The Message-Driven Processor:

A Multicomputer Processing Node with Efficient Mechanisms

The Message-Driven Processor, an integrated multicomputer node, provides efficient mechanisms for parallel computing. Rather than being specialized for a single model of computation, the MDP incorporates primitive mechanisms for communication, synchronization, and naming. These mechanisms efficiently support most proposed parallel programming models. Each processing node of MIT's J-Machine consists of an MDP with 1 Mbyte of DRAM. MDPs have been operational since June 1991, and J-Machines built from them went on line in July 1991.

William J. Dally

J.A. Stuart Fiske

John S. Keen

Richard A. Lethin

Michael D. Noakes

Peter R. Nuth

Massachusetts Institute of Technology

Roy E. Davison

Davison Design and Development Corp.

Gregory A. Fyler

Intel Corporation

The Message-Driven Processor is a 36-bit, 1.1-million transistor, VLSI microcomputer specialized to operate efficiently in a multicomputer. The MDP chip includes a processor, a 4,096-word by 36-bit memory, and a network port. An on-chip memory controller with error checking and correction (ECC) permits local memory to be expanded to one million words by adding external DRAM chips.

The processor is message-driven in the sense that it processes in response to messages, via the dispatch mechanism. No receive instruction is needed. The MDP creates a task to handle each arriving message. Messages carrying these tasks advance, or drive, each computation.

We designed the MDP with two primary goals in mind.

- We wanted 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.
- We wanted to create an inexpensive, VLSI component for cost-efficient parallel computers. Ideal nodes should be inexpensive and plentiful VLSI commodity parts—as inexpensive and plentiful as jellybean candies—that can network together to form a Jellybean Machine (J-Machine) multicomputer.

Efficient parallel mechanisms

Computer hardware provides primitive operations called mechanisms. These mechanisms build the abstractions that in turn make up a programming system.[1] 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 once without interference. The proper set of mechanisms can significantly improve 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 do not adequately 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, designers must implement these functions, inherent to any parallel model of computation, 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,[2] data parallel,[3] dataflow,[4] actors,[5] and explicit message-passing,[6] by providing low-overhead primitive mechanisms for communication, synchronization, and naming. Its communication mechanisms permit a user-level task on one node to send a message to any other node in a 4,096-node machine in less than 2 μs. This process doesn't consume any processing resources on intermediate nodes, and it automatically allocates buffer memory on the receiving node. On message arrival, the receiving node creates and dispatches a task in less than 1 μs.

Presence tags provide synchronization on all storage locations. Three separate register sets allow fast task switching. A translation mechanism maintains bindings between arbitrary names and values, and supports a global virtual address space. We selected these mechanisms to be both general and amenable to efficient hardware implementation. To support fine-grain, concurrent programming systems, we designed the mechanisms to efficiently handle small objects (eight words) and small tasks (20 instructions).

3D array of fine-grain, processing nodes

The MDP is an example of an inexpensive, fine-grain, multicomputer building block. A fine-grain node does not necessarily have a slow processor. We can build a competent processor in a fraction of a modern VLSI chip's area. Fine grain and small memory decrease the chip's cost, resulting in greater arithmetic performance and local memory bandwidth per unit cost. Fast communication and a 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, designers avoid costly features that give a small incremental return in processor performance (such as large caches) in favor of building systems with more nodes, an option not available to the designer of a sequential computer.

The 3D network that connects MDPs gives the highest throughput and lowest latency for a given wire density.[7] This network allows the processing nodes to be packed densely and results in uniformly short wires. It does not waste communication bandwidth by embedding an esoteric topology into physical space. Messages traveling through the network follow a Manhattan shortest path in physical space; they never backtrack. (A Manhattan path travels forward, to the side, and up or down, but not across diagonals.)

Background

The MDP builds on previous work in multicomputer design. Like the Caltech Cosmic Cube,[6] Intel's iPSC,[8] the Ncube,[9] and the Ametek,[10] 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 less than 2 μs on the J-Machine, as opposed to 5 ms on an iPSC-1 or 300 μs on an iPSC-2.

Like the BBN Butterfly[11] and the IBM RP3,[12] the MDP supports a global virtual address space. The same IDs (virtual addresses) reference local (on the same node) and remote (on a different node) objects. Like the Inmos transputer,[13] the Caltech Mosaic,[14] and the Intel iWarp,[15] the MDP is a single-chip processing element integrating a processor, memory, and a communication unit. The MDP is unique because it extends these previous efforts with efficient primitive mechanisms for communication, synchronization, and naming.[1] It uses a direct communication network based on work reported by Dally,[7] Dally and Seitz,[16] and Dally and Song.[17]

System architecture

To the hardware designer, the MDP appears as a component with a memory port, six two-way network ports, and a diagnostic port, as shown in Figure 1.

The memory port provides a direct (that is, no glue) interface to up to 1 Mwords of ECC DRAM, consisting of 11 multiplexed address lines, a 12-bit data bus, and three control signals. Static-column or page mode DRAMs cycle 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 three 1M x 4 memory parts to form a four-chip processing node with

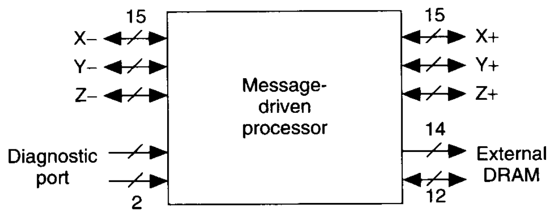


Figure 1. MDP pinout. The MDP has a memory port (26 pins), six network ports (15 pins each), and a diagnostic port (three pins).

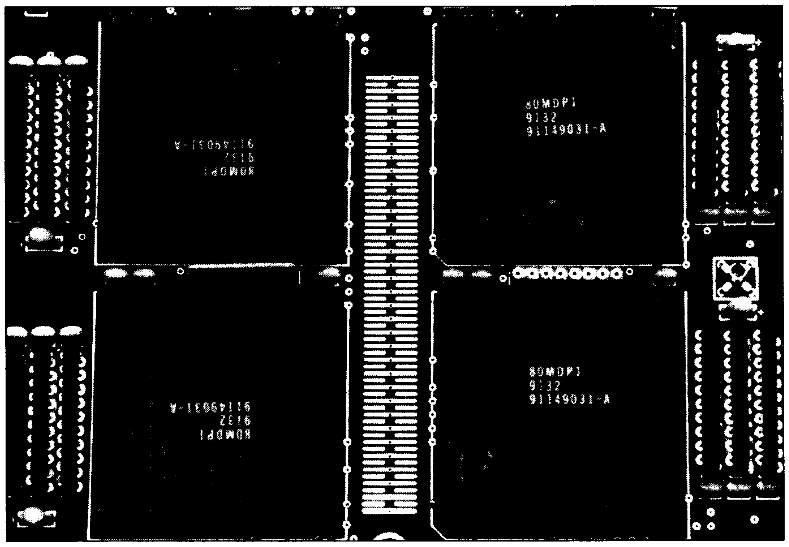


Figure 2. An array of four J-Machine processing nodes. Each node consists of one MDP chip and three 1M × 4 static-column DRAMs. With conventional packaging the node measures 2 in. × 2.75 in.

262,144 words of memory that measures 2 in. × 2.75 in., as shown in Figure 2.

The network ports connect MDPs together in a 3D 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. We give details of the 3D network later in this article.

The diagnostic port issues supervisory commands and reads and writes MDP memory from a console processor. The port consists of two control lines, a serial input line, and a serial output line. 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. To a systems programmer, a bare J-Machine appears as a collection of node memories and register files operable 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[18] 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.

We initialize the value to zero and the count to the number of inputs expected. To sum the values of a number of parallel processes, each node sends a COMBINE message containing the result of its process to a combining node. When the messages arrive, the processor containing the combining node creates a task to execute the COMBINE routine. The routine adds the message value to the node's value and decrements the count. When the count reaches zero, the node sends a COMBINE message to the node's parent.

Communication. 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 carries a message of arbitrary length to any node in the machine. Upon arrival at the receiving node, a hardware queue buffers the message. When the message reaches the head of the queue, the node dispatches a task to handle the message. The combining tree example uses a pair of SEND instructions 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. The MDP synchronizes using message dispatch and presence tags on all states. Because each message arrival dispatches a process, messages can signal events on remote nodes. For example, in the combining tree ex-

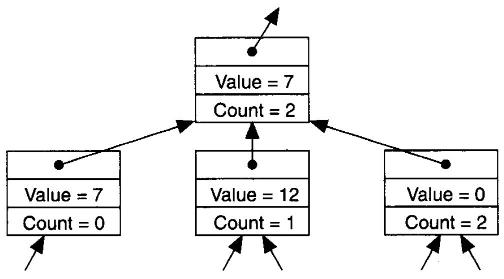


Figure 3. A combining tree sums results produced by a distributed computation. Each node sums the input values as they arrive and then passes a result message to its parent.

ample, each COMBINE message signals its own arrival and initiates the COMBINE routine.

In response to an arriving message, the processor may set presence tags for task synchronization. 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 suspend the reading task until the root of the tree writes the value. Synchronization on data availability in this manner is quite common in many parallel programs.

Naming. The MDP supports naming with segmented memory management and translation instructions. In the combining tree example, the MDP allocates a memory segment to hold the state of each combining node. Using a segment descriptor, it relocates and protects accesses to the node. To make combining nodes relocatable across processing nodes, the MDP translates a node's virtual address to find the processing node where it resides. Upon reaching this node, a second translation locates the segment descriptor for the combining node.

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. Although we describe here the MDP ISA, with particular emphasis on these mechanisms, readers can find more details in Dally et al.[19]

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 no messages are pending. Each arriving message creates a task and initiates execution at P0 or P1, depending on the message's priority. The MDP executes the highest priority task at any point in time. The arrival of a P1 message while the MDP is executing a P0 task causes the MDP to switch execution levels (and thus register sets). When the P1 task completes, the MDP resumes execution at P0 by switching to the P0 register set that holds the register state of the suspended task.

The register set at each priority level includes

- four general-purpose data registers, R0-R3,
- four address registers, A0-A3,
- four ID registers, ID0-ID3, and
- one instruction pointer, IP.

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

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 three words into A2's segment.

ID registers usually 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 states to change by loading a single register. The relatively small size of each register set facilitates quick task switching within an execution level.

Tags. The MDP uses tags 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, and Boolean) and for system data types, such as IP, Addr (a segment descriptor), and Msg (a message header). 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. It raises an exception if the operands are incompatible with the instruction.

Two tags, Fut and Cfut, support intertask synchronization. A Cfut tag initially marks a location empty. 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 checking whether operands are present or of the correct type. For normal cases in which no fault occurs, execution proceeds faster 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 general register operands, and a third operand that may be a register, a memory location, or a constant. Two 17-bit instructions fit into each 36-bit word. Any instruction stream word not tagged as an

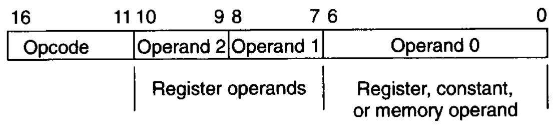


Figure 4. MDP instruction format.

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.

instruction is loaded as a constant into register R0. This provides a very efficient means to load arbitrary 36-bit constants. Figure 5 summarizes the MDP instruction set by category.

Naming. The MDP supports naming via translation instructions and segmented addressing. Addressing memory through segment descriptors permits arbitrary size 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 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. Figure 6 shows a typical message send. 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 four clock cycles (250 ns).

```
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 four-word message uses three variants of the SEND instruction.

The network delivers an injected message to the destination node, as described later. At the destination, a hardware-managed, FIFO queue in the internal RAM of the MDP buffers the message. Separate queues exist for P0 and P1 messages.

Task scheduling. When a message reaches the head of the highest priority nonempty queue, the MDP creates a task to handle it by changing the thread of control and creating a new addressing environment, as shown in Figure 7. Every message header contains a message opcode and the message length. The MDP loads the message opcode into the instruction pointer to start a new thread of control. The length field and the queue head 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 three cycles.

The dispatch mechanism directly processes messages requiring low latency (for example, combining and forwarding). Other messages, such as a remote procedure call, specify a handler that locates the required method (using the translation mechanism described earlier) and then transfers control to the method.

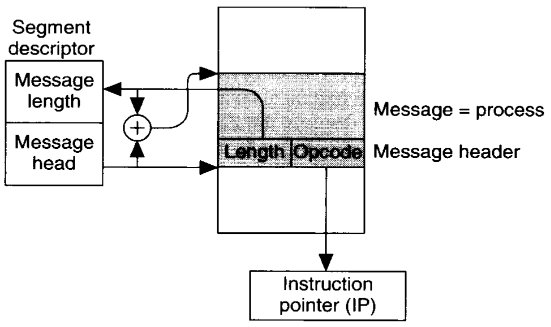


Figure 7. Message dispatch. In three clock cycles, a node creates a new task by setting the instruction pointer to change the thread of control and creating a message segment to provide the initial addressing environment.

```
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.

For example, Figure 8 shows the CALL handler code handling a remote procedure call. Figure 9 depicts the execution of the handler. The first instruction gets the method ID (offset one word 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 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 XLATE instruction translates argument object identifiers to physical memory base/length pairs. If the method needs space to store local state, it may create a context object. When the method finishes executing, 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 shown in Figure 3. Figure 10 displays the code for this routine. 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 executes in 21 cycles. It demonstrates several ways in which the MDP's communication mechanism reduces the overhead of message passing to the point where it can perform simple operations, such as combining. These ways include the following:

- The 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.
- The two SEND instructions transmit the four-word mes-

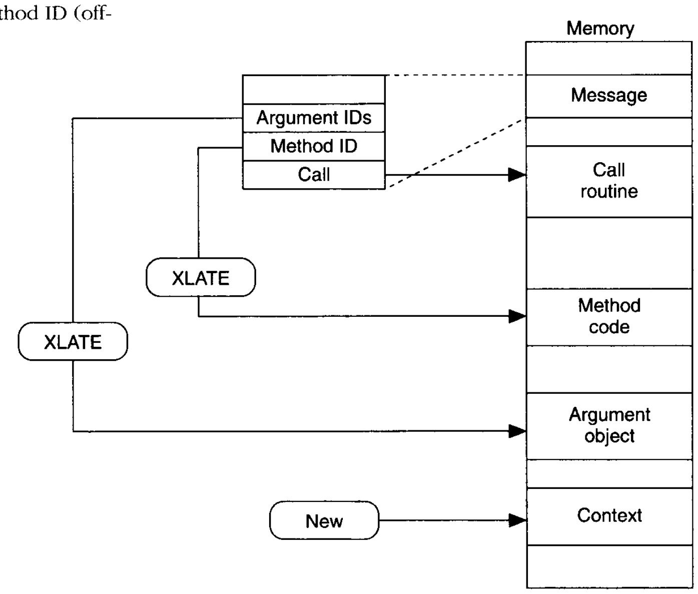


Figure 9. The CALL message invokes a method by translating the method identifier to find the code, creating a context (if necessary) to hold local state, and translating argument identifiers to locate arguments.

```
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.

sage to the parent task. The message transmits directly from register and memory variables with no 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, the processor dispatches a task to handle it two cycles after the execution of the SUSPEND instruction.

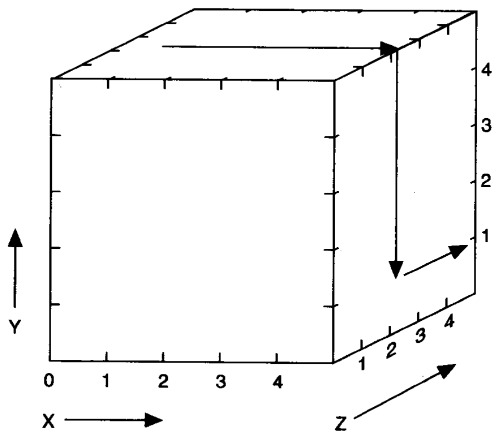


Figure 11. The J-Machine network is a 3D mesh or k-ary 3-cube. The network performs e-cube or destination tag routing. Messages route in each dimension in turn to the proper coordinate in that dimension. In this figure, a message routes from (1,5,2) to (5,1,4), routing first in X, then Y, then Z.

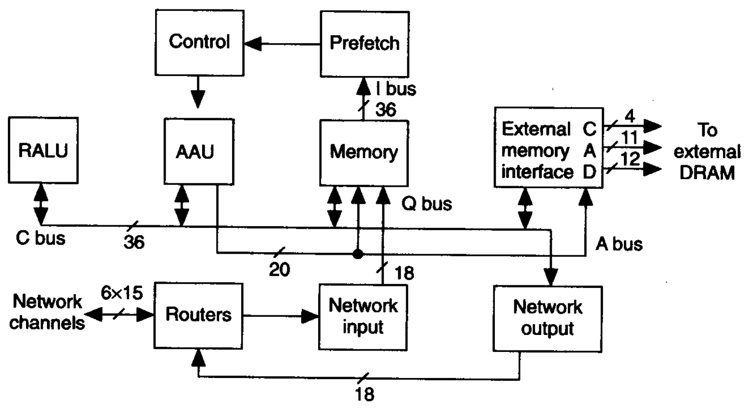


Figure 12. MDP block diagram.

Network architecture

The MDP 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 (less than 2 μs in a 4,096-node network) and high bandwidth (288 Mbits per second per channel). Message delivery occurs entirely within the routers of the machine and consumes no processor or memory resources at intermediate nodes.

Structure. The J-Machine network is a 3D grid, with two-way channels, dimension-order routing, and blocking flow control. (See Figure 11.) Addressing limits the size of the network to 65,536 nodes (32 x 32 x 64). Our initial prototype is a 1,024-node machine (8 x 8 x 16). The faces of the network cube are open for use as I/O ports to the machine. Each channel can sustain a data rate of 288 Mbps. All three dimensions may operate simultaneously for an aggregate data rate of 864 Mbps per node.

Three modules, shown in Figure 12, compose the network logic. The network output 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 network input module reassembles messages at their destination and buffers them into a message queue. We describe more details of implementation in the next section.

Engineering. We chose the 3D mesh topology of the J-Machine network as the most efficient arrangement subject to constraints of wiring density and component pinout.[7] These constraints set the width of the six bidirectional channels per MDP node at 9 data bits plus 6 control bits. We built the J-Machine as a stack of boards with dense board-to-board interconnections to implement the 3D network with short wires.

The MDP breaks with the tradition of asynchronous network routers by implementing a synchronous router.[16,17] This router operates at twice the rate of the processor, sending a pair of 9-bit *phits* between nodes each 62.5-ns processor cycle (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.)

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 reverses direction.[20] Messages route through the network with a latency of one 62.5-ns processor cycle per hop. Thus, message latency T is given by

$$T = T_c(2L + D),$$

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 1,024-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 1.4 μs. The bisection bandwidth (the bandwidth across a plane dividing the machine into two equal halves) of a 1,024-node machine is 18.4 Gbps. The aggregate bandwidth of the network channels is 864 Gbps, and the I/O bandwidth is 184 Gbps.

Routing and flow control. The J-Machine uses deterministic dimension order routing, also called e-cube routing. As shown in Figure 11, all messages route first in the X dimension, then in Y, then in Z. Since messages route in dimension order and messages running in opposite directions along the same dimension do not block, we avoid resource cycles, and leave the network provably deadlock free.[21]

Table 1 lists the format of a message. The first three flits of the message contain the X, Y, and Z addresses. Each node along the path compares the address in the head flit of the message with the node's index in the current dimension. If the two indices match, the node strips the head flit off the message and routes the rest to the next dimension. The MDP's network output node formats the address flits of the message. It also precomputes the direction (positive or negative) 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 (see Figure 13). When a message arrives at a router path 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 (two flits) of the message. When the blockage clears, the message uncompresses and proceeds to its destination, at a rate of one hop per cycle.

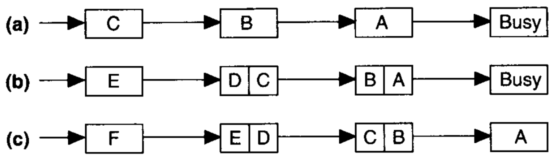


Figure 13. The J-Machine network performs blocking flow control with two stages of queueing per node. Message arrives at busy channel (a). Message becomes compressed by queueing (b). Channel is available; message continues advancing (c).

Table 1. A typical message in the J-Machine.

Flit	Contents	Remark
1	5:+	*X* address
2	1:–	*Y* address
3	4:+	*Z* address
4	MSG: 00	Method to call
5	00440	
6	INT: 00	Argument to method
7	0023	
8	INT: 00	Reply address
9	<1:5:2> T	

The first three flits contain the destination address. The final flit in the message is marked as the tail.

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, the system could not redistribute data that has caused hot spots in the network.

MDP implementation

Figure 12 shows the major subsystems in the MDP. 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 arithmetic unit (AAU) provides addressing functions. The MDP also includes a DRAM interface, control block, and diagnostic interface.

Communication subsystem. The communication subsystem contains the network output, the network input, and the routers. The network output block buffers messages from the registers or memory and injects them into the network. A FIFO buffer matches the speed of message transmission to the network. On each SEND instruction, the MDP transfers one or two words to its FIFO. When the message is complete, or the eight-word buffer is full, the buffer launches the message 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 module transfers messages from the network to the MDP's memory. Data from the network arrive in 18-bit flits, which are composed into a four-word queue row buffer. When the QRB fills, it writes its contents 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

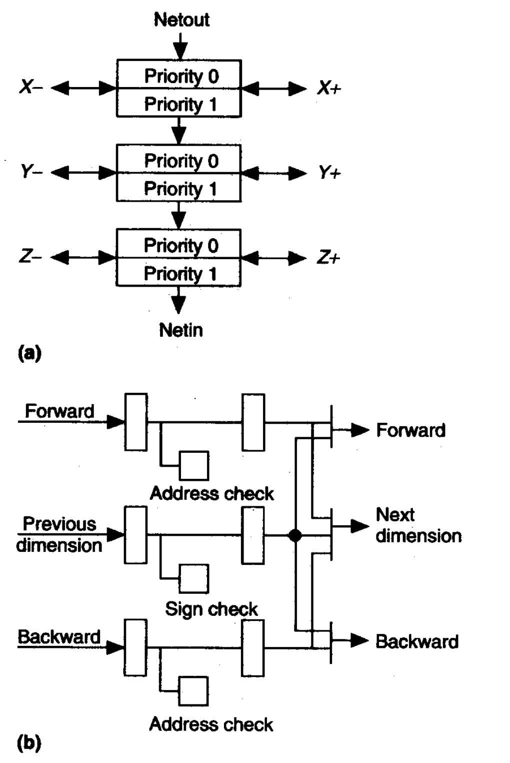


Figure 14. Block diagram of the routers. The two priorities per dimension are completely separate except where they share physical channels (a). Each priority contains forward, reverse, and previous to next dimension datapaths (b).

deliver messages to their destinations. As shown in Figure 14a, the MDP contains three independent routers, one for each bidirectional dimension of the network. Each router contains 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 contains buffers, comparators, and output arbitration (Figure 14b). On each data path, a comparator compares the lead flit, which contains the destination's address in 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 two-to-one switch. Once a message is granted this switch, any 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 AAU, the largest logic block in the MDP, performs all functions associated with memory addressing. To support naming and relocation, the AAU contains the address and ID registers. It protects 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 relocates) or to access beyond the end of an object raises an exception. A translation base/mask register defines an area of memory to be a two-way, set-associative translation buffer used by the XLATE, PROBE, 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 a queue head/length (QHL) register. (See Figure 15.) 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 the MDP dispatches a task to handle a message, it loads the A3 register with a segment descriptor for the message. The processor 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.

Layout. Figure 16 shows a floor plan of the chip with a die photograph for comparison. Table 2 breaks down the area usage.

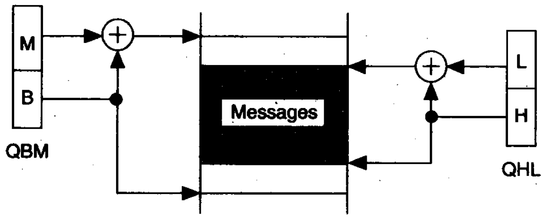
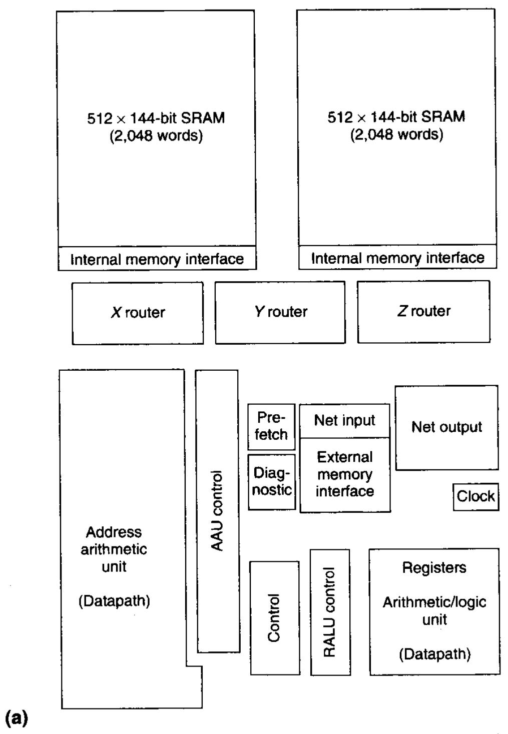


Figure 15. The AAU maintains the queue base/mask (QBM) registers, which specify the location of the message queues in main memory, and the queue head/length (QHL) registers, which specify the beginning and end of the messages received in each queue. Figure shows only one queue.


512 x 144-bit SRAM (2,048 words)
512 x 144-bit SRAM (2,048 words)
Internal memory interface
Internal memory interface
X router
Y router
Z router
Address arithmetic unit (Datapath)
AAU control
Pre-fetch
Diag-nostic
Net input
External memory interface
Net output
Clock
Control
RALU control
Registers
Arithmetic/logic unit (Datapath)
(a)

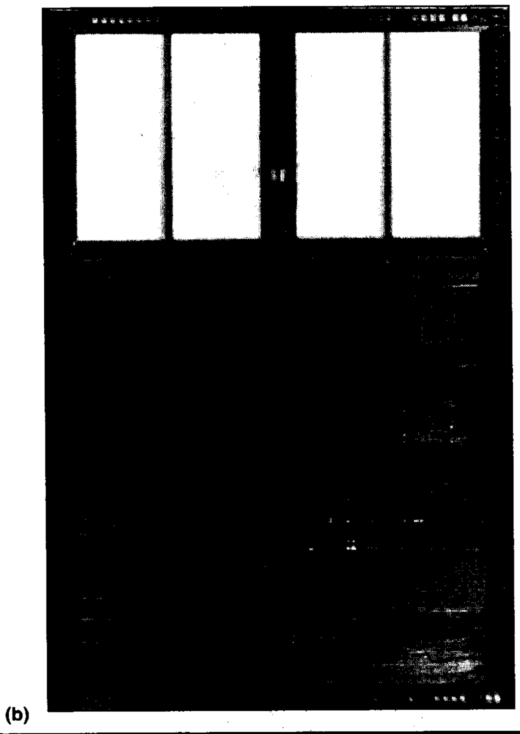


Figure 16. MDP chip floor plan (a) and die photograph (b).

Methodology. We implemented the MDP using Intel standard cells except for the on-chip RAM, clock generator, and pads. Using standard cells sacrificed a factor of three to four in area and two to three 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.

The 700 or so sheets of schematics drafted at MIT used 35,000 standard cells containing 210,000 transistors. (The remaining 890,000 devices are contained in the full custom portions of the chip, mostly in the RAM.) We sent these schematics to Intel for layout. Designers laid out many of the data paths by hand to exploit the regularity of the design. Automatic place and route CAD tools laid out the less regular collections of logic.

We began architecture studies leading to the MDP in October 1986. Work on the RTL model of the microarchitecture began in June 1988, and schematic entry at MIT started that November. The task of translating schematics into layout commenced in June 1989, and we finished the layout in December 1990. We received first silicon in June 1991 and were running programs on it within a few hours.

Table 2. Chip area breakdown.

Module	Dimensions (mm)	Area (mm^2)	Transistors ($\times 10^3$)
AAU	3.7 x 7.0	25.9	75.0
RALU	3.7 x 2.9	10.7	39.0
Diagnostic	0.9 x 1.1	1.0	3.7
Prefetch	0.9 x 1.1	1.0	3.2
Control	1.1 x 2.6	2.9	8.7
Internal memory interface	7.8 x 0.5	3.9	13.0
External memory interface	1.6 x 1.8	2.9	9.0
Net input	1.8 x 0.7	1.3	4.4
Net output	2.1 x 1.8	3.8	18.0
Routers	8.4 x 1.3	10.9	29.0
RAM	8.8 x 4.9	43.1	880.0
Clock	0.7 x 0.8	0.6	0.1
Pads	50.5 x 0.2	8.4	2.6
Full chip	10.2 x 15.0	153.0*	1,087.0

* Includes wiring between modules.

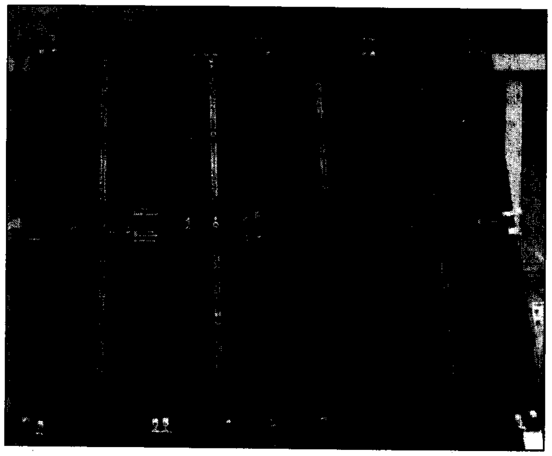


Figure 17. Photograph of 64-node J-Machine system.

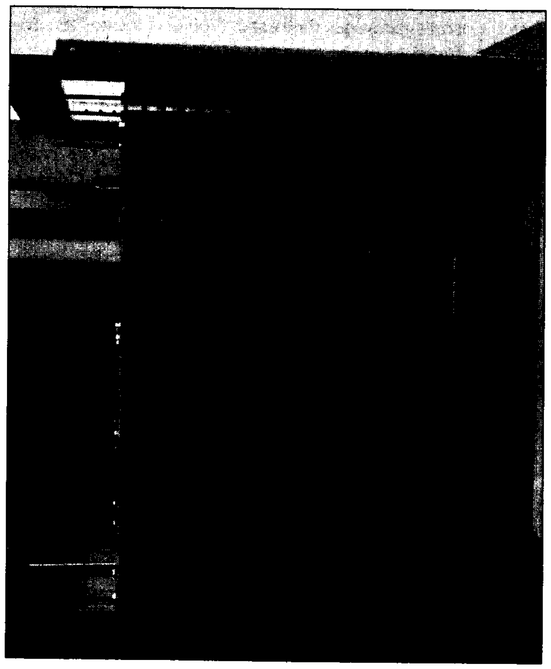


Figure 18. A 1,024-node J-Machine chassis.

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

System design. Figure 17 shows a photograph of a 64-node J-Machine processor board measuring 20.5 in. x 24 in. Each node consists of an MDP chip (in a 168-pin grid array package) and three 4-Mbit 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 in a stack. A total of 32 elastomeric connectors held in four connector holders provide 2,240 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. Bus bars supply power and ground directly to each board. The center area of the board contains the final stage of the clock distribution network, along with diagnostic fan-out, multiplexing logic, and temperature and airflow monitors.

Figure 18 shows a photograph of our chassis for a 1,024-node system. The chassis contain a stack of 16 processor boards, power supplies, and distribution bus bars. Twenty tie rods bind the boards and compress the elastomer connectors. A 4,096-node system can be built by combining four chassis. Each stack connects to its neighboring stacks by 128 (16 x 8) short, 60-pin, ribbon cables—one for each pair of nodes on the periphery. Each vertical pair of stacks shares a 3,000 cu ft/min. blower for cooling.

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

Software

We intended the J-Machine as a platform for software experiments in fine-grain, parallel programming. To this end, we have implemented and are studying software systems for different fine-grain programming models. Fine-grain programs typically execute from 10 to 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.

```
(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 19. Concurrent Smalltalk source to compute Size-Of-Tree. Method 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.

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 extracting parallelism from existing programs (an active and interesting area for many parallel programming researchers) or on adapting sequential languages for the parallel domain, we focus on languages where the expression of fine-grain parallelism is much cleaner. To date, we have implemented two languages on the J-Machine: the actor language Concurrent Smalltalk and the dataflow language Id.

Concurrent Smalltalk. CST[23] is a parallel, object-oriented, programming language (based on the Actor model[5]) with asynchronous message send and distributed objects. Its syntax is similar to that of Lisp or Scheme. It performs method or function invocation 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, called Optimist, and assisted at runtime by a small kernel called Cosmos. Cosmos provides a global virtual name space, 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 efficiently supports fine-grain concurrent computation in which tasks are very short (40 user instructions) and data objects are very small (eight 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.

Figure 19 shows a small sample program defining the Size-Of-Tree method for three object types: a Pair, the Null object, and a generic object. When called on a Lisp-style tree, these methods return the number of generic objects stored in the tree. For example, when called on the tree '((1 2 3)(4 5 6)(7 8 9)), Size-Of-Tree returns the value 9. Note that since Pair and Null are subclasses of Object, their more specific methods are selected when Size-Of-Tree is invoked on their types.

When Optimist, the CST compiler, compiles this example program, it defines a selector object and three function objects. The selector object (shown in Figure 20) lists the type and function correspondence. When a method applies to an object, Cosmos examines the object type and locates the appropriate function in the selector object. The MDP then invokes this function on the object. (In cases where the compiler can infer the type of the object or when the type of objects is explicitly declared, the compiler optimizes a method invocation directly to the correct function invocation.) The compiler marks the selector object as copyable, and Cosmos maintains it like any other object.

```
MODULE    OBJ:Selector.Size_Of_Tree

DC        Copyable|class_Selector       ; Identify properties of
                                        ;  Size_Of_Tree selector
DC        OBJ:Selector.Size_Of_Tree     ; Store own ID inside selector

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
```

Figure 20. Selector object generated by the example program.

Figure 21 shows the compiled code for the function for the class Pair. When a method applies to a particular object, Cosmos examines the object class and the selector object, and chooses 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 where Left exists. The function sends a message to Left that recursively applies the Size-Of-Tree method. It marks the slot that will hold the return value with a Cfut tag. Next, it applies Size-Of-Tree 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. In this case, the ADD instruction will fault trying to add Cfuts, and the MDP will suspend the process, saving its registers into the context.

```
MODULE     OBJ:function.Size_Of_Tree_2

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

  ;; 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 address
  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 for
  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 21. Compiled code for the Size-Of-Tree function for objects of class Pair.

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

Let us consider 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. Cosmos will find the object and move or copy it to the local node.

- Cosmos maintains functions and selectors like any other immutable object. If they are not present, Cosmos will copy them to the node, a process analogous to a distributed instruction cache.
- If the function were preempted and the object moved or migrated away, Cosmos would invalidate the address registers. Accesses to the object would cause a fault that would attempt to retranslate or reobtain the object.
- The A1 register points to the current context. The context contains storage to hold working variables or, if the context faults, to hold spilled register values. In the example, the futures are constructed in the context, and thus are named *context-future* (Cfut).

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

- 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 trade-off at compile and runtime still need to be developed and evaluated.
- How should we place objects in the machine, and how should they migrate in order to reduce the overhead of communication?
- In some cases, the amount of parallelism grows much larger than the machine can handle. We need to study how we can effectively and automatically throttle the parallelism created by the machine when it becomes saturated.

Horwat discusses these issues, and others related to the efficiency of programming fine-grain, parallel processors in more detail.[23]

Dataflow implementation. Id is a functional programming language originally designed for dataflow architectures.[24] The Id compiler converts an Id program into a dataflow graph, in which nodes represent operators and arcs represent dependencies. Originally, researchers executed these dataflow graphs directly on specialized dataflow machines. More recently, they have begun compiling dataflow graphs to run on general-purpose parallel machines.[25] Dataflow programs suit large parallel computers, because the abundance of fine-grain 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-grain mechanisms make it an excellent target for dataflow programs.

We experimented with several methods of executing dataflow programs on the J-Machine.[26] 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. To do so, it stores the first data value, matches it with the second value when it arrives, performs the dataflow operation, and sends the resulting value to two destinations. This process uses the Cfut tag and fault handler.

A more efficient approach increases the granularity of each task to reduce scheduling overhead. We are building a system on top of the Berkeley TAM project[25] that addresses the inefficiencies of our earlier systems.

WE BUILT THE MDP TO DEMONSTRATE THE UTILITY of general-purpose communication, synchronization, and naming mechanisms in a multicomputer building block. Its mechanisms efficiently support dataflow[26] and object-oriented programming[23] models using a global name space. 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. Its communication and synchronization performance competes with processing nodes specialized to a single model of computation, such as iWarp[15] (systolic) or the transputer[13] (communicating sequential processes).

Computers built from fine-grain processing nodes, such as the MDP, consisting of a small but powerful processor and a small memory, are more cost-effective than those built from fewer coarse-grain nodes. Fine-grain nodes devote a larger fraction of their silicon area to processing and have higher arithmetic, memory, and communication bandwidth per unit cost. Large-scale parallel machines built from fine-grain 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 65,536 nodes. A 128-node Jellybean Machine is currently operational and resources are in place to build several more machines, including a 1,024-node system at MIT and machines at a number of other research institutions.

The MDP project 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 completed the design of the chip, its system-level hardware, and several programming systems with a handful (less than eight) of graduate students and engineers in two and a half 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. We find the design of mechanisms for parallel computers particularly challenging because no well-established parallel benchmarks exist. Additionally, 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 and better mapping mechanisms would be useful. MDP's conservative implementation leaves opportunities for streamlining, by 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, we can put many powerful processing units on one chip. An interesting direction for further research is the extension of the MDP mechanisms to control intranode as well as internode concurrency. The MIT M-Machine project, now in its early phase, takes this approach. It employs a processor-coupling mechanism to allow local processors to interact with single-cycle latency.

Acknowledgments

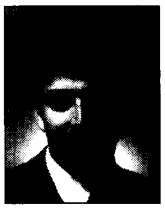
The MDP was built as part of a collaborative project between MIT and Intel Corporation. The Defense Advanced Research Projects Agency partially funded the project, under contracts N00014-88K-0738 and N00014-91-J-1698. Funding also came from a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation, and AT&T Corporation. Intel Corporation provided technical support and personnel.

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.

References

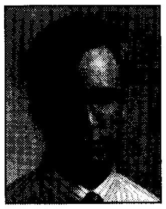
1. W.J. Dally, D.S. Wills, and R. Lethin, "Mechanisms for Parallel Computing," *Proc. NATO Advanced Study Institute on Parallel Computing on Distributed Memory Multiprocessors*, Springer-Verlag, New York, 1991.
2. A. Gottlieb et al., "The NYU Ultracomputer – Designing a MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers*, Vol. C-32, No. 2, Feb. 1983, pp. 175-189.
3. W.D. Hillis and G.L. Steele, "Data Parallel Algorithms," *Comm. of the ACM*, Vol. 29, No. 12, 1986, pp. 1170-1183.
4. A.H. Veen, "Dataflow Machine Architecture," *ACM Computing Surveys*, Vol. 18, No. 4, Dec. 1986, pp. 365-396.
5. G. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems," Tech. Report 844, Artificial Intelligence Laboratory, Massachusetts Inst. of Technology, Cambridge, Mass., 1985.
6. W.C. Athas and C.L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *Computer*, Vol. 21, No. 8, Aug. 1988, pp. 9-24.
7. W.J. Dally, "Performance Analysis of *k*-ary *n*-cube Interconnection Networks," *IEEE Trans. Computers*, Vol. 39, No. 6, June 1990, pp. 775-785.
8. R. Arlauskas, "iPSC/2 System: A Second-Generation Hypercube," *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, Assn. for Computing Machinery, New York, 1988, pp. 33-36.
9. J.F. Palmer, "The Ncube Family of Parallel Supercomputers," *Proc. IEEE Int'l Conf. Computer Design*, IEEE CS Press, Los Alamitos, Calif., 1986, p. 107.
10. *Series 2010 Product Description*, Ametek Computer Research Division, Monrovia, Calif., 1987.
11. "Butterfly Parallel Processor Overview," BBN Report 6148, Bolt, Beranek and Newman Advanced Computers, Inc., Cambridge, Mass., Mar. 1986.
12. W.C. Brantley, K.P. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *IEEE Trans. Computers*, Vol. C-34, No. 10, Sept. 1985, pp. 782-789.
13. I. Barron et al., "Transputer Does Five or More MIPS Even When Not Used in Parallel," *Electronics*, Nov. 1983, pp. 109-115.
14. C. Lutz et al., "Design of the Mosaic Element," *Proc. MIT Conf. Advanced Research in VLSI*, Artech Books, Dedham, Mass., 1984, pp. 1-10.
15. S. Borkar et al., "iWarp: An Integrated Solution to High-Speed Parallel Computing," *Proc. Supercomputing Conf.*, IEEE CS Press, Nov. 1988, pp. 330-338.
16. W.J. Dally and C.L. Seitz, "The Torus Routing Chip," *Distributed Computing*, Vol. 1, 1986, pp. 187-196.
17. W.J. Dally and P. Song. "Design of a Self-Timed VLSI Multicomputer Communication Controller," *Proc. Int'l Conf. Computer Design*, pp. 230-234. IEEE CS Press, Oct. 1987.
18. P.C. Yew, N.F. Tzeng, and D.H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Trans.*

Computers, Vol. C-36, No. 4, Apr. 1987, pp. 388-395.
19. W.J. Dally et al., "Architecture of a Message-Driven Processor," *Proc. 14th Int'l Symp. Computer Architecture*, IEEE CS Press, June 1987, pp. 189-205.
20. P.R. Nuth, "Router Protocol," internal memo 23, MIT Concurrent VLSI Architecture Group, 1990.
21. W.J. Dally and C.L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Computers*, Vol. C-36, No. 5, May 1987, pp. 547-553.
22. M. Noakes and W.J. Dally, "System Design of the J-Machine," *Proc. Sixth MIT Conf. Advanced Research in VLSI*, W.J. Dally, ed., MIT Press, Cambridge, Mass, 1990, pp. 179-194.
23. W. Horwat, "Concurrent Smalltalk on the Message-Driven Processor," master's thesis, MIT, May 1989.
24. R.S. Nikhil, *ID Version 88.1 Reference Manual*, Tech. Report 284, Computation Structure Group, MIT, 1988.
25. D.E. Culler et al., "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 164-175.
26. E. Spertus and W.J. Dally, "Experiments with Dataflow on a General-Purpose Parallel Computer," *Proc. Int'l Conf. Parallel Processing*, Aug. 1991, pp. II-231–II-235.

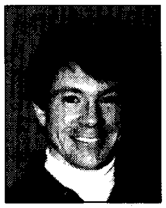


William J. Dally is an associate professor of computer science at the Massachusetts Institute of Technology, where he directs the group building the J-Machine. His research interests include concurrent computing, computer architecture, computer-aided design, and VLSI design.

Dally received a BS degree from Virginia Polytechnic Institute and an MS from Stanford University, both in electrical engineering. He earned his PhD in computer science from the California Institute of Technology.



J.A. Stuart Fiske is a PhD student at MIT. His main research interests include computer architecture, parallel processing, and VLSI design. He received the BE degree in electrical engineering from McGill University, and the MS in electrical engineering and computer science from MIT. Fiske is a member of the IEEE.



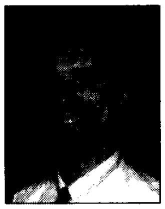
John S. Keen is a PhD student at the MIT AI Lab. He has worked at IBM Canada, Bell Canada, and Intel. His primary research interests include computer architecture, parallel processing, CAD tools, and concurrent database systems.

Keen graduated from the University of Waterloo with a BASc degree and an MASc degree, both in electrical engineering. He received a National Science and Engineering Research Council of Canada 1967 Scholarship and is a member of the IEEE and Sigma Xi.

Richard A. Lethin is also pursuing his PhD at MIT's AI Lab. Previously, he designed floating-point processor boards for the Trace-200 and Trace-300 VLIW (very long instruction word) mini-supercomputers at Multiflow Computer. His research interests include computer architecture and artificial intelligence.

Lethin received a BS degree in electrical engineering from Yale College and an MS in electrical engineering and computer science from MIT. He is a Hertz Foundation fellow.

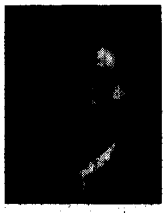
Michael D. Noakes is a member of the research staff of MIT's AI Lab, where he researches parallel computer architecture. Previously he helped develop the Concert multiprocessor at Harris Corporation. His research interests include computer architecture and parallel processing. Noakes earned a BS degree in electrical engineering from MIT.



Peter R. Nuth is a PhD student in the MIT AI Lab. He helped design MIT's Concert and J-Machine parallel computers. His primary research interests include computer architecture, communication, and parallel processing.

Nuth earned BS, MS, and Engineer's degrees in electrical engineering and computer science from MIT. He belongs to the IEEE, Sigma Xi, and Eta Kappa Nu.

Roy E. Davison, a 25-year veteran of the IC industry, heads Davison Design and Development in Ben Lomond, California. He has designed chips for several manufacturers, including Texas Instruments, Advanced Micro Devices, National Semiconductor, Intel, and Mas Par. His primary research interest is the design of microprocessors and peripherals. He attended Sam Houston State University.



Gregory A. Fyler is a project manager in Intel's Architecture Development Lab in Santa Clara, California. He has managed chip design for several microcontroller and microcomputer products, most recently the MDP project. Prior to joining Intel 15 years ago, he developed calculator chips for Fairchild Camera and Instrument.

Fyler earned a BS degree in electrical engineering and computer science from the University of California, Berkeley. He is a member of the IEEE.

Direct questions regarding this article to William J. Dally, MIT Artificial Intelligence Lab, 545 Technology Square, NE43-620, Cambridge, MA 02139; or via e-mail at billd@ai.mit.edu

Reader Interest Survey
Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 156	Medium 157	High 158