LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

*T: Integrated Building Blocks for Parallel Computing

CSG Memo 351
July 28, 1993

Gregory M. Papadopoulos

G. Andy Boughton

Robert Greiner

Michael J. Beckerle

To appear in Supercomputing '93, November 15-19, Portland, Oregon.

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

*T : Integrated Building Blocks for Parallel Computing

Gregory M. Papadopoulos[1]
G. Andy Boughton[3]
MIT

Robert Greiner[2]
Michael J. Beckerle[4]
Motorola, Inc.

Abstract

In this paper we present two hardware components for high performance parallel computing: a superscalar RISC microprocessor with an integrated 400 megabytes/sec user-level network interface (the 88110MP), and a companion 8×8 low-latency packet router chip (ARCTIC). The design point combines very low message overhead and high delivered communications bandwidth with a commercially competitive sequential processor core. The network interface is directly programmed in user mode as an instruction set extension to the Motorola 88110. Importantly, naming and protection mechanisms are provided to support robust multi-user space and time sharing. Thus, fine-grain messaging and synchronization can be supported efficiently, without compromising per-processor performance or system integrity. Preliminary performance modeling results are presented.

1 Introduction

There seems to be little debate that the future of high performance computing, and perhaps all computing, is MIMD parallelism [4, 16]. But there is also little debate that, at least in the near-term, the technology and marketing factors that drive the design of the individual processors will be their suitability as components as *single* processors in workstations, PC's, and *small* shared memory multiprocessor servers, not as components in large scale parallel

[1,3]MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA; greg@mit.edu, gab@lcs.mit.edu
[2]Motorola Computer Group, 2900 Diablo Way, Pheonix, AZ, USA: bobg@matl.phx.mot.com
[4]Motorola Cambridge Research Center, One Kendall Square, Cambridge, MA, 02139, USA: mikeb@mcrc.mot.com

machines or even workstation clusters. Unfortunately, this also means that these hot microprocessors are often less than ideal processing elements for large scale MIMD parallelism; the principal and most obvious shortcoming is a lack of mechanisms that are tailored for scalable interprocessor communication and synchronization. The result: individual nodes will offer highly competitive performance on sequential streams that exhibit good locality, but communication and synchronization among nodes will have substantial execution overhead.

The goal of the *T project (pronounced 'Start') is to bring very low cost, fine grain communications to high performance sequential processors. We have accomplished this by extending an existing superscalar microprocessor, the Motorola 88110, with an integrated network messaging unit. The design approach is very simple. We have added two new sets of register files, a transmit and a receive register file, and provided new instructions for moving data between these and existing floating point and integer registers, for launching messages into the network and for consuming incoming messages. We have also added extensive support for a variant of active messages and fine grain multithreading we call *microthreading.*

While this degree of explicit register-level coupling of a network within a processor datapath is not new (*cf* the J-machine MDP [7], iWarp [3], Monsoon [19, 20, 21], Epsilon [10] and the EM-4 [22]), we believe we have made significant contributions in accomplishing the integration in a way which is consistent with superscalar execution and, most importantly, which supports UNIX-style protection and naming. In fact, most of our architectural effort has been invested in supporting a user mode network interface, essential for our performance goals, that works within a protected multitasking, multi-user environment, and in dealing with the vagaries of speculative, multiple issue instruction execution.

A companion hardware component is an 8×8 packet routing chip that emphasizes high sustained network throughput while minimizing per-hop latency. The ARCTIC chip builds upon our previous experience with the very successful PaRC router [12] used in Monsoon and similarly provides sophisticated input packet buffer management, virtual cut-through routing, and robust error detection. In addition, ARCTIC provides specialized support for randomized routing on fat-tree networks, the network topology used in *T systems. Each ARCTIC link is a 16-bit wide, 100 Mbaud GTL (a low voltage swing CMOS technology), yielding a raw data rate of 200 Megabytes/sec/link. Our modified processor, the 88110MP, also directly drives two compatible GTL links, for an aggregrate peak per processor performance of 400 Megabytes/sec — equivalent to the existing local bus bandwidth of an unmodified 88110.

2 The High Cost of Communicating

The unavoidably high transport latency of interprocessor communications is often identified as a fundamental issue in any form of multiprocessing [2]. While this ultimately may be true, the fact is that in commercial parallel and distributed systems, the *overhead* of interprocessor communication dominates communication latency. That is, a processor will typically expend many more cycles on network protocol and in driving the network interface hardware than is incurred by the actual transport of a packet from one processor to another.

Figure 1 illustrates the relative contributions of network processing overhead and transport latency for the transmission of a single short message on several commercial and research machines. The overhead accounts for the sum of the time spent by the sending processor to format and launch a short packet into the network plus the time spent by the receiving processor to accept the packet and to dispatch to user code to handle to the packet (but not the time spent in the user code handler). The transport latency is the time spent from when the head of the packet leaves the sender to when the tail of the packet arrives at the receiver. In the figure, the various machines are sorted by their per-node peak floating point performance, a (very) rough measure of sequential node performance.

Rather than measuring transport latency and overhead in microseconds, we instead use floating point operation times: the total number of floating point issue opportunities (at peak) that were expended in communications activities. This is a useful metric because it reflects the cost of communications as computational energy that could have been applied to "useful" work, and normalizes across implementation technologies. In effect, it penalizes machines that have increased their floating point performance without making commensurate improvements in the network.

Observe that for the commercial machines the overhead for communication completely dominates the transport latency.[1] Most of this overhead is either due to user-kernel interactions for kernel-mode networks (Paragon, Meiko[2]), unoptimized message library software CM5), or a basic mismatch between the built-in communications mechanisms and message-passing (KSR1). We note that active messages [25] can substantially reduce the software overhead for those machines that provide a user mode network interface — for example the message overhead on the CM5 has been reduced to three microseconds using active messages. Still, we consider this well over an order of magnitude too high for fine grain message passing.

In contrast, the research machines have obtained much better levels of communications performance by employing novel processor architectures with deeply integrated network interfaces. But as interesting as these processor architectures might be in their own right, they clearly do not represent the overwhelming trend towards parallel machines built from commodity superscalar RISC microprocessors.

Even so, there are many lessons to learn in the way that the network is integrated into the processor pipelines of these research architectures. An appropriate view of one of the primary contributions of the *T effort is the assimilation of the network interface ideas — especially those from Monsoon (and the other multithreaded dataflow architectures), the J-machine, Alewife [1] and iWarp — into mainstream processor architecture. We have also been influenced by the network interface study of Henry and Joerg [11].

[1]The product literature [5] for Intel Paragon describe this as a feature. Intel labels the Paragon a "flat" communications architecture because the total time to communicate between two nodes is, to first order, independent of the network distance between them. That is, the software overhead so dominates the communication time that the transport latency is immaterial.

[2]The Meiko CS-2 supports direct user mode interaction with a network interface processor, but a substantial fraction of the overhead (about ten microseconds) is attributable to memory protection mechanisms enforced by the network interface.

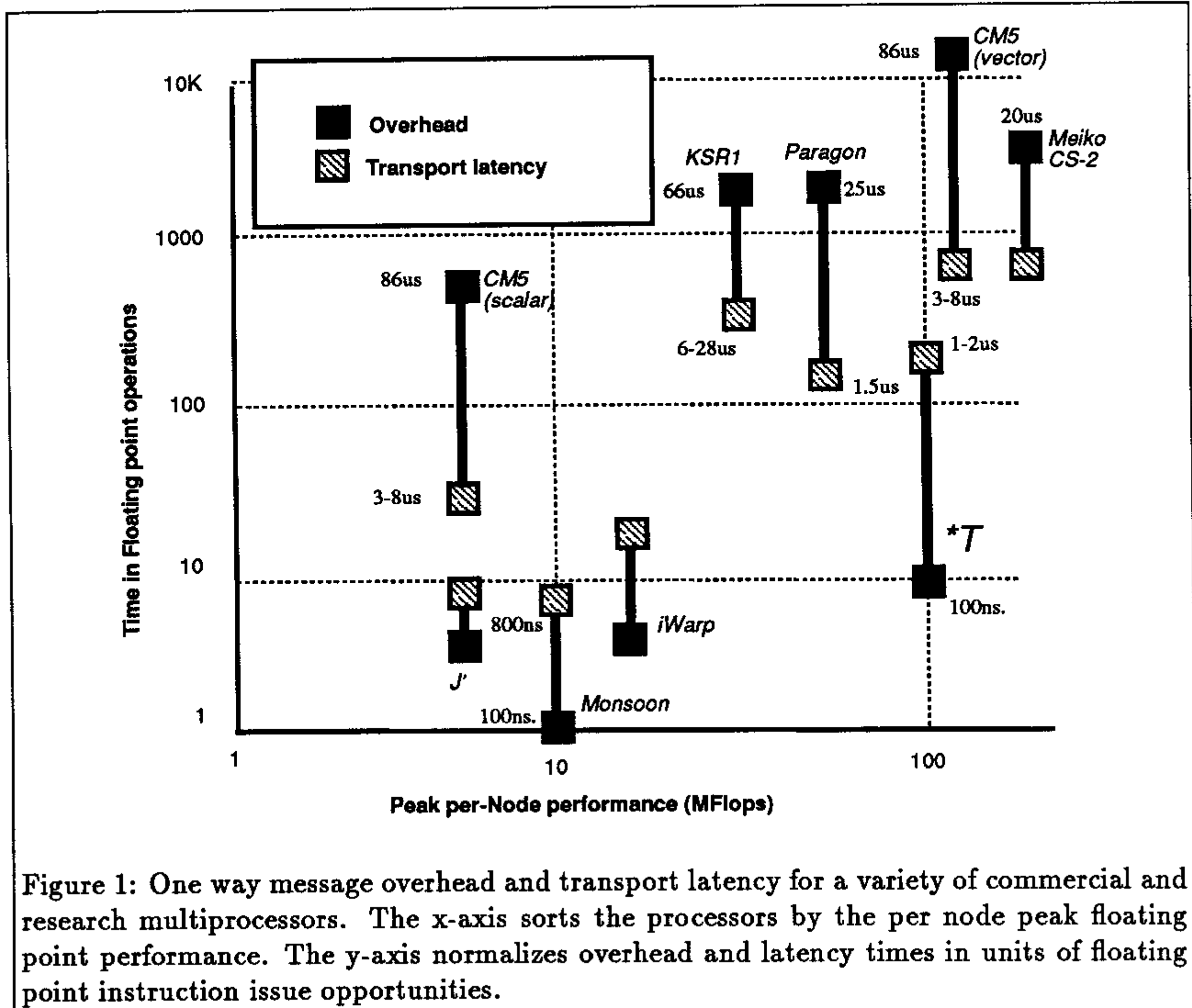

Figure 1: One way message overhead and transport latency for a variety of commercial and research multiprocessors. The x-axis sorts the processors by the per node peak floating point performance. The y-axis normalizes overhead and latency times in units of floating point instruction issue opportunities.

3 Processor-Integrated Networking

If frequent, fine grain interprocessor communication is to be at all practical, then messaging overheads need to be *orders of magnitude* smaller than what is experienced on contemporary commercial multiprocessors — overheads should be measured in a small number of instruction times, not hundreds or thousands. For example on Monsoon, a research multithreaded dataflow processor [19, 20, 21], a sender can format and launch a message in exactly one cycle, and a receiver can process an incoming message (storing its value and performing an n-ary join) in one or two cycles. In terms of throughput, the processor-network interface can sustain a rate of one message in and one message out every three cycles. The network interface is literally part of the processor pipeline and a visible aspect of the user instruction set architecture.

The primary architectural objective of *T was to assimilate Monsoon-like communications performance into high-end sequential processor design. To do so, we had to reconcile the often competing demands of lightweight and asynchronous fine grain messages with "stateful" and speculative superscalar processing. We call our approach *processor-integrated networking* to denote what we believe is a simple yet highly effective way of integrating low overhead communications directly into a RISC processor instruction set. We believe that this has been accomplished in a way that is completely consistent with single stream speculative and multi-issue execution and, most importantly, supports user-mode programming within a fully protected multiuser environment.

In this section, we develop the instruction set architecture of the integrated message unit (the MSU) on our modified superscalar processor, the 88110MP. This is done in three layers. First, we show the basic user-mode operations for transmitting and receiving messages. Second, we describe how active messages are directly implemented on top of the first layer with either polling or interrupts. Finally, we illustrate how the processor supports a fine grain multithreading and synchronization model called *microthreads.*

3.1 Layer 0: Register-level Transmit and Receive

The integration of basic message formatting and reception into the processor instruction set is very simple. As shown in figure 2, two new user mode register sets, a transmit file (`tx`) and a receive file (`rx`), have been added to the baseline 88110, a commercial dual-issue superscalar microprocessor [15, 9]. Each register file looks to user code like a conventional 24 word (32-bit) set of registers whose contents can be read and written by transferring data to and from these files and the existing general purpose integer and floating point files. There are three basic instructions provided in layer 0:

`sttx`	Store into Transmit Registers (and optionally send current message)
`ldrx`	Load from Receive Registers
`rxpoll`	Poll Receive Status (and optionally get next message)

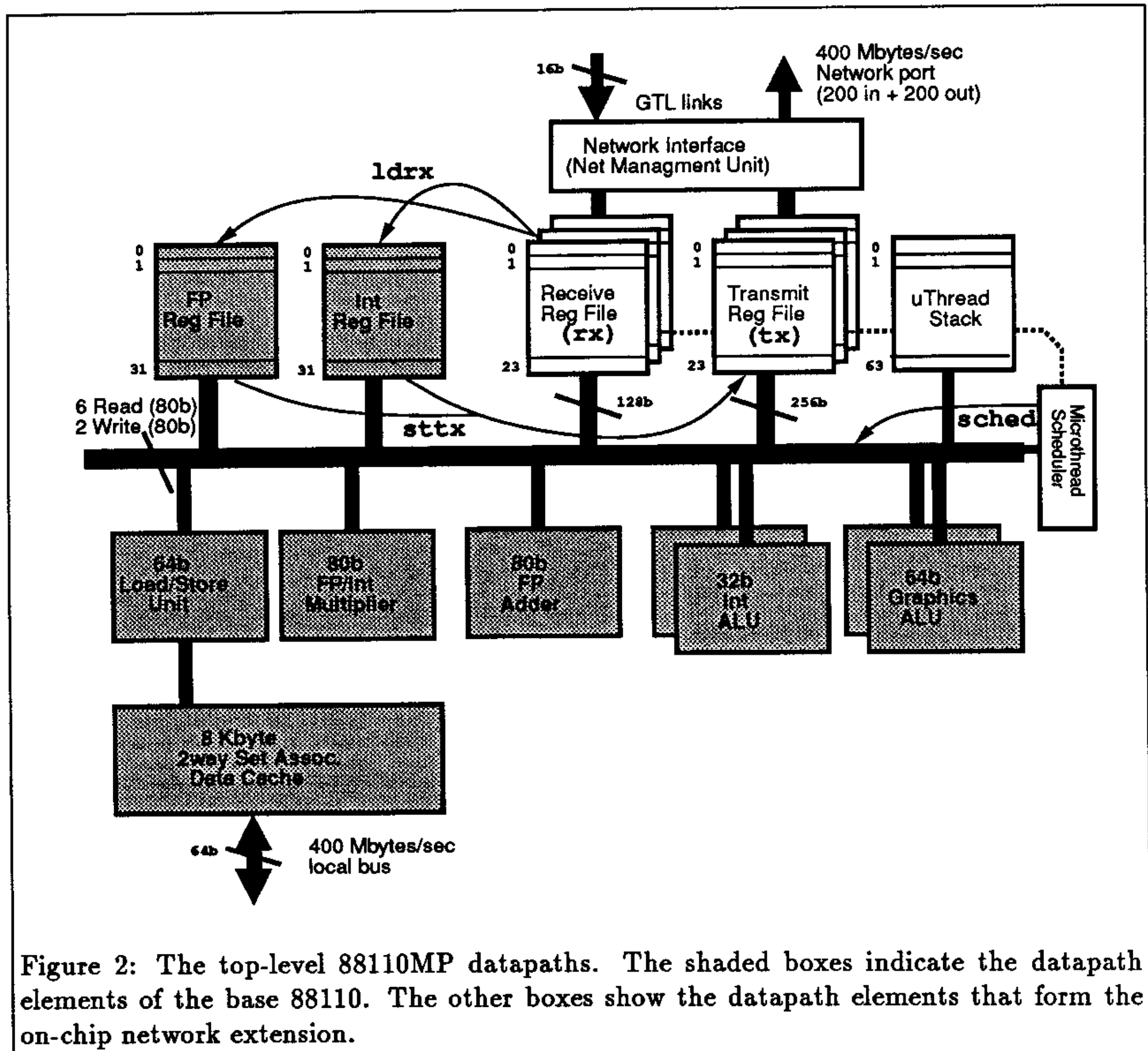

Figure 2: The top-level 88110MP datapaths. The shaded boxes indicate the datapath elements of the base 88110. The other boxes show the datapath elements that form the on-chip network extension.

Transmitting a Message

To transmit a message, the destination node is written into transmit register zero (tx0), the message length (in words) is written into tx1, and the message payload is written into registers tx2–tx23, as appropriate for the message length. Then, the message is submitted into the network using a go operation. Finally, the transmit code must check that the message was actually accepted by the network (network flow control being the common reason for rejection), and then retry transmission when necessary.

While these are the basic operations, the user code path length for message transmission has been heavily optimized by eliding several operations into a single instruction. The principal transmit instruction is sttx (store into transmit register) and comprises four operands:

```
sttx[.go]  r-status, r-src1, r-src2, tx-dest
```

where,

- .go is an option that, when present, causes the current message in the tx file to be transmitted (*after* the transfers indicated by the current instruction are effected).
- r-status is a register in the integer register file that is written with the current transmitter status.
- r-src1, r-src2 are two (single or double word) registers from either the integer or floating point files whose contents are to be deposited into tx registers.
- tx-dest is the (first) destination register in the tx file that is to receive the contents of the two source registers. Up to three subsequent tx registers will be written (in the same cycle) when r-src1 and r-src2 denote double word source registers.

The setting of the message length register tx1 is also optimized — if tx1 has not been written, it is automatically set when a sttx.go is executed. After an sttx.go is executed and the returned status r-status indicates that the message was accepted by the network interface, then the user is given a fresh transmit register file in which a new message may be composed on the very next clock.[3] The code fragment below shows how sttx can be used to create and send a short message comprising three double words (24 bytes) of payload:

```
send_msg:
        sttx.sd         r0,  r2, dr4, tx0    ; clock 1 - dest node (r2) and double word 1
        sttx.dd.go      r1, dr6, dr8, tx4    ; clock 1 - msg double word 2 and 3
        bcnd            ne0, r1, msg_sent    ; clock 2 - check status
msg_not_sent:
        bsr             _retry_tx            ; call out to retry handler
msg_sent:
        ...                                  ; clock 3 - continue normal execution
```

[3]In fact, both the transmit and receive files are only the visible part of *queues* of outgoing and incoming messages. The management of these queues also interacts with the protection mechanisms, as we shall see in a moment.

The first sttx instruction deposits the destination node (from integer register r2) into tx0 and copies the first double word of data from integer registers r4 and r5 (denoted by dr4) into tx2–3. Note that the message length register tx1 is *not* set by this first sttx. The second sttx stores two more double words into tx4–7 and sends the completed message, as indicated by the .go extension. This second sttx implicitly sets the message length in tx1.

Every sttx returns a transmit status value in register r-status. The first sttx discards this value (by writing into r0), but second sttx.go records the status into r1. The status reflects the disposition of this message; that is, was the message accepted by the transmitter and sent? There are several reasons why a sttx.go might not succeed. The most common would be that the transmitter is in flow control and all outgoing transmit buffers are full. In this case, the software could wait and then retry the transmission, or queue the message for later transmission.

Importantly, if a message is not transmitted then *it is still retained in the transmit register set*. The 88110MP also provides instruction set extensions for reading the transmit registers (ldtx) and writing the receive registers (strx). This means that message retry and queueing software can be written as generic user mode handlers — we do not require the user code to be able to reconstruct every message that could possibly fail transmission. In this regard, we consider the register model of a network interface to be superior to the FIFO models of the J-machine and iWarp.

Finally, note that the sttx instructions are dual-issued by the 88110MP. This is discussed in more detail below, but observe that a message with 24 bytes of payload can be sent in two cycles, *including* the time required to check the transmit status.[4] Larger messages can be constructed for transmission at the rate of thirtytwo bytes per cycle.

Receiving a Message

The receive register set is analogous to the transmit register set. An application code can directly poll for messages, and handle them entirely in user mode with no transfers of control into the operating system. As in transmission, this mechanism is highly optimized. There are two primary instructions used for receiving messages: pollrx and ldrx. The pollrx instruction is used to determine if a new message is ready to be received:

```
pollrx[.next]  r-status
```

where .next is an option that, when present, causes the next incoming message to be copied into the rx file. Any of the fields of the new message are available to subsequent instructions, including those immediately following the pollrx.next. R-status is a register in the integer register file that is written with the current receiver status.

The ldrx instruction is the dual of the sttx and is used for copying fields from the rx file into the integer or floating point registers:

[4]The code could be optimized further with a bcnd.n that exposes the extra instruction slot that occurs when the branch is issued. The branch should predict along the msg_sent path, so there is no delay slot after the branch in this case.

```
ldrx   r-dest, rx-source
```

`R-dest` is a single or double word register from either the integer or floating point files which is written with the contents of the indicated `rx-source` register from the `rx` file.

A receive code fragment is given below which performs the receive action for a 24-byte message sent by the transmit example code. In the code below we test for a message in the receiver using an `rxpoll.next` instruction. If a message is present, then this instruction (as indicated by the `.next` option) moves the message into the receive registers:

```
;;;
;;; message in rx registers:
;;; Word 0 = node, Word 1 = length.
;;; Words 2 to 7 contain the message data itself.
;;;
        ...
        rxpoll.next r1
        bcnd    NO_MSG,r1, out          ; leave if no msg
        ldrx    dr2, rx2                ; clock 1
        ldrx    dr4 ,rx4                ; clock 1
        ldrx    dr6, rx6                ; clock 2
        ldrx    dr8, rx8                ; clock 2
        ;; message now in general purpose regs dr2 to dr9
        ...
        ...
```

The incoming message can be consumed at sixteen bytes per cycle — half of the rate that messages can be constructed because the general purpose integer and floating register files have fewer write ports than read ports.

Multiple Instruction Issue

A principal advantage of processor-integrated networking is that the performance of the network interface can track processor performance. In the 88110MP implementation, we are able to devote almost all of the general purpose register file read and write bandwidth to message processing. Our ultimate goal is to establish a reasonably standardized instruction set extension that could be brought forward by new processors in an ISA family, and thereby enjoy a communications/computation balance that scales with higher instruction issue rates. In this way we hope that code generation and scheduling techniques that are developed for other function units will apply directly for the network instruction extension. Contrast this approach with bus-connected, or cache-connected schemes where the read latencies (as measured in instruction times) continue to grow as processors get faster, further decoupling the computation for communication and creating new challenges in code generation for each new processor.

In the 88110MP implementation, frequent network instruction combinations are permitted to dual-issue. In particular, it is possible to issue two `sttx` instructions (or two `ldrx` instructions) in the same cycle as long the `tx` (or `rx`) registers refer to different "banks" of the transmit (receive) register files. Specifically, these register files are two-way interleaved on

four-word boundaries for the transmitter and two-word boundaries for the receiver.[5] When the dual-issue constraints are met, the 88110MP delivers very high bandwidth message transmission (thirtytwo bytes per cycle) and reception (sixteen bytes per cycle). Instruction issue hardware is dedicated to checking constraints, so that instruction ordering only affects performance, not correctness.

Speculative Execution

While it is straightforward to design a network unit for multiple issue, speculative execution (branch prediction) introduces a set of subtle problems. The most obvious is that the transmit and receive register files are visible processor state, and the `sttx` and `strx` instructions modify that state. Thus, the effects of these instructions have to be undone if they are issued speculatively after a mispredicted branch. Actually, the challenge here is no harder than for the integer and floating point register files and the same set of techniques apply (*e.g.,* history buffers). But because we did not want to introduce the design complexity for our first version of the 88110MP, we have instead placed a single-assignment constraint on register updates: a given `tx` or `rx` register is written at most once along any program path with fewer than two branches.[6] We would expect future implementations to eliminate this code generation constraint.

The more interesting interactions with speculative execution happen with respect to `sttx.go` instructions to launch messages in the network and `.next` operations to advance the receive register set. Consider the speculative execution of a `sttx.go` which later turns out to be the result of a mispredicted branch. How do you *recall* a message from the network(!)? Our approach is to permit one outstanding `sttx.go` that is remembered by the message interface until the instruction retires, and only then is the message actually sent.[7] Even more subtle is the status returned (in `r-status`) by an `sttx.go`: it must represent the status of the message *as if it were actually sent.* This problem is made easier by the fact that the `sttx.go` atomically returns a status that is associated with the transmit state on the clock that the instruction is issued. Contrast this with a separate instruction to check the transmit status — if the status checking instruction is issued on a clock after the `.go`, there is the possibility that the flow control state of the transmitter has changed in the intervening cycles.

In summary, incorporating the message interface as an instruction set extension can yield very high performance by riding the trends in speculative and multi-issue execution. The

[5]That is, `tx0–3`, `tx8–11`, `tx16–19` are in Bank 0 and `tx4–7`, `tx12–15`, `tx20–23` are in Bank 1. A similar pattern holds for the receiver, except the boundaries are every two words.

[6]The 88110MP will speculate across only a single branch, so if two branches occur along a path then it is guaranteed that all instructions before the first branch will have been committed for execution before any instructions after the second branch are issued. Details of 88110 branch prediction and speculative execution are given in [15, 9].

[7]A second `sttx.go` causes instruction issuing to be stalled until the first retires. Of course, our solution adds latency from the time that message gets generated until it is actually sent in the case when the speculation pays off. Unfortunately, this latency grows as future processors permit a larger window of outstanding instructions. Perhaps there are some clever ways to speculatively launch messages into a network, but we'll leave this kind of hacking for implementors that have more fortitude than ourselves.

downside to this is that the implementation must deal with thorny issues of instructions that cause a number of interesting side effects.

3.2 Layer 1: Support for Active Messages

The layer 0 operations for transmitting and receiving messages offer a base transport-like layer for communicating messages. The underlying contract of the network is:

> If the network accepts a packet (indicated through `r-status` after a `sttx.go`), then it promises to ultimately deliver that packet error-free to the indicated destination node as long as all nodes in the network promise to consume incoming packets.

The network contract makes no guarantees about packet ordering and only promises to deliver packets if every node promises to receive them. This means that if the network *does not* accept a packet then the transmitting code has to be prepared to receive packets until the blocking condition no longer exists, else the contract is violated and network deadlock may occur. It also means that we have to have a place to put the contents of the packets we are receiving in the meantime *and* that the processing of the receive packets can't themselves require a response to be transmitted.

The traditional way of abstracting the network contract to the application is through a message-based communications library which provides scheduling and internal buffering for received messages. Various end-to-end flow control protocols overlay the packet transport and provide some immunity from network deadlock. Unfortunately, these kinds of message libraries invariably introduce an unacceptably high communications overhead (refer back to figure 1).

In contrast, Berkeley's *active messages* [25] rationalizes the network contract and exposes it directly to the application. Under the active messages model, every message carries an instruction pointer (IP) which points to the handler for that message on the destination node. Each handler is specialized to the particular message and is responsible for directly operating on the message contents, including copying any message data into local data structures and performing synchronization operations. Handlers are divided into two types: *reply handlers* that can execute to completion without creating network messages, and *request handlers*, such as a remote load server, that need to send response messages.

One can view message handlers as very lightweight threads that "interrupt" a background thread — the main computational thread on a node. The message handlers communicate to the background thread through shared data structures (*e.g.*, a stack frame) and synchronization variables (*e.g.*, decrementing a counting semaphore).

IP:FP Active Messages

88110MP supports a variant of active messages wherein the second double word of a message encodes a pair of pointers, an instruction pointer (IP) and a frame pointer (FP). The FP

should be considered a pointer to a context, such as a procedure activation frame, an object, or a heap location. The latter would be typical of request handlers for remote read and write operations. Refer to Figure 3.

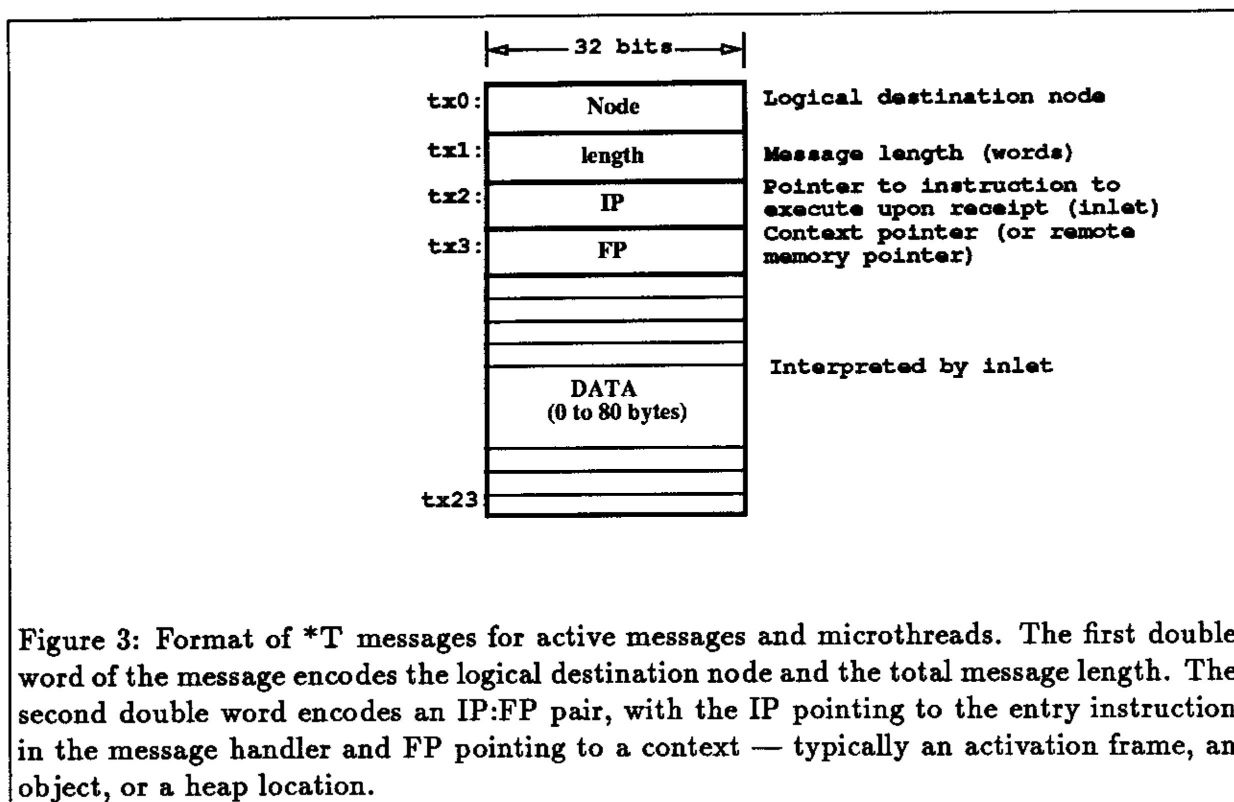

Figure 3: Format of *T messages for active messages and microthreads. The first double word of the message encodes the logical destination node and the total message length. The second double word encodes an IP:FP pair, with the IP pointing to the entry instruction in the message handler and FP pointing to a context — typically an activation frame, an object, or a heap location.

Of course, no extra hardware support is required to implement active messages — it possible to make an efficient implementation directly on top layer 0 message primitives. For example, a background thread could poll for an incoming message, extract the IP from `rx2` and then jump to the message handler.

Microscheduling Support

The 88110MP implementation provides *microscheduling* support to make the polling of active messages more efficient. The basic primitive is the `sched` instruction, which returns a double word value comprising an IP:FP pair:

```
sched    r-ip:fp
```

If there is a message ready at the receiver, the `sched` instruction advances the receive register set (*i.e.,* it implicitly performs a `.next` operation) and then extracts words two and three from the message (the assumed IP:FP) and returns them in `r-ip:fp`. If no message is ready at the receiver, then the `sched` instruction will return a default IP:FP pair that can be set by the user code. One way that the default pair can be set is through the `rxsched` instruction:

```
rxsched  r-ip:fp, r-default-ip:fp
```

Rxsched is just like sched except that the default IP:FP pair is supplied — if there is no message then this value is returned. Also, the message unit remembers the default pair and will return it as appropriate for any subsequent sched operation. To perform polling in the background thread requires only a two instruction sequence:

```
;;;  Assume that r_continue already set up with IP:FP
;;;  pointing to label 'continue'
;;;
   rxsched   r2_3, r_continue
   jmp       r2

continue:
```

Each active message handler ends with the idiom sched r2_3; jmp r2, and control is thereby threaded through all of the incoming messages and finally returned to the background thread once all messages are handled.

Interrupt Support

For a variety reasons relating to polling overhead and network congestion control, it may be desirable to have incoming messages interrupt the background thread. Although we have not modified the 88110 to vector these interrupts directly to user mode — something highly desirable but beyond our engineering resources for this version — we have made the incoming message interrupt maskable under user mode control. Thus, when the user code enabled interrupts, an incoming messages causes a lightweight kernel interrupt which immediately reflects the interrupt back to user mode, *i.e.*, the kernel handler does not save and restore user registers. The user mode access of the enable/disable bit permits inexpensive critical sections to be established without invoking the kernel. Importantly, the enable/disable operation is *serializing*: no subsequent instructions will be issued until the operation retires. This way, the user code can be assured that instructions which immediately follow an interrupt disable are part of the critical section.[8]

Some system implementations may desire that every incoming message interrupt the kernel. For this case, we note that the kernel mode has its own interrupt enable bit which takes priority over the user bit. Also, the kernel has a watchdog timer feature that can not be disabled in user mode — if the receiver flow controls the network for more than a specified number of cycles, then an interrupt is generated. See the discussion on protection in section 4.

[8] Again, contrast this with an off-chip based network unit. Even if the interrupt enable bit is mapped into the user mode address space, guarantees that following instructions are indeed part of a critical section requires synchronization of the processor with external bus cycles (for example, by performing a non-cacheable read and then touching the result).

3.3 Layer 2: Microthreading

Under active messages there is conceptually a single background thread that periodically yields to active message handlers. Synchronization between the incoming active messages and the background thread is *ad hoc,* typically involving shared semaphores that are queried by the background thread. In contrast, the 88110MP supports the message-driven *T *microthreading* model [18] wherein the execution of message handlers can cause *data threads,* short non-blocking instruction sequences, to be queued for execution. At a high level, the *T microthreading model is direct descendant of Berkeley's Threaded Abstract Machine (TAM) [23].

Figure 4 illustrates some of the basic ideas behind the TAM-like *T microthreading model. Incoming messages are processed by *inlets* that correspond to the message handlers under active messages. In general, an inlet will deposit the data-part of the message into an offset in an activation frame whose base is given by the FP field on the message. Then, the inlet will decrement a counting semaphore (also found in the activation frame) and, if the semaphore is zero, will queue a continuation for a data thread described by the pair IP_{DT}:FP, where IP_{DT} points to the first instruction in the data thread code. Data threads are executed as background threads: inlets will be executed as long as there are incoming messages, but then data thread continuations are dequeued and executed to completion.

Microthread Stack

The 88110MP provides an on-chip, 64-entry cache for a microthread stack that stores descriptors for data threads awaiting execution. The microthread stack is part of the user's process state and is manipulated in user mode. Entries on the stack are 64-bit IP:FP pairs. The IP points to the entry instruction for a data thread and FP points to the thread's environment, typically an activation frame. Pushing a new entry onto the stack can be performed several ways. The `fork` instruction unconditionally pushes an IP:FP pair onto the stack:

```
fork  r-status, r-ip:fp
```

where `r-ip:fp` is a double register containing the data thread microthread descriptor to push, and `r-status` is a destination register that is written with current stack status. The status indicates how many words are remaining in the stack cache and user software is responsible for not overflowing the cache. Even more useful for the counting semaphore model of TAM is the conditional `cfork`:

```
cfork  r-status, r-ip:fp, r-semaphore
```

which conditionally pushes the contents of `r-ip:fp` if the contents of `r-semaphore` is zero.

Although microthread descriptors can be explicitly popped from the stack using the poput instruction, the primary way of popping descriptors is *implicitly* with `sched` (see section 3.2). Specifically, the IP:FP pair returned by a `sched` or `rxsched` instruction will be, in priority

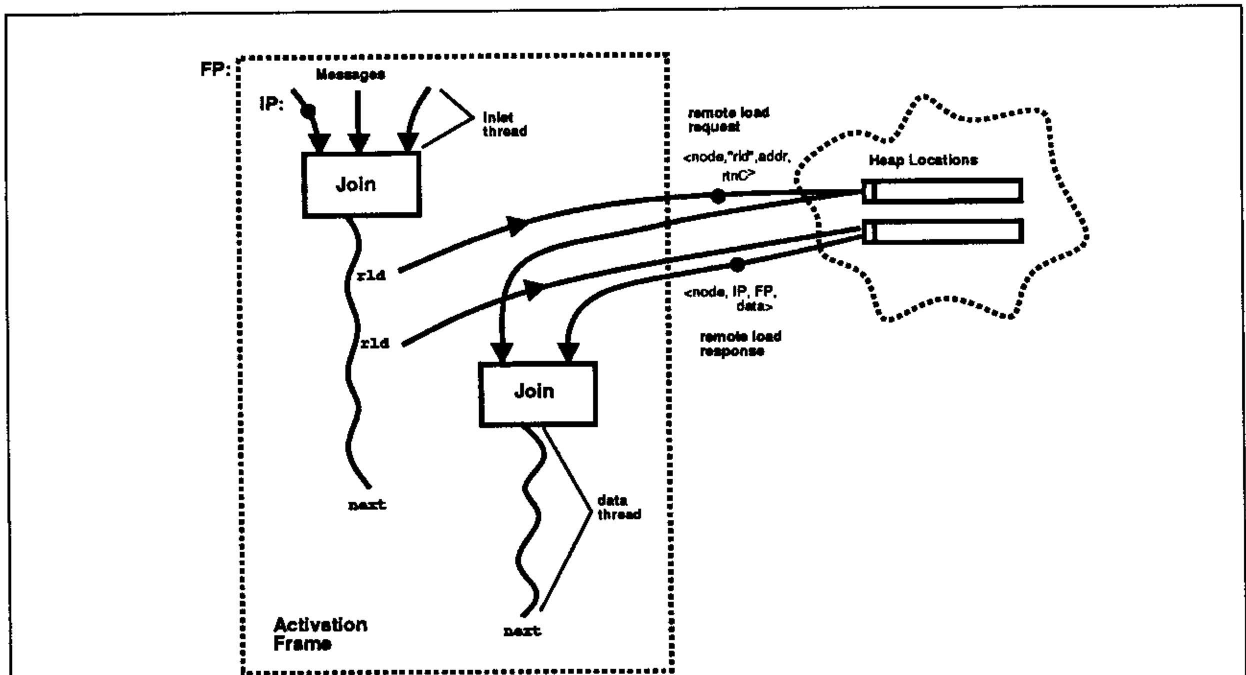

Figure 4: A high level view of TAM-like *T microthreaded code. There are two types of microthreads. *Inlets* correspond to active message reply handlers and process all incoming messages. Inlets manipulate counting semaphores to effect a join operation and post *data threads*, short background processing threads that are non-blocking and execute to completion. In the example, remote load requests (`rld`'s) are issued by one data thread while the responses are synchronized by a set of inlets that perform a join and then post a second data thread when all responses have arrived.

order, from either (1) the header of an incoming message, (2) the poll return IP:FP, (3) the top of the microthread stack, or (4) the "idle" microthread. These priorities can be adjusted[9], but the high level idea is that the hardware is providing a very simple scheduling veneer that collapses many tests — *e.g.,* incoming message present, in polling mode, microthread stack empty — into a single jump that dispatches directly to a message handler, data thread, idle handler, or returns to a polling thread. Put another way, a `sched` instruction is like asking "give me the next thing to do", where the answer is given as a pointer to a piece of code to execute (the IP) and a context in which to perform the execution (the FP).

A common case in message handlers is the execution of a conditional fork (`cfork`) of a data thread followed by a `sched` and `jmp` in order to dispatch to the next thing to do. As a further pathlength reduction, the `cfork` and `sched` instruction can be elided into a single `cpost` instruction:

```
cpost  r-next-ip:fp, r-fork-ip:fp, r-semaphore
```

where the contents of `r-fork-ip:fp` are pushed onto the microthread stack if the contents of `r-semaphore` are zero. The register `r-next-ip:fp` is written with the descriptor for the next thing to do. Note that if popping a descriptor from the microthread stack is the highest priority activity when the instruction is issued, then `r-next-ip:fp` is given the value of `r-fork-ip:fp` and the stack is not affected.

Putting it Together: A Five Cycle Inlet

The complexity of all of the instructions we have given so far are consistent with multiple issue execution on the 88110. Now we show how to code a inlet thread with synchronization that performs the following actions in five cycles:

1. Fetch an integer semaphore from the activation frame coded by the message (FP); decrement the semaphore; write the semaphore back to the frame.

2. Conditionally fork a microthread if the semaphore decrements through zero.

3. Copy a double word datum from the message into the activation frame.

4. Transfer control to the next message handler or microthread.

This kind of inlet occurs frequently as the reply handler for remote load requests, and as an entry point for separately passed arguments for remote procedures. Here is the 88110MP-code:

[9]There are in fact eleven priority levels that can be selectively enabled by user code. For example, is possible to reverse the "normal" priority of selecting incoming messages over the microthread stack. There are also hooks for *dynamically* escaping from the priority structure, but this is beyond the scope of this paper.

```
;;;
;;; The invariant assumption is that  r2 = IP, r3 = FP,
;;; i.e., the microthread descriptor of the message handler.
;;; The invariant is preserved on exit in that r2:r3 exits with
;;; the IP:FP for the next inlet or data thread.
;;; Registers r1 through r6 are modified.
;;;
message_inlet:
    ld.b     r1, r3, SEM       ; Clk 1 - get semaphore byte from current frame
    add.imm  r2, DT_label, r2  ; Clk 1 - compute IP for data thread relative to ours
    ldrx     dr4, tx4          ; Clk 2 - get double word payload from packet
    or       r6, r3, 0         ; Clk 2 - save FP -> r6
    st.d     dr4, r3, DATA     ; Clk 3 - save payload into current frame
    cpost    dr2, dr2, r1      ; Clk 3 - push IP_DT:FP if semaphore = 0
    sub      r1, r1, 1         ; Clk 4 - decrement semaphore
    jmp.n    r2                ; Clk 4 - jump to next uthread r2:r3 = IP:FP
    st.b     r1, r6, SEM       ; Clk 5 - use delay slot to store back semaphore
```

This code fragment is obviously highly optimized. The handler assumes that on entry `r2:3` contains the IP:FP for the handler. This invariant is preserved by the code upon exit — after the `jmp.n` either another inlet or a data thread is entered with `r2:3` pointing to the IP:FP for that inlet or thread[10]. The dual-issue features of the 88110MP allow the code fragment to execute in five clock cycles assuming all memory references hit the cache.

We note that the semaphores used for join counters are not at all like the conventional notion of OS semaphores. We are able to read and write them non-atomically because of the mutual exclusion property of microthreads. Any other microthread which would also be manipulating this semaphore cannot be in execution concurrently or in any way interleaved with the execution of any other microthread.

4 Naming and Protection

Giving direct user mode network access can be very efficient, but is of little value if it comes at the expense of security and system integrity. One of the fundamental challenges of providing an efficient user mode network interface is do so in a way that is compatible with the naming and protection mechanisms expected by modern operating systems.

Interference among separate processes sharing a network can occur in many ways. A malicious process might try to intercept messages destined to another process, forge new messages and inject them into another process, or congest the network in such a way as interfere with communications and induce deadlock in another process. These, of course, are in addition to the usual requirements of memory protection on each node. Our *T implementation provides naming and protection mechanisms to support user mode network access within a complete multiuser environment.

[10]Recall, if the semaphore is zero and there are no more incoming messages, then the `cpost` will return the data thread corresponding to this inlet.

4.1 Naming mechanisms

A *T *global virtual address* comprises a node-part and an offset-part, `node:offset`. `Node` is a logical node number that is translated by the transmitter hardware into a network route to a physical node (see section 5). The `offset` of a global virtual address is a local address that is valid on the target physical node. Thus, the IP:FP pairs on message are *local virtual addresses* that are translated into physical addresses at the physical node by using the pre-existing MMU mechanisms during load and store operations. More simply, name translation happens in two phases: a logical-to-physical translation of the node happens at transmit time on the sending node, while a virtual-to-physical translation of a memory offset happens on the receiving node and uses the existing virtual memory translation mechanisms.

The first phase of the translation happens in the *network management unit (NMU)* (see Figure 2) which employs a small content addressable map from logical nodes to network routes. The mapping is prioritized and allows the translation of variable-sized blocks of logical nodes into routes, similar to modern TLBs. The route table is accessible only by privileged instructions (*i.e.,* it is not loadable by user code). If the user code creates a message with node that is not mapped by the table, then the message is not sent and a failed status is returned. Note that a trap is not automatically issued in this case, it is up to the user code to invoke the kernel to modify the routing table, if so desired and if permitted.

4.2 Protection Mechanisms

Along with the physical route, the network management unit also inserts a parallel process Id called a *partition Id (PID)*[11] into the header of each outgoing message. Refer to figure 5. Whenever a message is received, this PID is examined and compared (by hardware) with the PID for the user process currently executing on the destination node. The incoming message is thereby sorted into one of four on-chip receive queues: the active user queue, high and low priority kernel queues, and the "others" queue called the *miss* queue.[12]

User mode messages received by a processor that is currently executing a parallel process corresponding to the message's PID, will be queued for that process without any other software intervention — in particular, the kernel is not invoked unless the user code has sensitized interrupts on incoming user mode messages. When a user mode message is received whose PID *does not* match the currently registered user process, the message is placed in the miss queue and an interrupt is posted to the kernel. Also, any packets that have been corrupted in transmission as determined by a CRC error are inserted in the miss queue. Kernel mode packets (PID $\equiv 0$) are inserted in either a high or low priority kernel queue depending on the header of the packet (this interacts with network priorities — see below) and an interrupt is posted to the kernel, unless the processor is already in kernel mode.

[11] We distinguish partition Id from process Id because, depending on the operating system, user mode processes on separate processors that correspond to the same parallel process may be assigned different process Ids. If these processes are given the same partition Id for communication, then messages can be exchanged without any operating system intervention.

[12] In the 88110MPthere is a common pool of twelve receive buffers (register sets) which are dynamically allocated to the four queues.

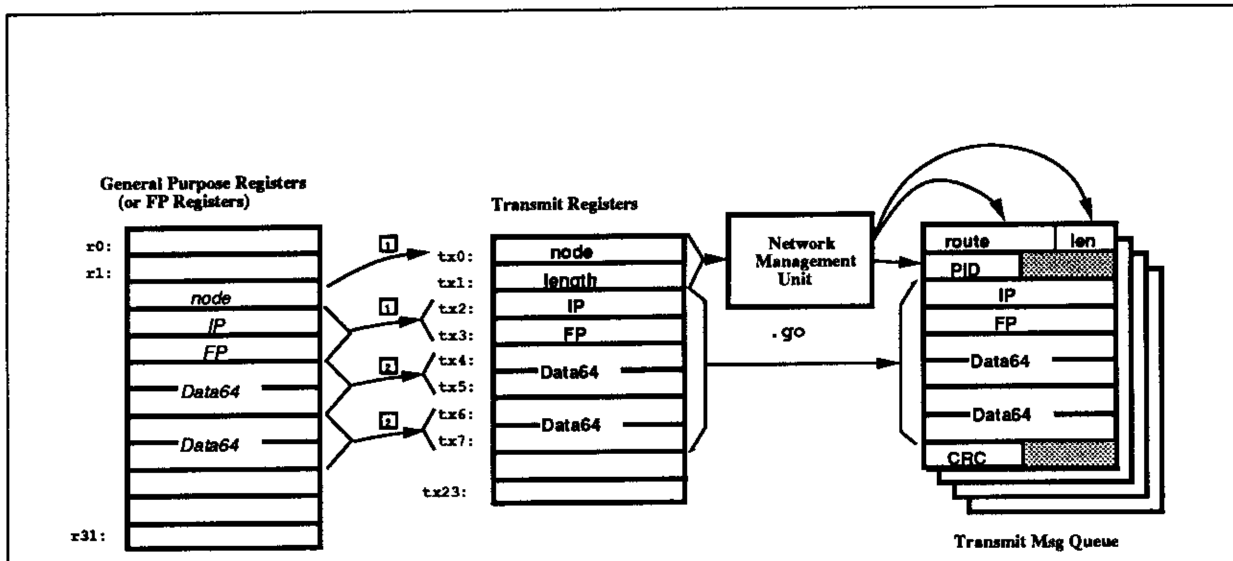

Figure 5: Naming and protection mechanisms are provided by the network management unit (NMU). This illustration shows how the short message from the example in section 3.1 is formatted for transmission. The NMU translates the destination node to a physical route using a small CAM, and inserts the currently active user PID in the message header. The PID is compared by the receiver hardware on the destination node in order to sort the message into one of four queues.

Optionally, the kernel can elect to be interrupt whenever a message arrives for the currently executing user process — even if there is a PID match. Alternatively, the kernel can elect to be interrupted only after the user mode message queue is full and has placed the network in flow control for specifiable number of cycles. This can be used to mitigate the effects of user-user network interference, as described below.

Because the head of the message queues actually form the `tx` and `rx` register sets, the kernel automatically gets its own set of these registers whenever the 88110MP changes mode. Thus, O/S does not need to save and restore the user's version of the `tx` and `rx` sets until it wants to swap user processes. Note that the microscheduling stack state is *not* replicated, however. If the kernel desires microscheduling support it must save and restore the stack state.

4.3 Network Interference and Virtual Channels

The protection mechanisms described above are sufficient to prevent a malicious or errant user mode process from aliasing or stealing packets for other user mode processes or the kernel. Further, they permit a processor to run its own scheduler and memory manager without synchronizing with other processors.[13] However, it is possible for a user process to

[13]Such synchronization (*e.g.*, gang scheduling) is not required to provide protection guarantees, but might be highly desirable for performance reasons.

interfere with other processes by deadlocking the network and thereby not allowing competing processes to make progress. This can happen whenever the physical buffering resources in the network are shared by two processes and when one process jams the network with messages and places links into flow control (this condition occurs frequently, even in correctly executing programs).

There are several different approaches to avoiding networking interference. One is to provide a *virtual channel* [8, 3] for each process. In effect, a virtual channel is created by dedicating message buffers that can not be consumed by other virtual channels. This way, a message in a virtual channel can always make progress independent of the state of the other channels. Another way to mitigate the effects of interference in a user mode network is to *space partition* the network where user processes are space and time scheduled such that two processes never share the same set of network links at the same time. Space partitioning is implemented on the CM5 by isolating users in separate subtrees of a fat-tree [6]. Kernel messaging (including I/O) can traverse through the root. Unfortunately, a pure reliance on space partitioning restricts system organization, and in particular the way in which I/O devices can be distributed throughout the machine.[14]

The *T system architecture takes a mixed strategy. The network is virtualized into low and high priorities. User messages are always sent in low priority whereas the kernel can label a message for high priority. High priority messages are guaranteed to make progress even if the low priority channel is deadlocked. Beyond this, multiple user processes are given separate space partitions (*T uses a fat-tree network topology). We note that the packet timeout mechanism and PID tagging *does* permit users to physically share space partitions, but with a potential degradation of performance.

5 ARCTIC: A Companion Message Routing Chip

ARCTIC[15] is a high performance 8×8 packet routing chip based upon the highly successful PaRC [12], a 4×4 100 Mbytes/sec/link packet router used in Monsoon. Like PaRC, ARCTIC routers can store and forward packets during times of high network utilization, but use virtual cut-through routing to avoid the latency of buffering when there is no contention for a desired output link. While an ARCTIC router might be used in a wide variety of networks, we have added special routing support for use in a fat-tree topology [13], the network topology used in *T systems.

The link signalling conventions are directly compatible with the links on the 88110MP. Each link is unidirectional and capable of sustaining a raw transfer rate of 200 Mbytes/sec. Flow control and buffer management algorithms provide a high and low priority virtual channel for each link. The 88110MP uses the high priority channel for kernel packets, and the low priority channel for user and non-critical O/S services.

[14] Kernel mode-only fabrics (*cf* Intel Paragon) avoid the deadlock aspect of network interference. Of course, this is at the expense of the overhead of the user-kernel interaction required to send and receive messages. One view of this overhead, then, is from the buffer management and flow control algorithms employed by the kernel to avoid network interference.

[15] A Routing Chip That Is Cool

A key design goal was to make ARCTIC a *modeless router.* Aside from configuration registers to establish the physical routing addresses algorithm, an ARCTIC chip has no knowledge of user versus kernel packets, space partitioning or context switching. All ARCTIC routing decisions are deterministic with respect to the physical routing address on each packet. Space partitioning, route randomization and faulty link avoidance are all properties of the route generation algorithm at the source 88110MP. While this design position gives up some potential performance under highly loaded conditions[16], it permits deterministic management of network resources by the nodes and avoids dynamic configuration and context switching issues in the router. The route generation CAM in the NMU is sophisticated enough to provide random up-route generation in a fat tree while being able to suppress paths that traverse known-bad links.

ARCTIC Core

The block diagram for ARCTIC is show in figure 6. There are eight identical input sections and eight identical output sections. Packet buffering is provided by the input sections in the form of four 768 bit buffers per section. A 768 bit buffer (corresponding to the maximum message size) is allocated for each incoming packet. In most cases the buffer will not be fully utilized because the packet will likely be less maximum size and, because virtual cut-through routing is used, the head of the packet will likely be forwarded before the tail is received.

Three of the four input buffers can be used by high or low priority packets, while the fourth can be consumed only by a high priority packet. This is required to support the high versus low priority virtual channels — low priority packets can never consume all of the routers buffers, so progress of high priority packets will be guaranteed[17].

Inputs sections are connected to output sections by a 32×8 crossbar switch. The crossbar has an input connected to each buffer of each input section and an output connected to each output section. Thus, it is possible to simultaneously transfer packets from any subset of eight input buffers to the eight output sections. For example, the crossbar can simultaneously transfer three packets from input section 0 and one packet each from input sections 1 through 5.

Input links are configured to either be an *up-link* or a *down-link,* depending on whether the link is a member of the up-part of the fat-tree or the down-part. For up-links, routing is determined by each input section by comparing the *up-route* part of the physical route in message header with the contents of a configuration register that establishes the position (height) of the router in the fat-tree. A decision is made to forward the message further up the tree according to the up-route, or to send the packet down the tree as indicated by the *down-route.* Down-links route according to the *down-route* field in the message header. We emphasize that all routing decisions are deterministic. Pseudo-randomization of up paths can be performed at the source by suitably programming the route generation CAM in the 88110MP NMU.

[16]Permitting a router to make a local routing decision on the basis of instantaneous link load can improve throughput in some circumstances.

[17]Note that we permit high priority packets to use any buffer. This adds to performance when there is little low priority traffic and does not cause concern for deadlock.

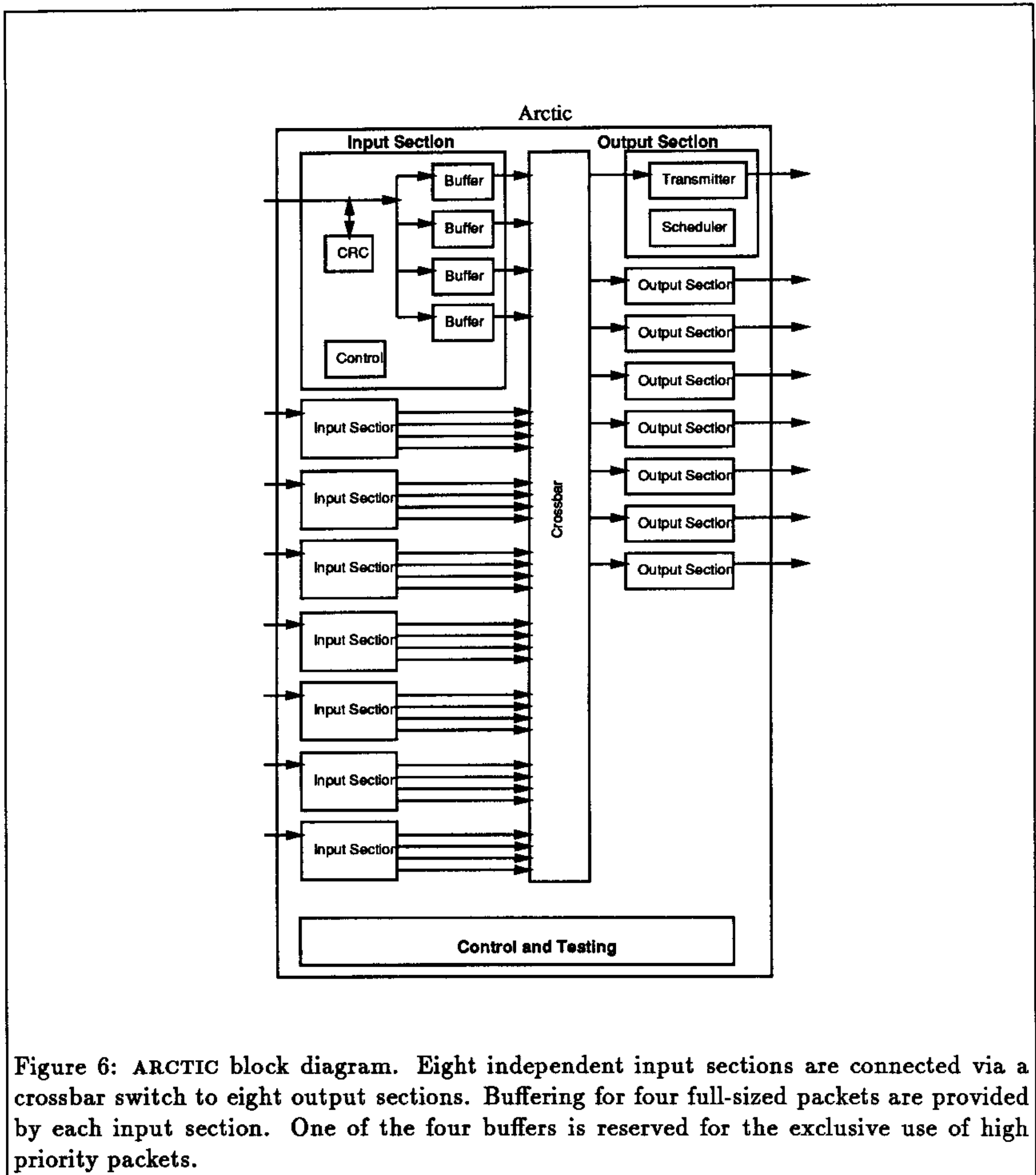

Figure 6: ARCTIC block diagram. Eight independent input sections are connected via a crossbar switch to eight output sections. Buffering for four full-sized packets are provided by each input section. One of the four buffers is reserved for the exclusive use of high priority packets.

Link Protocol and Flow Control

The data links that connect to ARCTIC's inputs and outputs have the structure shown in Figure 7. Gunning Transceiver Logic (GTL), a low-voltage swing CMOS technology, is used for all signals and each signal is parallel terminated to a termination plane at the receiver end. The design goal is to support very low bit error rates on 1.5 meter copper stripline traces on standard epoxy-Fiberglas printed circuit boards.

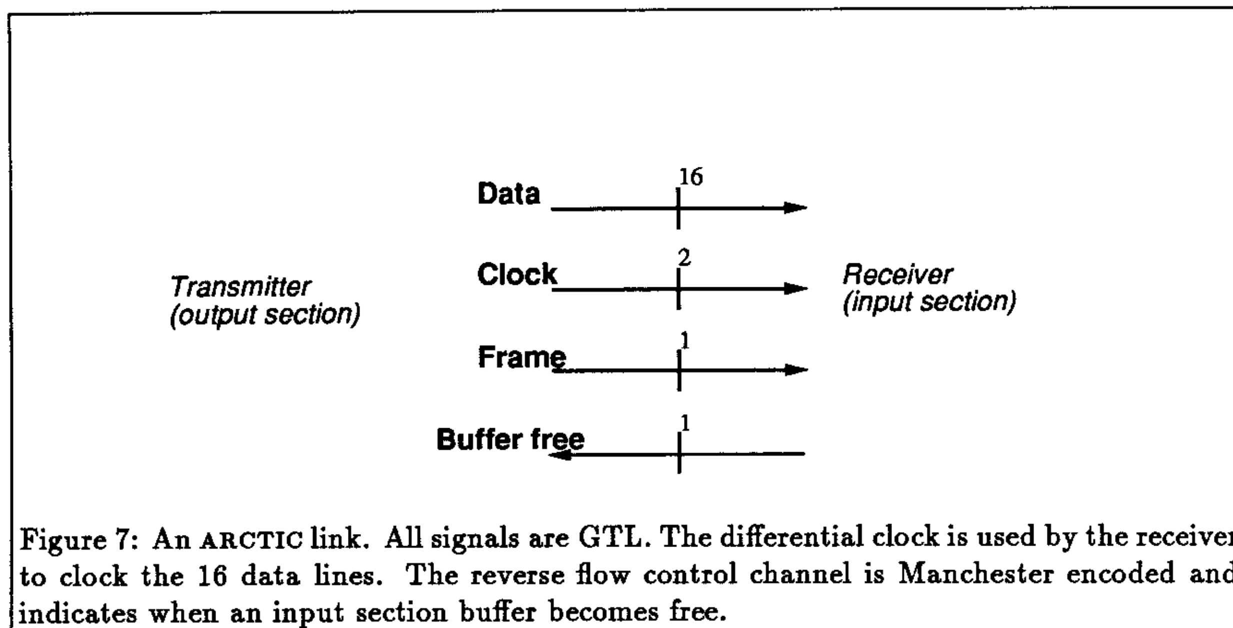

Figure 7: An ARCTIC link. All signals are GTL. The differential clock is used by the receiver to clock the 16 data lines. The reverse flow control channel is Manchester encoded and indicates when an input section buffer becomes free.

The data link uses a 16 bit wide data path and each bit is transmitted using a single ended GTL driver. Data is sent at 100Mbits/sec on each signal line synchronous with a 50MHz differential clock that is transmitted with the data. The ARCTIC clocking convention is *plesiochronous* [14]: each router runs on its own independent 50MHz clock. A router transmits its clock along with the data on each output link. The input sections use the incoming clock clock to safely latch link data, and to clock the link data into a free input buffer. There are thus nine clock domains, one for each of the eight input sections and one for the ARCTIC core and all output sections. This technique proved to be highly reliable on PaRC, which to date has yet to record a single packet delivery failure in fielded systems.

Also as in PaRC, flow control is performed on packet boundaries. The ARCTIC protocol differs, however, in that the output section of the transmitter keeps track of the number of available buffers in the input section of the receiver to which it is connected. The input section has a single back link on which it modulates a Manchester encoded buffer free signal every time one of its input buffers becomes available.

This flow control strategy has two advantages compared with a traditional "back pressure" flow control line. First, it cuts the effects of the link latency in half — a back pressure line needs to assert early enough to account for two link delays worth of *in situ* data. Second, the link delay itself is not part of the flow control algorithm, so the link delay does not have to be parameterized in the flow control logic. The flow control self-adapts to link length.

6 Preliminary Performance Modeling

Although both the 88110MPand ARCTIC chips are still under development,[18] we have used a variety of techniques to model expected system performance. Here, we briefly report on an extension of performance modeling technique used to compare the J-machine and the CM5 under fine grain parallelism [24]. This approach develops a comprehensive file of "machine costs" that detail the expected number of cycles it takes a processor, including its network interface, to perform the abstract operations under TL0, Berkeley's intermediate language for fine grain multithreading. These costs are then factored into instruction frequency statistics gathered from instrumenting the threaded execution of a variety of benchmark codes written in Id [17]. The average execution time in terms of cycles per TL0 instructions are reported.

Figure 8 reproduces the data from the original J-machine and CM5 study and adds our data for *T . Overall, the 88110MPimplementation yields an average of 3.5 cycles/TL0 instruction, in contrast to about 12 cycles for the J-machine (modified to include a floating point unit and a data cache) and 14 cycles for the CM5 (modified to include an improved network interface). Of course, the *T implementation also benefits from the dual-issue nature of the 88110MP. In our model, however, we have conservatively rated the 88110MPat 1.3 instructions per clock. We believe that these preliminary data attest to the broad scale yet balanced attack on network overhead without compromising sequential code performance.

7 Conclusion

We believe that we have struck a good balance in combining the demands of competitive sequential performance with the tightly coupled network interface of experimental machines such as Monsoon, the J-machine and iWarp. While the mapping of the network into user mode registers is not difficult by itself, we have done so in a way that is consistent with speculative superscalar execution and, importantly, supports a naming and protection model consistent with modern operating systems. Our preliminary modeling shows that the payoff may be overall cycle efficiencies of better than a factor of three when compared with the latest commercial and research machines. We hope that this design helps the pave the way to network interfaces that are as closely coupled to processors in the future as load/store units are today.

[18] First silicon for both parts is expected in 4Q93.

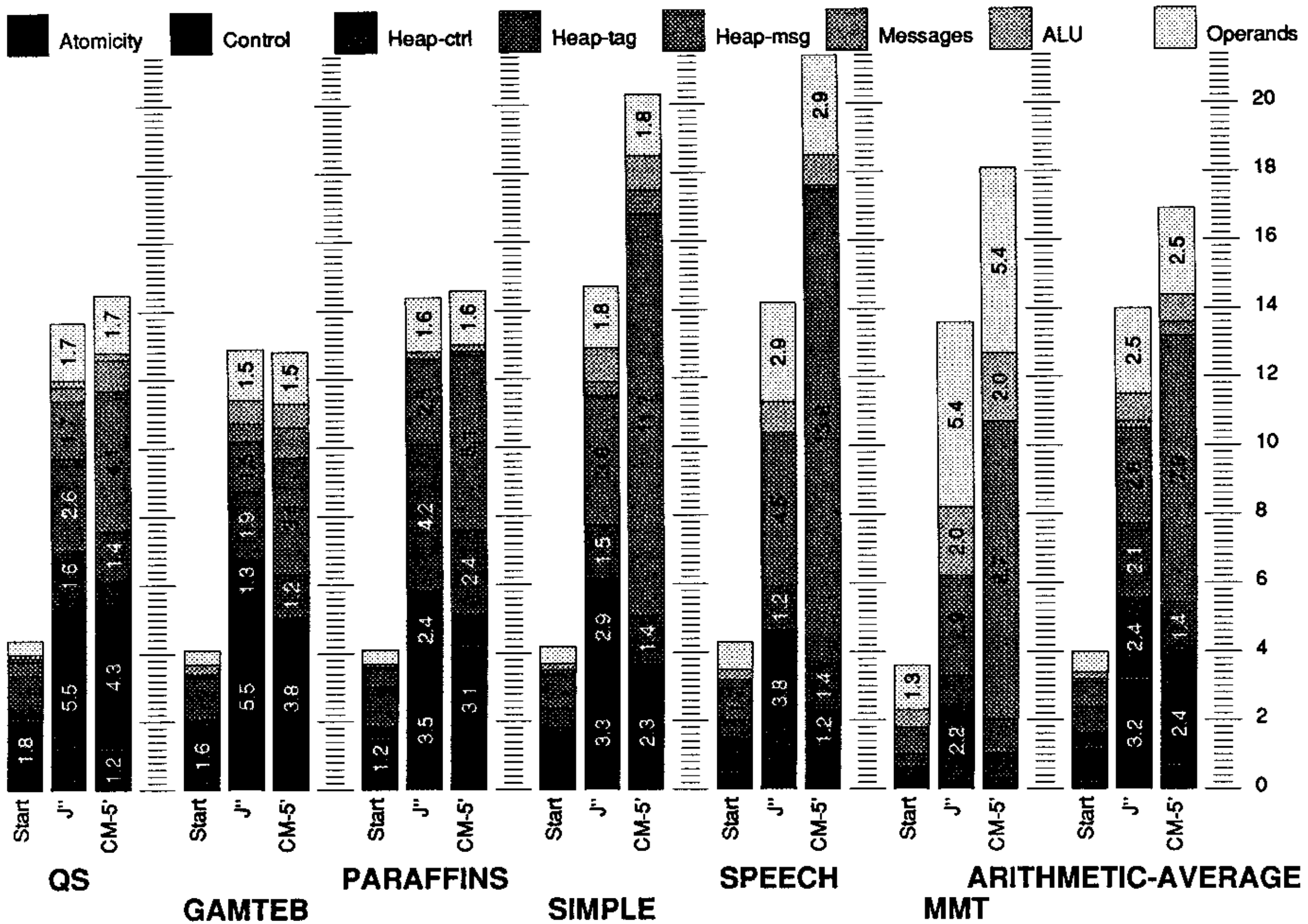

Figure 8: A cycle per TL0 instruction comparison of the J-machine (modified to include floating point hardware and a cache) and the CM5 (modified to include a better network interface). The x-axis is in units of cycles per TL0 instructionThe benchmark codes were written in Id and compiled into TL0. QS is a quicksort, GAMTEB is a neutral particle transport problem, PARAFFINS enumerates paraffin isomers, SIMPLE is a fluid dynamics problem with heat conduction, and MMT is a blobked matrix multiply.

References

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. 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.

[2] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, Springer-Verlag LNCS 295*, June 25-29 1987.

[3] S. Borkar, R. Cohn, G. Cox, T. Gross, H. 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.

[4] E. Brooks. The Attack of the Killer Micros, 1989. Presentationtation in the Teraflop Computing Panel Discussion, Supercomputing '89, Reno, Nevada.

[5] I. Corporation. *Paragon XP/S Product Overview.* 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006, 1991.

[6] T. M. Corporation. The Connection Machine CM-5 Technical Summary. Technical report, Thinking Machines Corporation, 245 First Street, Cambridge, MA, Jan. 1992.

[7] W. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a Message-Driven Processor. In *Proc. 14th. Annual Intl. Symp. on Computer Architecture, Pittsburgh, PA*, pages 189–196, June 1987.

[8] W. J. Dally, R. Davison, J. A. S. Fiske, G. Fyler, J. S. Keen, R. A. Lethin, M. Noakes, and P. R. Nuth. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, April 1992.

[9] K. Diefendorff and M. Allen. Organization of the Motorola 88110 Superscalar RISC Microprocessor. *IEEE Micro*, Apr. 1992.

[10] V. Grafe, G. Davidson, J. Hoch, and V. Holmes. The Epsilon Dataflow Processor. In *Proceedings of the 16th. Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 36–45, May 29-31 1989.

[11] D. Henry and C. Joerg. A Tightly Coupled Processor-Network Interface. In *Proc. 5th Intl. Conf. on Achitectural Support for Programming Languages and Systems (ASPLOS), Boston, MA*, pages 111–122, October 12-15 1992.

[12] C. F. Joerg. Design and Implementation of a Packet Switched Routing Chip. Technical Report MIT/LCS/TR-482, Dec. 1990.

[13] C. E. Leiserson. Fat Trees: Universal Networks for Hardware-Efficient Supercomputing. C-34(10), Oct. 1985.

[14] D. G. Messerschmitt. Synchronization in Digital System Design. *IEEE Journal on Selected Areas in Communications*, October 1990.

[15] Motorola, Inc. *MC88110 Second Generation RISC Microprocessor User's Manual*, 1991. Part number MC88110UM/AD.

[16] D. E. Nielsen. General Purpose Parallel Supercomputing. Technical Report UCRL-ID-108228, Lawrence Livermore National Laboratory, June 1991.

[17] R. S. Nikhil. Id (Version 90.1) Reference Manual. Technical Report CSG Memo 284-2, MIT Lab. for Computer Science, July 15 1991.

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

[19] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. MIT Press, 1991.

[20] G. M. Papadopoulos and D. E. Culler. Monsoon: An Explicit Token Store Architecture. In *Proc. 17th. Intl. Symp. on Computer Architecture, Seattle, WA*, May 1990.

[21] G. M. Papadopoulos and K. R. Traub. Multithreading: A Revisionist View of Dataflow Architectures. In *Proc. 18th. Intl. Symp. on Computer Architecture, Toronto, Canada*, March 1991.

[22] S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 46–53, May28-June 1 1989.

[23] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pages 50–72, August 1991. Springer-Verlag LNCS 523.

[24] E. Spertus, W. J. Dally, S. Goldstein, K. Schauser, T. von Eicken, and D. E. Culler. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proceedings of the 20th. Annual International Symposium on Computer Architecture, San Diego, CA*, May 17-19 1993.

[25] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th ISCA*, May 1992.