 crossbars can be built with off-the-shelf parts, and 128×128 crossbars are possible with custom VLSI.

Re-engineering the software in the critical path of communication is as important for achieving low latency as building fast network hardware. Typical profiles of networking implementations on UNIX[2] show that the time spent in the software dominates the time spent on the wire [3,5,11].

There are three main sources of inefficiency in current networking implementations. First, existing application interfaces incur excessive costs due to context switching and data copying between the user process and the node operating system. Second, the node must incur the overhead of higher-level protocols that ensure reliable communications for applications. Third, the network interface burdens the node with interrupt handling and header processing for each packet.

The Nectar software architecture alleviates these problems by restructuring the way applications communicate. User processes have direct access to a high-level network

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

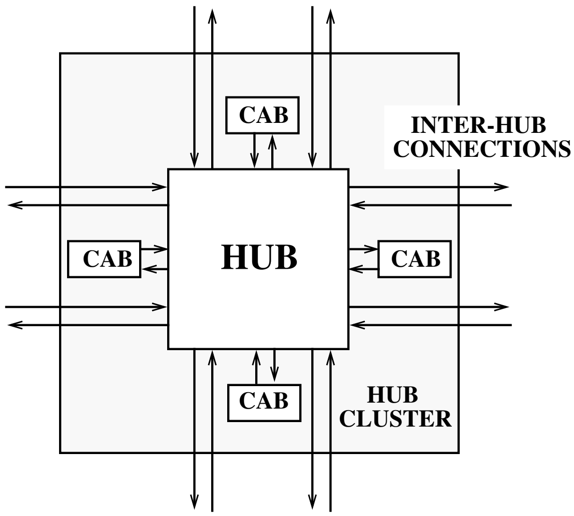

Figure 3: HUB cluster

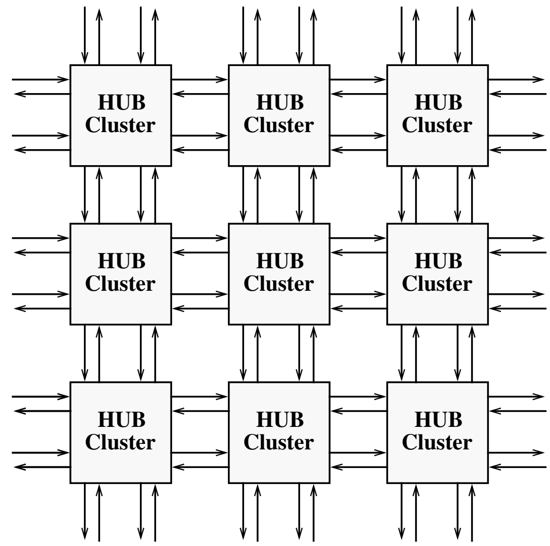

Figure 4: Multi-HUB system connected in a 2-D mesh

interface mapped into their address spaces. Communication overhead on the node is substantially reduced for three reasons. First, no system calls are required during communication. Second, protocol processing is offloaded to the CAB. Third, interrupts are required only for high-level events in which the application is interested, such as delivery of complete messages, rather than low-level events such as the arrival of control packets or timer expiration.

3.2 Prototype Implementation

We have built a prototype Nectar system to carry out extensive systems and applications experiments. The system includes three board types: CAB, HUB I/O board, and HUB backplane. As of early 1989 the prototype consists of 2 HUBs and 4 CABs. The system will be expanded to about 30 CABs in Spring 1989.

In the prototype, a node can be any system running UNIX or Mach [12] with a VME interface. The initial Nectar system at Carnegie Mellon will have Sun-3s, Sun-4s and Warp systems as nodes.

To speed up hardware construction, the prototype uses only off-the-shelf parts and 16×16 crossbars. The serial to parallel conversion is performed by a pair of TAXI chips manufactured by Advanced Micro Devices. The effective bandwidth per fiber line is 100 megabits/second, a limit imposed by the TAXI chips.

When the prototype has demonstrated that the Nectar architecture and software works well for applications, we plan to re-implement the system in custom or semi-custom VLSI. This will lead to larger systems with higher performance and lower cost.

4 The HUB

The Nectar HUB establishes connections and passes messages between its input and output fiber lines. There are four design goals for the HUB:

1. *Low latency.* The HUB provides custom hardware to minimize latency. In the prototype system, the latency to set up a connection and transfer the first byte of a packet through a single HUB is ten cycles (700 nanoseconds). Once a connection has been established, the latency to transfer a byte is five cycles (350 nanoseconds), but the transfer of multiple bytes is pipelined to match the 100 megabits/second peak bandwidth of the fibers.

2. *High switching rate.* In the prototype Nectar system, the HUB central controller can set up a new connection through the crossbar switch every 70 nanosecond cycle.

3. *Efficient support for multi-HUB systems.* Because of the low switching and transfer latency of a single HUB, the latency of process to process communication in a multi-HUB system is not significantly higher. Flow control for inter-HUB communication is implemented in hardware (see Section 4.2.3).

4. *Flexibility and high efficiency.* The HUB hardware implements a set of simple commands for the most frequently used operations such as opening and closing connections. These commands can be executed in one cycle by the central HUB controller. By sending different combinations of these

simple commands to the HUB, CABs can implement more complicated datalink protocols such as multicast and multi-HUB connections. The HUB hardware is flexible enough to implement point-to-point and multicast connections using either circuit or packet switching. In addition, HUB commands can be used to implement various network management functions such as testing, reconfiguration, and recovery from hardware failures.

4.1 HUB Overview

The HUB has a number of *I/O ports*, each capable of connecting to a CAB or a HUB via a pair of fiber lines. The I/O port contains circuitry for optical to electrical and electrical to optical conversion. From the functional viewpoint, a port consists of an *input queue* and an *output register* as depicted in Figure 5.

The HUB has a *crossbar switch*, which can connect the input queue of a port to the output register of any other port (see Figure 7). An input queue can be connected to multiple output registers (for multicast), but only one input queue can be connected to an output register at a time. A *status table* is used to keep track of existing connections and to ensure that no new connections are made to output registers that are already in use. The status table is maintained by a *central controller* and can be interrogated by the CABs.

The I/O port extracts commands from the incoming byte stream, and inserts replies to the commands in the outgoing byte stream. Commands that require serialization, such as establishing a connection, are forwarded to the central controller, while "localized" commands, such as breaking a connection, are executed inside the I/O port.

For the prototype Nectar system, the HUB has 16 I/O ports. Two HUB *I/O boards*, each consisting of eight I/O ports, can be plugged into the HUB *backplane*. The backplane contains an 8-bit wide 16×16 crossbar and the central controller. Each I/O port interfaces to a pair of fiber lines at the front of the I/O board. This packaging scheme is depicted in Figure 6. An additional instrumentation board can be plugged into the backplane, as shown in the figure; it can monitor and record events related to the crossbar and its controller.

Each I/O board in the prototype uses 305 chips and has a typical power consumption of 110 watts; the boards are 15×17 inches. The backplane uses 92 chips for the 16×16 crossbar and 132 chips for the central controller. (47 chips in the crossbar and 20 chips in the controller are for hardware debugging.) The backplane has a typical power consumption of 70 watts.

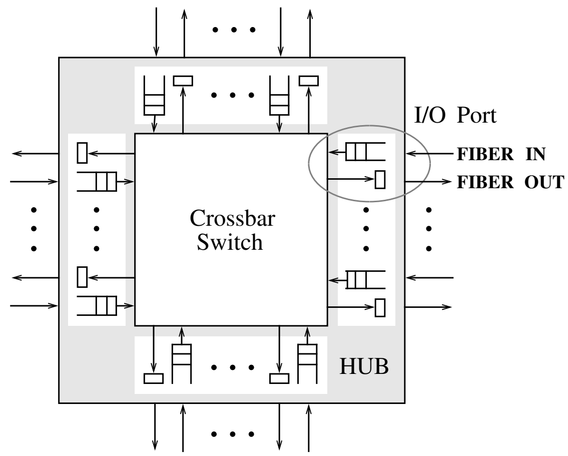

Figure 5: HUB overview

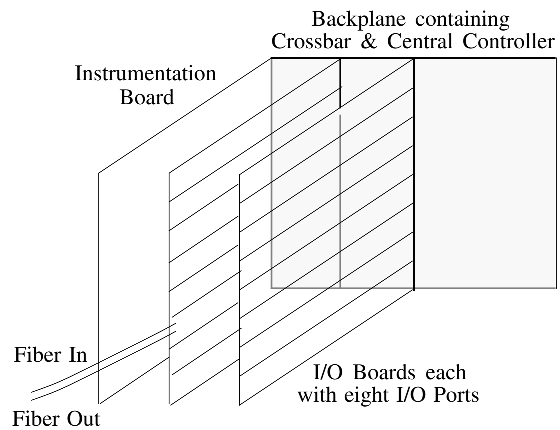

Figure 6: HUB packaging in the Nectar prototype

4.2 HUB Commands and Usage

The HUB hardware supports 38 user commands and 14 supervisor commands for various datalink protocols. Supervisor commands are for system testing and reconfiguration purposes, whereas user commands are for operations concerning connections, locks, status, and flow control.

For the Nectar prototype each command is a sequence of three bytes:

command HUB ID param

The first byte specifies a HUB command, the second byte specifies the HUB to which the command is directed, and the third byte is a parameter for the command, typically the ID of one of the ports on that HUB.

In the following we mention some of the user commands and describe how they can be used to implement

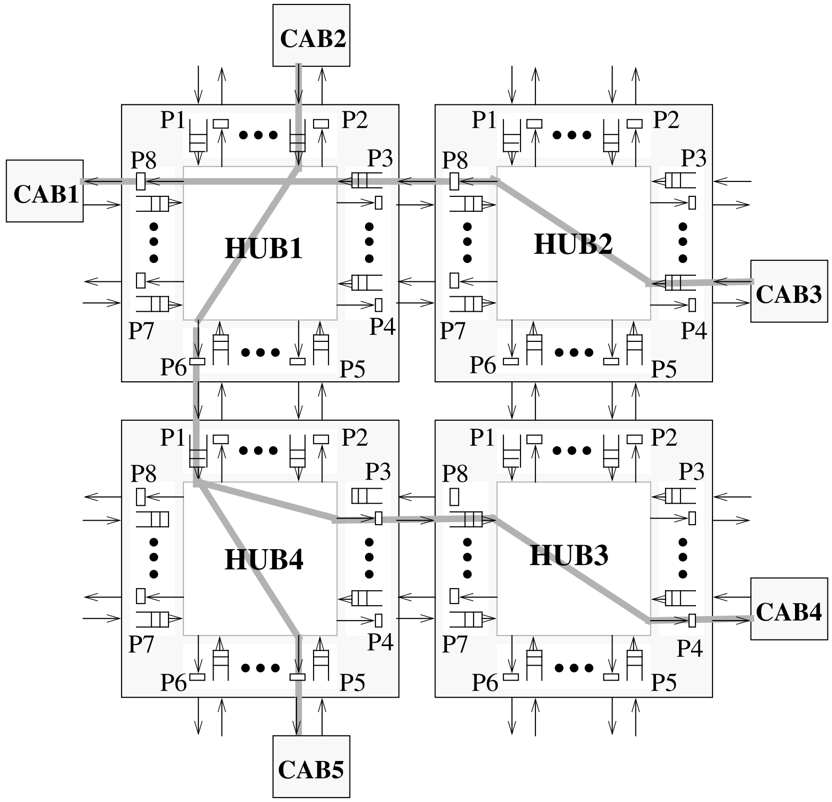

Figure 7: Connections on a 4-HUB system

several datalink protocols. The four-HUB system depicted in Figure 7 will be used in all the examples.

4.2.1 Circuit Switching

Using circuit switching, the entire route is set up first using a *command packet*, before a *data packet* is transmitted. (A data packet is framed by **start of packet** and **end of packet**.) To send *data* from CAB3 to CAB1, CAB3 first establishes the route by sending out the following command packet:

open with retry	**HUB2**	**P8**
open with retry and reply	**HUB1**	**P8**

HUB2 will keep trying to open the connection from P4 to P8. After the connection is made, the **open with retry and reply** command is forwarded to HUB1 over the connection. After the connection from P3 to P8 in HUB1 is established, HUB1 sends a reply over the route established in the opposite direction, using another set of fiber lines, input queues, and output registers. By stealing cycles from these resources whenever necessary, the reply is never blocked and can reach CAB3 within a bounded amount of time. After receiving the reply, CAB3 knows that all the requested connections have been established. Then CAB3 sends *data*, followed by a **close all** command that travels over the established route.

The **close all** command is recognized at the output register of each HUB in the route. After detecting the **close all**, the HUB closes the connection leading to the output register. Therefore all the connections will be closed after the data has flowed through them. Alternatively, **close all** can be replaced with a set of individual **close** commands, closing the connections in reverse order.

If CAB3 does not receive a reply soon enough, it can try to get the connection status of the HUBs involved to find out what connections have been made, and can send another command packet requesting a different route (possibly starting from some existing connections). CAB3 can also decide to take down all the existing connections by using **close all**, and attempt to re-establish an entire route.

4.2.2 Circuit Switching for Multicasting

To illustrate how multicasting is implemented, consider the case in which CAB2 wants to send a data packet to both CAB4 and CAB5, as depicted in Figure 7. To establish the required connections, CAB2 can use the following command packet:

open with retry	**HUB1**	**P6**
open with retry and reply	**HUB4**	**P5**
open with retry	**HUB4**	**P3**
open with retry and reply	**HUB3**	**P4**

After receiving replies to both of the **open with retry and reply** commands, CAB2 sends the data packet.

4.2.3 Packet Switching

The HUB has two facilities to support packet switching:

1. *An input queue for each port.* For the prototype Nectar system the length of the input queue, and thus the maximum packet size, is 1 kilobyte. (Circuit switching must be used for larger packets but, since the overhead of circuit setup is small compared to the packet transmission time, this does not add significantly to latency.)

2. *Support for flow control.* A *ready bit* is associated with each port of a HUB. The ready bit indicates whether the input queue of the next HUB connected to it is ready to store a new packet. Consider for example port P8 of HUB2 in Figure 7. This port is connected to port P3 of HUB1. If the ready bit associated with P8 of HUB2 is 1, then the input queue of P3 of HUB1 is guaranteed to be ready to store a new packet.

   The ready bit associated with each port is set to 1 initially. When **start of packet** is detected at the output register of the port, the ready bit is set to 0. Upon receipt of a signal from the next HUB indicating that the **start of packet** has emerged from the input queue connected to the port, the bit is set to 1.

Suppose that CAB3 wants to send a data packet to CAB1 using the route shown in Figure 7. Using packet switching, CAB3 can send out the following packet:

test open with retry	**HUB2**	**P8**
test open with retry	**HUB1**	**P8**
data		
close all		

The **test open with retry** command is used to enforce flow control. For example, the first **test open with retry** command ensures that HUB2 will not succeed in making the connection from P4 to P8 until port P3 of HUB1 is ready to store the entire data packet. Otherwise HUB2 will keep trying to make the connection. Thus the packet is forwarded to the next HUB as soon as the input queue in that HUB becomes available.

4.2.4 Packet Switching for Multicasting

Consider again the multicasting example of Section 4.2.2. Using packet switching, CAB2 can use the following commands to multicast a packet to CAB4 and CAB5:

test open with retry	**HUB1**	**P6**
test open with retry	**HUB4**	**P5**
test open with retry	**HUB4**	**P3**
test open with retry	**HUB3**	**P4**
data		
close all		

5 The CAB

The CAB is the interface between a node and the Nectar-net. It handles the transmission and reception of data over the fibers connected to the network. Communication protocol processing is off-loaded from the node to the CAB thus freeing the node from the burden of handling packet interrupts, processing packet headers, retransmitting lost packets, fragmenting large messages, and calculating checksums.

5.1 CAB Design Issues

The design of the CAB is driven by three requirements:

1. The CAB must be able to keep up with the transmission rate of the optical fibers (100 megabits/second in each direction).

2. The CAB should ensure that messages can be transmitted over the Nectar-net with low latency. The HUB can set up a connection and begin transferring a data packet in less than one microsecond. Thus any latency added by the CAB can contribute significantly to the overall latency of message transmission.

3. The CAB should provide a flexible environment for the efficient implementation of protocols and selected applications. Specifically, it should be possible to implement a simple operating system on the CAB that allows multiple lightweight processes to share CAB resources.

Together these design goals require that the CAB be able to handle incoming and outgoing data at the same time as meeting local processing needs. This is accomplished by including a hardware DMA controller on the CAB. The DMA controller is able to manage simultaneous data transfers between the incoming and outgoing fibers and CAB memory, as well as between VME and CAB memory, leaving the CAB CPU free for protocol and application processing.

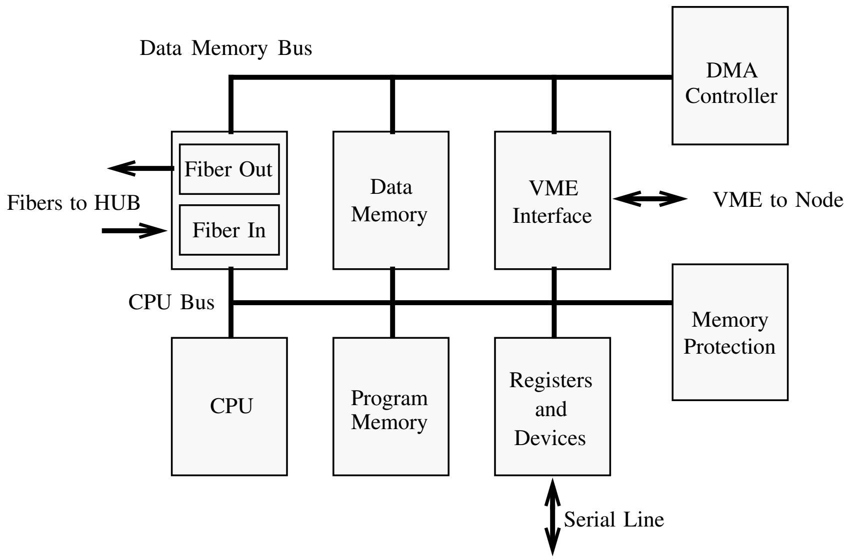

Figure 8: CAB block diagram

To meet the protocol and application processing requirements, we designed the CAB around a high performance RISC CPU and fast local memory. The choice of a high-speed CPU, rather than a custom microengine or lower performance CPU, distinguishes the CAB from many I/O controllers. The latest RISC chips are competitive with microsequencers in speed, but offer additional flexibility and a familiar development environment. Older general-purpose microprocessors would be unable to keep up with the protocol processing requirements at fiber speeds, let alone provide cycles for user tasks.

Allowing application software to run on the CAB is important to many applications but has dangers. In particular, incorrect application software may corrupt CAB operating system data structures. To prevent such problems, the CAB provides memory protection on a per-page basis and hardware support for multiple protection domains.

The CAB design also includes various devices to support high-speed communication: hardware checksum computation removes this burden from protocol software; hardware timers allow time-outs to be set by the software with low overhead.

5.2 CAB Implementation

The prototype CAB implementation uses as its RISC CPU a SPARC processor running at 16 megahertz. A block diagram of the CAB is shown in Figure 8.

A VME interface to the node was the natural choice in our environment, allowing Sun workstations and Warp systems to be used in the Nectar prototype. The initial CAB implementation supports a VME bandwidth of 10 megabytes/second, which is close to the speed of the current fiber interface.

Two fibers (one for each direction) connect each CAB to the HUB. The fiber interface uses the same circuit as the HUB I/O port (see Section 4.1). Data can be read or written to the fiber input or output queue by the CPU, but for data transfers of more than small numbers of words the DMA controller should be used to achieve higher transmission rates. The DMA controller also handles flow control during a transfer: the DMA controller waits for data to arrive if the input queue is empty, or for data to drain if the output queue is full.

The on-board CAB memory is split into two regions: one intended for use as program memory, the other as data memory. DMA transfers are supported for data memory only; transfers to and from program memory must be performed by the CPU. The memory architecture is thus optimized for the expected usage pattern, although still allowing code to be executed from data memory or packets to be sent from program memory.

In the prototype, the total bandwidth of the data memory is 66 megabytes/second, sufficient to support the following concurrent accesses: CPU reads or writes, DMA to the outgoing fiber, DMA from the incoming fiber, and DMA to or from VME memory. The program memory has the same bandwidth as data memory and is thus able to sustain the peak CPU execution rate.

The program memory region contains 128 kilobytes of PROM and 512 kilobytes of RAM. The data memory region contains 1 megabyte of RAM. Both memories are implemented using fast (35 nanosecond) static RAM. Using static RAM in the prototype rather than less expensive dynamic RAM plus caching was worth the additional cost—it allowed us to focus on the more innovative aspects of the design instead of expending effort on a cache.

The CAB's memory protection facility allows each 1 kilobyte page to be protected separately. Each page of the CAB address space (including the CAB registers and devices) can be assigned any subset of read, write, and execute permissions. All accesses from the CAB CPU or from over the VME bus are checked in parallel with the operation so that no latency is added to memory accesses. The flexibility, safety, and debugging support that memory protection affords the CAB software is worth the non-trivial cost in design time and board area.

The memory protection includes hardware support for multiple protection domains, with a separate page protection table for each domain. Currently the CAB supports 32 protection domains. The assignment of protection domains is under the control of the CAB operating system kernel. The kernel can therefore ensure that the CAB system software is protected from user tasks and that user tasks are protected from one another. In addition, accesses from over the VME bus are assigned to a VME-specific protection domain.

The CAB occupies a 24-bit region of the node's VME address space. Every device accessible to the CAB CPU is also visible to the node, allowing complete control of the CAB from the node. In normal operation, however, the node and CAB communicate through shared buffers, DMA, and VME interrupts.

The CAB prototype is a 15 $\times$ 17 inch board, with a typical power consumption of 100 watts. Of the nearly 360 components on the densely packed board about 25% are for the data memory and DMA ports, 15% for the VME interface, 15% for the CPU and program memory, and 13% for the I/O ports. The remaining 120 or so chips are divided among the DMA controller, CAB registers, hardware checksum computation, memory protection, and clocks and timers.

6 Software

The design of the Nectar software has two goals:

1. *Minimize communication latency between user processes.* Since processing overhead on the sending and receiving nodes accounts for most of the communication latency over local area networks [3,5,11], the software organization plays a critical role in reducing the latency. To achieve low latency, data copying and context switching must be minimized.
2. *Provide a flexible software environment on the CAB.* This will convert a bare "protocol engine" into a customizable network interface. When appropriate, the CAB can also be used to off-load application tasks from the node.

The software currently running on the prototype system consists of the CAB kernel, communication protocols, and *Nectarine*, a library for programming applications.

6.1 The CAB Kernel

Candidate run-time systems for the CAB range from a minimal single-task system to a complete UNIX implementation. To provide the required efficiency and flexibility, we built the CAB kernel around lightweight processes similar to Mach *threads* [8]. Threads support multitasking so the CAB can execute multiple activities concurrently in a time-shared fashion, but, since threads have little state associated with them, the cost of context switching is low. Thread switching takes between 10 and 15 microseconds; almost all of this time is spent saving and restoring the SPARC register windows.

Threads execute as a set of coroutines, using a simple, non-preemptive scheduler. This organization matches the intended use of the CAB: a thread will be awakened by an event (such as the arrival of a packet), will take some action (such as processing transport protocol headers), and will voluntarily go back to waiting for another event.

The CAB kernel provides support for simple, time-critical operations such as memory management and timers, but it relies on the node operating system for more complicated operations such as file I/O. The CAB invokes these services by interrupting the node over the VME bus.

Another CAB function is to provide temporary buffer space for messages in an efficient way. This is achieved using *mailboxes* in CAB memory. In the common single-reader, single-writer case, allocating and reclaiming space is simple because mailboxes behave like FIFOs. Mailboxes also support multiple readers, multiple writers, and out-of-order reads. These access patterns occur, for example, when multiple servers operate on different messages in the same mailbox.

6.2 Communication

The communication software has three main parts: the datalink protocol on the CAB, transport protocols on the CAB, and node software dealing with the CAB-node interface. The main guideline in the design of the communication software was to avoid data copying and context switching on both the CAB and the node.

6.2.1 Datalink Protocol

The datalink protocol transfers data packets between CABs using HUB commands, manages HUB connections, and recovers from framing errors and lost HUB commands. The most frequently used simple operations, such as sending a packet to a node in the same HUB cluster, are implemented in hardware as a single HUB command, while more complicated and less frequent operations, such as multicasting and error recovery, are implemented in software.

The interface between the datalink and transport layers passes packets by reference, thus avoiding the copying of data. During a send, the datalink gathers the packet when it transfers the data to the fiber output queue using DMA. During a receive, the datalink interrupt handler, invoked by the **start of packet** signal, executes an *upcall* [6] to a transport layer routine. This routine uses the transport header to determine the destination mailbox for the packet. The datalink layer then sets up the DMA to transfer the incoming data to the destination mailbox. The transport layer upcalls must determine the destination mailbox and return to the datalink layer before incoming data overflows the CAB input queue.

The datalink code is executed entirely by interrupt handlers and by procedures that are called from transport or application threads, so there is no context switching overhead at the datalink-transport interface. The SPARC architecture helps reduce the overhead for critical interrupts by reserving a register window for trap handling.

6.2.2 Transport Protocols

The transport layer is responsible for message transfer between mailboxes on different CABs. This involves breaking messages into packets, reassembling messages, flow control, and retransmission of lost and damaged packets. Three protocols have been implemented:

- The *datagram protocol* has low overhead but does not guarantee packet delivery; it is a direct interface to the datalink layer and should only be used by applications that can tolerate or recover from lost packets.
- The *byte-stream protocol* provides reliable communication using acknowledgments, retransmissions, and a sliding window for flow control.
- The *request-response protocol* supports client-server interactions such as remote procedure calls.

The current transport protocols are simple and Nectar-specific. We plan to experiment with the corresponding Internet protocols (IP, TCP, and VMTP [4]) over Nectar in the coming year.

Each transport protocol can be invoked through a procedure call or by placing a command in a special mailbox. In both cases, the message to be sent is specified as a list of areas located in CAB memory or in CAB-accessible VME memory. When sending large messages between nodes, it is important to overlap packet transfers over the Nectar-net and over the VME bus at each end, in order to reduce latency and increase throughput. The CABs at the sender and receiver sides are well suited for setting up this "packet pipeline": they can select an optimal packet size, synchronize the various DMAs, and manage the buffers.

6.2.3 CAB-Node interface

Three CAB-node interfaces are provided, with different tradeoffs between efficiency and transparency:

- The most efficient CAB-node interface is based on shared memory: the CAB memory is mapped into the address space of the node process, and the node process builds or consumes messages in place in CAB memory. Node processes invoke services by placing a command in a special mailbox on the CAB. This interface is efficient since it eliminates copying the message between the node and the CAB and does not involve the operating system on the node. Messages are received by polling CAB memory.
- A second approach is to provide a Berkeley UNIX socket interface to Nectar. This interface is less efficient since it involves system call overhead and data copying on the node. But the transport protocol overhead is off-loaded onto the CAB. This approach allows existing source code to be used on Nectar with minimal modification.
- The third interface is a Berkeley UNIX network driver for Nectar. In this case, Nectar is used as a "dumb" network and all transport protocol processing is performed on the node. The advantage of this approach is binary compatibility for current applications such as network file systems and window systems.

6.3 Programming Nectar: Nectarine

Nectarine is a programming interface that gives the programmer access to the Nectar hardware and low-level software. It is similar to the communication interface available on other distributed-memory machines such as hypercubes [2]. An important difference is that Nectarine must accommodate heterogeneous nodes, operating systems, memories, attached processors, and other devices.

Nectarine presents the programmer with a simple communication abstraction: applications consist of *tasks* that communicate by transferring messages between user-specified buffers. Tasks are processes on any CAB or node. Messages can be located in any memory. Using Nectarine, the programmer can create tasks, manage buffers, and send and receive messages. Nectarine minimizes the number of copy operations and uses DMA whenever possible.

The fact that Nectarine hides much of the heterogeneity does not free the programmer from knowing the characteristics of nodes and memories, because the allocation of tasks and data to processors and memories has a serious impact on performance. For example, whether a message is allocated in CAB or node memory influences how efficiently the message can be built and how fast it can be sent.

Work has started on higher-level programming tools for Nectar. We are developing a high-level language that will be mapped onto a specific Nectar configuration by a compiler. Automating the mapping process will not only simplify the programming task, but will also make programs portable across multiple Nectar configurations or other distributed systems.

7 Applications

The prototype Nectar system will be used for a wide spectrum of applications ranging from parallel programs, in which Nectar is used as a multicomputer, to research in distributed operating systems, in which Nectar is viewed as a fast LAN.

One of the first Nectar applications is in the area of vision. The application uses a Warp machine [1] for low-level vision analysis and Sun workstations for manipulating image features that are stored in a distributed spatial database. It requires both high bandwidth for image transfer and low latency for communication between nodes in the database. This application has a static computational model: the assignment of tasks to nodes is done when the application is started.

We are implementing a parallel production system as an example of an application that requires run-time load balancing. Matching is performed in parallel using a distributed RETE network, and tokens that propagate through the network are stored in a distributed task queue [14]. The low latency communication of Nectar provides good support for the fine-grained parallelism required by this application.

The flexibility of Nectar allows it to run applications originally written for other parallel systems. For example, to run hypercube applications on Nectar, we have implemented the Intel iPSC communication library [10] on top of Nectarine. Since Nectarine is functionally a superset of the iPSC primitives, this implementation is relatively simple. Several large applications are being ported to Nectar using this approach, including simulated annealing and a solid modeling system based on the octree data structure.

Large-scale scientific applications that execute well on loosely-coupled arrays of processors [7] are also easily ported to Nectar. Powerful, general-purpose Nectar nodes can provide sufficient processing power and memory to meet the computational demands of these applications and the Nectar-net has the bandwidth to meet their communication needs.

The high bandwidth and low latency provided by Nectar also make it an attractive architecture for communication-intensive distributed applications. Examples of such applications include distributed transaction systems, such as Camelot [13], and the simulation of shared virtual memory over a distributed system using Mach [9]. In these applications, the CAB will play a critical role as an operating system co-processor.

8 Conclusions

Architectures to exploit the high-level, irregular parallelism found in large applications should support heterogeneity, low-latency communication, and scalability. The Nectar architecture meets these goals: it is based on a "network backplane" that consists of a high-speed fiber-optic network, large crossbar switches, and powerful network interface processors.

The hardware design takes advantage of the availability of fast crossbar chips, fiber-optic technology, and RISC processors that can keep up with fiber speeds. The software architecture has benefited from experience with network communication, operating systems, and parallel applications.

The prototype shows that the proposed architecture can be implemented. Developing real applications on the prototype is the next step; we will use these applications to evaluate the basic ideas of Nectar, to judge how well the design meets its goals, and to guide us in evolving the prototype into a large-scale system with hundreds of nodes in production use.

Acknowledgements

The Nectar project is a team effort. The following people have made invaluable contributions to the project: Matthieu Arnould, Bernd Bruegge, Vincent Cate, Ming-Jen Chan, Cheng Chen, Chiun-Hong Chien, Takashi Hotta, Mike MacPherson, Onat Menzilcioğlu, Martha Moran, Ravishankar Mosur, Hudson Ribas, Steven Schlick, Wilhelm Schmid, Venkatadri Seshadri, Bruce Siegell, Bernd Spiegel, I-Chen Wu, and Brian Zill. We are also grateful to Sun Microsystems for technical assistance with the SPARC architecture.

References

[1] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica Lam, Onat Menzilcioğlu, and Jon A. Webb.
The Warp computer: architecture, implementation and performance.
IEEE Transactions on Computers, C–36(12):1523–1538, December 1987.

[2] William C. Athas and Charles L. Seitz.
Multicomputers: message-passing concurrent computers.
Computer, 21(8):9–24, August 1988.

[3] Luis-Felipe Cabrera, Edward Hunter, Michael J. Karels, and David A. Mosher.
User-process communication performance in networks of computers.
IEEE Transactions on Software Engineering, SE–14(1):38–53, January 1988.

[4] David R. Cheriton.
VMTP: Versatile Message Transaction Protocol.
RFC 1045, Stanford University, February 1988.

[5] Greg Chesson.
Protocol engine design.
In *Proceedings of the Summer 1987 USENIX Conference*, pages 209–215, June 1987.

[6] David D. Clark.
The structuring of systems using upcalls.
In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180, ACM, December 1985.

[7] E. Clementi, J. Detrich, S. Chin, G. Corongiu, D. Folsom, D. Logan, R. Caltabiano, A. Carnevali, J. Helin, M. Russo, A. Gnuda, and P. Palamidese.
Large-scale computations on a scalar, vector and parallel "Supercomputer".
In E. Clementi and S. Chin, editors, *Structure and Dynamics of Nucleic Acids, Proteins and Membranes*, pages 403–450, Plenum Press, 1986.

[8] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report CMU–CS–88–154, Computer Science Department, Carnegie Mellon University, June 1988.

[9] Allesandro Forin, Joseph Barrera, and Richard Sanzi.
The shared memory server.
In *Winter USENIX Conference*, Usenix, San Diego, January 1989.

[10] *iPSC/2 C Programmer's Reference Manual*.
Intel Corporation, March 1988.

[11] Hemant Kanakia and David R. Cheriton.
The VMP network adaptor board (NAB): high-performance network communication for multiprocessors.
In *Proceedings of the SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 175–187, ACM, August 1988.
Also published as *Computer Communications Review*, 18(4).

[12] Richard F. Rashid, Avadis Tevanian, Jr., Michael W. Young, David B. Golub, Robert V. Baron, David L. Black, William Bolosky, and Jonathan J. Chew.
Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures.
IEEE Transactions on Computers, C–37(8):896–908, August 1988.

[13] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Daniel J. Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson.
The Camelot project.
Database Engineering, 9(4), December 1986.
Also published as Technical Report CMU-CS-86-166, Computer Science Department, Carnegie Mellon University, November 1986.

[14] Milind Tambe, Dirk Kalp, Anoop Gupta, Charles Forgy, Brian Milnes, and Allen Newell.
Soar/PSM-E: investigating match parallelism in a learning production system.
In *Proceedings of the ACM/SIGPLAN PPEALS 1988: Parallel Programming: Experience with Applications, Languages, and Systems*, pages 146–161, ACM, July 1988.
Also published as *SIGPLAN Notices*, 23(9).