Architecture of a Message-Driven Processor [1]

William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat,
Jon Kaplan, Paul Song, Brian Totty, and Scott Wills

Artificial Intelligence Laboratory and Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

We propose a machine architecture for a high-performance processing node for a message-passing, MIMD concurrent computer. The principal mechanisms for attaining this goal are the direct execution and buffering of messages and a memory-based architecture that permits very fast context switches. Our architecture also includes a novel memory organization that permits both indexed and associative accesses and that incorporates an instruction buffer and message queue. Simulation results suggest that this architecture reduces message reception overhead by more than an order of magnitude.

1 Introduction

1.1 Summary

The message-driven processor (MDP) is a processing node for a message-passing concurrent computer. It is designed to support fine-grain concurrent programs by reducing the overhead and latency associated with receiving a message, by reducing the time necessary to perform a context switch, and by providing hardware support for object-oriented concurrent programming systems.

Message handling overhead is reduced by directly executing messages rather than interpreting them with sequences of instructions. As shown in Figure 1, the MDP contains two control units, the instruction unit (IU) that executes instructions and the message unit (MU) that executes messages. When a message arrives it is examined by the MU which decides whether to queue the message or to execute the message by preempting the IU. Messages are enqueued without interrupting the IU. Message execution is accomplished by immediately vectoring the IU to the appropriate memory address. Special registers are dedicated to the MU so no time is wasted saving or restoring state when switching between message and instruction execution.

Context switch time is reduced by making the MDP a memory rather than register based processor. Each MDP instruction may read or write one word of memory. Because the MDP memory is on-chip, these memory references do not slow down instruction execution. Four general purpose registers are provided to allow

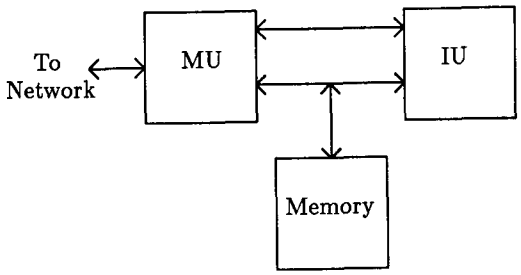

Figure 1: Message Driven Processor Organization

instructions that require up to three operands to execute in a single cycle. The entire state of a context may be saved or restored in less than 10 clock cycles. Two register sets are provided, one for each of two priority levels, to allow low priority messages to be preempted without saving state.

The MDP memory can be accessed either by address or by content, as a set-associative cache. Cache access is used to provide address translation from object identifier to object location. This translation mechanism is used to support a global address space. Object identifiers in the MDP are global. They are translated at run time to find the node on which the object resides and the address within this node at which the object starts.

The associative access of the MDP memory is also used to look up the method to be executed in response to a message. The cache acts as an ITLB [3] and translates a selector (from the message), and class (from the receiver) into the starting address of the method. Because the MDP maintains a global name space, it is not necessary to keep a copy of the program code (and the operating system code) at each node. Each MDP keeps a method cache in its memory and fetches methods from a single distributed copy of the program on cache misses.

[1]The research described in this paper was sponsored by the Defense Advanced Research Projects Agency in part under contract number N00014-80-C-0622 and in part under contract number N00014-85-K0124.

The MDP is a tagged machine. Tags are used both to support dynamically-typed programming languages and to support concurrent programming constructs such as futures [8].

The MDP is intended to support a fine-grain, object-oriented concurrent programming system in which a collection of objects interact by passing messages [1]. In such a system, addresses are object names (identifiers). Execution is invoked by sending a message specifying a method to be performed, and possibly some arguments to an object. When an object receives a message it looks up and executes the corresponding method. Method execution may involve modifying the object's state, sending messages, and creating new objects. Because the messages are short (typically 6 words), and the methods are short (typically 20 instructions) it is critical that the overhead involved in receiving a message and in switching tasks to execute the method be kept to a minimum.

1.2 Background

Several message-passing concurrent computers have been built using conventional microprocessors for processing elements. Examples of this class of machines include the Cosmic Cube [13], the Intel iPSC [7], and the S-NET [2]. The software overhead of message interpretation on these machines is about 300μs. The message is copied into memory by a DMA controller or communication processor. The node's microprocessor then takes an interrupt, saves its current state, fetches the message from memory, and interprets the message by executing a sequence of instructions. Finally, the message is either buffered or the method specified by the message is executed.

This large overhead restricts programmers to using coarse-grained concurrency. The code executed in response to each message must run for at least a millisecond to achieve reasonable (75%) efficiency. Much of the potential concurrency in an application cannot be exploited at this coarse grain size. For many applications the natural grain-size is about 20 instruction times [4] (5μs on a high-performance microprocessor). Two-hundred times as many processing elements could be applied to a problem if we could efficiently run programs with a granularity of 5μs rather than 1 ms.

For many of the early message-passing machines, the network latency was several milliseconds, making the software overhead a minor concern. However, recent developments in communication networks for these machines [5] [6] have reduced network latency to a few microseconds making software overhead a major concern.

The MDP is not the first processing element designed explicitly for a message-passing concurrent computer. The N-CUBE family of parallel processors is built around a single chip processing element that is used in conjunction with external memory [11]. The Mosaic processor integrates the processor, memory, and communication unit all on one chip [10]. Neither of these processors addresses the issue of message reception overhead. The N-CUBE processor uses DMA and interrupts to handle its messages, while the Mosaic receives messages one word at a time using programmed transfers out of receive registers. Closer in spirit to the MDP is the The InMOS Transputer [9]. The Transputer supports a static, synchronous model of programming based on CSP [12] in much the same way that the MDP supports a dynamic asynchronous model based on actors [1].

Some of the ideas used in the MDP have been borrowed from other processors. Multiple register sets have been used in microprocessors such as the Zilog Z-80 [16], and in microcoded processors such as the XEROX Alto [15]. The Alto uses its multiple register sets to perform micro-tasking. By switching between the register sets, context switches can be made on microinstruction boundaries with no state saving required. Spector [14] used micro-tasking on the Alto to implement remote operations over an Ethernet, an idea similar to direct method execution.

1.3 Outline

The remainder of this paper describes the MDP in detail. The user architecture of the MDP is presented in Section 2. The machine state, message set, and instruction set are discussed. The MDP micro architecture is the topic of Section 3. This section includes a description of our novel memory architecture. Section 4 discusses support for concurrent execution models. We show how a programming system that combines reactive objects, dynamic typing, fetch-and-op combining, and futures can be efficiently implemented on the MDP. Performance estimates for the MDP are discussed in Section 5.

2 User Architecture

2.1 Machine State

The programmer sees the MDP as a 4K-word by 36-bit/word array of read-write memory (RWM), a small read-only memory (ROM), and a collection of registers.

The MDP registers are shown in Figure 2. The registers are divided into *instruction registers* and *message registers*. There are two sets of instruction registers, one for each of two priority levels. Each set consists of four general registers `R0-R3`, four address registers `A0-A3`, and an instruction pointer `IP`. The general registers are 36 bits long (32 data bits + 4 tag bits) and are used to hold operands and results of arithmetic operations.

The 28-bit address registers are divided into 14-bit base and limit fields that point to the base and limit addresses of an object in the node's local memory. Associated with each address register is an invalid bit, and a queue bit. The invalid bit is set when the register does not contain a valid address. The queue bit is set when the register is used to reference the current message queue. Address registers are not saved on a context switch since the object they point to may be relocated. Instead, the object's identifier (OID) is re-translated into the object's base and limit addresses when the context is restored. All address registers as well as the queue and translation buffer registers, appear to the programmer to have two adjacent 14-bit fields.

The instruction pointer is a 16-bit register that is used to fetch instructions. The low order 14-bits select a word of memory, bit 14 selects one of the two instructions packed in the word, and bit 15 determines whether the IP is an absolute address, or an offset into A0. Because instructions are prefetched, the value of the IP may be ahead of the next instruction.

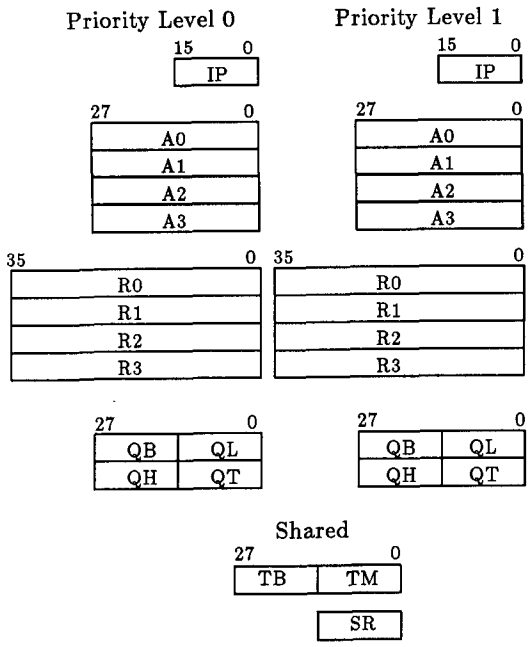

Figure 2: MDP Registers

The small register set allows a context switch to be performed very quickly. Only five registers must be saved and nine registers restored. Because the on-chip memory can be accessed in a single clock cycle, the fact that few intermediate results can be kept in registers does not significantly degrade performance.

The message registers consist of two sets of queue registers, a translation buffer base/mask register, and a status register. A set of queue registers is provided for each of the two receive queues. Each queue register set contains a 28-bit base/limit register, and a 28-bit head/tail register. The queue base/limit register contains 14-bit pointers to the first and last words allocated to the queue while the head/tail register contains 14-bit pointers to the first and last words that hold valid data. As with the address registers all these 14-bit fields contain physical addresses into local memory. Special address hardware is provided to enqueue or dequeue a word in a single clock cycle.

We have omitted a send queue from the MDP for two reasons. First, analysis of the networks we plan to use [6] indicate that the network will be able to accept messages as fast as the nodes can generate them. Second, if network congestion does occur, the absence of a send queue allows the congestion to act as a governor on objects producing messages. With a send queue, these objects would fill their respective queues before they blocked. Because both the MDP and the network support multiple priority levels, higher priority objects will be able to execute and clear the congestion.

The translation buffer base/mask register is used to generate addresses when using the MDP memory as a set-associative cache. This register contains a 14-bit base and a 14-bit mask. As shown in Figure 3, each bit of the the mask, $MASK_i$, selects between a bit of the association key, KEY_i, and a bit of the base, $BASE_i$, to

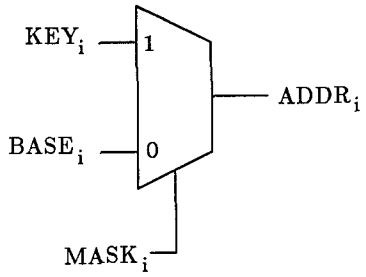

Figure 3: Translation Buffer Address Formation

generate the corresponding address bit, $ADDR_i$. The high order ten bits of the resulting address are used to select the memory row in which the key might be found. The operation of the memory as a set-associative cache is described in Section 3.2.

The status register contains a set of bits that reflect the current execution state of the MDP including: current priority level, a fault status bit, and an interrupt enable bit.

2.2 Message Set

The MDP controller is driven by the incoming message stream. The arrival of a message causes some action to be performed by the MDP. This action may be to read or write a memory location, execute a sequence of instructions, and/or send additional messages. The MDP controller reacts to the arrival of a message by scheduling the execution of a code sequence.

Rather than providing a large message set hard-wired into the MDP, we chose to implement only a single primitive message, EXECUTE. This message takes as arguments a priority level <priority> (0 or 1), an opcode <opcode>, and an optional list of arguments, <arg>. The message opcode is a physical address to the routine that implements the message. More complex messages, such as those that invoke a method or dereference an identifier, can be implemented as almost as efficiently using the EXECUTE message as they could if they were hard-wired.

```
EXECUTE <priority> <opcode> <arg> .. <arg>
```

When a message arrives at a message-driven processor, it is buffered until the node is either idle or executing code at lower priority level. If the node is already executing at a lower priority, no buffering is required. This buffering takes place without interrupting the processor, by stealing memory cycles. The processor then examines the header of the message and dispatches control to an instruction sequence beginning at the <opcode> field of the message in physical memory. Saving state is not required as the new message is executed in the high priority registers. Message arguments are read under program control. The processor's control unit rather than software, decides (1) whether to buffer or execute the message and (2) what address to branch to when the message is accepted.

In the MDP, all messages *do* result in the execution of instructions. The key difference is that no instructions are required to receive or buffer the message, and very few instructions are required to

locate the code to be executed in response to the message. The MDP provides efficient mechanisms to buffer messages in memory, to synchronize program execution with message arrival, and to transfer control rapidly in response to a message. By performing these functions in hardware (not microcode), their overhead is reduced to a few clock cycles (<500ns).

We choose not to implement complex messages in microcode because they will run just as fast using macrocode and implementing them in macrocode gives us more flexibility. Since the MDP is an experimental machine we place a high value on providing the flexibility to experiment with different concurrent programming models and different message sets, and to instrument the system. The MDP uses a small ROM to hold the code required to execute the message types listed below. The ROM code uses the macro instruction set and lies in the same address space as the RWM, so it is very easy for the user to redefine these messages simply by specifying a different start address in the header of the message.

```
READ        <base> <limit> <reply-node> <reply-sel>
WRITE       <base> <limit> <data> ... <data>
READ-FIELD  <obj-id> <index> <reply-id> <reply-sel>
WRITE-FIELD <obj-id> <index> <data>
DEREFERENCE <oid> <reply-id> <reply-sel>
NEW         <size> <data> ... <data> <reply-id> <reply-sel>
CALL        <method-id> <arg> ... <arg>
SEND        <receiver-id> <selector> <arg> ... <arg>
REPLY       <context-id> <index> <data>
FORWARD     <control> <data> ... <data>
COMBINE     <obj-id> <arg> ... <arg> <reply-id> <reply-sel>
GC          <obj-id> <mark>
```

The READ, WRITE, READ-FIELD, WRITE-FIELD, DEREFERENCE, and NEW messages are used to read or write memory locations. READ WRITE read and write blocks of physical memory. They deal only with physical memory addresses, <base> <limit>, and physical node addresses, <reply-node>. The READ-FIELD and WRITE-FIELD read and write a field of a named object. These messages use logical addresses (object identifiers), <obj-id>, <reply-id>, and will work even if their target is relocated to another memory address, or another node. The DEREFERENCE method reads the entire contents of an object. NEW creates a new object with the specified contents (optional) and returns an identifier. The <reply-sel> (reply-selector) field of the read messages specifies the selector to be used in the reply message.

The CALL and SEND messages cause a method to be executed. The method is specified directly in the CALL message, <method-id>. In the SEND message, the method is determined at run-time depending on the class of the receiver.

The REPLY, FORWARD, COMBINE, and GC messages are used to implement *futures*, message multicast, fetch-and-op combining, and garbage collection respectively.

2.3 Instruction Set

Each MDP instruction is 17-bits in length. Two instructions are packed into each MDP word (the INST tag is abbreviated). Each instruction may specify at most one memory access. Registers or constants supply all other operands.

As shown in Figure 4, each instruction contains a 6-bit opcode field, two 2-bit register select fields, and an 7-bit operand descriptor field. The operand descriptor can be used to specify: (1) a memory location using a offset (short integer or register) from an address register, (2) a short integer or bit-field constant, (3) access to the message port, or (4) access to any of the processor registers.

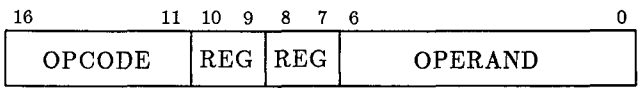

Figure 4: Instruction Format

In addition to the usual data movement, arithmetic, logical, and control instructions, the MDP provides instructions to:

- Read, write, and check tag fields.
- Look up the data associated with a key using the TBM register and set-associative features of the memory.
- Enter a key/data pair in the association table.
- Transmit a message word.
- Suspend execution of a method.

All instructions are type checked. Attempting an operation on the wrong class of data results in a trap. Traps are also provided for arithmetic overflow, for translation buffer miss, for illegal instruction, for message queue overflow, etc....

3 Micro Architecture

Figure 5 shows a block diagram of the MDP. Messages arrive at the network interface. The message unit (MU) controls the reception of these messages, and depending on the status of the instruction unit (IU), either signals the IU to begin execution, or buffers the message in memory. The IU executes methods by controlling the registers and arithmetic units in the data path, and by performing read, write, and translate operations on the memory. While the MU and IU are conceptually separate units,

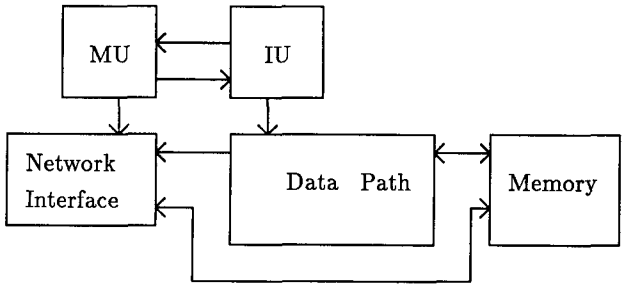

Figure 5: MDP Block Diagram

in the current implementation they are combined into a single controller.

3.1 Data Path

As shown in Figure 6, the data path is divided into two sections. The arithmetic section (left) consists of two copies of the general registers, and an arithmetic unit (ALU). The ALU unit accepts one argument from the register file, one argument from the data bus, and returns its result to the register file.

The address section (right) consists of the address, queue, IP,

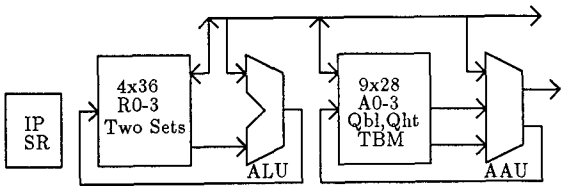

Figure 6: MDP Data Path

and TBM registers and an address arithmetic unit (AAU). Each register in the address section holds two 14-bit fields that are bit-interleaved so that corresponding bits of the two fields can be easily compared. The AAU generates memory addresses, and may modify the contents of a queue register. In a single cycle it can (1) perform a queue insert or delete (with wraparound), (2) insert portions of a key into a base field to perform a translate operation, (3) compute an address as an offset from an address register's base field and check the address against the limit field, or (4) fetch an instruction word and increment the corresponding IP.

3.2 Memory Design

A block diagram of the MDP memory is shown in Figure 7. The memory system consists of a memory array, a row decoder, a column multiplexor and comparators, and two row buffers (one for instruction fetch and one for queue access). Word sizes in this figure are for our prototype which will have only 1K words of RWM.

In the prototype, the memory array will be a 256-row by 144-column array of 3 transistor DRAM cells. In an industrial version of the chip, a 4K word memory using 1 transistor cells would be feasible. We wanted to provide simultaneous memory access for data operations, instruction fetches, and queue inserts; however, to achieve high memory density we could not alter the basic memory cell. Making a dual port memory would double the area of the basic cell. Instead, we have provided two row buffers that cache one memory row (4 words) each. One buffer is used to hold the row from which instructions are being fetched. The other holds the row in which message words are being enqueued. Address comparators are provided for each row buffer to prevent normal accesses to these rows from receiving stale data. We are considering using additional address comparators to provide spare memory rows that can be configured at power-up to replace defective rows.

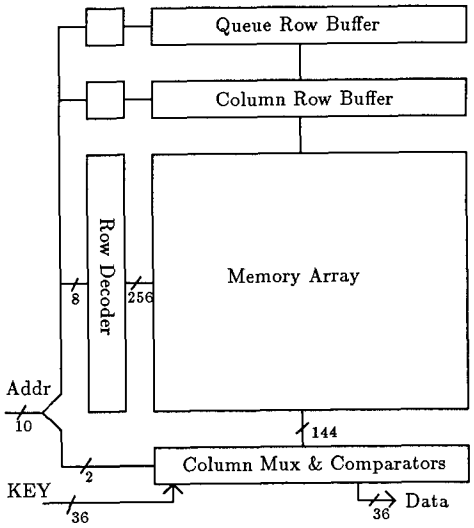

Figure 7: MDP Memory Block Diagram

The MDP memory is used both for normal read/write operations, and as a set-associative cache to translate object identifiers into physical addresses and to perform method lookup. These translation operations are performed as shown in Figure 8. The TBM register selects the range of memory rows that contain the translation table. The key being translated selects a particular row within this range. Comparators built into the column multiplexor compare the key with each odd word in the row. If a comparator indicates a match, it enables the adjacent even word onto the data bus. If no comparator matches the data a miss is signaled, and the processor takes a trap. For clarity, Figure 8 shows the words brought out separately. In fact, to simplify multiplexor layout, the words in a row are bit-interleaved.

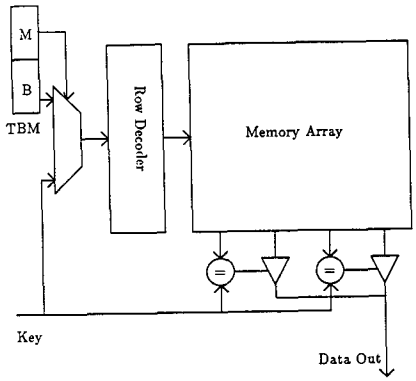

Figure 8: Associative Memory Access

3.3 Area Estimate

Our data paths use a pitch of 60λ (λ is half the minimum design rule) per bit giving a height of 2160λ. We expect the data path to be $\approx 3000\lambda$ wide for an area of $\approx 6.5M\lambda^2$. A 1K word memory array built from 3T DRAM cells will have dimensions of $\approx 2450\lambda \times 6150\lambda \approx 15M\lambda^2$. We expect the memory peripheral circuitry to add an additional $5M\lambda^2$. We plan to use an on chip communication unit similar to the Torus Routing Chip [5] which will take an additional $4M\lambda^2$. Allowing $8M\lambda^2$ for wiring gives a total chip area of $\approx 40M\lambda^2$ (or a chip about 6.5mm on a side in 2μ CMOS) for our 1K word prototype.

4 Execution Model

4.1 CALL and SEND

In a concurrent, object-oriented programming system, programs operate by sending messages to objects. Each method results in the execution of a method. The MDP supports this model of programming with the CALL and SEND messages.

The execution sequence for a CALL message is shown in Figure 9. The first word of the message contains the priority level (0), and

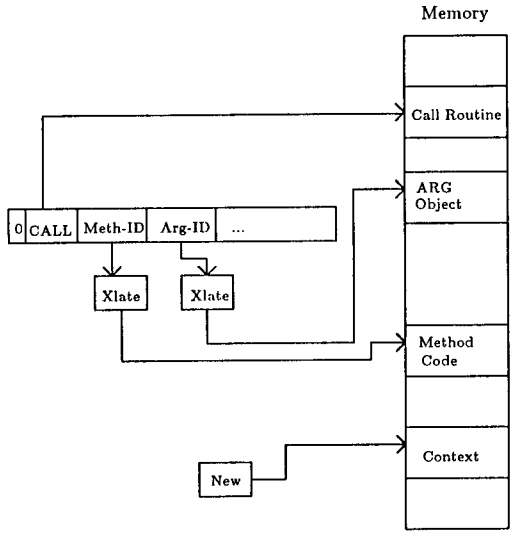

Figure 9: Processing a CALL Message

the physical address of the CALL subroutine. If the processor is idle, in the clock cycle following receipt of this word, the first instruction of the call routine is fetched. The call routine then reads the object identifier for the method. This identifier is translated into a physical address in a single clock cycle using the translation table in memory. If the translation misses, or if the method is not resident in memory, a trap routine performs the translation or fetches the method from a global data structure.

Once the method code is found, the CALL routine jumps to this code. The method code may then read in arguments from the message queue. This is accomplished by setting the queue-bit of A3 on message arrival. Subsequent accesses through A3 read words from the message queue. If the method faults, the message is copied from the queue to the heap. Register A3 is set to point to the message in the heap when the code is resumed. The argument object identifiers are translated to physical memory base/limit pairs using the translate 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 passing control to the next message.

A SEND message looks up its method based on a selector in the message, and the class of the receiver. This method lookup is shown in Figure 10. The receiver identifier is translated into a base/limit pair. Using this address, the class of the receiver is fetched. The class is concatenated with the selector field of the message to form a key that is used to look up the physical address of the method in the translation table. Once the method is found, processing proceeds as with the CALL message.

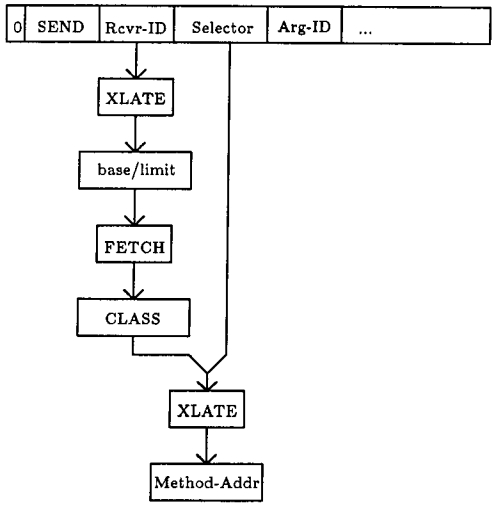

Figure 10: Method Lookup

4.2 Non-Local References and Futures

If either operand of an instruction is not of the proper type, a trap will occur. This hardware support for run-time type checking not only allows us to support dynamically-typed languages such as LISP and Smalltalk, but also allows us to handle local and non-local data uniformly. For example, suppose we attempt to access an instance variable of an object using the instruction `temp <- anObject at: aField`. If `anObject` is resident on the local node a simple memory reference is generated; however, if `anObject` is resident on a different node a message send results. This uniform handling of objects regardless of their location relieves the programmer and the compiler from keeping track of object locations. More importantly, it facilitates dynamically moving objects from node to node.

Futures are supported through the use of tags. Consider the instruction mentioned in the previous paragraph: `temp <- anObject at: aField`. If `anObject` is not local, a message will be sent with the `Reply-To:` slot of the message specifying the variable `temp` in the current context, and `temp` will be tagged as a `context future`. When the reply message arrives, as shown in Figure 11, it looks up the context object, and overwrites the specified slot with the proper value. In the meantime, execution continues until the program attempts to use the value in `temp` perhaps by executing `aVar <- temp + 1`. If when this instruction examines `temp` it is still tagged `Future`, the current context is suspended until the value of `temp` is available. If the `at:` message had already replied with the value of `temp`, however, the tag of `temp` would have signified a value and the context would not be suspended.

Futures can be handled in a more general sense by creating an object of class `future` to which the pending computation is to reply. References to this `future` object may then be passed outside of the local context. When the result of the pending computation is available, the future object `becomes` this value.

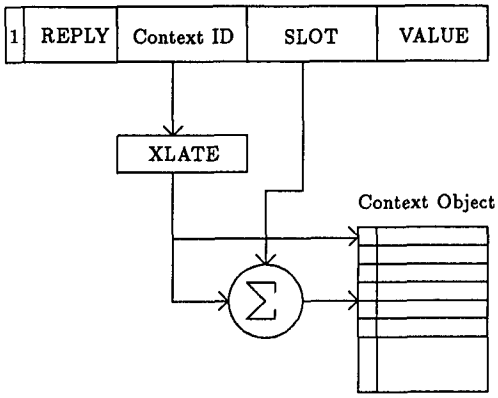

Figure 11: Processing A Reply Message

4.3 Multicast and Combining

In concurrent computations it is often necessary to fan data out to many destinations, and to accumulate data from many sources with an associative operator. In the MDP, these functions are performed by the `FORWARD` and `COMBINE` messages respectively.

The `FORWARD` message contains the identifier of a control object, and a message to be forwarded as specified in that object. The control object is a list of destinations to which the message should be forwarded along with the header (if any) which should precede the message. When the message arrives, the control object is located and a buffer is created in memory to hold the message. The message is read into the buffer and at the same time transmitted to the first destination in the list. The message is then transmitted to the subsequent destinations on the list, and the buffer is deallocated.

The combine message specifies the identifier of a combine object, and a message to be combined or forwarded. The combine object contains the destination to which combined messages are to be forwarded, buffers for combined messages awaiting a reply, and identifiers for the methods to be executed in response to combine or reply messages. The combining performed is controlled entirely by these user specified methods. The combine message is quite similar to a `CALL` differing only in that the method to be executed is implicit.

5 Performance

We have constructed both instruction-level and a register-transfer (RT) level simulators for the MDP. Using these simulators we have evaluated the time required by the MDP to perform a number of simple operations. These operations are tabulated in Table 1.

In this table, W specifies the number of words transferred, and N specifies the number of destinations for the `FORWARD` message. The times for `CALL`, `SEND`, and `COMBINE` are the time from message reception until the first word of the appropriate method is fetched. Times are expressed in clock cycles. We expect the clock period of our prototype to be 100ns.

Operation	Time
READ	5 + W
WRITE	4 + W
READ-FIELD	7
WRITE-FIELD	6
DEREFERENCE	6 + W
CALL	5
SEND	8
REPLY	7
FORWARD	5 + N × W
COMBINE	5

Table 1: MDP Message Execution Times (in clock cycles)

In the near future we plan to run benchmarks on a simulated collection of MDPs to measure the hit ratios in translation buffer and method cache (as a function of cache size), and effectiveness of the row buffers.

6 Conclusion

The message-driven processor (MDP) is able to process a set of messages that support an object-oriented concurrent programming system with an overhead of less than ten clock cycles per message. This performance, more than an order of magnitude improvement over existing message-passing systems, enables the MDP to efficiently run programming systems that exploit concurrency at a grain size of $\approx$ 10 instructions. In contrast existing machines operate efficiently only at a grain size of several hundred instructions. We conjecture that by exploiting concurrency at this fine grain size we will be able to achieve an order of magnitude more concurrency for a given application than is possible on existing machines.

The MDP achieves much of its performance by using a message-driven control mechanism. The MU handles reception and buffering of arriving messages as well as directing the operation of the IU. The IU simply executes instructions. It never makes a deci-

sion concerning whether to buffer or execute an arriving message. For each message, it is vectored to the proper entry point by the MU. A single message type, EXECUTE, with two priority levels, provides all the mechanism necessary to implement a concurrent programming system.

The MDP uses a memory based instruction set and two register sets to implement fast context switches. The dual register sets allow a high priority message to interrupt a lower priority message without saving state. The memory based instruction set allows a context to save its state in five clock cycles. Operating out of memory is almost as fast as operating out of registers since the memory is implemented on chip and can be accessed in a single clock cycle.

The MDP memory adds functionality in its peripheral circuitry while preserving the density of a simple memory array. The memory supports both indexed and associative access by placing comparators in the column multiplexor. The associative access mechanism speeds the execution of concurrent programs by allowing address translation and method lookup to be performed in a single clock cycle. This translation mechanism is made visible to the programmer so it can be applied in other situations (e.g., method lookup).

The MDP has been motivated by the development of high-performance message-passing networks [5]. In early message passing machines, message latency, in the milliseconds, was the limiting factor. Now that the message latency has been reduced to a few microseconds, we can no longer ignore processor latencies in hundreds of microseconds.

Some may argue that the MDP is unbalanced according to the rule of thumb stating that a 1MIP processor should have a 1MByte memory. The MDP is an $\approx$ 4MIP processor and only has a 16KByte memory (4KByte in the prototype). We argue however that it is not the size of the memory in a single node that is important, but rather the amount of memory that can be accessed in a given period of time. In a 64K node machine constructed from MDPs and using a fast routing network, a processor will be able to access a uniform address space of 2^{28} words (2^{30} Bytes) in less than $10\mu s$.

The MDP provides many of the advantages of both message-passing multicomputers and shared-memory multiprocessors. Like a shared-memory machine, it provides a single global name space, and needs to keep only a single copy of the application and operating system code. Like a message-passing machine, the MDP exploits locality in object placement, uses messages to trigger events, and gains efficiency by sending a single message through the network instead of sending multiple words. While we plan to implement an object-oriented programming system on the MDP, we also see the MDP as an emulator that can be used to experiment with other programming models.

Acknowledgement

The authors thank Tom Knight, Gerry Sussman, Chuck Seitz, and Bill Athas for their comments on the MDP architecture and the referees for their suggestions on how to improve this paper.

References

[1] Agha, Gul A., *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Artificial Intelligence Laboratory, Technical Report 844, June 1985.

[2] Ahuja, S.R., "S/Net: A High Speed Interconnect for Multicomputers," *IEEE Jounal on Selected Areas in Communications,* November 1983, pp. 751-756.

[3] Dally, William .J., and Kajiya, James T., "An Object Oriented Architecture," *Proc. 12th Symposium on Computer Architecture,* 1985, pp. 154-160.

[4] Dally, William J., *A VLSI Architecture for Concurrent Data Structures,* Ph.D. Thesis, Department of Computer Science, California Institute of Technology, Technical Report 5209:TR:86, 1986.

[5] Dally, William J. and Seitz, Charles L., "The Torus Routing Chip," to appear in *J. Distributed Systems,* Vol. 1, No. 3, 1986.

[6] Dally, William J., "Wire Efficient VLSI Multiprocessor Communication Networks," to appear in *Stanford Conference on Advanced Research in VLSI,* 1987.

[7] Intel Scientific Computers, *iPSC User's Guide,* Order No. 175455-001, Santa Clara, Calif., Aug. 1985.

[8] Halstead, R.H., "Parallel Symbolic Computing," *Computer,* Vol. 19, No. 8, Aug. 1986, pp. 35-43.

[9] Inmos Limited, *IMS T424 Reference Manual,* Order No. 72 TRN 006 00, Bristol, United Kingdom, November 1984.

[10] Lutz, C., et. al., "Design of the Mosaic Element," *Proc. MIT Conference on Advanced Research in VLSI,* Artech Books, 1984, pp. 1-10.

[11] Palmer, John F., "The NCUBE Family of Parallel Supercomputers," *Proc. IEEE International Conference on Computer Design, ICCD-86,* 1986, p. 107.

[12] Hoare, C.A.R., "Communicating Sequential Processes," *CACM,* Vol. 21, No. 8, August 1978, pp. 666-677.

[13] Seitz, Charles L., "The Cosmic Cube," *CACM,* Vol. 28, No. 1, Jan. 1985, pp. 22-33.

[14] Spector, Alfred, Z. "Performing Remote Operations Efficiently on a Local Computer Network," *CACM,* Vol. 25, No. 4, April 1982, pp. 246-260.

[15] Thacker, C.P., et. al., "Alto: A Personal Computer," in *Computer Structures: Principles and Examples,* Siewiorek, Bell, and Newell, Ed., McGraw Hill, 1982.

[16] Z-80 Product Description, Zilog Corporation, 1977.