The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms

William J. Dally, Roy Davison*, J.A. Stuart Fiske, Greg Fyler*,
John S. Keen, Richard A. Lethin, Michael Noakes, Peter R. Nuth

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

* Platform Architecture Group
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95052-8131

Abstract

The Message-Driven Processor (MDP) is an integrated multicomputer node that provides efficient mechanisms for parallel computing. It incorporates a 36-bit integer processor, a memory management unit, a router for a 3-D mesh network, a network interface, a 4K-word × 36-bit SRAM, and an ECC DRAM controller in a single 1.1M transistor VLSI chip. Rather than being specialized for a single model of computation, the MDP incorporates efficient primitive mechanisms for communication, synchronization and naming. These mechanisms efficiently support most proposed parallel programming models. Each processing node of the MIT J-Machine consists of an MDP with 1 MByte of DRAM. MDPs have been operational since June 1991 and J-Machines built from them have been on-line since July 1991.

1 Introduction

The Message-Driven Processor (MDP) is a 36-bit single-chip microcomputer specialized to operate efficiently in a multicomputer. The MDP chip includes the processor, a 4K-word by 36-bit memory, and a network port. An on-chip memory controller with ECC permits local memory to be expanded to 1M-words by adding external DRAM chips.

The processor is *message driven* in the sense that processing is performed in response to messages (via the dispatch mechanism). There is no *receive* instruction. A task is created to handle each arriving message. A computation is advanced (driven) by the messages carrying tasks about the network.

The design of the MDP was driven by two primary goals:

1. To implement a general-purpose multicomputer processing node that provides the communication, synchronization, and naming mechanisms required to efficiently support several different parallel programming models.
2. To create an inexpensive *jellybean* VLSI component that can be used to build cost-efficient parallel computers. The nodes should be inexpensive and plentiful VLSI commodity parts (like real jellybean candies) that can be networked together to form a *Jellybean Machine* (J-Machine) multicomputer.

Efficient Parallel Mechanisms Support Many Programming Models: *Mechanisms* are the primitive operations provided by a computer's hardware. The abstractions that make up a programming system are built from these mechanisms [15]. For example, most sequential machines provide some mechanism for a push-down stack to support the last-in-first-out (LIFO) storage allocation required by many sequential programming models. Most machines also provide some form of memory relocation and protection to allow several processes to coexist in memory at a single time without interference. The proper set of mechanisms can provide a significant improvement in performance over a brute-force interpretation of a programming model.

Over the past 40 years, sequential von Neumann processors have evolved a set of mechanisms appropriate for supporting most sequential programming models. It is clear, however, from efforts to build concurrent machines by wiring together many sequential processors, that these highly-evolved sequential mechanisms are not adequate to support most parallel models of computation. These mechanisms do not efficiently support synchronization of events, communication of data, or global naming of objects. As a result, these functions, inherent to any parallel model of computation, must be implemented largely in software with prohibitive overhead. For example, sequential machines require hundreds of instructions to create a new process. This cost prohibits the use of fine-grain programming models where processes typically last only a few tens of instructions.

The MDP supports a broad range of parallel programming models (including shared-memory [16], data parallel [17], dataflow [26], actors [1], and explicit message-passing [5]) by providing low-overhead primitive mechanisms for communication, synchronization, and naming. Communication mechanisms are provided that permit a user-level task on one node to send a message to any other node in a 4K-node machine in $< 2\mu$s. No processing resources on intermediate nodes are consumed and buffer memory is automatically allocated on the receiving node. On message arrival, a task is created and dispatched in $< 1\mu$s. Synchronization is provided by presence tags on all storage locations. Three separate register sets allow fast task switching. The translation mechanism maintains bindings between arbitrary names and values; it may be used to support a global virtual address space. These mechanisms have been selected to be both general and amenable to efficient hardware implementation. To support fine-grain concurrent programming systems, the mechanisms are designed to efficiently handle small objects (8 words) and small tasks (20 instructions).

A 3-D Array of Fine-Grain Processing Nodes is Physically Efficient: The MDP is an example of an inexpensive fine-grain multicomputer building block. Being fine-grain does not mean a node has a slow processor; a competent processor can be built in a fraction of a modern VLSI chip's area. A fine-grain node implies an inexpensive node with a small amount of memory. The result is greater arithmetic performance and local memory bandwidth per unit cost. The fast communication and global address space prevent the small local memories from limiting programmability or performance.

In a multicomputer, system cost is very sensitive to processor cost. A less expensive node results in a comparably priced system with more processors and, to first order, higher performance. In these systems, costly features that give a small incremental return in processor performance (such as large caches) are avoided in favor of building systems with more nodes, an option not available to the designer of a sequential computer.

The 3-D network that connects MDPs gives the highest throughput and lowest latency for a given wire density [10]. This network makes use of all three dimensions of physical space. It allows the processing nodes to be packed densely and results in uniformly short wires. No communication bandwidth is wasted by embedding an esoteric topology into physical space. Messages traveling through the network follow a Manhattan shortest path in physical space; they never backtrack.

Background: The MDP builds on previous work in multicomputer design. Like the Caltech Cosmic Cube [4], the Intel iPSC [3], the N-CUBE [24], and the Ametek [2], each MDP in the J-Machine has a local memory and communicates with other nodes by passing messages. Because of its low overhead, the MDP can exploit concurrency at a much finer grain than these early message-passing multicomputers. Delivering a message and dispatching a task in response to the message's arrival takes $< 2\mu$s on the J-Machine, as opposed to 5ms on an iPSC-1 or 300μs on an iPSC-2. Like the BBN Butterfly [19] and the IBM RP3 [8], the MDP supports a global virtual address

Figure 1: MDP Pinout: The MDP has a memory port (26 pins), six network ports (15 pins each) and a diagnostic port (3 pins).

space. The same IDs (virtual addresses) are used to reference local (on the same node) and remote (on a different node) objects. Like the InMOS transputer [6], the Caltech MOSAIC [20], and the Intel iWARP [7], the MDP is a single chip processing element integrating a processor, memory, and a communication unit. The MDP is unique in that it extends these previous efforts with efficient primitive mechanisms for communication, synchronization and naming [15]. The MDP uses a direct communication network based on the work reported in [12, 10, 14].

The MDP was built as part of a collaborative project between MIT and Intel Corporation. The project was funded by the Defense Advanced Research Projects Agency.

Outline: The next section describes the system architecture of the MDP from hardware and software points of view. Section 3 describes the MDP instruction set with emphasis on instructions that support parallelism. The network architecture is discussed in Section 4. Section 5 briefly describes the implementation of the MDP with attention given to the subsystems that implement the MDP's mechanisms. MDP software is addressed in Section 6.

2 System Architecture

Hardware View: To the hardware designer, the MDP appears as a component with a memory port, six bidirectional network ports, and a diagnostic port as shown in Figure 1.

The memory port provides a direct (no glue) interface to up to 1M-word of ECC DRAM memory consisting of 11 multiplexed address lines, a 12-bit data bus, and 3 control signals. Static-column or page mode DRAMs are cycled three times to access a 36-bit data word and a fourth time to check or update the ECC check bits. Current J-Machines use 3 1Mx4 memory parts to form a 4-chip processing node with 260KW of memory that measures 2x2.75 inches as shown in Figure 2.

The network ports are used to connect MDPs together into a 3-dimensional mesh network. Each of the six network ports corresponds to one of the six cardinal directions ($+X, -X, +Y, -Y, +Z, -Z$) and consists of nine data and six control lines. Each port connects directly to the opposite port on an adjacent MDP. Details of the 3-D network are given in Section 4.

The diagnostic port is used to issue supervisory commands and to read and write MDP memory

Figure 2: An array of four J-Machine processing nodes. Each node consists of one MDP chip and three 1Mx4 static-column DRAMS. With conventional packaging, the node measures 2x2.75 inches.

Figure 3: A combining tree sums results produced by a distributed computation. Each node sums the input values as they arrive and then passes a result message to its parent.

from a console processor. The port consists of two control lines, a serial input, and a serial output. Using this port, a console processor can read or write any location in the MDP's address space, as well as reset, interrupt, halt, or single-step the processor.

Software View: To a systems programmer, a bare J-Machine appears as a collection of node memories and register files that can be operated on by an instruction set that includes communication, synchronization, and naming mechanisms. The systems programmer uses these mechanisms to implement a programming model. For example, one can build a shared memory model that gives the application programmer a single, shared address space.

The implementation of a combining tree [27] illustrates the use of the MDP mechanisms. The combining tree (Figure 3) consists of a number of nodes each containing a value, a count, and a pointer to a parent node. The value is initialized to zero and count is initialized to the number of inputs expected. Summing a set of values computed by a number of processes distributed across the nodes of the machine is accomplished by each process sending a `combine` message containing its result to a combining node. When the message arrives at the processor containing the combining node, a task is created that executes a `combine` routine. The routine adds the message value to the node's value and decrements the count. When the count reaches zero, a `combine` message is sent to the node's parent.

The MDP supports communication using a `SEND` instruction for message formatting, a fast network for delivery, automatic message buffering, and task creation upon message arrival. A series of `SEND` instructions is used to send a message of arbitrary length to any node in the machine. Upon arrival, the message is buffered in a hardware queue. When the message arrives at the head of the queue, a task is dispatched to handle the message. In the combining tree example, a pair of `SEND` instructions is used to send the `combine` message to a node. Upon message arrival, the MDP buffers the message and creates a task to execute the `combine` routine.

Synchronization in the MDP is performed using message dispatch and presence tags on all state.

The dispatch of a process on each message arrival allows messages to be used to signal events on remote nodes. For example, in the combining tree example above, each combine message signals its own arrival and causes the `combine` routine to be executed.

Presence tags are used to synchronize on the arrival of data. For example, access to the value produced by the combining tree may be synchronized by initially tagging as *empty* the location that will hold this value. An attempt to read this location before the combining tree had written it would raise an exception and cause the reading task to be suspended until the value is written by the root of the tree. Synchronization on data availability in this manner is quite common in many parallel programs.

The MDP supports naming with segmented memory management and with translation instructions. In the combining tree example, a memory segment would be allocated to hold the state of each combining node. Accesses to the node would be relocated and protected using a segment descriptor. To make combining nodes relocatable across processing nodes, a node's virtual address would be translated to find the processing node on which it resides. Upon reaching this node, a second translation would locate the segment descriptor for the combining node.

3 Instruction Set Architecture

The MDP extends a conventional microprocessor instruction-set architecture (ISA) with instructions to support parallel processing. Specifically, the MDP provides efficient hardware mechanisms for communication, synchronization and naming. This section describes the MDP ISA with particular emphasis on these mechanisms. Further details of the MDP ISA are described in [11].

Register Set: The MDP provides separate register sets to support rapid switching between three execution levels: background, priority 0 (*P0*), and priority 1 (*P1*). The MDP executes at the background level when there are no pending messages. Each arriving message creates a task and initiates execution at *P0* or *P1* depending on the priority of the message. The MDP executes the highest priority task at any point in time. Thus, the arrival of a *P1* message while the MDP is executing a *P0* task will cause the MDP to switch execution levels (and thus register sets). When the *P1* task completes, execution at *P0* is resumed by switching to the *P0* register set which holds the register state of the suspended task.

The register set at each priority level includes:

4 general-purpose data registers:	`R0-R3`
4 address registers:	`A0-A3`
4 ID registers:[1]	`ID0-ID3`
1 instruction pointer:	`IP`

Most instructions operate on the general registers `R0-R3`. Each address register `A0-A3` contains a segment descriptor consisting of a base and a length field. Memory addresses are specified by an offset and an address register. For example, the operands `[R0,A1]` and `[3,A2]` specify an indexed access to the segment described by `A1` and a displacement of 3 into `A2`'s segment. `ID` registers are less accessible general registers that by convention are used to hold object IDs. The instruction pointer includes process status bits that control virtual addressing, type checking, and fault handling. Placing these bits in the instruction pointer enables control and execution state to be changed by loading a single register. Quick task switching within an execution level is facilitated by the relatively small size of each register set.

[1]The background register set does not include ID registers. They only exist at *P0* and *P1*.

Figure 4: MDP Instruction Format

Tags: Tags in the MDP are used for type checking and synchronization. Every 36-bit word of register and memory state holds a 32-bit value and a 4-bit tag that indicates the type of the value. Tag values are defined for primitive user data types (such as `SYMBOL`, `INTEGER`, `BOOLEAN`, etc.) and for system data types (such as `IP`, `ADDR` (a segment descriptor), `MSG` (a message header), etc.). Four tag values are user-definable. If type checking is enabled, the MDP checks operand tags to determine which form of an instruction to execute. An exception is raised if the operands are not compatible with the instruction.

Two tags, `FUT` and `CFUT`, support inter-task synchronization. A location is initially marked empty by writing it with a `CFUT` tag. When a task produces the value for the location, it overwrites the `CFUT` with the final value and tag. Any attempt to read from the location before the value is produced invokes the `CFUT` fault handler which typically suspends the reading task until the location is written. `FUT` is used for global synchronization, and `CFUT` for local.

Hardware support for tags makes software more efficient and robust. A program can perform an operation without having to check that operands are present or of the correct type. For the "normal" cases in which no fault occurs, execution proceeds more quickly than if special test and branch instructions were required to check for type and presence. Only exceptional cases incur the overhead of running a fault handler.

Instructions: The MDP executes 17-bit, fixed format, three-address instructions with the format shown in Figure 4. Each instruction specifies an operation, two register operands, and a third operand that may be a register, a memory location, or a constant. Two 17-bit instructions are packed in each 36-bit word. Any instruction-stream word not tagged as an instruction is loaded as a constant into register `R0`. This provides a very efficient means to load arbitrary 36-bit constants. The MDP instruction set is summarized by category in Figure 5.

Naming: Naming is supported in the MDP via translation instructions and segmented addressing. Addressing memory through segment descriptors permits arbitrary sized objects to be relocated and protected. The `ENTER` instruction enters an arbitrary translation from a 36-bit key to a 36-bit data value in a set-associative cache (translation table) mapped into the on-chip memory. The `XLATE` instruction looks up the data value (if any) associated with a key. These instructions can be used to translate an object's name into a physical segment descriptor or a node number to support a global virtual address space.

Communication: The MDP provides hardware support for end-to-end message delivery including formatting, injection, delivery, buffer allocation, buffering, and task scheduling.

An MDP transmits a message using a series of `SEND` instructions each of which injects one or two words into the network at either priority 0 or 1. A typical message send is shown in Figure 6.

General Movement and Type Instructions:

READ	WRITE	READR	WRITER	RTAG	WTAG	LDIP
LDIPR	CHECK					

Arithmetic and Logic Instructions:

CARRY	ADD	SUB	MULH	MUL	ASH	LSH
ROT	AND	OR	XOR	FFB	NOT	NEG
LT	LE	GE	GT	EQUAL	NEQUAL	EQ
NEQ						

Network Instructions:

SEND	SENDE	SEND2	SEND2E

Associative Lookup Table Instructions:

XLATE	ENTER	PROBE

Special Instructions:

NOP	INVAL	SUSPEND	CALL

Branches:

BR	BNIL	BNNIL	BF	BT	BZ	BNZ

Figure 5: Six Categories of MDP Instructions.

```
SEND    R0,0          ; send net address (priority 0)
SEND2   R1,R2,0       ; header and receiver (priority 0)
SEND2E  R3,[3,A3],0   ; selector and continuation - end msg. (priority 0)
```

Figure 6: MDP assembly code to send a 4 word message uses three variants of the `SEND` instruction.

Figure 7: Message dispatch. In three clock cycles, a new task is created by (1) setting the IP to change the thread of control and (2) creating a message segment to provide the initial addressing environment.

The first `SEND` instruction reads the absolute address of the destination node in $< x, y, z >$ format from `R0` and forwards it to the network hardware. The `SEND2` instruction reads the first two words of the message out of registers `R1` and `R2` and enqueues them for transmission. The final instruction enqueues two additional words of data, one from `R3`, and one from memory. The use of the `SEND2E` instruction marks the end of the message and causes it to be transmitted into the network. This sequence executes in 4 clock cycles (250ns).

An injected message is delivered to the destination node by the network (Section 4). At the destination, it is buffered in a hardware-managed FIFO queue in the internal RAM of the MDP. Separate queues are maintained for *P0* and *P1* messages.

Task Scheduling: When a message reaches the head of the highest priority non-empty queue, a task is created to handle it by changing the thread of control and creating a new addressing environment as shown in Figure 7. Every message header contains an *opcode* and the message *length*. The message opcode is loaded into the `IP` to start a new thread of control. The length field is used along with the queue head to create a message segment descriptor (automatically written to `A3`) that represents the initial addressing environment for the task. The message handler code may open additional segments by translating object IDs in the message into segment descriptors. Creating a task to handle a message takes 3 cycles.

The dispatch mechanism is used directly to process messages requiring low latency (e.g., combining and forwarding). Other messages (e.g., remote procedure call) specify a handler that locates the required method, using the translation mechanism described above, and then transfers control to it.

For example, a remote procedure call message is handled by the `CALL` handler code as shown in Figure 8. The execution of this handler is depicted in Figure 9. The first instruction gets the method ID (offset 1 into the message segment referenced by `A3`). The next instruction translates this method ID into a segment descriptor for the method and places this descriptor in `A0`. In one of its operating modes, the MDP can use `A0` as a pointer to a segment of code, and `IP` as an index into that segment. This allows code to be easily relocated at runtime. The final instruction of the `CALL`

```
MOVE    [1,A3],R0   ; get method id
XLATE   R0,A0       ; translate to segment descriptor
LDIP    INITIAL_IP  ; load instruction pointer to transfer control to method
```

Figure 8: MDP assembly code for the `CALL` message.

Figure 9: The CALL message invokes a method by translating the method identifier to find the code, creating a context (if necessary) to hold local state, and translating argument identifiers to locate arguments.

handler transfers control to the method by loading the `IP` with a short integer offset. Thereafter, the MDP will fetch instructions from the called method.

The method code may then read in arguments from the message queue. The argument object identifiers are translated to physical memory base/length pairs using the `XLATE` instruction. If the method needs space to store local state, it may create a context object. When the method has finished execution, or when it needs to wait for a reply, it executes a `SUSPEND` instruction which dequeues its message and passes control to the next message in the queue.

An example of a direct message handler is the combine routine of Figure 3 in Section 2. The code for this routine is shown in Figure 10. If the node is idle, execution of this routine begins three cycles after message arrival. The routine loads the combining node pointer and value from the message, performs the required add and decrement, and, if `count` reaches zero, sends a message to its parent.

This 12-instruction routine takes 21 cycles to execute. It demonstrates several ways in which the MDP's communication mechanism reduces the overhead of message passing to the point where it can be used for simple operations such as combining.

- MDP hardware dispatches the combine task by setting the instruction pointer to `COMBINE` and initializing message pointer `A3` to allow direct access to message words. This avoids the overhead otherwise associated with control transfer and with setting up an addressing environment.

```
COMBINE: MOVE   [1,A3], COMB            ; get node pointer from msg
         MOVE   [2,A3], R1              ; get value from msg
         ADD    R1, COMB.VALUE, R1      ;
         MOVE   R1, COMB.VALUE          ; store result
         MOVE   COMB.COUNT, R2          ; get count
         ADD    R2, -1, R2              ;
         MOVE   R2, COMB.COUNT          ; store decremented count
         BNZ    R2, DONE
         MOVE   HEADER,R0               ; get message header
         SEND2  COMB.PARENT_NODE, R0    ; send message to parent
         SEND2E COMB.PARENT, R1         ; with value
DONE:    SUSPEND
```

Figure 10: MDP Assembly code for the combining tree example.

- The two `SEND` instructions transmit the four word message to the parent task. The message is transmitted directly from register and memory variables without any need to first format it in memory.
- The `SUSPEND` instruction terminates the task and simultaneously dequeues the message. If another message is pending in the queue, a task to handle it will be dispatched two cycles after the execution of the `SUSPEND` instruction.

4 Network Architecture

The Message Driven Processor contains a network interface and a router that support a communication network closely integrated with the processor. In a J-Machine composed of MDPs, the network provides end-to-end message delivery with low latency ($< 2\mu$s in a 4K-node network) and high bandwidth (288 Mbits/sec per channel). Message delivery is handled entirely within the routers of the machine and consumes no processor or memory resources at intermediate nodes.

Structure: The J-machine network is a three dimensional grid, with bidirectional channels, dimension-order routing, and blocking flow control. (See Figure 11). Addressing limits the size of the network to 64K nodes ($32 \times 32 \times 64$). Our initial prototype is a 1K node machine ($8 \times 8 \times 16$). The faces of the network cube are left open for use as I/O ports to the machine. Each channel can sustain a data-rate of 288Mbits/sec. All three dimensions may operate simultaneously for an aggregate data rate of 864Mbits/sec per node.

The network logic is composed of three modules as shown in Figure 14. The netout module buffers words and injects them into the network. The three routers, one for each dimension of the network, route messages from node to node. The netin module reassembles messages at their destination and buffers them into a message queue. Section 5 contains more implementation details.

Engineering: The three-dimensional mesh topology of the J-Machine network was chosen as the most efficient arrangement subject to constraints of wiring density and component pinout [10]. These constraints set the width of the six bidirectional channels per MDP node at 9 data bits plus 6 control bits. The J-machine is built as a stack of boards with dense board-to-board interconnections (Section 5) to implement the 3-D network with short wires.

Figure 11: The J-Machine Network is a 3-D mesh or k-ary 3-cube. The network performs e-cube or destination tag routing. Messages are routed in each dimension in turn to the proper coordinate in that dimension. In this figure, a message is routed from (1,5,2) to (5,1,4) routing first in x, then y, then z.

The MDP router is synchronous[1] and operates at twice the rate of the processor sending a pair of 9-bit *phits* between nodes each 62.5ns processor cycle[2]. Each of the six bidirectional channels can be turned around on alternate cycles with no contention penalty. A novel pad design tolerates clock skew between routers and eliminates the potential for conduction overlap when the channel is turned around [23]. Messages are routed through the network with a latency of one 62.5ns processor cycle per hop. Thus message latency T is given by

$$T = T_c(2L + D), \tag{1}$$

where T_c is the processor cycle time, L is message length in words, and D is the distance (number of nodes) a message must traverse. For example, in a 1K-node machine, an L=6 word message to a random destination traverses an average of D=10 nodes for a latency of T=22 cycles or 1.4μs. The bisection bandwidth[3] of a 1K-node machine is 18.4Gbits/s, the aggregate bandwidth of the network channels is 864Gbits/s, and the I/O bandwidth is 184Gbits/s.

Routing & Flow control: The J-machine uses deterministic dimension-order routing (also called *e-cube* routing). As shown in Figure 11, all messages are first routed in the X dimension, then in Y, then in Z. Since messages are routed in dimension order and messages running in opposite directions along the same dimension do not block, there are no resource cycles, and the network is provably deadlock free [13].

Figure 12 shows the format of a message. The first three flits of the message contain the X, Y and Z addresses. At each node along the path, the address in the head flit of the message is compared with the node's index in the current dimension. If the two indices match, the head flit is stripped off the message, and the rest is routed down to the next dimension. The MDP's

[1] This breaks with a tradition of asynchronous network routers [12, 14].

[2] A *phit* is a physical digit, the width of the physical channel. A pair of phits form a *flit*, or flow-control digit, the granularity of flow-control in the network. An 18 bit flit is half an MDP data word.

[3] The bandwidth across a plane dividing the machine into two equal halves.

Flit	Contents	Remark
1	5:+	X address
2	1:−	Y address
3	4:+	Z address
4	MSG:440	Method
5		to call
6	INT:23	Argument
7		to method
8	INT:<1:5:2>:T	Reply
9		address

Figure 12: A typical message in the J-machine. The first three flits contain the destination address. The final flit in the msg is marked as the "tail".

Figure 13: The J-Machine Network performs blocking flow control with two stages of queueing per node. (a) Message arrives at busy channel. (b) Message becomes compressed by queueing. (c) Channel is available; message continues advancing.

netout block formats the address flits of the message. It also precomputes the direction (positive or negative) in which the message must travel along each dimension, setting additional bits in the address flits. This reduces the latency and complexity of the router nodes.

The network uses blocking flow control to resolve contention for a physical channel (Figure 13). When a message arrives at a router path that is already in use by a message of the same priority, it is blocked. The blocked message compresses into routers along its path, occupying one node per word (2 flits) of the message. When the blockage clears, the message uncompresses and proceeds to its destination, at a rate of one hop per cycle.

Two priorities of messages share the physical wires, but use completely separate buffers and routing logic. This allows priority 1 messages to proceed through blockages at priority 0. Without this ability, there would be no way to redistribute data that has caused hot-spots in the network.

5 MDP Implementation

The major subsystems in the MDP are shown in Figure 14. The chip includes a conventional microprocessor with prefetch, control, register file and ALU (RALU), and memory blocks. The communication system comprises the routers and network input and output interfaces. The *Address*

Figure 14: MDP block diagram.

Arithmetic Unit (AAU) provides addressing functions. In addition, a DRAM interface and control block and diagnostic interface are included.

Communication Subsystem: The communication subsystem contains the network output, the network input, and the routers.

The network output block buffers messages from the RALU and injects them into the network. A FIFO buffer is used to match the speed of message transmission to the network. On each `SEND` instruction, the RALU transfers one or two words to the FIFO. When the message is complete, or the eight-word buffer is full, the message is launched into the network. In cases where the MDP cannot send message words as fast as the network can transmit them, the FIFO prevents *bubbles* (absence of words) from entering the network pipeline and degrading performance.

The network input block transfers messages from the network to the MDP's memory. Data from the network arrives in 18-bit flits, which are composed into a four word *Queue Row Buffer* (QRB). When the QRB fills, its contents are written to the on-chip memory in one cycle. Writing memory a row (4×36 bits) at a time reduces the number of memory cycles consumed by the network, leaving more memory bandwidth for the CPU.

The routers form the switches in a J-Machine network and deliver messages to their destinations. As shown in Figure 15A, the MDP contains three independent routers, one for each bidirectional dimension of the network. Each router is split into two separate virtual networks with different priorities that share the same physical channels. The priority 1 network can preempt the wires even if the priority 0 network is congested or jammed.

Each of the 18 router paths (Figure 15B) contains buffering, comparators and output arbitration. On each datapath, a comparator compares the lead flit, which contains the destination's address in

Figure 15: Block diagram of the routers. (A) The two priorities per dimension are completely separate except where they share the physical channels. (B) Each priority contains forward, reverse, and previous to next dimension datapaths.

this dimension, to the node coordinate. If the head flit does not match, the message continues in the current direction. Otherwise the message is routed to the next dimension. Messages entering the dimension compete with messages continuing in the dimension at a 2-to-1 switch. Once one message is granted this switch, the other input is locked out for the duration of the message. Once the head flit of the message has set up the route, subsequent flits follow directly behind it.

Address Arithmetic Unit: The Address Arithmetic Unit (AAU), the largest logic block in the MDP, performs all functions associated with memory addressing. To support naming, the AAU contains the address registers, performs relocation and protection of memory accesses, and implements the translation instructions. Each memory reference is offset by the selected address register's base field and checked against its length field. An attempt to access through an invalid address register (which may occur when an object is relocated) or to access beyond the end of an object raises an exception. A *translation base-mask* (`TBM`) register is used to define an area of memory to be a two-way set-associative translation buffer used by the `XLATE` and `ENTER` instructions. The AAU hashes the keys used to access this table using an exclusive-or network to improve hit-rate in the translation buffer.

The AAU maintains two queues to buffer incoming messages and schedule the associated tasks. Associated with each queue are a *queue base-mask* (`QBM`) and *queue head-length* (`QHL`) register (Figure 16). The `QBM` registers define the position and length in main memory of the message queues. Queues are circular, so messages at the end of the queue wrap around to the beginning. The `QHL` registers point to the beginning of the first message in the queue and its length field encompasses exactly all of the messages currently in the queue. When a task is dispatched to handle a message, the `A3` register is loaded with a segment descriptor for the message. The MDP dispatches a task as soon as the first four words of a message are written. If the task attempts to read a word of the message which has not yet arrived, a special `EARLY` fault occurs.

Figure 16: The AAU maintains the Queue Base/Mask (QBM) registers which specify the location of the message queue in main memory, and the Queue Head/Length QHL register which specifies the beginning and end of the messages received. Only one of the two queues is shown.

Layout: Figure 17 shows a floorplan of the chip with a die photograph for comparison. The area breakdown is shown in Figure 18.

Methodology: The MDP was implemented using Intel standard cells except for the on-chip RAM, clock generator, and pads. Using standard cells sacrificed a factor of 3-4 in area and 2-3 in performance over what would be possible with full-custom design. The advantage was a significant increase in productivity which was essential to completing the chip successfully with our small design team.

About $35K$ standard cells containing $\approx 210K$ transistors[4] were used in the ≈ 700 sheets of schematics drafted at MIT. These schematics were transferred to Intel for layout. Many of the datapaths were laid out by hand to exploit the regularity of the design. Less regular collections of logic were laid out using automatic place and route CAD tools.

Architecture studies leading to the MDP were started in October 1986. Work on the RTL model of the microarchitecture began in June 1988. Schematic entry at MIT started in November 1988. The task of translating schematics into layout commenced in June 1989. Layout was completed in December 1990. First silicon was received in June 1991 and was running programs within a few hours.

Although we thoroughly simulated the logic design, running our validation tests and applications on the hardware has uncovered 12 bugs. Some of these bugs have simple software work-arounds, but for performance reasons a second revision of the layout with modified control logic and some metal fixes was sent for fabrication in January 1992. We plan to use several thousand of these chips to build research multicomputers at MIT.

System Design: A photograph of a J-Machine processor board is shown in Figure 19. The board measures 20.5 by 24 inches and contains 64 processing nodes. Each node consists of an MDP chip (in a 168-pin PGA package) and three 4Mbit DRAMs. Each pair of nodes shares a set of elastomeric connectors to communicate with the corresponding nodes on the boards above or below the board

[4] The remaining $\approx$ 890K devices are contained in the full custom portions of the chip; mostly in the RAM.

Figure 17: MDP chip floorplan and die photograph

Module	Dimensions $mm \times mm$	Area mm^2	Transistors $\times 10^3$
AAU	3.7×7.0	25.9	75
RALU	3.7×2.9	10.7	39
DIAG	0.9×1.1	1.0	3.7
PREF	0.9×1.1	1.0	3.2
CNTL	1.1×2.6	2.9	8.7
IMEM	7.8×0.5	3.9	13
EMI	1.6×1.8	2.9	9
NETIN	1.8×0.7	1.3	4.4
NETOUT	2.1×1.8	3.8	18
ROUTERS	8.4×1.3	10.9	29
RAM	8.8×4.9	43.1	880
CLOCK	0.7×0.8	0.6	0.1
PADS	50.5×0.2	8.4	2.6
Full Chip	10.2×15.0	153.0	1087

Figure 18: Chip Area Breakdown.

in a stack. A total of 32 elastomeric connectors held in 4 connector holders provide 2240 electrical connections between adjacent boards. Of these connections, 960 are used for signalling and the remaining are ground returns. No power is supplied through the elastomers. Power and ground are supplied to each board directly from bus bars. The center area of the board contains the final stage of the clock distribution network along with diagnostic fanout and multiplexing logic and temperature and airflow monitors.

A photograph of our chassis for a 1024-node system is shown in Figure 20. The chassis contains a *stack* of 16 processor boards, power supplies, and distribution bus bars. The boards are held together by a set of 20 tie-rods to compress the elastomer connectors. A 4096-node system can be built by combining four chassis. Each stack is connected to its neighboring stacks by 128 (16×8) short 60-pin ribbon cables (one for each pair of nodes on the periphery). Each vertical pair of stacks shares a 3000ft^3/min blower for cooling.

In addition to the processor board and chassis, we have also designed a diagnostic interface board, and are in the process of designing a SCSI disk interface, a distributed graphics frame buffer, and an S-bus interface. More details of the J-Machine system design are given in [22].

6 Software

The J-machine is intended as a platform for software experiments in fine-grained parallel programming. To this end, we have implemented and are studying software systems for different fine-grain programming models. Fine-grained programs typically execute from 10-100 instructions between communication and synchronization actions. Reducing the grain size of a program increases both the potential speedup due to parallel execution and the potential overhead associated with parallelism. Special hardware mechanisms to reduce the overhead due to communication, process switching, synchronization, and multithreading are therefore central to the design of the MDP. Software issues such as load balancing, scheduling, and locality remain open questions and are the focus of current research efforts.

A parallel processor creates programming challenges. It is difficult to extract the fine-grain parallelism needed from stock programs written in C or FORTRAN. Instead of concentrating on

Figure 19: Photograph of 64-node J-Machine System

Figure 20: 1024-node J-Machine Chassis

```
(defmethod size-of-tree pair ()
  (+ (size-of-tree left)
     (size-of-tree right)))

(defmethod size-of-tree object ()
  1)

(defmethod size-of-tree null ()
  0)
```

Figure 21: Concurrent Smalltalk Source to compute the size of a tree. Methods definitions specify the class to which they apply. The class `pair` contains two elements, `left` and `right`, each of which may hold an object or another pair.

the extraction of parallelism from existing programs (an active and interesting area of research for many parallel programming researchers) or on the adaptation of sequential languages for the parallel domain, our current approach focuses on languages where the expression of fine-grained parallelism is much cleaner. To date, two languages have been implemented to the J-machine: The actor language Concurrent Smalltalk (CST) and the dataflow language Id.

Concurrent Smalltalk: CST [18] is a parallel object-oriented programming language (based on the Actor model [1]) with asynchronous message send and distributed objects. Syntax is similar to that of LISP or SCHEME. Method or function invocation is performed by sending a message to the first argument of the method. The message contains the method selector and the rest of the arguments.

Functions and methods in the language are compiled into MDP assembly code by an optimizing compiler, and assisted at run time by a small kernel called COSMOS. COSMOS provides a global virtual namespace, object-based memory management, support for distributed objects, and low-overhead context switching. Its memory management system provides fast, transparent access to storage distributed across the machine. COSMOS is designed to efficiently support fine-grain concurrent computation where tasks are very short (40 user instructions) and data objects are very small (8 words). The CST compiler and the COSMOS runtime system also provide floating-point arithmetic, simple arrays, and garbage collection for CST programs. COSMOS manages contexts, futures, and objects, and therefore plays an important role in providing services that exploit the communication, synchronization, and naming mechanisms of the J-machine.

A small sample program is shown in Figure 21. A method called `size-of-tree` is defined for three object types, a `pair`, the null object, and a generic object. When called on a LISP style tree, these methods will return the number of generic objects stored in the tree. For example, called on the tree `'((1 2 3)(4 5 6)(7 8 9)) size-of-tree` returns the value 9. Note that since a `pair` and `null` are subclass of `object`, their more specific methods are selected when `size-of-tree` is invoked on their types.

When this example program is compiled, a selector object and three function objects are defined. The selector object (shown in Figure 22) is a list of the type/function correspondence. When a method is applied to an object, the object type is examined and the appropriate function is located in the selector object. This function is then invoked on the object[5]. The selector object is marked as copyable and maintained like any other object.

[5]In cases where the compiler can infer the type of the object or when the type of objects is explicitly declared, a method invocation is optimized directly to the correct function invocation.

```
MODULE  OBJ:Selector.Size_Of_Tree

DC      Copyable|class_Selector         ; Identify Properties of object
DC      OBJ:Selector.Size_Of_Tree       ; Store own id inside object

DC      3                               ; Number of functions
DC      CLASS:Object                    ; Class identifier for Object
DC      {function.Size_Of_Tree}         ; Function for class object
DC      CLASS:Null                      ; Class identifier for Null
DC      {function.Size_Of_Tree_1}       ; Function for class Null
DC      CLASS:Pair                      ; Class identifier for Pair
DC      {function.Size_Of_Tree_2}       ; Function for class Pair
END
```

Figure 22: Selector object generated by the example program

An example of the compiled code for the function for the class `pair` is shown in Figure 23. When a method application directed at a particular object is received, COSMOS will examine the object class and the selector object, and choose the correct function to invoke.

The function first does an `XLATE` operation to get the address of the `pair`, and uses that address to get the object ID for `left`. It then calls COSMOS to find the node on which `left` lives. A message is sent to `left` that recursively applies the `size-of-tree` method. The slot that holds the return value of this call is marked `CFUT`. Next, `size-of-tree` is applied to `right` without waiting for the result of the first remote procedure to return. However, when the function attempts to add the two return values, the results will probably not have returned yet, so the `add` instruction will fault trying to add `CFUT`s, and the process will be suspended, saving its registers into the context.

Assuming this happens, when the replies from the methods are received, after writing the value into the future slot, COSMOS checks to see if the process was waiting for that particular future. If so, the context is reactivated. The reactivated function would then perform the sum of the two results and forward them to the continuation specified in the original method invocation.

Some interesting points:

- If the object of the function is not present or if the translation cache does not have an entry for the object, the `XLATE` instruction will fault and COSMOS will find the object, moving or copying it to the local node.
- Functions and selectors are objects maintained like any other immutable object. If they are not present, they will be copied to the node. This process is analogous to a distributed instruction cache.
- If the function were preempted and the object moved or migrated away, the address registers would be invalidated, and accesses to the object would cause a fault that would attempt to re-translate or re-obtain the object.
- The `A1` register points to the current context. The context contains storage that could be used to spill register values if the context faults, and might also contain working values. In the example, the futures are constructed in the context (and thus the name *context-future* (`CFUT`).

This example illustrates some important research questions related to the efficiency of this model of computation:

```
    MODULE  OBJ:function.Size_Of_Tree_2

    DC      Copyable|class_Function         ; Identify properties of object
    DC      {OBJ:function.Size_Of_Tree_2}   ; Store own ID inside object

    ;; Incoming: A1 points to the context
    ;;          A3 points to the message

START:
    MOVE    [2,A3],R3                       ; Get the pair's object ID
    XLATE   R3,A2                           ; Find the pair's local addr
    MOVE    [2,A2],R0                       ; Get the object ID of LEFT
    CALL    objectNode,R1                   ; Find LEFT's node -> R1
    DC      MSG:Apply_Selector              ; Send a message to LEFT to apply
    SEND2   R1,R0                           ;   the method specified by the
    DC      {OBJ:Selector.Size_Of_Tree}     ;   selector for Size_Of_Tree.
    SEND    R0                              ;   Includes our context ID
    SEND    [2,A2]                          ;    and a continuation.
    MOVE    5,R0                            ;
    SEND2E  [1,A1],R0
    WTAG    R0,CFUT,R0                      ; Make a future for LEFT's
    MOVE    R0,[5,A1]                       ;   result.
    MOVE    [3,A2],R0                       ; Do the same for RIGHT as to
    CALL    objectNode,R1                   ;   LEFT.
    DC      MSG:Apply_Selector
    SEND2   R1,R0
    DC      {OBJ:Selector.Size_Of_Tree}
    SEND    R0
    SEND    [3,A2]
    MOVE    6,R0
    SEND2E  [1,A1],R0
    WTAG    R0,CFUT,R0
    MOVE    R0,[6,A1]
    MOVE    [6,A1],R2                       ; Do the sum.
    ADD     R2,[5,A1],R1
    MOVE    [3,A3],R3                       ; Get the continuation for
    BNIL    R3,^L001                        ;   this context.  Reply if
    DC      MSG:Reply                       ;   non-nil.
    SEND2   R3,R0
    SEND    R3
    SEND2E  [4,A3],R1
L001:
    SUSPEND
    END
```

Figure 23: Compiled code for the `size-of-tree` function for objects of class `pair`.

- When is it better to spawn processes nonlocally rather than locally? This is probably a strong function of the amount of associated overhead. The MDP architecture attempts to reduce this overhead, but algorithms for making this tradeoff at compile and run time still need to be developed and evaluated.
- How should objects be placed on the machine, and how should they migrate in order to reduce the overhead of communication?
- In some cases, the amount of parallelism grows to be much larger than the machine can handle. We need to study how we can effectively and automatically throttle the parallelism being created when the machine is saturated.

These issues, and others related to the efficiency of programming fine-grained parallel processors, are discussed in more detail in [18].

Dataflow Implementation: Id [21] is a functional programming language originally designed for dataflow architectures. An Id program can be converted into a dataflow graph, in which operators are represented by nodes and dependencies by arcs. Originally, these dataflow graphs were executed directly on specialized dataflow machines; more recently, researchers have begun compiling dataflow graphs to run on general-purpose parallel machines[9]. Dataflow programs are well-suited to large parallel computers, because the abundance of fine-grained tasks, each of which can be as small as a single dataflow operator, makes it easy to mask communication latency with task switches. Conversely, the J-machine's fine-grained mechanisms make it an excellent target for dataflow programs.

We have experimented with several methods of executing dataflow programs on the J-machine [25]. The simplest of the systems translates each node of the dataflow graph into a sequence of MDP instructions. A dataflow node with two inputs takes 20 MDP instructions to simulate, i.e., to store the first data value, match it with the second value when it arrives, perform the dataflow operation, and send the resulting value to two destinations. This makes use of the `CFUT` tag and fault handler.

A more efficient approach is to increase the granularity each task in order to reduce scheduling overhead. We are currently building a system on top of the Berkeley TAM project[9] that addresses the inefficiencies of our earlier systems.

7 Concluding Remarks

The MDP has been built to demonstrate the utility of general-purpose communication, synchronization and naming mechanisms in a multicomputer building block. Its mechanisms have been shown to efficiently support dataflow [25] and object-oriented programming [18] models using a shared address space. Our studies have shown that the use of a few simple mechanisms provides orders of magnitude lower communication and synchronization overhead than is possible with multicomputers built from off-the-shelf microprocessors. Communication and synchronization performance is competitive with processing nodes specialized to a single model of computation such as iWARP [7] (systolic) or the Transputer [6] (CSP).

Computers built from fine-grain processing nodes, such as the MDP, consisting of a small but powerful processor and a small amount of memory, are more cost-effective than those built from fewer coarse-grain nodes. Fine-grain nodes have a larger fraction of their silicon area devoted to processing and have higher arithmetic, memory, and communication bandwidth per unit cost. Large scale parallel machines built from fine-grained processors have a larger total amount of memory within a given latency of a processor. An efficient network design provides global memory latency and bandwidth competitive with coarse grain machines.

The MDP is a component for building scalable computer systems. It is useful in configurations ranging from one node to 64K nodes. A 128-node Jellybean machine (J-Machine) is currently operational and resources are in place to build several more machines, including a 1024-node system at MIT and machines at a number of other research institutions.

The MDP project has demonstrated the feasibility of building experimental computer systems with limited resources. By concentrating on the novel mechanisms of the MDP and keeping the design simple and modest in other respects we were able to complete the design of the chip, its system level hardware, and several programming systems with a handful (< 8) of graduate students and engineers in 2.5 years.

With the MDP we have begun exploring mechanisms for parallel computers. Much work remains to be done to tune the MDP's mechanisms and compare them to alternatives. The demands of parallel software that drive these mechanisms are very different from the demands placed on sequential computers. The design of mechanisms for parallel computers is particularly challenging because there are no well established parallel benchmarks and most parallel programs are very biased by the mechanisms (or lack thereof) of the machines for which they were initially written.

Our software studies have suggested improvements that could be made to the MDP. More registers, as well as better mapping mechanisms would be useful. The current implementation is very conservative, so there are certainly opportunities for streamlining the implementation, decreasing the cycle time and number of clocks per instruction. A commercial custom VLSI product based on the architectural mechanisms in the MDP is very plausible.

As technology scales, it becomes feasible to put many powerful processing units on a single chip. An interesting direction for further research is to extend the MDP mechanisms to control intra-node as well as inter-node concurrency. This is the approach taken in the MIT M-Machine project, currently in its early phase, which employs a processor coupling mechanism to allow local processors to interact with single-cycle latency.

Acknowledgments

Andrew Chien, Waldemar Horwat, and D. Scott Wills helped develop the MDP architecture. Scott Furman, Shaun Kaneshiro, Ellen Spertus, Brian Totty, and Deborah Wallach were instrumental in developing the J-Machine system software. Salim Ahmed, Paul Carrick, Steve Lear, Ted Nguyen and Mark Vestrich of Intel Corporation provided engineering and management support. Discussions with Chuck Seitz, Steve Ward, Anant Agarwal, and Tom Knight were helpful during the early stages of the J-Machine project. We thank Lisa Sardegna for her help preparing this manuscript. R. Lethin is supported by a fellowship from the John and Fannie Hertz Foundation. The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-91-J-1698, in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation and AT&T Corporation, and in part by assistance from Intel Corporation.

References

[1] Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Technical Report 844, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA 02139, 1985.

[2] Ametek Computer Research Division. *Series 2010 Product Description*, 1987.

[3] Ramune Arlauskas. iPSC/2 System: A Second Generation Hypercube. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 33–36. ACM, 1988.

[4] William C. Athas and Charles L. Seitz. Multicomputers. Technical Report 5244:TR:87, Computer Science Dept., Caltech, Pasadena, CA 91125, 1987.

[5] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *Computer*, 21(8):9 – 24, August 1988.

[6] Iann Barron, Peter Cavill, David May, and Pete Wilson. Transputer Does Five or More MIPS Even When Not Used in Parallel. *Electronics*, pages 109–115, November 1983.

[7] Shekhar Borkar et al. iWARP: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–338. IEEE, November 1988.

[8] W.C. Brantley, K.P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. *IEEE Transactions on Computers*, pages 782–789, 1985.

[9] David E. Culler, Anurag Sah, Klaus Erik Shauser, Thorsten von Eicken, and John Wawrzynek. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

[10] William J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6), June 1990.

[11] William J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 189–205. IEEE, Computer Society Press, June 1987.

[12] William J. Dally and Charles L. Seitz. The Torus Routing Chip. *Distributed Computing*, 1:187–196, 1986.

[13] William J. Dally and Charles L. Seitz. Deadlock Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–53, May 1987.

[14] William J. Dally and Paul Song. Design of a Self-Timed VLSI Multicomputer Communication Controller. In *Proceedings of the International Conference on Computer Design*, pages 230–234. IEEE, Computer Society Press, October 1987.

[15] William J. Dally, D. Scott Wills, and Richard Lethin. Mechanisms for Parallel Computing. In *Proceedings of the NATO Advanced Study Institute on "Parallel Computing on Distributed Memory Mutliprocessors"*. Springer, 1991.

[16] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer – Designing a MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, c-32(2):175–189, February 1983.

[17] W. Daniel Hillis and Guy L. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[18] Waldemar Horwat. Concurrent Smalltalk on the Message-Driven Processor. Master's thesis, MIT, May 1989.

[19] BBN Advanced Computers Incorporated. Butterfly Parallel Processor Overview. BBN Report No. 6148, March. 1986.

[20] C. Lutz et al. Design of the Mosaic Element. In *Proc. MIT Conference on Advanced Research in VLSI*, pages 1–10. Artech Books, 1984.

[21] Rishiyur S. Nikhil. ID Version 88.1 Reference Manual. Technical Report 284, Computation Structure Group, MIT, Cambridge, MA 02138, 1988.

[22] Michael Noakes and William J. Dally. System Design of the J-Machine. In William J. Dally, editor, *Sixth MIT Conference of Advanced Research in VLSI*, pages 179–194, Cambridge, MA 02139, 1990. The MIT Press.

[23] Peter R. Nuth. Router Protocol. MIT Concurrent VLSI Architecture Group Internal Memo 23, 1990.

[24] John F. Palmer. The NCUBE Family of Parallel Supercomputers. In *Proc. IEEE International Conference on Computer Design*, page 107. IEEE, 1986.

[25] Ellen Spertus and William J. Dally. Experiments with Dataflow on a General-Purpose Parallel Computer. In *Proceedings of International Conference on Parallel Processing*, pages II231–II235, Aug 1991.

[26] Arthur H. Veen. Dataflow Machine Architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

[27] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.