LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A Tightly-Coupled Processor-Network Interface

Computation Structures Group Memo 342
March 16, 1992

Christopher F. Joerg
Dana S. Henry

The order of authors' names has been chosen by a random number generator.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Tightly-Coupled Processor-Network Interface

Christopher F. Joerg
Dana S. Henry[1]

By carefully designing the processor-network interface, we can dramatically reduce the software overhead of interprocesssor communication. Our interface architecture typically achieves a three fold improvement over the best existing interfaces. Most of our performance gain comes from simple, low cost hardware support mechanisms for fast dispatching, forwarding of messages, and replying to messages. The remaining improvement is gained by mapping the network interface directly to the processor's register file rather than its memory. Using our hardware mechanisms, a register-mapped interface can receive, process, and reply to a remote read request in a total of two RISC instructions. We have implemented an RTL model of an off-chip memory-mapped interface which provides our hardware mechanisms. Our industrial partner, Motorola, is implementing a similar network interface on-chip in an experimental version of the 88110 processor.

1 Introduction

To have a fast parallel computer, the wisdom goes, one needs a fast processor and a fast network. This paper is not concerned with either. The fastest processor and the fastest network will not perform well if there is too much overhead when the processor posts and receives network messages. On machines in which sending and receiving takes even tens of microseconds, programmers carefully write their programs in order to avoid sending too many messages. In the process, they sacrifice parallelism and, ultimately, performance.

In this paper, we report on our work in designing a significantly faster message posting and message receiving interface. Our goal has been to reduce the software cost of sending or receiving a message to the point where a programmer will not have to worry about sending a message any more than about performing a floating point operation. We have achieved this level of efficiency by handing the compiler explicit control over a very simple, user-level interface. In order to tightly couple sending and receiving into the processor, we have mapped the network interface into processor registers. To speed up the interface, we have folded frequent operations such as dispatching, forwarding, replying, and testing for boundary conditions into simple hardware mechanisms. Our interface is asynchronous and targeted to short messages. It will not deadlock if the network flow control backs up the interface.

Our design has followed four basic principles:

- The processor-network interface should be user-mode programmable. It should not invoke the operating system except to handle messages belonging to non-resident applications.
- The processor-network interface ought to map to processor registers rather than memory, in order to avoid needless loading and storing of message values.

[1]The order of authors' names has been chosen by a random number generator.

- Frequent message operations, such as dispatching, should be assisted by simple hardware mechanisms.

- The sending and receiving of messages should be under explicit control of the user level program, giving it the flexibility to optimize message traffic and to better interleave computation and communication.

The first three principles affect the efficiency of the interface while the fourth principle advocates a certain model of the interface.

To understand how we derived these principles, let us examine existing network interface architectures. Existing designs can be broadly grouped into three categories: DMA based interfaces, memory-mapped interfaces, and hardwired interfaces. We will look at each category separately.

1.1 DMA-Based Interfaces

The first, and the oldest, category consists of parallel processors which relegate message handling to the DMA interface under the operating system's control. The best known of these are the NCUBE [Pal88] and the iPSC/2 [Bra88]. At the hardware level, both machines send and receive messages by initiating a DMA transfer between the main memory and the node's network channel. At the software level, the sending of a message is accomplished by writing the message into the memory and executing a "send" system call which initiates the DMA transfer from the memory to the channel. Receiving messages also involves the operating system, and also requires the program on the receiving node to explicitly perform a "receive" operation.

Since these machines involve the operating system to handle messages, the latency of sending messages can be quite high. [Bra88] shows that doing a simple send and reply with small messages takes 705 microseconds on the iPSC/2 and 1140 microseconds on an NCUBE. Most of this time is due to software overhead. According to [Nug88], on the iPSC/2, 95% of the message latency is due to software and only 5% is due to transport delays in the network.

Our first design principle is that the network interface should not invoke the operating system in order to handle a message belonging to the currently active application. Involving the operating system in the handling of each message leads to large, and often unnecessary, overhead. Even if the operating system routines that are called are very fast and efficient, the costs will still be high because just switching from the user's context to an operating system and back typically takes many microseconds. Having rewritten the NCUBE operating system, [vECCS92] still found overhead of 11 microseconds for sending and 15 for receiving a message.

One justification for accepting this overhead is that it is needed to provide protection among different applications. However, as long as it can be guaranteed that the message is destined for the currently active application, this protection is unnecessary. Instead of involving the operating system, the application itself can handle its arrived messages at a much lower cost. We can accomplish this by checking the process id of the message in the network interface against the id of the processor's resident application and by invoking the operating system only if the id's do not match. Alternately, we can insure that all arriving messages belong to the currently resident application. The network interface or the router can guarantee this by preventing messages from escaping the application's assigned space slice and time slice.

1.2 Memory-Mapped Interfaces

More recent processor-network interface designs place the interface of the processor and the network at a faster processor/memory interface. This interface is placed, to varying degrees, under user-level software control. Examples of this approach are the the Alewife Machine [ACD+91] [Kub91], the CM-5 [Cor91], and the MDP Machine [DDF+92]. These machines provide memory mapped hardware to queue a small number of incoming and outgoing messages, to send messages to the network channel, and to interrupt the processor on message arrival. This hardware sits on one of the memory buses — the main memory bus in the CM-5, the external cache bus in the Alewife, and the internal double bandwidth cache bus in the MDP. Messages are sent by the user's process writing the message directly into the memory mapped hardware and executing a SEND instruction. Messages are received either by polling the memory mapped hardware or by an interrupt on message arrival. For instance, sending a single packet message in the CM-5 takes 1.6 microseconds, or about 50 cycles [vECCS92].

The three machines involve the operating system to different degrees. The Alewife machine uses a privileged send instruction to send a message and an interrupt to receive a message. As a result, both sending and receiving involves transfer of control to the operating system. The MDP's interface is completely under user control. Messages are composed by a user-level instruction which stores one or two message words from general registers into the interface and, optionally, initiates message transmission. To receive a message, another user-level instruction suspends the current thread and polls for a new message. The CM-5 supports both user-level and operating system based messages.

Our second principle is that the processor-network interface should be mapped to the processor's general registers. The memory-mapped network interface designs improve upon the earlier DMA-based ones by placing the network interface at a faster processor/memory interface, and by providing more user-level software control. However they still leave room for further improvement. In order to send a message, these processors have to execute a series of store operations to the memory mapped network interface. In order to receive a message, they have to execute a series of load operations from the memory mapped network interface. A processor with an on-chip network interface could eliminate these loads and stores by mapping the interface into the processor's registers rather than its cache. An arrived message could implicitly appear in a predetermined general registers. Words of an outgoing message could be computed directly into other predetermined general registers.

1.3 Hardwired Interfaces

The final category of interface designs consists of those designs which completely bind the sending, the receiving, and the interpretation of arrived messages in hardware. Examples of these designs include shared memory machines, such as the shared memory interface of the MIT Alewife machine [Kub91], and dataflow machines, such as the MIT Monsoon machine [Pap90]. Since the messages are controlled without software intervention, they can be handled very efficiently. For example, a Monsoon processor can receive, dispatch on, and create messages at the rate of one per cycle. However these machines provide no explicit user-level model of the network. The meaning of messages is bound in hardware and the programmer has no control over when and how communication occurs. In Alewife's shared memory interface, any load or store instruction can initiate a message, or a set of messages, in order to access a potentially remote memory location and to maintain coherence among its copies. In Monsoon, a "send" instruction can explicitly send a token

to another procedure's frame. However, once the token arrives at its destination, it enters that node's token queue and becomes indistinguishable from any local token.

Our third principle is that message handling should remain under programmer and compiler control. True, the hardwired message interface does gain execution efficiency by completely hiding the messages from the programmer. But because the network interface is not exposed to the software, the programmer loses some control. For one, the programmer cannot control the generated network traffic. She cannot prevent the clogging of the network or the idling of the processors by attempting to apportion time between message sending operations, message receiving operations, and other computation. She also cannot lower the network bandwidth by sending messages of the correct size, as opposed to a cache line or an individual token. For these reasons, we find it desirable to leave message handing under the programmer's control.

Our last principle is to preserve the benefits of hardware assistance whenever possible. We do not want to use the completely hardwired-approach because it takes away too much control from the compiler and the programmer. However, just because the programmer is aware of the sending and receiving of messages does not mean that the hardware cannot assist the programmer in efficient message handling. The appealing aspect of the shared memory and dataflow approach is that the arrived message is processed very efficiently in hardware. Each message type is processed by the corresponding hardware and the value saved to a hardware determined location. In software, on the other hand, we will need to execute a series of instructions to determine the type of the arrived message, compute the corresponding message handler, and invoke that handler. We wish to provide hardware support to optimize this task, while still allowing the user control over when (and if) messages are sent and received.

1.4 Outline

The rest of this paper describes a network architecture which we have designed, details an implementation of this architecture which we have simulated, and draws conclusions from a network interface performance study we have carried out. In Section 2, we describe our interface architecture. This architecture follows the four principles described above. In Section 3, we report on an implementation of this architecture, NIC, which we have designed and simulated at register transfer level. We also mention preceding efforts which motivated the design of NIC. NIC allows an efficient network interface to be added to an existing off-the-shelf processor. In Section 4, we evaluate and compare various implementations of our architecture in terms of the dynamic instruction overhead which they generate. We show that simple improvements to the network interface can have a significant impact on the performance of programs. Finally, in Section 5, we conclude by summarizing the lessons and the outcome of our project.

2 The Network Interface Architecture

We designed a network interface architecture based on the outlined principles. As we will show in Section 3, this architecture can be implemented, with different performance implications, on-chip or off-chip. Because we were interested in a concrete programming model, the Berkeley TAM model [CSS+91], we limited our attention to short, 5-word messages typical of that model. However, the architecture could be extended to handle variable length messages, as in MDP, Alewife, or CM5. The architecture takes no position on whether the interface is polled or interrupt driven and could be implemented as either for different types of messages. We have also ignored process protection,

assuming that only a single application will be run or that the network will be drained in between time slices.

In Section 2.1 we will describe the basic architecture. This architecture meets goals 1, 2, and 4; it provides a user-level, register-based, programmer visible interface between the network and the processor. The remainder of Section 2 describes several additional features of our architecture which allow us to meet goal 3 — to match the performance of hardwired designs.

2.1 The Basic Design

As Figure 1 shows, the interface architecture consists of 14 interface registers together with an input message queue and an output message queue. The registers should preferably be part of the processor's general-purpose registers; however, they could also be memory mapped and addressed through explicit loads and stores. In the rest of this section, we will assume that the interface register have been mapped to the processor's general-purpose registers and can be addressed via the register fields of any RISC instruction. Five of the interface registers, the output registers o0 through o4, contain the words of the message being composed. Another five, the input registers i0 through i4, contain the words of a received message. The Control register is used to set values which control the operation of the network interface. For instance, bits in the control register determine what should be done if a new message is to be sent and the output queue is full. The bits in the Status register are set to indicate the current status of the network interface. For instance, one field in the status register reports the number of messages in the input queue. The CodeBase and MsgIP registers are used for optimizing message dispatch and will be described later. The input message queue continuously receives messages from the network and buffers them until the processor is ready to receive them. Similarly, the output message queue buffers messages sent by the processor until the network is able to transport them.

Each message has the same format. It consists of five words, m0 through m4, plus a 4-bit field typically used to indicate the type of the message. The use of the type field will be described later. The address of the destination processor is specified by the high bits of the first word of the message.

The network interface is driven by two new processor commands (Figure 2). The SEND command forms a message out of the contents of the output registers and its constant 4-bit argument; and queues the message for transmission in the output queue. The NEXT command pops the next message from the input queue, if there is one, and writes it into the input registers. Since both the SEND and the NEXT commands take up very few bits, they need not be implemented as separate instructions. They can be incorporated into the unused bits of other instructions, and execute in parallel with other useful work.

The network interface, together with the network, enforces flow control at the sender and prevents deadlock. If the receiving processor does not process messages as fast as the network delivers them, its input message queue backs up into the network. As the network becomes clogged, processors can no longer transmit messages and eventually their output queues fill up. When a sender's output queue is full, an attempt to send a message either signals an exception or stalls the processor until the output queue empties. The choice of action is determined by settings in the CONTROL register. Interrupting the processor will prevent deadlock. The interrupted processor can empty its own input queue and help unclog the network. In contrast, stalling the processor, while more efficient, could lead to deadlock. This is because all processors could stall as a result of a clogged network, leading to no further progress. It is safe to stall the processor only in the rare architectural case where a processor is guaranteed never to receive messages, i.e. lacks a network

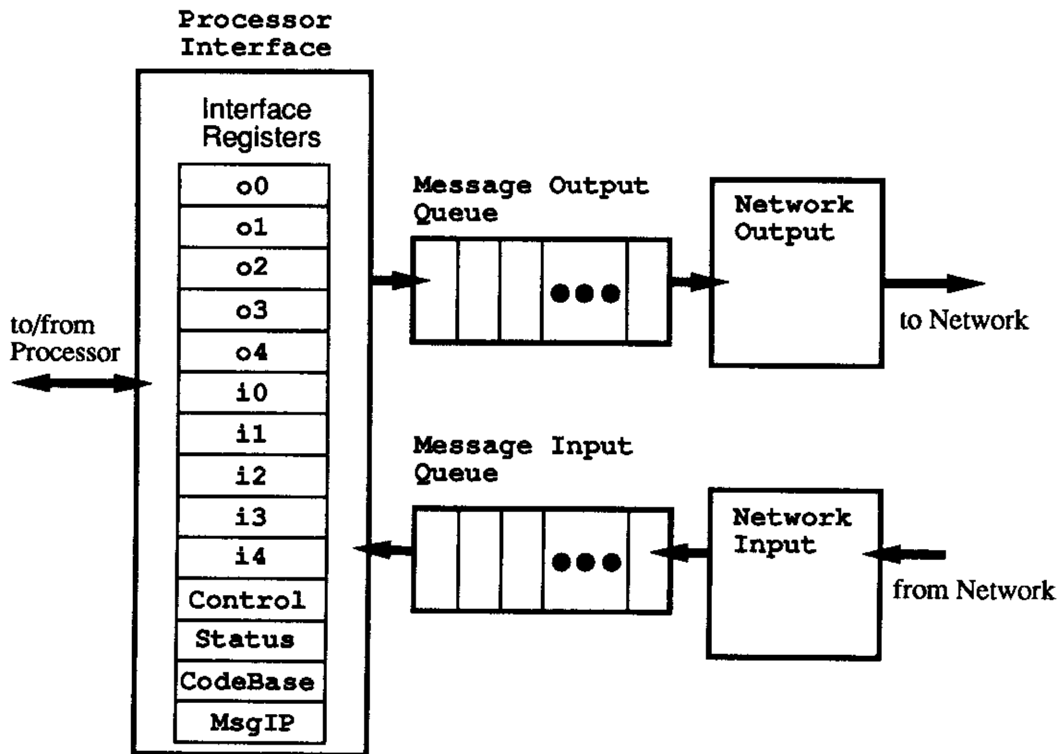


Figure 1: Processor state associated with the network interface.

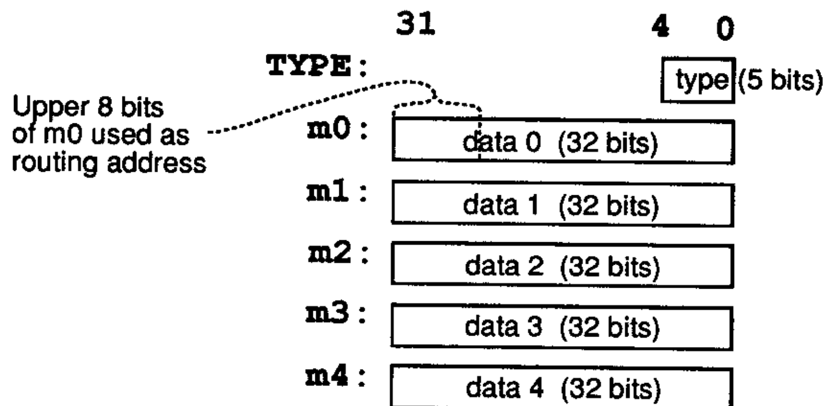


Figure 2: The message format defined by the architecture.

SEND type mode ;;; Send a message
NEXT ;;; Advance to next incoming message

where

type: the 4-bit type field of the message to be sent

mode: an optional argument
if mode is REPLY, m0 and m1 are assigned the contents of i1 and i2
if mode is FORWARD, m2 and m3 are assigned the contents of i2 and i3

Figure 3: The two new processor commands defined by the architecture, SEND and NEXT, their arguments and mnemonics.

---

input port.

2.2 Encoded types

Each message must somehow identify the message handler to be invoked when the message arrives at its destination. In many message passing models, the number of frequently invoked handlers is relatively small and messages identify each message handler by a short constant known as the message type. This type is typically stored in one of the words of each message. When sending a message the software must spend a cycle to explicitly store the type into the message. To save this extra cycle, we provide a four bit type field as part of each message. Instead of storing the message type in a separate instruction, the type field is specified as a constant in the SEND instruction. Even in systems where there are more message types than can be represented in four bits this feature is still useful. Typically the large majority of the messages will be made up of only a few types. These common types can be stored in the type field, and an "escape" type can be stored there when one of the less common message types is to be sent.

2.3 Fast Reply/Forward

When a programmer decides to send a message, the data composing the message may be in registers other than the output registers. Having to move data into the output registers costs cycles and can defeat the purpose of a register-based interface. Replying to or forwarding part of a received message are the two most frequent examples. In both cases, part of the outgoing message should consist of the words of the input message that is currently in the input registers. The address of a reply message should be the same as the address in the input message; the data of a forwarding message should be the same as the data in the input message.

The SEND command takes an additional 2-bit argument, the mode, in order to optimize these frequent cases. In the default case, when the mode is not set, the outgoing message words, m0 through m4, are taken from the output registers, o0 through o4. In the REPLY mode, the address words of the outgoing message, m0 and m1, are instead taken from the return address in the input registers, i1 and i2. Similarly, in the FORWARD mode, the two data words of the outgoing message, m3 and m4, are instead formed from the data in the input registers i3 and i4.

2.4 Hardwired Message Interpretation

The basic architecture we have described so far handles arrived messages completely in software. To dispatch an arrived message, software must check to see if there is a valid message, get the type of the arrived message, use this to compute the instruction address of the handler for that message, and then jump to that handler. In contrast, shared memory and dataflow architectures can analyze the type of the arrived message and invoke the correct message handler in hardware. There is no reason why a programmer visible interface could not do the same.

We achieve hardware efficiency via an additional register, the MsgIP register. The MsgIP register precomputes in hardware the instruction address of the message handler corresponding to the message type of the current input message. Figure 4 shows how MsgIP is computed. To dispatch an incoming message to the correct message handler, we only need to jump to the contents of the MsgIP register. When using this feature another register, the CODE-BASE register, must have been loaded with the starting address of the area in which the message handlers are stored.

The MsgIP register imparts an additional hardware interpretation to the message format. It interprets messages of every type, other than type 0, to have a unique message handler which resides at a location determined by the message type and the CODE-BASE register. Type 0 is the "escape" type. Messages of type 0 are interpreted as carrying the instruction address of their message handler explicitly in the first word of the message, m1. In addition, two unique message handlers are defined for the case when the input registers do not have a valid input message and when an exceptional condition occurs. We discuss exceptional events in more detail in the next section.

2.5 Optimized Boundary Conditions

In addition to message handling, we also use hardware assistance to handle infrequent boundary conditions. For example, we use hardware assistance to avoid network clogging. Recovering from a clogged network can be very costly as multiple processors take exceptions to handle overflow of their output queues. One way to keep this situation from happening is by frequently reading the STATUS register to determine the size of the input and output queues. If either queue starts getting too full, we can take actions to ease the network load. If the input queue starts getting too full, we may want to handle all queued messages before yielding the processor to other computation. If the output queue starts getting too full, we may want to stop sending messages for a while.

However, continuously checking queue lengths in software in order to prevent the, hopefully, rare condition of network overflow is not very efficient. Instead, we offer hardware assistance through the MsgIP register. Whenever the input queue length or the output queue length exceeds a certain threshold, a bit in the MsgIP register gets set. The queue threshold at which the MsgIP register bit gets set can be set independently for each queue in the CONTROL register. Figure 4 illustrates this.

As can be seen from Figure 4, we have chosen to define four versions of each message handler instead of defining two new message handlers to handle the special condition of the input queue and/or the output queue exceeding its threshold. Although this choice uses up more instruction space, it allows each message handler to independently decide how to respond to these conditions. For instance, a message handler which does not send any new messages may decide to ignore the state of the output queue. A short message handler may decide to process its message even when the input queue is getting full.

There are also other rare conditions which we do not want to continuously check for in software

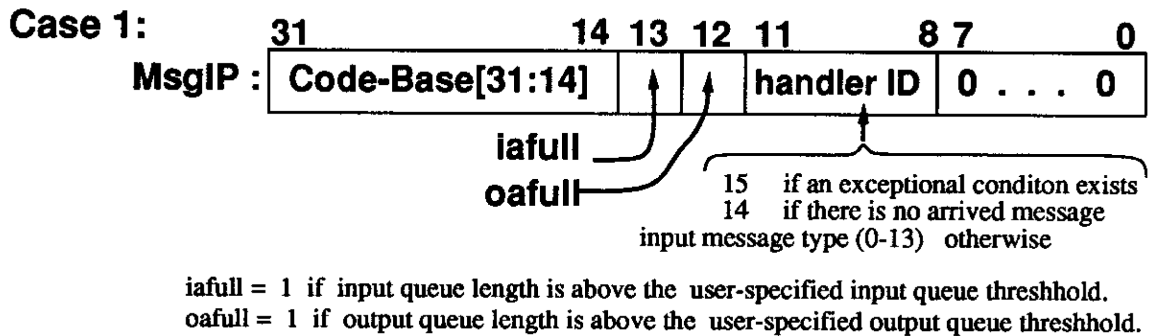


Figure 4: Hardware computation of MsgIP. Typically case 1 applies. If there are no exceptional conditions, neither queue is over its threshold, and the arrived message is of type 0, then case 2 applies.

type:	12
m0:	address of the requested location
m1:	procedure frame location where to deposit the reply
m2:	instruction pointer to be invoked on reply
m3:	—
m4:	—

Figure 5: The message format of a remote read request.

(such as an error in the message input port). The hardware also checks for these exceptional conditions and will set the MsgIP to the instruction address of an exception handler rather than the input message handler if any exceptional condition has occurred. The exception handler can then check the STATUS register to see precisely which exceptional condition has occurred.

2.6 Example

Finally, we illustrate the sending, receiving, and interpreting of messages under our architecture in Figures 6 and 7. Both figures give the Motorola 88100 assembly code for receiving and replying to a remote read request. The message format of a remote read request is illustrated in Figure 5. Both code segments assume that the SEND and NEXT commands can be added to the unused bits of all Motorola 88100 triadic (three-register) instructions. Figure 6 makes no use of the architecture's special features; it is meant to illustrate the basic architecture. Figure 7 employs the encoded message type, the fast reply mode, as well as hardwired message interpretation. Using these features, we can receive, interpret, and reply to a remote read request in only two instructions.

```
dispatcher:
    ; check if there is a valid message in the input registers
    bb0 VALID STATUS no-message
    if there is, and the message is of type 0, jump to type0_msg
    bcnd eq0, TYPE, type0-msg
    ; otherwise, extract the type field of the message
    and r1 STATUS 0x0F00
    ; compute the address of the message handler from its type
    or r1 r1 CODE-BASE
    ; jump to the message handler
    jmp r1

remote read request message handler:
    ; load contents of location specified by i0
    ; into output register o2
    load o2 i0
    ; copy requester's IP into output register o1
    move o1 i2
    ; copy requester's FP into output register o1
    ; send a reply message of type 0
    ; load the next message into the input registers
    move o0 i1, SEND 0, NEXT
```

Figure 6: The eight RISC instructions necessary to receive and process a remote read request using only the basic architecture.

```
dispatcher:
    ; jump to the message handler
    jmp MsgIP

remote read request message handler:
    ; load contents of location specified by i0
    ; into output register o2,
    ; send a reply message of type 0, and
    ; load the next message into the input registers
    load o2 i0, SEND reply 0, NEXT
```

Figure 7: The two RISC instructions necessary to receive and process a remote read request using the encoded message type, the fast reply mode, and the hardwired message interpretation.

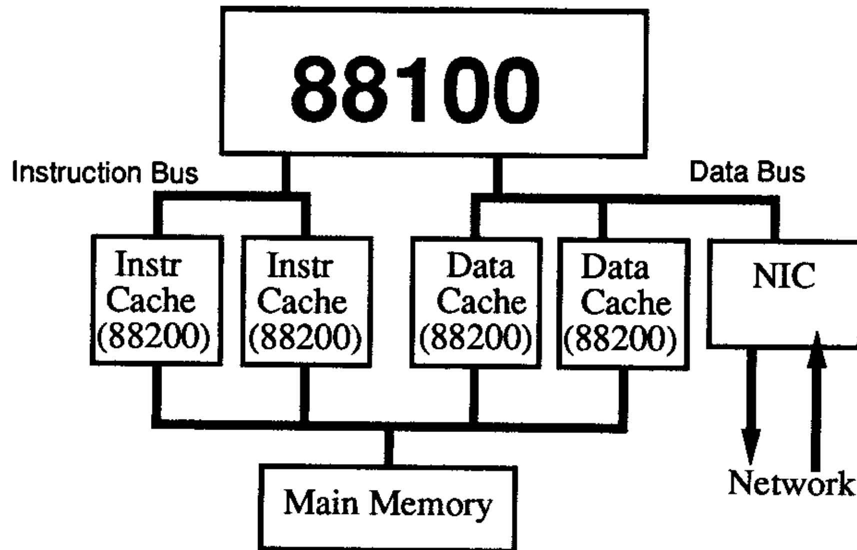


Figure 8: An 88100 system augmented with NIC.

3 An Implementation: The Network Interface Chip

This section describes our implementation of the network interface architecture, the network interface chip (NIC), and the events which motivated our design. In the fall of 1990, together with several other members of our group, we set out to design a multiprocessor for the Berkeley TAM programming model. Unlike conventional RISC processors, this processor was to have an on-chip network interface together with fast synchronization primitives. As the group finished the block level design of the multiprocessor, TIM [ACH+], the project evolved and merged into the *T multiprocessor design [NPA92], a joint effort of our research group and the Motorola Corporation. With the group's attention focused on *T, we proceeded with the design of a network interface, only now, off-chip.

We designed the Network Interface Chip (NIC), whose goal was to provide an efficient network interface which could be used with an existing off-the-shelf microprocessor. Figure 8 gives an overview of a system using NIC. The Motorola 88100 can have up to four 88200 cache chips sitting on its data cache bus. NIC can be used in place of one of these caches, and at the hardware level it appears to the processor as just another 88200. NIC is mapped into a 2^{16} byte region of the processor's virtual address space. It is enabled whenever the high 16 bits of a virtual address match the region into which NIC is mapped. At the network end, NIC interfaces to the PaRC chip[Joe90][JB91], a 100MB/sec/channel packet switched routing chip. NIC's state reflects the architecture described in the preceding section. NIC contains the 14 registers of Figure 1 together with a 16 message deep input queue and a 16 message deep output queue.

Since the off-chip interface is not part of the 88100 processor design, the processor's instruction set cannot access the interface registers or issue a SEND or a NEXT command directly. Instead, the processor communicates with NIC via load and store instructions. In a single load or store instruction, the processor can load from or store to one interface register and, at the same time execute a SEND and a NEXT command.

This is done in the following way. Whenever the high 16 bits of the load or store address match

Address Lines	Information
5:2	interface register number
10:6	type of message to be sent
11	NEXT command
13:12	01 – SEND command 10 – SEND reply command 11 – SEND forward command

Figure 9: The address lines which are read by NIC and the information which is communicated across these pins.

```
dispatcher:
    ; jump to the message handler
    load r3 rl (MsgIP)
    jmp r3

remote read request message handler:
    ; load i1, the address of the location to be read
    load r3 rl (i1)
    ; now load the value stored in that address
    load r4 r3 0
    ; store the contents into output register o2,
    ; send a reply message of type 0, and
    ; load the next message into the input registers
    store r4 rl (o2, SEND reply 0, NEXT)
```

Figure 10: The five RISC instructions necessary to receive and process a remote read request using the NIC chip. The high 16 bits to which NIC is mapped are assumed to be stored in register r1.

the memory mapped position of the chip, the load or store will be processed by NIC. This leaves the lower bits of the address available to be used to communicate what operations NIC should perform. Figure 9 shows how the processor uses these bits to communicate to NIC the SEND and NEXT commands, as well as the interface register being accessed. The data value to be stored into, or loaded from, the interface register is communicated on the data bus. For instance, if the register r1 contains the high 16 bits to which NIC is mapped, then the load instruction:

ld r3 rl 0b10100111011000

loads the contents of interface register 6, i1, into processor register 3; sends a reply message of type 7; and loads the input registers with the next message.

To further illustrate the NIC interface, Figure 10 shows how a remote read request is handled in NIC. We saw in Figure 7 that an on-chip implementation of our network interface can receive, interpret, and reply to a remote read request in two instructions. An off-chip implementation, such as NIC, takes only three more instructions, a relatively mild decrease in performance.

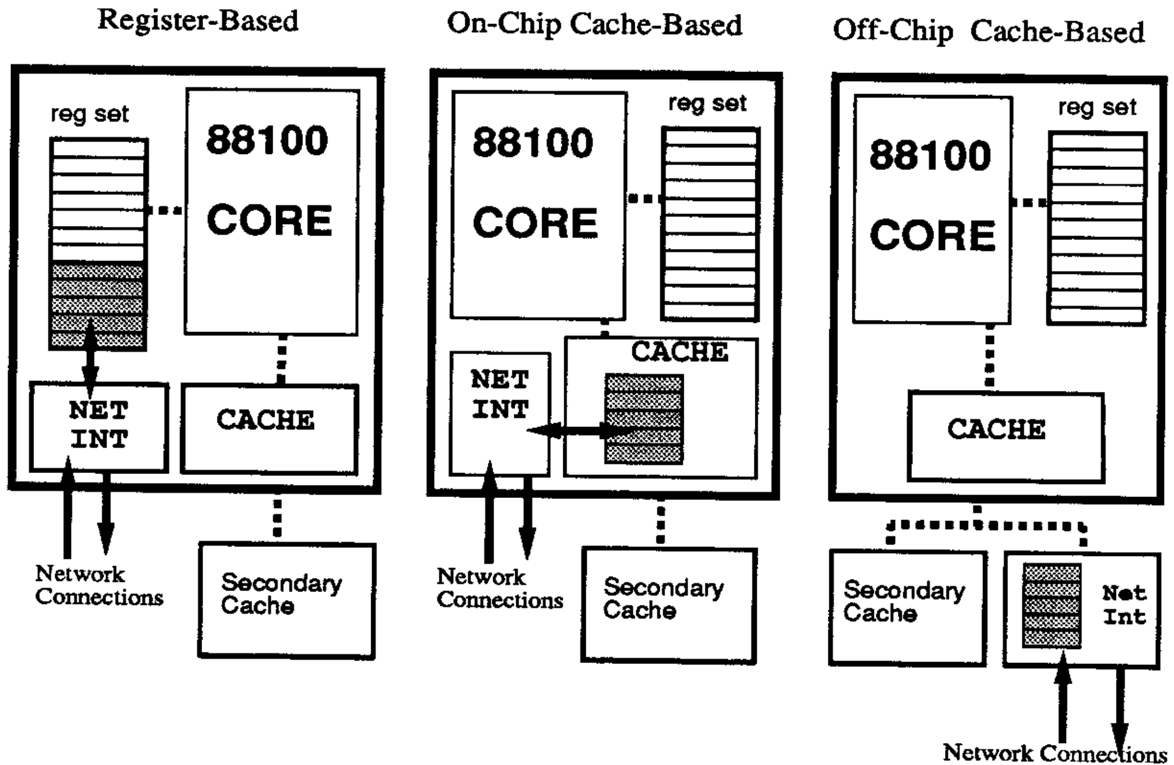


Figure 11: The three network interface implementations studied. Shaded area marks the placement of network interface registers: inside the general-purpose register file, in an on-chip cache, or in an off-chip cache.

4 Performance Evaluation

Finally, in this section, we look at the performance implications of our architecture. We consider the effects of mapping the network interface registers into the processor's register file, the processor's on-chip cache, or the processor's off-chip cache. We also evaluate the impact of the optimizing features described in Sections 2.2 through 2.5. We take two approaches to evaluating the performance of various network interfaces. First, we compare the dynamic number of RISC instructions necessary to send, dispatch, and handle typical types of messages under each interface. Second, we evaluate the dynamic instruction counts of two scientific programs under each interface.

We consider three different implementations of our processor-network architecture from Section 2. Figure 11 outlines the three implementations. All three implementations start out with the 88100 instruction architecture and assume on-chip instruction and data caches. They differ in their access to the network interface registers. The first implementation, the register-based implementation, places the network interface registers in the register file. It augments the 88100 instruction set so that the network interface registers, like all other registers, can be addressed directly by the instruction set. SEND and NEXT commands are incorporated into the unused bits of triadic instructions. This implementation tightly couples the network interface to the processor; the chip's design as well as the instruction set have been modified to incorporate the network interface. It is very similar to our original network interface implementation in the TIM processor [ACH+].

The second implementation of our architecture memory maps the network interface registers

into an on-chip cache. The communication between the processor and the network interface is the same as that described in the previous section. The processor loads from and stores into network interface registers using the 88100's load and store commands. SEND and NEXT commands can be sent simultaneously along the low bits of load and store addresses. Because the interface is on-chip, we assume that communication via a load or store instruction takes one cycle. Unlike the register-based implementation, this implementation does not modify the processor's instruction set; however, it does incorporate an on-chip network interface into the processor chip design.

The third implementation of our architecture memory maps the network interface registers into a secondary, off-chip cache. The network interface in this case is identical to our NIC implementation described in the previous section. The only difference between the previous on-chip implementation and this off-chip memory mapped implementation is the speed of communication between the network interface and the processor. We assume that communication via a load instruction requires two delay slots while communication via a store instruction still takes only one cycle. Unlike the previous two implementations, this implementation makes no changes to the processor; the instruction set as well as the processor chip design are unaffected. As a result, this implementation can be used with existing processors.

We further consider two versions of each processor-network interface implementation of Figure 11. The first version, the basic version, only uses the subset of our interface architecture described in Section 2.1. This subset does not provide any hardware support for encoded message types, fast interpretation of the message, for fast forwarding and replys to incoming messages, or for detection of abnormal, boundary conditions. Without these features, the basic version closely corresponds to existing network interfaces. In contrast, our second version of each interface in Figure 11, the optimized version, makes full use of our architecture's optimizing features.

4.1 Software Cost of Each Message

We first look at how the cost of sending, receiving, and processing a typical message is influenced by the network interface. As our measure of cost, we use the dynamic number of RISC instructions necessary to perform each task. We derive these numbers from code segments which we have handwritten for the 88100 Motorola RISC processor.

We consider the types of messages used by the TAM model [CSS+91]. These are:

Send A Send message sends 0, 1, or 2 words of data to a handler whose address is specified by the message. This type of message is used to pass procedure arguments and procedure results and to reply to all other message requests.

I-fetch An I-fetch message fetches a value from a remote array location with presence bits. If the word has already been written (the location is "full") the receiving handler replies right away. If the word is yet to be written, the handler defers the I-fetch request. The first deferred request takes longer because it initializes a queue of deferred requests.

I-store An I-store message stores a value into a remote array location with presence bits. If there are deferred readers waiting for the value, the receiving handler forwards the value to each of the n deferred readers.

I-allocate An I-allocate message requests an array to be allocated on a remote processor. The receiving handler allocates storage for the array and replies with the address of the allocated array.

F-allocate An F-allocate message requests a procedure frame to be allocated on a remote processor. The receiving handler allocates storage for the frame and informs both the requesting parent frame and the newly allocated child frame, of each other's address.

F-free An F-free message deallocates a procedure frame. No reply is generated.

Table 1 shows our results. It gives the dynamic instruction counts for sending a message, for dispatching an arrived message to the appropriate handler, and for processing the arrived message. The sending of a message consists of communicating the message fields to the network interface and telling the interface to send the message. The dispatching of a message consists of reading the appropriate message field(s) and jumping to the correct message handler. Because of the high cost, we assume that basic implementations, unlike the optimized implementations, do not preventively check for abnormal conditions when dispatching a message. The processing of a message inside the message handler may involve replying or forwarding of the message, as well as some other work such as allocation of memory or management of deferred read lists.

Table 1 shows substantial improvement from a basic, off-chip implementation of our interface to an optimized, on-chip implementation. Several instructions are saved sending a message. These savings come from not having to write out the message type in a separate instruction (Section 2.2) and from not having to store the message fields to memory. Probably the largest number of instructions are saved in dispatch. Most of these savings come from using the hardwired message interpretation of Section 2.4. Savings in processing a message can be largely attributed to the fast reply/forward mode of Section 2.3.

4.2 Impact of Interface on Program Speed

The different software costs of sending, dispatching, and processing a message using each network interface impact the final speed of a parallel program. In this section, we estimate this impact by comparing the total number of RISC instructions to execute each program. Although we concentrate on relatively fine-grain parallel programs, our results apply equally to more coarse-grain ones; the relative advantage of one network interface over another remains the same even as the number of non-message related instructions grows.

We examine the dynamic instruction counts of two scientific programs: a 100 by 100 matrix multiply program and a program which generates paraffins of size 14. The matrix multiply program first subdivides matrices into 4 by 4 blocks and computes their products. The paraffins program generates every distinct paraffin isomer, a hydrocarbon, up to size 14. Both programs have been written in a functional language with logic variables, a non-imperative subset of the Id [Nik90] language, and compiled for the TAM programming model, a relatively fine-grain parallel model.

Figure 12 shows the total number of RISC intructions for the two programs under each network interface model. We have computed these numbers by first compiling the two programs into TL0, the intermediate language for the TAM programming model. We ran the TLO programs by compiling them to TLC, a C programming language back end and running them on a SPARC simulator of TAM. Finally, we computed the dynamic frequency of each TL0 instruction and replaced it by the appropriate number of RISC instructions.

Each bargraph in Figure 12 is divided into two components. The clear, top component corresponds to the total number of instructions executed in order to send, dispatch, and process messages. Although some of the instructions inside message handlers do perform useful work, such as memory allocation or queueing of deferred read requests, most of these instructions can be con-

Table 1: The number of RISC instructions it takes each network interface implementation to send a message, to dispatch an arrived message to the appropriate message handler, and to process a message. Basic implementations only use the basic network interface architecture and do not check for abnormal conditions at dispatch. Optimized implementations make use of special features: the immediate type field, fast forward and reply modes, and hardware assisted message interpretation.

(a) Number of cycles to send a message:

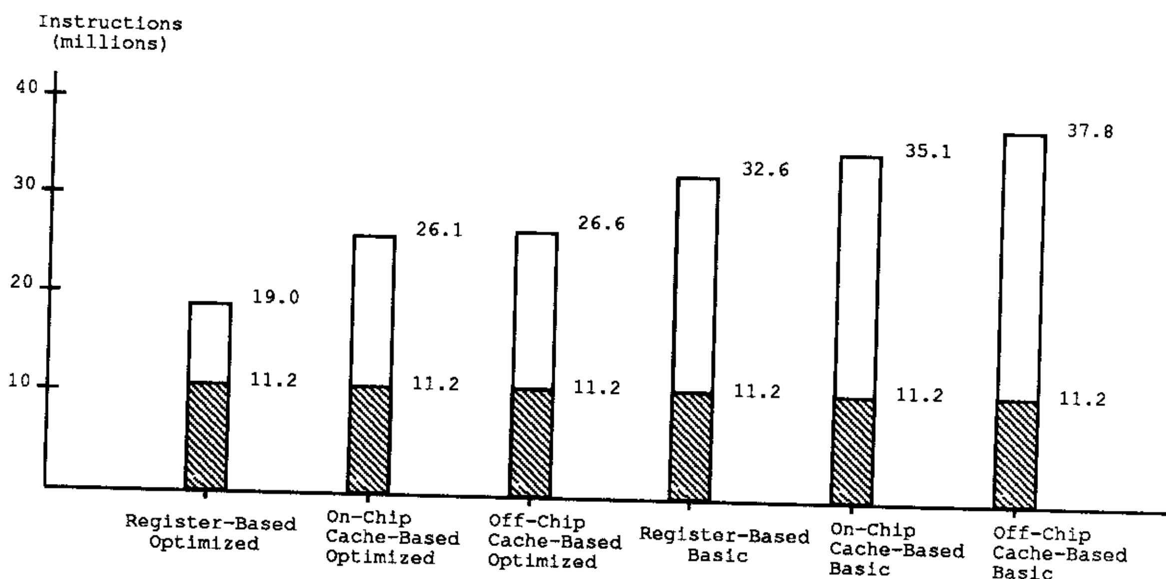
Message Type	Network Interface Implementation					
	Register-Based		On-Chip Cache-Based		Off-Chip Cache-Based	
	Basic	Optimized	Basic	Optimized	Basic	Optimized
Send (0 words)	3	2	5	3	5	3
Send (1 word)	3	2	6	4	6	4
Send (2 words)	3	2	7	5	7	5
I-fetch	4	3	7	5	7	5
I-store	3	2	5	3	5	3
Fallocate	4	3	7	5	7	5
Ffree	2	1	3	1	3	1
Iallocate	4	3	7	5	7	5

(b) Number of cycles to dispatch a message:

Network Interface Implementation					
Register-Based		On-Chip Cache-Based		Off-Chip Cache-Based	
Basic	Optimized	Basic	Optimized	Basic	Optimized
5	1	7	2	9	2

(c) Number of cycles to process a message:

Message Type	Network Interface Implementation					
	Register-Based		On-Chip Cache-Based		Off-Chip Cache-Based	
	Basic	Optimized	Basic	Optimized	Basic	Optimized
Send (0 words)	3	3	4	4	4	4
Send (1 word)	4	4	6	6	6	6
Send (2 words)	5	5	8	8	8	8
I-fetch (full)	12	9	17	12	18	13
I-fetch (empty)	19	19	23	23	25	25
I-fetch (deferred)	15	15	19	19	20	20
I-store (empty)	14	14	17	17	17	17
I-store(deferred)	17+6n	15+6n	20+8n	18+8n	20+9n	18+8n
Fallocate	12	9	16	12	16	12
Deallocate	4	4	5	5	5	5
Iallocate	12	9	16	12	16	12



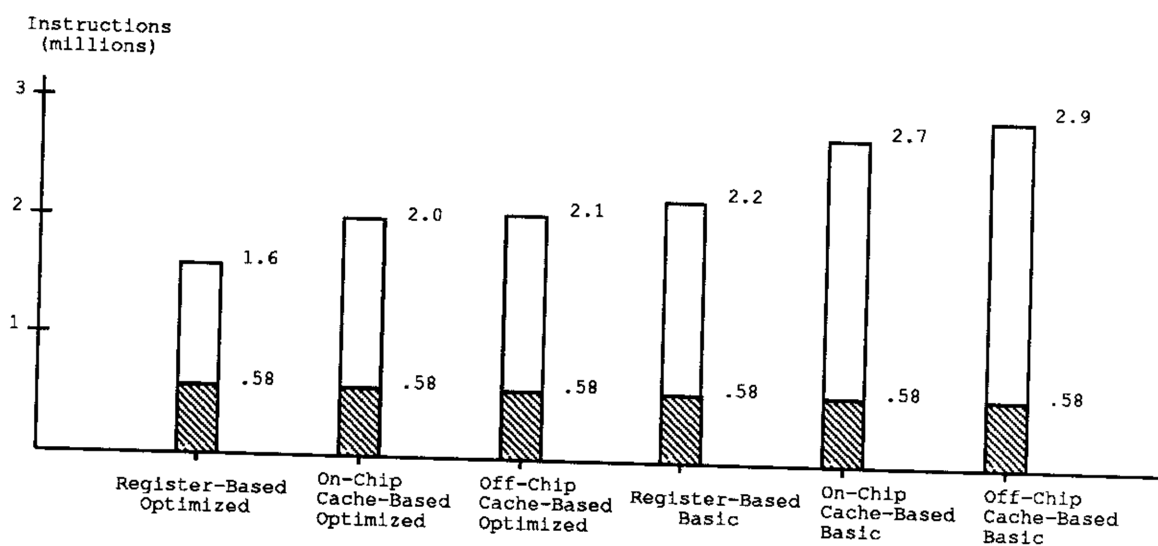


Figure 12: Dynamic instruction counts for 100 by 100 matrix multiply and 14 paraffins using the six different network interface implementations. The shaded section of each bargraph corresponds to computation other than message sending, message dispatching, and message processing.

sidered network interface overhead.[2] The shaded, bottom component corresponds to the remaining instructions, ones which are not involved in communication.

One result, and not a surprising one, is that for these fine grain parallel programs, the cost of communicating has a first order effect on the total number of instructions. Although the dynamic frequency of executing a message sending instruction, such as Send or I-fetch, is only 9.4% for matrix multiply and 11.6% for paraffins; each sending and each receiving of a message expands into a large number of RISC instructions. As a result, in all experiments, except for one, instructions related to interfacing to the network and handling remote requests dominate the total instruction count.

More interesting results can be drawn from the varying number of instructions dedicated to interfacing to the network and handling remote requests under the six different network interface models. The results contrast the importance of our architectural features and different implementations of our architecture. We see that hardware optimizations of the network interface are more important than the actual placement of the interface. Even off-chip cache-based interface with optimizations performs better than a register-based interface without optimizations. This result gives hope to existing processor designs. Simply by attaching an optimized network interface, such as NIC, to the cache bus; today's processors ought to considerably lower their network interface costs.

Most importantly, the gains achieved by using a register-based, optimized interface as opposed to a cache-based, unoptimized interface are substantial. The overhead for interfacing to the network and handling remote requests decreases two to three fold as we optimize the network interface and incorporate it into the register file. This result confirms our belief that substantial performance gains can be achieved relative to existing processor-network interfaces.

5 Conclusion

In this paper, we have advocated a tighter, more optimized coupling of the network and the processor. We have enumerated a set of principles which an efficient processor-network interface should follow, and we have detailed an architecture which follows them. The most important features of our architecture are simple, low-cost hardware support mechanisms for fast dispatching, forwarding, and replying to messages. When mapped into the processor's register file, our processor-network interface allows a remote read request to be received, processed, and replied to in a total of two RISC instructions. We have also demonstrated, through our performance studies, that our architecture could significantly reduce communication overhead from that of existing network interface designs. By mapping the network interface into the processor's general-purpose registers instead of its cache, and by making use of our hardware support mechanisms, we have been able to reduce communication overhead about three fold in our benchmarks.

Another significant outcome of our study is that most of the performance gain comes from our hardware support mechanisms rather than from the placement of the network interface inside the register file. Even if the network interface is moved back into an off-chip cache, our hardware support mechanisms improve its performance more than two fold. We conclude that even existing processors could significantly benefit by incorporating our hardware support mechanisms into an off-chip network interface. We have designed and simulated at low level such an interface, NIC, for the Motorola 88100 processor. In our experiments we have modeled a load from an off-chip memory

[2] TO THE REFEREES: For our final copy, we plan to separate out the useful computation in message handlers from network interface overhead.

mapped interface as having two delayed slots. While this is consistent with the load cost of the 88100 processor, as processor speeds increase, we can expect the cost of an off-chip load to rise. As a result, relegating the network interface offchip will likely not remain a viable alternative for future generations of multiprocessors. Influenced by our work, our industrial partner, Motorola, is implementing a similar network interface on-chip in an experimental version of the 88110 processor.

Acknowledgements

We thank Gregory Papadopoulos, our academic advisor, for letting us take on this project, for advising us, and for enthusiastically supporting our work. We thank Bradley Kuszmaul and members of the TIM design team who have helped us develop many of the ideas in this paper. We thank Michael Flaster and Derek Chiou who have helped us collect statistics. We thank the Berkeley TAM team who have provided us with a TLO and a TLC compiler. We thank past and present members of CSG who have surrounded us with a great working atmosphere and who have written and maintained the Id language and the Id benchmarks which we have used in this paper.

References

[ACD+91] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. Technical Report MIT LCS TM-454, MIT Laboratory for Computer Science, Cambridge, MA, 1991.

[ACH+] Shail Aditya, Jack Costanza, Dana Henry, Christopher Joerg, Paul Johnson, and Ralph Tiberio. TIM Instruction Set Architecture. MIT LCS Computation Structures Group internal document.

[Bra88] D. K. Bradley. First and Second Generation Hypercube Performance. Technical Report UIUCDCS-R-88-1455, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, USA, September 1988.

[Cor91] Thinking Machines Corp. The Connection Machine CM5 Technical Summary. Technical report, Thinking Machines Corp., Cambridge, MA, October 1991.

[CSS+91] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[DDF+92] William J. Dally, Roy Davison, J. A. Stuart Fiske, Greg Fyler, John S. Keen, Richard A. Lethin, Michael Noakes, and Peter R. Nuth. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, April 1992. to be published.

[JB91] Christopher F. Joerg and Andy Boughton. The Monsoon Interconnection Network. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 156–159, October 1991.

[Joe90] Christopher F. Joerg. The Design and Implementation of a Packet Switched Routing Chip. Technical Report TR-482, MIT Laboratory for Computer Science, Cambridge, MA, December 1990.

[Kub91] John Kubiatowiz. Users Manual for the Alewife 1000 Controller Version 0.69. Technical Report Alewife Systems Memo #19, MIT LCS, Cambridge, MA, November 1991.

[Nik90] R.S. Nikhil. Id Version 90.0 Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge MA 02139, USA, September 1990.

[NPA92] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[Nug88] Steven F. Nugent. The iPSC/2 Direct-Connect Communications Technology. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 51–60, January 1988.

[Pal88] John F. Palmer. The NCUBE Family of High-Performance Parallel Computer Systems. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 847–851, January 1988.

[Pap90] Gregory M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* Pitman/MIT Press, 1990.

[vECCS92] Thorsten von Eicken, David E. Culler, Seth C. Culler, and Klaus E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.