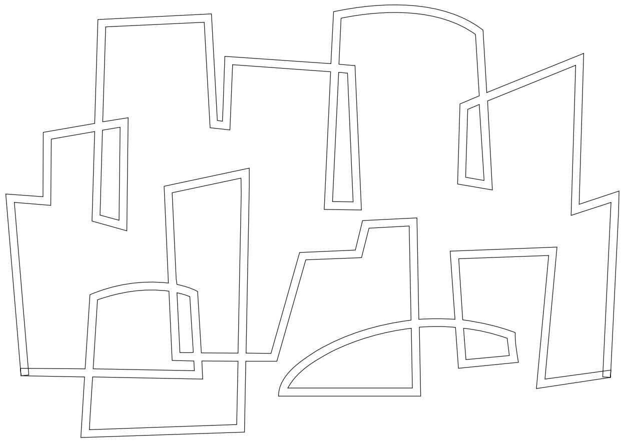
A Tightly-Coupled Processor-Network Interface

Christopher Joerg, Dana Henry

In Proceedings of the Fifth International Conference
on Architectural Support for Programming
Languages and Operating Systems, Boston, MA, October 1992

Architecture, 1992, October

Computation Structures Group
Memo 342



A Tightly-Coupled Processor-Network Interface

Computation Structures Group Memo 342
March 1992

Christopher F. Joerg[†] and Dana S. Henry[†]

[†] Both authors have contributed equally and have chosen the order of their names at random.

This paper appeared in Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992.

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.

A Tightly-Coupled Processor-Network Interface[*]

Dana S. Henry[†]and Christopher F. Joerg[†]
Lab for Computer Science
Massachusetts Institute of Technology
Cambridge MA, 02139
dana@lcs.mit.edu, cfj@lcs.mit.edu

Abstract

Careful design of the processor-network interface can dramatically reduce the software overhead of interprocessor communication. Our interface architecture reduces communication overhead five fold in our benchmarks. Most of our performance gain comes from simple, low cost hardware mechanisms for fast dispatching on, forwarding of, and replying to messages. The remaining improvement can be gained by implementing the network interface as part of the processor's register file. For example, 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 sends and receives network messages. On machines in which sending and receiving takes even a few microseconds, programmers carefully write their programs in order to avoid sending messages. In the process they sacrifice parallelism and, ultimately, performance.

In this paper, we report on our work in designing a significantly faster message sending 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. To speed up the interface, we propose folding frequent operations such as dispatching, forwarding, replying, and testing for boundary conditions into simple hardware mechanisms. In order to tightly couple sending and receiving into the processor, we propose incorporating the network interface into the processor's register file.

We are concerned with improving the performance of systems which use the message passing model. Typical messages in this model are used for passing arguments and results between function invocations and for accessing remote memory locations. Some of the earliest machines which supported the message passing model were the Cosmic Cube [Sei85] and the Transputer [WS85].

Moreover, our study is mainly targeted toward short messages. To efficiently handle short messages the start-up and reception costs of messages must be extremely small. As the size of messages increases, start-up and reception costs become amortized over larger payloads. Our proposed optimizations, though still helpful, become less important. We do not address megabyte size messages typical of I/O or long data streams typical of systolic communication; for these other communication mechanisms might be appropriate.

To motivate the mechanisms we will propose, we first examine existing network interface architectures. Existing designs can be broadly grouped into four categories: operating system based DMA interfaces, user-level memory mapped interfaces, user-level register mapped interfaces, and hardwired interfaces. We review each category separately.

1.1 OS-Level DMA-Based Interfaces

The first category of network interfaces which we consider consists of parallel processors that relegate message handling to the DMA interface under the operating system's control. Examples include the NCUBE [Pal88] and the iPSC/2 [Bra88]. At the hardware level, both machines send and receive messages by initiating a DMA transfer between 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 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. A simple send with small messages takes 267 μs on the iPSC/2 and 437 μs on an NCUBE/four [Bra88]. Much of this time is due to high overhead operating system routines [Nug88]. One group of users rewrote parts of the nCUBE/2 operating system and reduced the overhead by nearly an order of magnitude[vECGS92]. However, even the remaining time was still quite large (11/15 μs) due to

[*]This research was supported in part by Defense Advanced Research Projects Agency under contract no. N00014-89-J-1988

[†]Both authors have contributed equally and have chosen the order of their names at random.

the expense of the DMA instructions and instructions which switch to and from the operating system.

This leads us to believe 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 leads to large, and often unnecessary, overhead. One justification for accepting this overhead is that it is needed to provide protection among different applications. However, we will show in Section 2.1.3 that protection between individual processes can be efficiently maintained by user-level interfaces.

1.2 User-Level Memory-Mapped Interfaces

More recent processor-network interface designs make sending and receiving messages user level operations. Most of these interfaces are memory mapped. The important feature of these interfaces, however, is not that the hardware is actually memory mapped, but that the bandwidth and latency of accessing the network interface is similar to that of accessing memory. Typically, messages are sent by the user's process composing the message and executing a SEND command. The processor finds out that a message has arrived either by polling to check if a message has arrived or by an interrupt which is generated on message arrival.

Examples of this approach include, among others, the MDP Machine [DDF$^+$92], the CM-5 [LAD$^+$92], the memory communication of iWARP, and the message passing interface of the MIT Alewife machine [ACD$^+$91]. In these machines, the interface resides either directly in memory or in special purpose hardware attached to one of the memory buses. In iWARP the processor accesses messages by directly reading and writing off-chip main memory. In Alewife, the processor accesses messages in special purpose hardware which sits on the off chip cache bus.[1] In the CM-5, the processor accesses messages in special purpose hardware which sits on the M-bus, the off-chip memory bus. In the MDP, the processor writes outgoing messages to on-chip special purpose hardware but reads incoming messages directly from the on-chip memory.

Having eliminated the involvement of the operating system, these interfaces are much faster than those of the previous category. For instance, sending a single packet message in the CM-5 takes 1.6 microseconds [vECGS92]. Much of this time is spent accessing the network interface over the external message bus. The MDP is even faster since it accesses messages over an on-chip bus, and because it is able to send two words of a message in a single cycle.

1.3 User-Level Register-Mapped Interfaces

The memory mapped network interface designs 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 register file rather than its memory. An arrived message could implicitly appear in a predetermined set of general registers. Words of an outgoing message could be computed directly into other predetermined general registers.

[1] In Alewife, the arrival of a message is signalled by a supervisor-level interrupt. However, accesses to the interface and the sending of messages are under user-level control.

There are several existing interfaces which map the interface directly into the register file. As far as we know, none of these are used to support the message passing model. Two examples which support other communication models are the grid network of the CM-2[Hil85] and the systolic communication in iWARP[BCC$^+$90].

The CM-2 grid network supports synchronous, unbuffered communication appropriate for the SIMD model. It allows a node to communicate with its four nearest neighbors. For each neighbor, a processor has one register which can be read by that neighbor. A value written into one of these registers can be read by the neighbor on the next cycle.

The systolic network interface in iWARP supports a systolic communication model. A one-way systolic connection can be set up between any two nodes in the machine. The connection includes a special *gate* register at each end. Any write to this register by the sending node implicitly causes the data written to be sent to the receiver. Similarly any read from a special gate register by the receiving node automatically returns the next piece of data received. Unlike the CM-2, there is buffering of the data communicated between nodes, so the processors need not execute in lock step.

By mapping the processor-network interface into the register file, both these mechanisms allow for low-overhead, high-bandwidth communication of data. Although these particular interfaces are not used for a general message passing model, an interface designed to support a message passing model should be able to take advantage of a register mapped interface as well.

1.4 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. Again, designs which use this approach do not implement the general message passing model. 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, once a message (ie. token) arrives at its destination, it becomes indistinguishable from any local token.

We believe that these hardwired interfaces are not appropriate for a general message passing model because they take control away from the programmer and the compiler. One premise of the message passing model is that the application programmer, rather than the runtime system, is best equipped to optimize the placement of data. Contrary to this goal, hardwired shared memory interfaces migrate data from underneath the programmer. Another common belief of the message passing model is that the software system should use all available information in order to efficiently allocate resources. Not knowing which threads will generate or receive communication may hinder the efficient use of the network as well as the processors. Finally, without explicit access to the network, the programmer cannot fine-tune the generated network traffic by sending messages of the correct size, as opposed to a cache line or an individual token.

However, just because the programmer is aware of the sending and receiving of messages does not mean that the hardware cannot assist in efficient message handling. Typically, software based message passing interfaces execute a series of instructions to determine whether there is an incoming message and whether any abnormal conditions have occurred, to determine the type of the arrived message, to compute the corresponding message handler, and, finally, to invoke that handler. Most do not provide any hardware support for message handling. An exception is the MDP network interface which can, in three cycles, read an IP from a hardwired field of the arrived message and transfer control to that IP. Optimizing message passing interfaces by taking advantage of the same hardware mechanisms used by the shared memory and dataflow models, should speed up message passing programs and enable a fairer comparison with competing computation models.

1.5 Summary

We have drawn four principles from examining existing network interface designs:

- The processor-network interface should be user-mode programmable. It should not invoke the operating system to handle messages belonging to the resident application.
- The sending and receiving of messages should be under explicit control of the user level program, giving it the ability to control migration of data, to better interleave computation and communication, and to optimize message traffic.
- The processor-network interface ought to map to processor registers rather than memory, in order to avoid needless loading and storing of message values.
- Frequent message operations, such as dispatching, should be assisted by simple hardware mechanisms.

The first two of these principles are widely accepted by the message passing community. One goal of this paper is to quantitatively show the importance of the last two.

1.6 Outline

The rest of this paper describes our network interface architecture, details several implementations of this architecture, and draws conclusions from a network interface performance study we have carried out. In Section 2, we describe a basic interface architecture and extend it with our proposed optimizations. This architecture benefits from the insights gained in the above survey. In Section 3, we describe three possible implementations of this architecture. These implementations differ in how closely the network interface is coupled to the processor. In Section 4, we evaluate and compare the 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 outcome and lessons of our project.

2 Architectural Optimizations

In this section, we will describe our proposed network interface optimizations and set up the background for evaluating their effects. We start by outlining a very basic network interface architecture which is comparable to existing message passing network interfaces (Section 1.2). This basic architecture will serve as the basis for our performance comparisons. After describing the basic architecture, we proceed to extend it with our proposed hardware support mechanisms. We limit our discussion in this section to the programmer's model of the interface and leave implementation details to Section 3.

2.1 Basic Architecture

The basic architecture described in this section provides minimal network interface services to the programmer. It lets the programmer compose a short message and send it. It also lets the programmer examine the least recently arrived message and dispose of it. 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 will assume a polled interface for the remainder of this paper.

Figure 1 shows the programmer's view of the interface. The interface consists of 15 interface registers together with an input message queue and an output message queue. 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 specify what should be done if a new message is to be sent and the output queue is full. The bits in the **STATUS** register 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 remaining registers are used for optimizing message dispatch and will be described later. The mechanism by which the processor accesses the interface registers is implementation dependent.

The queues provide buffering for messages. The input message queue continuously receives messages from the network and buffers them until the processor receives them. Similarly, the output message queue buffers messages sent by the processor until the network accepts them.

Each message has the same format (Figure 2). It consists of five words, m0 through m4, plus a 4-bit type field which is ignored by the basic architecture. The use of the type field will be described in Section 2.2.1. The logical address of the destination processor is specified by the high bits of the first word of the message. The translation to a routing address is implementation dependent.

The network interface is controlled by two commands, **SEND** and **NEXT**. The **SEND** command forms a message out of the contents of the output registers and then 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 stores it into the input registers. The mechanism by which the processor issues these commands is implementation dependent.

2.1.1 Flow Control

The network interface, together with the network, enforces flow control at the sending processor. If the receiving proces-

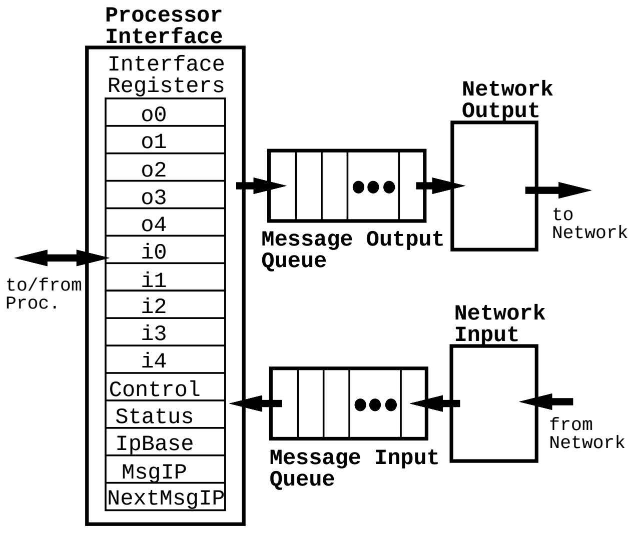


Figure 1: State associated with the network interface.

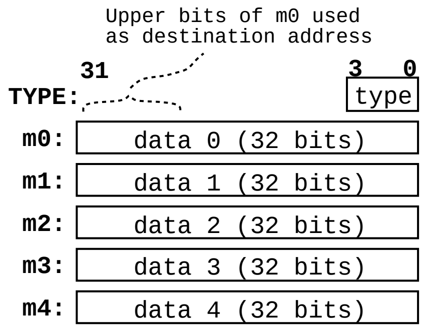


Figure 2: The message format defined by the architecture.

sor 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 processor's output queue becomes full, no more messages can be sent by that processor. Depending on the **CONTROL** register settings, an attempt to send a message when the queue is full will either signal an exception or stall the processor until the output queue empties. Stalling the processor should not be done if the processor needs to participate in emptying the network.

2.1.2 Variable Length Messages

The basic architecture we have outlined can only handle 5-word messages. However, the architecture could easily be extended to send and receive variable length messages including infinite length systolic streams.

To handle messages longer than 5-words, we can view the input and output registers as scrolling windows which expose to the programmer a 5-word segment of the message. An additional command, **SCROLL-IN**, scrolls the next five words of an incoming message into the input registers. Analogously, **SCROLL-OUT**, sends the five message words in the output registers and continues composing the same message.

type: —	
word0:	address of the requested location
word1:	FP of the thread which will handle reply
word2:	IP of the thread which will handle reply
word3:	—
word4:	id of a remote read request message

Figure 3: The message format of a remote read request.

2.1.3 Protection

Although none of our performance studies measures the cumulative performance of multiple applications, it is important to note that our basic architecture could be easily extended to handle a multi-user environment. Moreover, the necessary extensions would not affect the optimizations which we will propose.

There are two types of messages which require special handling in a multi-user environment: messages destined for the operating system, and messages destined for processes other than the currently active process. Messages destined for the operating system must be treated as privileged messages. To guarantee protection, the arrival of a privileged message could interrupt the processor or the message could be stored in privileged state until the operating system can handle it.

There are two basic mechanisms for handling messages destined for inactive processes. If all processors context switch synchronously, or time-slice, then such messages can be avoided by draining the network in between time-slices. This is the strategy used by the CM-5[LAD$^+$92]. On the other hand, if each processor context switches independently, then each message must be tagged with its process identification number (PIN). As a message arrives, the network interface will check whether its PIN matches the PIN of the currently active process. If not, the network interface will treat the message as a privileged message.

2.1.4 Example

We conclude our description of the basic architecture with a simple example - a remote read message. Consider the following remote read protocol. When a computation thread needs a remote value, it sends out a non-blocking remote read request message. This message carries the address of the remote value as well as the identity of the thread to be invoked when the reply value arrives. On receiving this message, the destination processor reads the requested location from its local memory and sends a reply. When the reply message arrives, the thread whose identity is supplied by the message is invoked. This thread saves the reply value and reschedules any threads waiting for the reply.

Let us consider one convention for remote read and reply messages. Figures 3 and 4 show the assumed format for these messages. For all messages, the type of message being sent is identified by the fourth word of the message. All other words of the messages are interpreted differently for each type of message. In case of a remote read request, the first word of the message contains the remote address of the requested memory location; Words 1 and 2 identify the thread which will handle the reply message. Word 1

type:	—
word0:	FP of the thread which will handle reply
word1:	IP of the thread which will handle reply
word2:	reply value
word3:	—
word4:	id of a reply message

Figure 4: The message format of a remote read reply.

contains the procedure frame pointer (FP) of the thread; Word 2 contains the instruction pointer (IP) of the thread. In case of a reply message, the first two words send back the FP and IP of the thread which will handle the arrived message; Word 2 contains the reply value.

Let us consider what happens as a remote read request message advances into the input registers of the destination processor's network interface. Figure 5 shows the polling, dispatching, and processing of the arrived remote read message as performed by the destination processor. Each line in this basic sequence corresponds to one logical step which must be taken to process a remote read request. It does not directly correspond to any number of processor instructions. For instance, the explicit reading and writing of network interface registers is not meant to imply memory loading and storing. As we will see in Sections 3 and 4 the number of instructions for each step will vary with each network interface implementation.

The processor first polls to see if a new message has arrived (Lines 1–2) It checks a bit in the network interface's **STATUS** register to determine if the input registers contain a valid message. If so, the processor goes on to dispatch the arrived message (Line 3) This dispatch routine computes the instruction pointer of the message handling thread using the message's id (Lines 4–5) and jumps to it (Line 6). In our example, the processor jumps to the remote read message handler.

The remote read message handler generates and sends a reply to the read request. Lines 7–10 copy from the input message into the output message the FP and IP of the thread which will serve as the message handler for the reply value. Lines 11–13 read the requested memory location and write its value to output register **o3**. Lines 14–15 generate the message id of a remote read reply message and write the id to output register 4. Finally, Lines 16–17 send the reply message and load the next incoming message into the network interface input registers.

```
 poll:
1.   read STATUS into stat
2.   check a flag in stat to find out if there is a message
     in the input registers
3.   if there is a message, goto dispatch

 dispatch:
4.   read i4 into msg_id
5.   compute address of message handler from msg_id
6.   jump to the message handler

 remote read message handler:
7.   read i1 into reply_FP
8.   write reply_FP into o0
9.   read i2 into variable reply_IP
10.  write reply_IP into o1
11.  read i0 into variable address
12.  load from memory address into variable value
13.  write value into o2
14.  generate the reply_msg_id
15.  write reply_msg_id into o4
16.  SEND
17.  NEXT
```

Figure 5: The logical sequence of actions necessary to receive and process a remote read request using only the basic architecture.

```
 poll and dispatch:
1.    read MsgIP into msg_ip
2.    jump to msg_ip

 remote read message handler:
3.    read i0 into variable address
4.    load from memory address into variable value
5.    write value into o2
6.    SEND -reply type=reply_msg_id
7.    NEXT
```

Figure 6: The logical sequence of actions necessary to receive and process a remote read request using proposed architectural optimizations.

2.2 Optimized Architecture

Having described a basic network interface architecture, we next extend this architecture with several optimizations. We illustrate the benefits of these optimizations by rewriting our example using these optimizations. Figure 6 shows the new and shorter sequence of steps necessary to receive and process a remote read request. We will refer to this optimized handler throughout this section.

2.2.1 Encoded types

We have seen in the previous example that each message had to identify its message handler. When sending a message the processor had to explicitly generate a 32-bit identifier and store it into the fourth word of each message (Lines 14–15.). Our first optimization will remove this overhead for the most common message types. We will allow the processor to supply a 4-bit compile-time constant, the message type, as part of each **SEND** instruction. This new 4-bit message field will be transmitted with the rest of the message and will show up in the **STATUS** register when the message is being processed.

Since, in many message passing models the number of frequently invoked handlers is relatively small, most messages will no longer need to generate and send a 32-bit identifier. In systems where there are more message identifiers than can be represented in four bits, an "escape" type can be be stored in the four bit field when one of the less common message types is to be sent.

Lines 14–15 of the basic example can now be omitted. Instead we can specify the type of the message directly in the **SEND** command (Line 6 of the optimized handler).

2.2.2 Fast Reply/Forward

Our next optimization optimizes the common case of replying to or forwarding part of a message. When replying to the remote read request in our example, the processor had to explicitly copy fields from the incoming message into fields of the outgoing message (Lines 7–10). In general, every handler which replys to a message will have to copy the FP/IP pair of the reply handler from the incoming message to the outgoing message. Similarly, every handler which forwards part of a message will have to copy the data fields from the incoming message into the outgoing message.

In order to avoid this overhead, we have defined two special modes of the **SEND** command: **REPLY** and **FORWARD**. When one of these modes is used, the **SEND** command composes an outgoing message using certain input registers in place of certain output registers, thus removing the need to copy. For example, in the **REPLY** mode, the **SEND** command composes a message using registers **i1** and **i2**, in place of **o0** and **o1**.

This feature allows us to remove Lines 7–10 of the basic example and replace them by the "-reply" option to the **SEND** command (Line 6 of the optimized handler).

2.2.3 Hardware Assisted Message Interpretation

The basic architecture we have described so far dispatches arrived messages completely in software. To dispatch an arrived message, the software must 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. These steps correspond to Lines 4–6 of the basic 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.

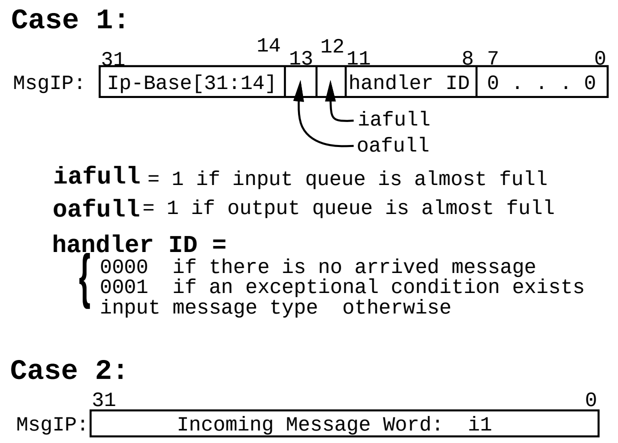
We achieve hardware efficiency via an additional register, the **MsgIp** register. The **MsgIp** register precomputes in hardware the instruction address of the handler for the current input message. To dispatch an incoming message to the correct message handler, we only need to jump to the contents of the **MsgIp** register.

To compute the address of the handler we must have some conventions on where the handlers are stored. First, we use an additional register: **IpBase**. This register should be loaded with the starting address of the area in which the message handlers are stored. To compute **MsgIp**, the network interface replaces certain bits of the **IpBase** register with the type bits of the arrived message. This allows each message type to dispatch to a different handler. We must also handle messages, such as the remote read reply message, which specify the address of their handler in the message. We mark these messages type 0. For type 0 messages, the **MsgIp** register assumes that the address of the handler is given by Word 1 of the message, and that is the value it returns.

As described so far, before using the value in **MsgIp** we must first check to make sure that a message has arrived. (This corresponds to Lines 1–3 in the above example.) When there is no new message, **MsgIp** will return a special handler address, corresponding to replacing bits of the **IpBase** register with type bits 0000.

Using these features, we can now replace Lines 1–6 of the example by simply reading and jumping to **MsgIp**. These commands are shown in Lines 1–2 of the optimized handler.

The code of the optimized handler may experience delays after Lines 1–2. There may be a delay after Line 1 caused by



Case 1:
31
14
13
12
11
8
7
0
MsgIP:
Ip-Base[31:14]
handler ID
0 . . . 0
iafull
oafull
iafull = 1 if input queue is almost full
oafull = 1 if output queue is almost full
handler ID =
0000 if there is no arrived message
0001 if an exceptional condition exists
input message type otherwise
Case 2:
31
0
MsgIP:
Incoming Message Word: i1

Figure 7: 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.

the latency between the read and use of the **MsgIp** value. The jump at Line 2 may cause additional delay because we cannot fill any delay slots after the jump. We can mask these delays by overlapping the processing of one message with the dispatching of the next message. However, without further optimizations, this is not always possible. In our example, we cannot read **MsgIp** of the next message until the next message has advanced into the input registers. This does not happen until the last instruction of this handler.

With an additional register, **NextMsgIp**, we can fill additional delay slots in the dispatch routine. **NextMsgIp** computes the handler address for the next message, just as the **MsgIp** computes it for the current message. Dispatching to **NextMsgIp** instead of **MsgIp**, allows the software to always overlap the processing of one message with the dispatching of the next.

2.2.4 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. For example, 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 very inefficient. We have not even included this step in our basic handler. To do this efficiently, we provide hardware assisted checks through the **MsgIp** register.

Figure 7 illustrates the computation of **MsgIp**, including the checking of queue lengths. Whenever the input queue length or the output queue length exceeds a certain threshold, the **iafull** or the **oafull** bit in the **MsgIp** register gets set. The queue threshold at which these bits get set can be set independently for each queue in the **CONTROL** register. We have chosen to define four versions of each message handler instead of defining two new message handlers to handle the special condition of a queue exceeding its threshold. While this choice uses up more memory to hold the handler code, it allows each message handler to independently decide how to respond to these conditions. For instance, a message handler that does not send any new messages may ignore the state of the output queue. A short message handler may 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 (such as an error in the message input port). The hardware also checks for and reports these exceptional conditions through the **STATUS** register. To be able to report exceptional conditions through the **MsgIp** register, we disallow messages of type 1. Whenever there is an exception, the four **handler ID** bits of **MsgIp** are set to 0001. This forms an instruction pointer to the exception handler. The exception handler can then check the **STATUS** register to see precisely which exceptional condition has occurred.

3 Three Implementations

Section 2 described the programmer's view of a network interface, not the implementation. This section will suggest several possible implementations, and outline their costs and benefits. We will refer to these implementations in our performance studies of Section 4.

The architecture we have described in the previous Section could be implemented in several different ways depending on:

- the encoding of network interface commands into processor's instruction set, and
- the mapping of network interface registers into processor's storage.

There are many possible ways in which a given implementation could encode network interface commands. These commands could be incorporated into the processor's instruction set. They could be assigned new opcodes or assume reserved coprocessor opcodes. Since these commands, including **SEND**'s 4-bit type and forward/reply mode, take up only seven bits, they could also be incorporated into the unused bits of many existing instructions. For implementations which do not wish to modify the processor instruction set, the **NEXT** and **SEND** commands could be memory mapped. A store into a certain memory location(s) could initiate a **NEXT** or a **SEND** command.

There are also several ways in which a given implementation could access the network interface registers. We have already discussed several of these in our survey. The interface registers could be included in the processor register file and accessed by the register field of any instruction. They could also sit in separate storage and be accessed via new opcodes or reserved coprocessor opcodes. Finally they could be mapped to certain memory locations in the processor cache or main memory.

For example, the 88110MP microprocessor [Gre92] [Bec92] contains an on-chip network interface with dispatch hardware optimizations similar to those described in this paper. The 88110MP network interface sits on the processor's

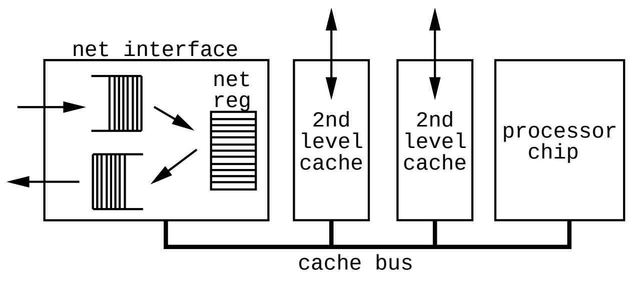


Figure 8: An off-chip cache based implementation of the network interface.

Address Lines	Information
5:2	interface register number
9:6	type of message to be sent
11:10	01 – SEND command 10 – SEND reply command 11 – SEND forward command 00 – Don't perform a send
12	NEXT command

Figure 9: Encoding of network interface commands and register number into a memory address.

source and write-back buses along with all the other functional units. The 88110MP is dual issue and the network interface can execute two coprocessor network instructions per cycle. Each network instruction can send two values or read one value from the interface.

We have chosen three likely implementations for our performance studies. These are:

- Off-chip cache mapped interface registers with memory mapped commands.
- On-chip cache mapped interface registers with memory mapped commands.
- Register-file mapped interface registers with commands encoded in unused bits of triadic instructions.

Each of our three implementations has different design implications in terms of processor chip real estate and complexity of processor design. The instruction sets of all three implementations are based on the 88100 Motorola processor - a typical RISC processor. We believe that other RISC instruction sets should yield similar performance results.

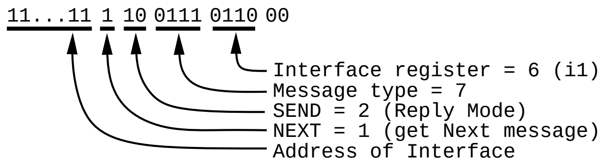
3.1 Off-Chip Cache-Based Implementation

The first implementation we consider maps the network interface into a second level off-chip cache (Figure 8). This interface becomes another chip on the processor's external data cache bus. The interface chip follows the cache interface protocol and behaves, from the viewpoint of the processor, as another data cache chip. A processor's load or store instruction is processed by the interface if the upper bits on the address bus match a preset constant. In a single load or store instruction, the processor can do any combination of the following: access one interface register, execute a **SEND** command, and execute a **NEXT** command. The command to the network interface is encoded in the low order bits of the memory address, as shown in Figure 9.

Consider a typical RISC load instruction:

```
load r3 r1 C
```

This instruction sign extends the 16-bit constant, `C`, to a 32-bit value and adds it to the contents of register `r1` in order to compute a memory address. The processor outputs this address on the address bus. The value returned on the data bus is then stored in register `r3`. Consider the case where the address `r1 + C` transmitted along the address bus is:



Assume that the address to which the interface is mapped consists of all 1's. This load request will activate the network interface instead of a data cache. The low order bits of the address will direct the network interface to:

- return the contents of the sixth interface register, `i1`, to the processor along the data bus,
- send a reply message of type 7, and
- load its input registers with the next message.

This interface is easy to implement; of the three network interface implementations which we present in this section, this is the only implementation which requires no modifications of the processor chip. We designed such a network interface chip, NIC, for use with the 88100 processor. We designed, simulated, and thoroughly tested NIC at the RTL level[HJ91].

The modularity of NIC's design is both a strength and a weakness. Because this network interface is not part of the processor chip, it is slower than an on-chip interface. Access to an off-chip interface is likely to take several processor cycles. For example, in the 88100 processor, a loaded value cannot be used in the two cycles following the load. We expect this weakness to get worse in the future as processor speeds further outpace off-chip memory latency.

3.2 On-Chip Cache-Based Implementation

The second implementation we consider maps the network interface into an on-chip cache. This implementation is identical to the previous one, except that the network interface now sits on an internal data cache bus rather than an external one (Figure 10).

This network interface implementation is only slightly more intrusive than the previous one. Although, we have added a new module - the network interface - to the processor chip, we have not modified the processor core. The network interface only communicates with the rest of the processor via its internal cache bus. The processor's instruction set, its control, and its datapaths remain unchanged. Moreover, because the network interface is on-chip, network interface accesses are somewhat faster. Access to an on-chip interface is likely to take only a single cycle.

The cost of this approach is that additional I/O pins and chip area are needed on the processor. Most of the area taken up by the network interface is memory, namely the message queues and, to a lesser degree, the interface registers. The area needed for control, including the proposed optimizations, is negligible. If, for example, each message

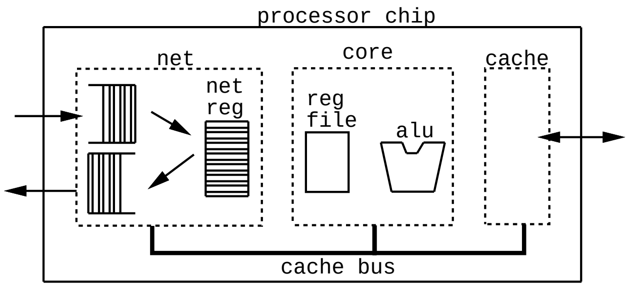


Figure 10: An on-chip cache based implementation of the network interface.

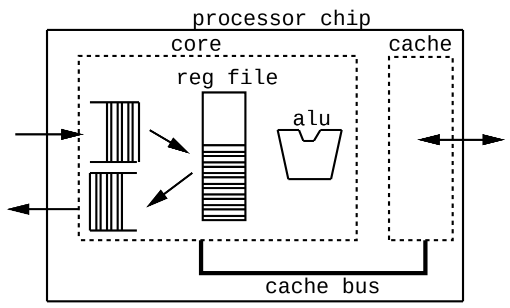


Figure 11: A register-file based implementation of the network interface.

queue is 16 messages long, the total memory needed is about 3/4 of a kilobyte. This corresponds to only a small fraction of the die on current microprocessors.

3.3 Register-File-Based Implementation

The final network interface implementation that we consider maps the network interface into the processor's register file. The network interface registers take up part of the register file and can be accessed as any other scalar register. The network interface commands, **NEXT** and **SEND**, are encoded into the unused bits of every triadic (three-register) 88100 instruction. Figure 11 illustrates the register-file-based implementation.

Although the cost of this implementation in terms of processor size and I/O count is comparable to that of the on-chip cache version, the increase in design complexity is much greater. Comparing Figure 11 to Figure 10, we see that this implementation is more intrusive than the previous implementation. The processor's decoder must be modified to forward **SEND** and **NEXT** fields of each triadic instruction to the network interface module. Moreover, some register file registers must contain an additional port. Network interface input registers must contain one more write port from the input queue; network interface output registers must contain one more read port to the output queue. Since the decode and register read stage of today's RISC processors tends not to be the critical one, these modifications are realistic. However, to the degree that these modifications increase the complexity of the processor design, their performance benefit must be carefully weighed against increased design time.

Of the three interfaces we have considered, the register-file-based interface is the most efficient. As long as other useful work can be performed in the same instruction, it takes no additional cycles to access up to three network interface registers and to send either or both commands to the network interface. For instance, consider the following triadic 88100 instruction:

add o1 i1 i2, **SEND** type=5, **NEXT**.

In one cycle, this instruction adds the contents of two input registers, stores the result in an output register, sends a message of type 5, and advances a new message into the input registers. The preceding memory mapped implementations require four 88100 RISC instructions to accomplish the same.

4 Performance Evaluation

This section examines the performance implications of our architecture. We consider the effects of two variables: the placement of the interface (off-chip cache, on-chip cache, register) and the presence or absence of our optimizations. To evaluate the impact of different interface placements, we consider the three implementations described in the previous section. To evaluate the impact of our proposed optimizations, we consider each implementation with and without the optimizations of Section 2.2. Without our optimizations, the memory mapped implementations of Sections 3.1 and 3.2 correspond closely to existing network interfaces (Section 1.2) and form a good basis for comparison.

We take two approaches to evaluating the performance of network interfaces. First, we compare the dynamic number of RISC cycles necessary to send, dispatch, and handle typical types of messages. Second, we evaluate the dynamic cycle counts of two parallel, scientific programs.

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 number of cycles necessary to perform each task. We derive these numbers from code segments which we have handwritten for the 88100 Motorola RISC processor.

We include here the instruction counts for several common types of messages. These include all message types necessary to communicate arguments and results between procedures, to access remote memory, and to access remote memory with presence bits. They are:

Send The **Send** message is the most general message we consider[2]. It is used to send procedure arguments and results and replies to all other message requests. The **Send** message invokes a thread whose procedure frame pointer and instruction pointer are supplied by the message. By convention, this thread first stores 0, 1, or 2 words of the message in the procedure frame. We do not consider its behavior after this point since it varies with each thread. An example of a **Send** message was the reply to the the remote read request, as described in Section 2.1.4.

[2]It is identical to the Start message in the proposed Start architecture [NPA92]

Read The **Read** message is used to request a read of a remote memory location. We showed the message handler for this message as an example in Figures 5 and 6.

Write The **Write** message writes a value to a remote memory location.

PRead The **PRead** message requests the value in a remote array location with presence bits. If the word has already been written (the presence bits are set to "full") the receiving handler replies right away. If the word is yet to be written, the handler defers the **PRead** request. The response will be sent as soon as the word is written. **PRead** and **PWrite** are used to implement I-Structures [ANP89].

PWrite The **PWrite** message writes a value to a remote array location with presence bits. If there are deferred readers waiting for the value, the **PWrite** message handler forwards the value to each of the n deferred readers.

Table 4.1 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 comparison somewhat favors the basic implementations in that the basic implementations, unlike the optimized implementations, do not check for abnormal conditions when dispatching a message.

In the sending of messages using the register based model, the number of instructions needed may depend on whether the values in the message can be computed directly into the output registers, or whether they must be saved for future use. In these cases we have given a range of values. We expect that the cost will typically be in the low to middle part of this range.

Table 4.1 shows substantial improvement from a basic, off-chip implementation to an optimized, on-chip implementation. Several instructions are saved when sending out a message. These savings come from not having to write out the message type in a separate instruction and from not having to store the message fields to memory. Typically, the largest number of instructions are saved in dispatch. Most of these savings come from using the hardware assisted message interpretation. Savings in processing a message can be attributed to the fast reply/forward modes, and to not having to load (and store) message words from memory.

It is important to note that the results of Table 1 are independent of the particular program being executed and the granularity of parallelism in the programming model. Any program which sends these types of messages will experience these savings.

4.2 Impact of Interface on Program Speed

The different software costs of sending, dispatching, and processing a message impact the final speed of a parallel program. In this section, we estimate this impact by comparing the total number of RISC cycles to execute a parallel program under each of the six models.

We have examined the dynamic cycle counts of several scientific programs. We report the results of two: Matrix Multiply and Gamteb; the rest give similar results. The matrix multiply program subdivides matrices into 4 by 4 blocks and computes their products. Gamteb performs a Monte Carlo photon transport simulation. Both programs have been written in Id [Nik90], a high level imperative language which implicitly uncovers parallelism.

Action	Message Type	Network Interface Implementation					
		Optimized			Basic		
		Register Mapped	On-chip Cache	Off-chip Cache	Register Mapped	On-chip Cache	Off-chip Cache
SENDING	Send (0 words)	2	3	3	3	4	4
	Send (1 word)	2-3	4	4	3-4	5	5
	Send (2 words)	2-4	5	5	3-5	6	6
	PRead	2-4	5	5	3-5	7	7
	PWrite	0-3	3	3	1-4	5	5
	Read	2-3	4	4	3-4	6	6
	Write	0-2	2	2	1-3	4	4
DISPATCHING	-	1	2	2	5	7	8
PROCESSING	Send (0 words)	1	1	3	1	1	3
	Send (1 word)	2	3	5	2	3	5
	Send (2 words)	3	5	6	3	5	6
	Read	1	3	5	4	8	8
	Write	1	3	4	1	3	4
	PRead (full)	9	12	13	12	17	17
	PRead (empty)	19	23	23	19	23	23
	PRead (deferred)	15	19	19	15	19	19
	PWrite (empty)	14	17	17	14	17	17
	Pwrite (deferred)	15+6n	19+8n	19+8n	16+6n	20+8n	20+8n

Table 1: The number of 88100 RISC processor cycles 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.

4.2.1 Environment

Both programs have been compiled to Berkeley's Threaded Abstract Machine (TAM) [CSS$^+$91]. TAM is targeted towards performing fine-grain parallel computations. Both programs were compiled so that any two procedure invocations would communicate across the network. To illustrate the grain size, there were, on average, 3 floating point operations performed for every message sent in our matrix multiply program.

We computed the total number of TAM instructions executed by each parallel program using a TAM instruction set simulator developed at Berkeley. This simulator reported the number of times each TAM instruction, or class of TAM instructions was executed. It computed these results by sequentially executing each thread of the parallel program. The simulator did not model any number of processors or any network latency. Both of these could introduce idle cycles due to communication delay and load imbalance. Our results do not include any idle cycles.

We determined the ratio of deferred, full, and empty PReads and PWrites by running the same programs on a simulator of the Monsoon dataflow processor, Mint, using LIFO scheduling of dataflow tokens [Sha89].

4.2.2 Scope of Program Speed Results

The results of this section are more limited in scope than the instruction counts reported in Table 1. First, as with any benchmark results, they only report the performance of specific programs. In additions, they reflect the types and frequencies of messages used by the TAM model running Id. However, we think that these programs are fairly typical of fine-grain parallel programs and expect that results from other fine-grain parallel models should be similar.

For coarser grained models the message types and frequencies may be substantially different from those of our fine-grain model; so the results of this section do not directly apply. But the results of Table 1 are still relevant. Namely, register based implementations will out perform non-register based implementations, and implementations with our optimizations will outperform those without. How much of an effect these choices have will greatly depend on the model.

4.2.3 Program Speed Results

Figure 12 shows the total number of 88100 instruction cycles for the two programs under each network interface model. We computed these by simulating each program and replacing the dynamic instruction count of each TAM intermediate instruction by the appropriate number of RISC instructions.

Each bargraph in Figure 12 is divided into several components. The bottom component corresponds to the total amount of cycles spent performing non-message passing work. The number of cycles spent performing communication related work is broken into two parts. The lower of the two components shows the amount of time spent dispatching. The top component shows the time spent on all other communication work, namely the sending and receiving of message values.

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 PRead, is under 10%; each sending and each receiving of a message expands into a large number of RISC instructions. As a result, in some experiments, more time is spent performing communication instructions than non-communication instructions.

More importantly, the gains achieved by using an opti-

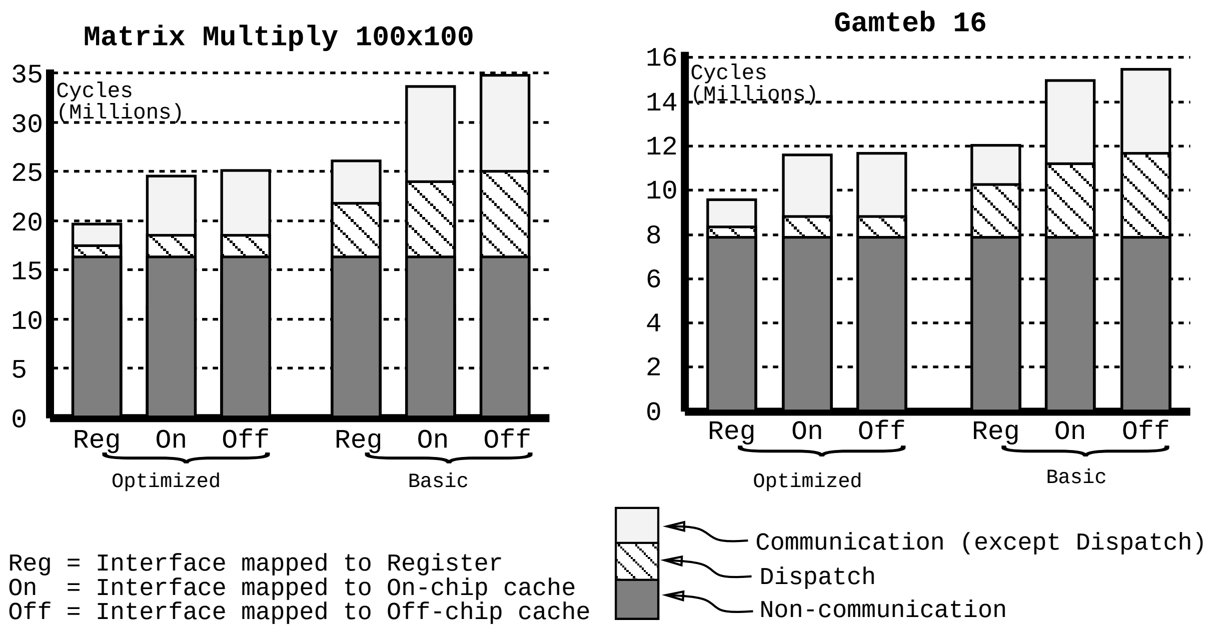


Figure 12: Dynamic instruction counts for 100 by 100 matrix multiply and 16 Gamteb using the six different network interface implementations.

mized, register-based interface as opposed to an unoptimized off-chip interface are substantial. The cost of sending, reading and dispatching messages decreases as much as five fold as we optimize the network interface and incorporate it into the register file. Total execution time of these fine-grain parallel programs is cut by about 40%. And the percentage of the execution time due to message passing overhead is reduced from 51% to only 17%. This result supports our hypothesis that substantial performance gains can be achieved relative to existing processor-network interfaces.

Other interesting results can be obtained by contrasting the gains due to placement versus the gains due to optimizations. Although both significantly cut down on network overhead, we see that hardware optimizations of the network interface are more important than the actual placement of the interface. Even the slowest optimized implementation is better than the fastest unoptimized implementation. This result suggests that effective parallel computers can be built with existing processor designs. Simply by attaching an optimized network interface to the cache bus, today's processors can considerably lower their network interface costs.

Although the optimizations currently seem somewhat more important than placement, we do not expect this to remain true for long. In Figure 12, the performance of the on-chip and off-chip memory mapped interfaces are fairly close. But as processors get faster, the latency of performing an off-chip operation will increase. Figure 12 assumes a two cycle latency for reads from the off-chip interface. If, however, the latency is increased to 8 cycles instead of 2, then the communication costs of the off-chip optimized model will double. As a result, relegating the network interface off-chip will not remain a viable alternative for future generations of multiprocessors.

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 message passing 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 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 as compared to that of existing network interface designs. By making use of our hardware mechanisms, and by mapping the network interface into the processor's general-purpose registers, we have been able to reduce communication overhead about five fold in our benchmarks.

Another significant outcome of our study is that, for the TAM execution model, most of the performance gain comes from our hardware mechanisms rather than from the placement of the network interface inside the register file. Even if the network interface resides off-chip, our hardware mechanisms improve its performance two fold. We conclude that even existing processors could significantly benefit by incorporating our hardware mechanisms into an off-chip network interface. We have designed and simulated at low level such an interface, NIC, for the Motorola 88100 processor. Influenced by our work, our industrial partner, Motorola, is implementing a similar network interface on-chip in an experimental version of the 88110 processor.

Acknowledgments

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 C. Kuszmaul who has helped us develop many of the ideas in this paper and has painstakingly proofread our draft. We thank the members of the TIM design team who have also 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 TL0 compiler and simulator. 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.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, pages 598–632, October 1989.

[BCC+90] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proc. of the 17th ISCA*, June 1990.

[Bec92] Michael J. Beckerle. An Overview of the START (*T) Computer System. Technical Report MCRC-TR-28, Motorola CRC, Cambridge, MA, July 1992.

[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.

[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 *Proc. of the Fourth ASPLOS*, 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.

[Gre92] Robert J. Greiner. *88110 Message Coprocessor User's Manual.* Motorola Computer Group, February 1992. DRAFT.

[Hil85] Daniel W. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[HJ91] Dana S. Henry and Christopher F. Joerg. The Network Interface Chip. Technical Report CSG Memo 331, MIT Laboratory for Computer Science, Cambridge MA 02139, June 1991.

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

[LAD+92] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *SPAA*, pages 272–285, June–July 1992.

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

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

[Nug88] Steven F. Nugent. The iPSC/2 Direct-Connect Communications Technology. In *Proc. of the Third Conf. 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 *Proc. of the Third Conf. 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.

[Sei85] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, pages 22–33, January 1985.

[Sha89] Andy Shaw. A Design and Implementation of MINT: A Monsoon Simulator. Technical Report CSG Memo 297, MIT Laboratory for Computer Science, Cambridge, MA, June 1989.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th ISCA*, May 1992.

[WS85] Colin Whitby-Strevens. The Transputer. In *Proc. of the 12th ISCA*, pages 292–300, June 1985.