DTIC FILE COPY

(4)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY VLSI PUBLICATIONS

VLSI Memo No. 88-454
August 1988

FINE-GRAIN MESSAGE-PASSING CONCURRENT COMPUTERS

William J. Dally

Abstract

Fine-grain concurrent computers, by operating at a fine grain, increase the amount of concurrency that can be efficiently exploited in a given problem. Programming is simplified because programs may be partitioned into natural units of methods and objects and these objects are addressed uniformly whether they are local or remote. The construction of these machines poses challenging problems in reducing overhead, increasing communication bandwidth, and developing resource management techniques. This paper describes this class of machines, the challenges posed by their construction, and recent progress toward meeting these challenges. Keywords: computers; manufacturing; Fine grain concurrent computers, (KR)

Microsystems Research Center Room 39-321 Massachusetts Institute of Technology Cambridge Massachusetts 02139 Telephone (617) 253-8138

Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-80-C-0622, N00014-87-K-0825 and N00014-85-K-0124, and in part by a National Sciences Foundation Presidential Young Investigators Award with matching funds from General Electric Corporation.

Author Information

Dally: Department of Electrical Engineering and Computer Science, Artificial Intelligence Laboratory and Laboratory of Computer Science, MIT, Cambridge, MA 02139; Room NE43-417, (617) 253-6043.

Fine-Grain Message-Passing Concurrent Computers [1]

William J. Dally
Artificial Intelligence Laboratory and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

Fine-grain concurrent computers, by operating at a fine grain, increase the amount of concurrency that can be efficiently exploited in a given problem. Programming is simplified because programs may be partitioned into natural units of methods and objects and these objects are addressed uniformly whether they are local or remote. The construction of these machines poses challenging problems in reducing overhead, increasing communication bandwidth, and developing resource management techniques. This paper describes this class of machines, the challenges posed by their construction, and recent progress toward meeting these challenges.

1 Introduction

The *grain size* of a program refers to the size of the tasks and messages that make up the program. Coarse-grain programs have a few long ($\approx$ 10ms) tasks, while fine-grain programs have many short ($\approx 5\mu s$) tasks. With more tasks that can execute at a given time - viz. more concurrency - fine-grain programs (in the absence of overhead) result in faster solutions than coarse-grain programs.

The *grain size* of a machine refers to the physical size and the amount of memory in one processing node. A coarse-grain processing node requires hundreds of chips (several boards) and has $\approx 10^7$ bytes of memory while fine-grain node fits on a single chip and has $\approx 10^4$ bytes of memory. Fine-grain nodes cost less and have less memory than coarse-grain nodes, however, because so little silicon area is required to build a fast processor, they need not have slower processors than coarse-grain nodes.

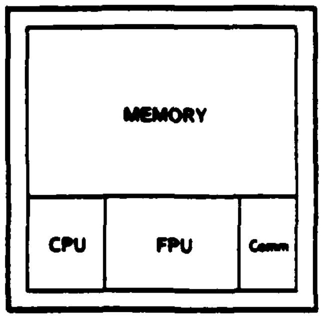

Figure 1: In the area of a 1Mbit DRAM chip one can construct a processing node with a 32-bit processor, a floating point unit, a communication controller, and 512Kbits of memory.

At MIT we are developing the J-Machine [12] as a research vehicle to investigate problems involved in the design and programming of concurrent computers with fine-grain processing nodes that efficiently execute fine-grain programs.

Processors are Inexpensive

VLSI technology makes it possible to build small, powerful processing elements. A 1M-bit DRAM chip has an area of $256M\lambda^2$ (λ is half the minimum line width [23].). In the same area we can build a single chip processing node as shown in Figure 1. The chip includes

A 32-bit processor	$16M\lambda^2$
A floating-point unit	$32M\lambda^2$
A communication controller	$8M\lambda^2$
512Kbits RAM	$128M\lambda^2$

[1] The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-80-C-0622 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation.

Such a single-chip processing node would have the same processing power as a board-sized node but significantly less memory. We refer to a machine built from these nodes as a *jellybean machine* as it is built with commodity part (jellybean) technology.

A fine-grain processing node has two major advantages: density and memory bandwidth. Several hundred single-chip nodes can be packaged on a single printed circuit board permitting us to exploit hundreds of times the concurrency of machines with board-sized nodes. With on-chip memory we can read an entire row of memory (128 or 256 bits) in a single cycle without incurring the delay of several chip crossings. This high memory bandwidth allows the memory to simultaneously buffer messages from a high bandwidth network and provide the processor with instructions and data.

Fine grain machines are quite efficient. We measure efficiency as

$$e_A = 1/AT \tag{1}$$

(where A is area and T is time) rather than

$$e_N = 1/NT \tag{2}$$

(where N is the number of processors). Proponents of coarse-grain machines argue that a machine constructed from several thousand single-chip nodes would be inefficient because many of the processing nodes will be idle. N is large, hence e_N is small. A user, however, is not concerned with N, but rather with what the machine costs, A, and how long it takes to solve a problem, T. Fine-grain machines have a very high e_A because they are able to exploit more concurrency in a smaller area.

Concurrency is Plentiful

Many computationally demanding problems have an abundance of concurrency. This concurrency exists at many levels: at the coarsest grain we iterate over the gridpoints of a problem. For each gridpoint we may perform some vector operations that can be carried out in parallel. Each operation may involve the evaluation of some expressions or method that can be performed simultaneously. Within one expression, several arithmetic operations can be performed in parallel.

At the level of methods (subroutines), the natural grain-size of a computation is 10 instructions [5]. The message transmission and reception overhead (the time for one edge in Figure 2) on existing message-passing computers is in excess of 500 instruction times. As a result these machines operate at a grain size of 2000 instructions. Conceptually 100 vertices of the fine-grain computation graph are grouped together to amortize the communication and synchronization overhead. By reducing communication and synchronization overhead to permit efficient execution at a grain size of 10 instructions we can exploit 100 times as much concurrency.

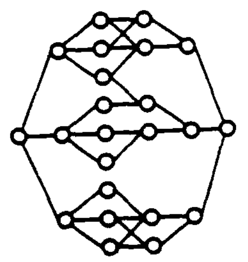

Figure 2: The computation graph of a concurrent program. The vertices represent a local computation being performed at a node of a concurrent computer. The edges represent communication actions between vertices. The time required to perform the computation is bounded below by the sum of edge and vertex times along the critical path for the computation.

A Global Address Space Simplifies Programming

A fine-grain machine with a global address space simplifies programming. Because the machine executes programs at their natural partition of methods and objects, the problem of partitioning the program into appropriate sized pieces (the grouping of vertices in Figure 2) is eliminated. Each object is a separate partition and each method is separately scheduled.

A global address space eliminates much of the bookkeeping required in a system with non-uniform naming. In many existing concurrent computers local objects are referenced through a pointer while global objects require an explicit send and receive [30]. Providing a global address space allows objects to be referenced via a single mechanism (the virtual address) regardless of their location, and relieves the programmer of the bookkeeping required to keep track of node numbers. Programs become both easier to write and more portable.

Background

The J-Machine builds on previous work in the design of message-passing and shared memory machines. Like the Caltech Cosmic Cube [28], the Intel iPSC [18], and the N-CUBE [24], each node of the J-Machine has a local memory and communicates with other nodes by passing messages. The J-Machine can exploit concurrency at a much finer grain than these early message passing computers. Delivering a message and dispatching a task in response to the message arrival takes 5µs on the J-Machine as opposed to 5ms on an iPSC. Like the BBN butterfly [4] and the IBM RP3 [25] the J-Machine provides a global virtual address space. The same IDs (virtual addresses) are used to reference on and off node objects. Like the InMOS transputer [17] and the Caltech MOSAIC [22] a J-Machine node is a single chip processing element integrating a processor, memory, and a communication unit. The J-Machine extends these previous efforts by providing efficient mechanisms for supporting fine-grain concurrent programming systems.

Outline

The major challenge in building a machine to exploit fine-grain concurrency is to reduce the overhead associated with message sending and task switching to a level that is small compared with the task size. This overhead has two components, T_{net}, the latency due to networks, and T_{node}, the latency due to task switching in a node. Low latency communication networks are described in Section 2. It is shown that low-dimensional k-ary n-cube networks outperform binary n-cubes (hypercubes). To exploit the low-latency of these networks requires processing elements that can react quickly to the arrival of messages. The architecture of such a message-driven processor is described in Section 3.

2 Interconnection Networks

VLSI systems are wire limited. The cost of these systems is predominantly that of connecting devices, and the performance is limited by the delay of these interconnections. Thus, an interconnection network must make efficient use of the available wire. The topology of the network must map into the three physical dimensions so that messages are not required to *double back* on themselves, and in a way that allows messages to use all of the available bandwidth along their path. Also, the topology and routing algorithm must be simple so the network switches will be sufficiently fast to avoid leaving the wires idle while making routing decisions.

Our recent findings suggest that low-dimensional k-ary n-cube interconnection networks [7] using *wormhole routing* [27] [19] and *virtual channels* [8] are capable of providing the performance required by fine-grain concurrent architectures. To test these ideas, we have constructed two prototype VLSI routing chips, the torus routing chip (TRC) [6], and the network design frame (NDF) [10]. The mesh routing chip MRC [14], based on similar principles, has been applied in a commercial product [2].

Wormhole Routing

With *wormhole routing* (Figure 3B) as soon as each *flit* (flow-control digit) of a message arrives at a node it is forwarded to the next node. With *store-and-forward routing* (Figure 3A), the method used by most existing concurrent computers, the entire message is received before forwarding the packet to the next node. Using wormhole routing gives a network latency, T_{WH}, that is the sum of a component due to message length normalized to channel width $\frac{L}{W}$, and a component due to the distance the message must travel, D. With *store-and-forward* routing, on the other hand, the latency, T_{SF}, is the

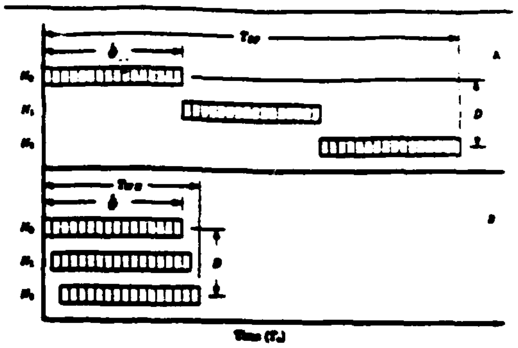

Figure 3: The latency of *store-and-forward routing* (A) compared to *wormhole routing* (B). Wormhole routing reduces latency from the product of $\frac{L}{W}$ and D to the sum of these two components.

product of these two components.

$$T_{WH} = T_C\left(\frac{L}{W} + D\right), \quad (3)$$

$$T_{SF} = T_C\left(\frac{L}{W} \times D\right), \quad (4)$$

where T_C is the channel transmission time, L is the message length in bits, W is the channel width in bits, and D is the number of channels the message must traverse (distance).

Consider a concurrent computer with 64K nodes connected as a 16-ary 4-cube with 8-bit wide channels ($W = 8$). Assuming no locality, the average distance a message must travel in this machine is $D = 15$. For 256-bit messages, $T_{WH} = 47T_C$, an order of magnitude less than $T_{SF} = 480T_C$.

Low-Dimensional k-ary n-Cubes

Many concurrent computers have been built using binary n-cube (hypercube) interconnection networks because these networks are optimal when all channels are considered equal. However, considering a channel in a binary n-cube to be equal to a channel in a low-dimensional network is not a reasonable assumption. Because binary n-cubes have long wires and high bisection widths their channels are typically narrower and slower than the channels in a low-dimensional network. When these factors are taken into account, the low-dimensional networks out-perform the high-dimensional networks.

Consider the networks shown in Figure 4. Suppose the binary 6-cube has 4-bit wide channels (as in the Caltech Cosmic Cube [28]). An 8-ary 2-cube with 16-bit wide channels has the same wiring complexity. With wormhole routing and 256-bit messages the 6-cube has a latency of $67T_C$ while the 2-cube has a latency of only $20T_C$. Increasing the radix, k, of a k-ary n-cube while holding wiring complexity (bisection width) constant increases both $W \propto k$ and $D \propto kn$. This decreases the component of latency due to message length, $\frac{L}{W}$, while increasing the component due to distance, D. The minimum latency occurs when these two components are nearly equal (Figure 5). For $L \approx 200$ the optimum dimension, n, is two for up to 1K nodes and three for 1K to 32K nodes, and four for 32K to 1M nodes.

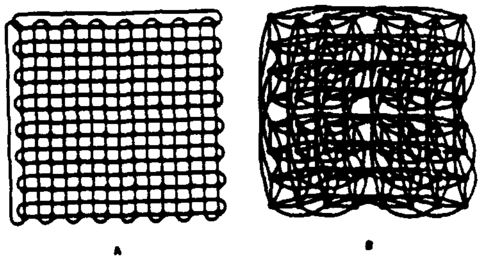

Figure 4: Two 64-node k-ary n-cubes: an 8-ary 2-cube (A) and a binary 6-cube (B). Network A has a bisection width of 16 channels while B has a bisection width of 64 channels. Thus the channels in A can be made four times as wide as the channels in B for the same wiring complexity.

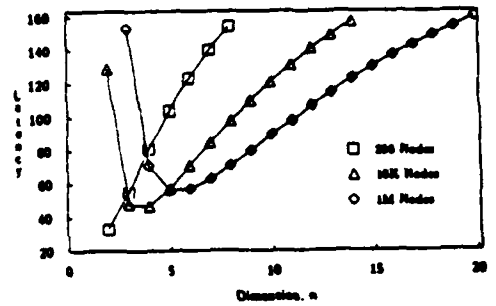

Figure 5: Latency as a function of dimension for networks of constant bisection width (B=N, L=150). Low-dimensional networks (left) are distance limited, while high-dimensional networks (right) are message-length limited.

The throughput of a network is the maximum number of messages that can be delivered per unit time. It is often expressed as a fraction of the network's capacity, the number of messages that would be delivered if every channel of the network was fully used. As the amount of traffic in the network increases, the latency of a message is increased. The latency given by (3) assumes an unloaded network.

We have developed a queueing model of k-ary n-cube wormhole networks that accurately predicts the latency as a function of network traffic, and allows us to calculate the maximum throughput for a given network configuration [7]. Figure 6 shows how latency varies with traffic for a 32-ary 2-cube

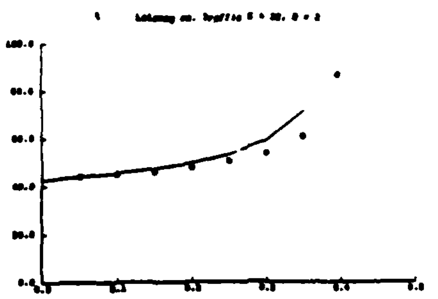

Figure 6: Latency vs. Traffic for a 32-ary 2-cube, L=200bits. Solid line is predicted latency, points are measurements taken from a simulator.

(1024 nodes). The solid line is the predicted latency. The points are measurements taken from a simulator. The model agrees with the simulation within 5%, with the model being slightly pessimistic, until the network approaches saturation. Latency increases less than 20% as traffic is increased from zero to 30% capacity. Saturation (maximum throughput) occurs at $\approx$ 40% capacity.

Low-dimensional networks have several other advantages.

- Because wires are shorter, the channels in these networks typically operate faster than in high dimensional networks, increasing throughput and further decreasing latency.
- Low-dimensional networks have better queueing performance. If one thinks of channels as being servers, these networks have fewer servers with greater capacity resulting in a lower average service time.
- Because the control logic for a network switch typically scales with the number of dimensions, the switches for low-dimensional networks are simpler than those for high-dimensional networks.

Virtual Channels

Until recently there was no known algorithm for deadlock-free routing in k-ary n-cube, wormhole networks. The conventional *structured buffer pool* algorithms that are used in store-and-forward networks are not applicable to networks that use wormhole routing. These algorithms interleave the items being buffered (packets in a store-and-forward network), but wormhole networks buffer flits that cannot be interleaved.

We have developed a new class of algorithms for deadlock free routing based on the concept of *virtual channels*. Shown in Figure 7, virtual channel algorithms operate by restrict-

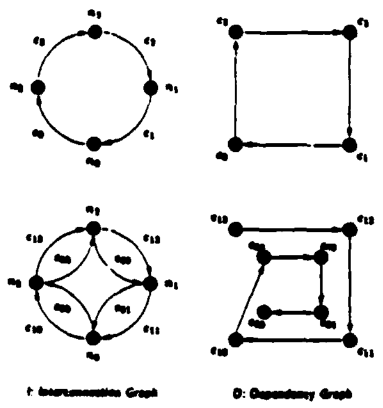

Figure 7: Considering routing to be a function $C \times N \mapsto C$ rather than the conventional $N \times N \mapsto C$ deadlock corresponds to cycles in the channel dependency graph (right) rather than the interconnection graph (left). By multiplexing two virtual channels on each physical channel, we can restrict the routing function to eliminate deadlock (bottom).

ing routing rather than by restricting buffer allocation. To do this requires that routing be a function of the channel a message arrives on and the destination node, $C \times N \mapsto C$, rather than the node a message is on and the destination node, $N \times N \mapsto C$. Projecting this function gives a dependency relation among channels. By multiplexing several virtual channels on each physical channel we can restrict routing in a manner that avoids deadlock without loosing strong connectivity. A set of virtual channels all share the same physical wires. Each virtual channel requires only a single flit buffer. The virtual channel method can be used to route deadlock free in any strongly connected network [8].

The Torus Routing Chip

The Torus Routing Chip (TRC), shown in Figure 8, is a self-timed [26] VLSI chip that performs wormhole routing in k-ary n-cube networks, and uses virtual channels to prevent deadlock [6]. A single TRC provides 8-bit data channels in two dimensions and can be cascaded to add more dimensions. A TRC network can deliver a 150-bit message in a 1024 node 32-ary 2-cube with an average latency of 7.5μs.

The Network Design Frame

The Network Design Frame (NDF) [10] incorporates a partitioned switch architecture [14], bidirectional data channels, and low-voltage output drivers to achieve a worst-case latency of 5μs in a 4K node 64-ary 2-cube. In the partitioned switch architecture, shown in Figure 9, the routing logic is partitioned into two-way switches. The partitioned switch's data paths and control logic are simpler (and thus smaller and faster) than the centralized crossbar design used in the TRC. A signal passes through only 10 gate delays from input to output for a propagation delay of 20ns (estimated).

Bidirectional data channels are used in the NDF to reduce latency and to exploit locality. Because wire density is a major limitation, the two directions of communication will share the same data wires. While the NDF is constructed using CMOS technology, communication on these bidirectional data wires uses ECL signal levels to improve speed, reduce power dissipation, and reduce noise. The NDF uses low-voltage swing output pads based on a design by Knight [20]. Reducing the voltage swing by a factor of 5 makes these pads 5 times as fast as conventional pads. Also, because power goes as the square of voltage, $P = CV^2f$, these pads dissipate 1/25 (4%) as much power as conventional pads. Since much of the power in the machine goes into driving the inter-node wires, this savings represents a considerable reduction in total power dissipation.

Adaptive Routing

The TRC and NDF are oblivious routers - viz. the route selected for a message is determined only by the source and destination nodes. In particular, they route a message first in the X direction and then in the Y direction. As shown in Figure 10 if several sources having the same Y coordinate transmit messages to several destinations having the same X coordinate only one message can proceed at a time[3].

As shown in Figure 11, simply relaxing the X-Y routing order could result in deadlock. The deadlock can be avoided by doubling the virtual channels in the north and south directions to separate eastbound messages from westbound messages [21]. We have recently undertaken the design of an adaptive router chip (ARC) based on this technique.

3 A Message-Driven Processor

Conventional instruction processors are ill-suited to serve as processing nodes in a concurrent computer. Their I/O systems are designed to handle high-latency peripherals (e.g., disks) and thus they respond slowly ($\approx$ 100 instruction times) to messages arriving over the network. Also, their register-oriented instruction sets, designed to match a fast processor with a slow memory in programming environments where context switches are infrequent (1 in $\approx$ 25000 instructions), are not appropriate in a processing node containing a fast local memory and in an environment where context switches happen every 10 instructions.

[3]Only one of the two conditions (source Y coordinates or destination X coordinates) must be present to cause congestion.

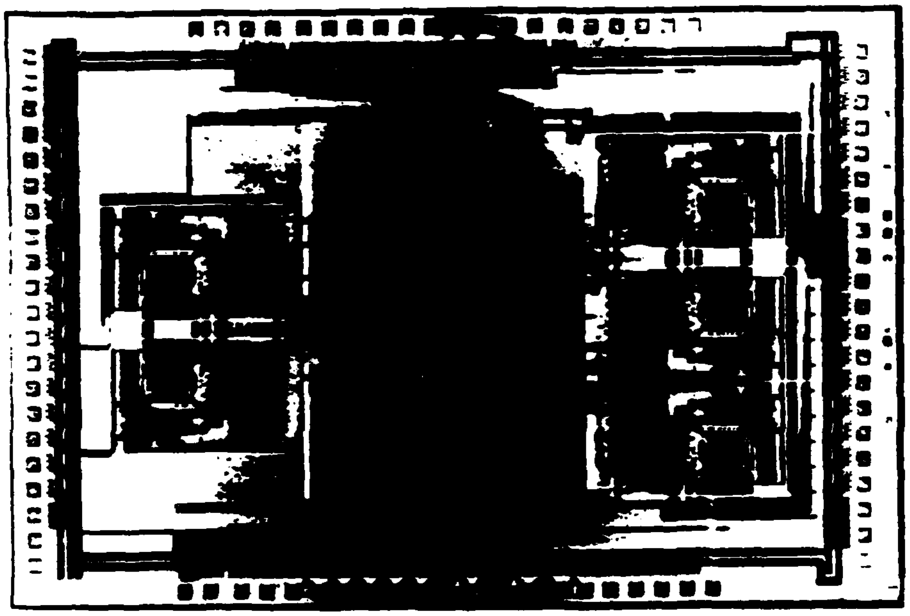

Figure 8: Photomicrograph of the Torus Routing Chip (TRC).

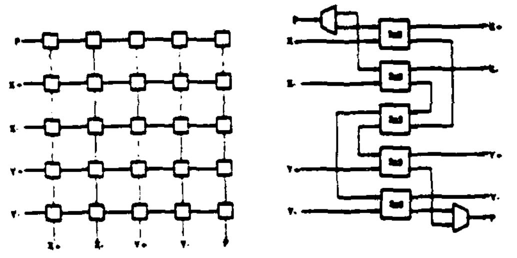

Figure 9: By using a partitioned datapath (right) the NDF requires less area and runs faster than the TRC which uses a centralized crossbar switch (left).

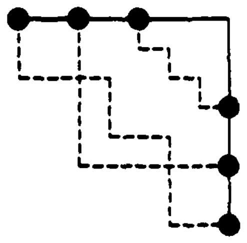

Figure 10: A pathological message pattern. Three sources with the same Y coordinate transmit messages to three destinations with the same X coordinates. With oblivious X-Y routing (solid lines) only one message can proceed at a time. An adaptive router (dashed lines) can make use of alternate paths to route the messages without interference.

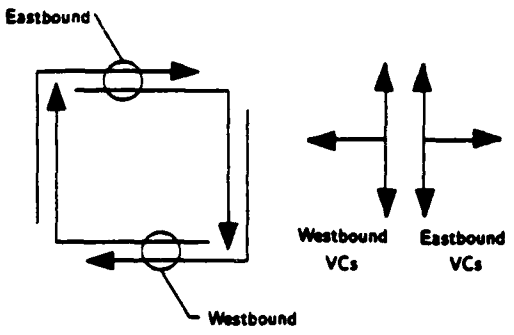

Figure 11: (A) Relaxing the X-Y routing order results in cycles in the channel dependency graph and thus a potential deadlock. (B) To prevent deadlock we can add additional virtual channels to separate eastbound messages from westbound messages.

The solution adopted in many machines is to increase the memory size of the node so a larger part of the problem can be performed in each node. This has the effect of reducing the concurrency to a point where the number of instructions executed between messages exceeds 10^3. This increases the perceived efficiency from 10% to 90% when measured in terms of e_N (2). This measure of efficiency, however, ignores the cost of the node. If instead we measure efficiency in terms of e_A (1), the actual efficiency has been reduced by making the node larger. To truly increase the efficiency, we must build small, efficient nodes.

At MIT, we are developing the message-driven processor (MDP), a small, efficient processing node for a message-passing concurrent computer [9]. 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.

The MDP provides the following mechanisms

1. A send instruction to inject short messages into the network with a minimum of delay.
2. A message unit that controls the reception and buffering of messages.
3. A scheduling mechanism that decides when to preempt execution and selects a message to be executed when a method suspends.
4. A general translation mechanism.
5. A small processor state and two sets of processor registers to support fast task switches.

Send Instruction

The MDP injects messages into the network using a send instruction that transmits one or two words (at most one from memory) and optionally terminates the message. The first word of the message is interpreted by the network as an absolute node address (in x,y format) and is stripped off before delivery. The remainder of the message is transmitted without modification. A typical message send is shown in Figure 12. The first instruction sends the absolute address of the destination node (contained in R0). The second instruction sends two words of data (from R1 and R2). The final instruction sends two additional words of data, one from R3, and one from memory. The use of the SENDE instruction marks the end of the message and causes it to be transmitted into the network. In a Concurrent Smalltalk message, the first word is a message header, the second specifies the receiver, the third word is the selector, subsequent words contain arguments, and the final word is a continuation. On our register-transfer simulator, this sequence executes in 4 clock cycles.

Early in the design of the MDP we considered making a message send a single instruction that took a message template, filled in the template using the current addressing environment, and transmitted the message. Each template entry specified one word of the message as being either a constant, the contents of a data register, or a memory reference offset from an address register (like an operand descriptor). The template approach was abandoned in favor of the simpler

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

Figure 12: MDP assembly code to send a 4 word message uses three variants of the SEND instruction.

one or two operand SEND instruction because the template did not significantly reduce code space or execution time. A two operand SEND instruction results in code that is nearly as dense as a template and can be implemented using the same control logic used for arithmetic and logical instructions.

Message Reception

Message reception overhead is reduced to $\approx 1\mu s$ by buffering, scheduling, and dispatching messages in hardware. The MDP maintains two message/scheduling queues (corresponding to two priority levels) in its on-chip memory. As messages arrive over the network, they are buffered in the appropriate queue. The queues are implemented as circular buffers. It is important that the queue have sufficient performance to accept words from the network at the same rate at which they arrive. Otherwise, messages would backup into the network causing congestion. To achieve the required performance, special addressing hardware is used to enqueue or dequeue a message word with wraparound and full/empty check in a single clock cycle. A queue row buffer allows enqueuing to proceed using one memory cycle for each four words received. Thus a program can execute in parallel with message reception with little loss of memory bandwidth.

The MDP schedules the task associated with each queued message. At any point in time, the MDP is executing the task associated with the first message in the highest priority non-empty queue. If both queues are empty, the MDP is idle – viz., executing a background task. Sending a message implicitly schedules a task on the destination node. The task will be run when it reaches the head of the queue. This simple two-priority scheduling mechanism removes the overhead associated with a software scheduler. More sophisticated scheduling policies may be implemented on top of this substrate.

Messages become *active* either by arriving while the node is idle or executing at a lower priority, or by being at the head of a queue when the preceding message *suspends* execution. When a message becomes active, a handler is dispatched in one clock cycle. The dispatch forces execution to a physical address specified in the message header. This mechanism is used directly to process messages requiring low latency (e.g., combining and forwarding). Other messages (e.g., remote procedure call) specify a handler that locates the required method (using the translation mechanism described below) and then transfers control to it.

For example, the call handler code is shown in Figure 13 and its execution is depicted in Figure 14. The first instruction gets the method ID (offset 1 into the message). To facilitate access to the message arguments, hardware initializes register A3 to contain an address descriptor (base/length) for the current message. The next instruction translates the method ID into an address descriptor for the method. If the translate faults, because the method is not resident or the descriptor is not in the cache, the fault handler *fixes* the problem and reschedules the message. If the translation succeeds, the final instruction (resume) transfers control to the method. The method code may then read in arguments from the message queue. The argument object identifiers are translated to physical memory base/length pairs using the 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.

```
MOVE   [1,A3],R0 ; get method id
XLATE  R0,A0     ; translate to address descriptor
RES    2         ; transfer control to method
```

Figure 13: MDP assembly code for the CALL message.

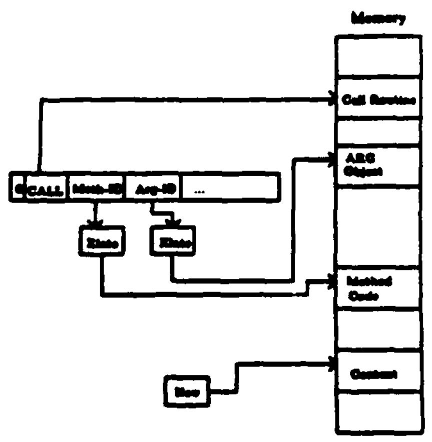

Figure 14: The CALL message invokes a method by translating the method identifier to find the code, creating a context (if necessary) to hold local state, and translating argument identifiers to locate arguments.

An early version of the MDP had a fixed set of message handlers in microcode. An analysis of these handlers showed that their performance was limited by memory accesses. Thus there was little advantage in using microcode. The microcode was eliminated, the handlers were recoded in assembly language, and the *message opcode* was defined to be the physical address of the handler routine. Frequently used handlers are contained in an on-chip ROM. This approach simplifies the control structure of the machine and gives us flexibility to redefine message handlers to fix bugs, for instrumentation (e.g., to count the number of sends), and to implement new message types.

The message queue originally allocated storage from the heap for each incoming message. This eliminated the need to copy messages when a method suspended for intermediate results. However, the cost of allocating and reclaiming storage for each message proved to be prohibitive. Instead, we settled on the preallocated circular buffer. When a method suspends for intermediate results, message arguments are copied into a context object. The overhead of this copying is small since the context must be created anyway to specify a continuation and to hold live variables. The fixed buffer also provides a convenient layering. Priority zero messages are sent when the memory allocator runs out of room and priority one messages are sent when the priority zero queue fills.

Translation

The MDP is an experiment in unifying shared-memory and message-passing parallel computers. Shared-memory machines provide a uniform global name space (address space) that allows processing elements to access data regardless of its location. Message-passing machines perform communication and synchronization via node-to-node messages. These two concepts are not mutually exclusive. The MDP provides a virtual addressing mechanism intended to support a global name space while using an execution mechanism based on message passing.

The MDP implements a global virtual address space using a very general translation mechanism. The MDP memory allows both indexed and set-associative access. By building comparators into the column multiplexer of the on-chip RAM, we are able to provide set-associative access with only a small increase in the size of the RAM's peripheral circuitry.

The translation mechanism is exposed to the programmer with the `ENTER` and `XLATE` instructions. `ENTER Ra,Rb` associates the contents of `Ra` (the key) with the contents of `Rb` (the data). The association is made on the full 36 bits of the key so that tags may be used to distinguish different keys. `XLATE Ra,Ab` looks up the data associated with the contents of `Ra` and stores this data in `Ab`. The instruction faults if the lookup *misses* or if the data is not an address descriptor. `XLATE Ra,Rb` can be used to lookup other types of data. This mechanism is used by our system code to cache ID to address descriptor (virtual to physical) translations, to cache ID to node number (virtual to physical) translations, and to cache class/selector to address descriptor (method lookup) translations.

Tags are an integral part of our addressing mechanism. An ID may translate into an address descriptor for a local object, or a node address for a global object. The tag allows us to distinguish these two cases and a fault provides an efficient mechanism for the test. Tags also allow us to distinguish an ID key from a class/selector key with the same bit pattern.

Most computers provide a set associative cache to accelerate translations. We have taken this mechanism and exposed it in a pair of instructions that a systems programmer can use for any translation. Providing this general mechanism gives us the freedom to experiment with different address translation mechanisms and different uses of translation. We pay very little for this flexibility since performance is limited by the number of memory accesses that must be performed.

Context Switches

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 instructions that require up to three operands to execute in a single cycle. The entire state of a context may be saved and restored in less than 12 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.

Synchronization using Tags

An MDP word is 36-bits: a 4-bit tag and a 32-bit datum. Tags are used both to support dynamically-typed programming languages and to support concurrent programming constructs such as relocatable objects and futures.

For example, consider the case where an object, A, sends a message to an object, B, instructing B to perform some computation and then to return the result in a *reply* message to update A's local variable x. To synchronize with the reply, A, first tags x as a `C-FUT` (for context future) then sends the message and proceeds without waiting for a reply. If the reply arrives before A uses x execution simply continues. An attempt to use x before the reply, however, results in a trap that suspends execution until the reply arrives.

The Effects of a Small Memory

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. In fact, a copy of the entire operating system will not fit into a node's memory. Each MDP keeps a method cache in its memory and fetches methods from a single distributed copy of the program on cache misses.

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 36KByte memory. 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^{29} words (2^{31} 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.

4 Conclusion

The J-Machine efficiently executes fine-grain concurrent programs by providing mechanisms that reduce the overhead of message-passing, translation, and context switching to $\approx 5\mu s$. Reducing overhead to a time comparable with the natural grain size of many concurrent programs allows the programmer to exploit all of the concurrency present in these programs rather than grouping many grains together - reducing the concurrency to improve the efficiency.

Low-dimensional k-ary n-cube networks that use wormhole routing and virtual channels can send a 6-word message across the diameter of a 4K-node concurrent computer in $4\mu s$. These low-dimensional networks ($8 \leq k \leq 64$ and $2 \leq n \leq 4$) outperform binary n-cubes ($k = 2$) because they balance the component of latency due to message length with the component due to distance. These networks are implemented with VLSI chips such as the TRC [6], the NDF [10], and the MRC [14] that perform all routing and buffering internally using no memory bandwidth or CPU time on intermediate nodes. Adaptive routers are being developed that will further improve routing performance by reducing contention.

The Message-Driven Processor (MDP) can perform a task switch on message arrival in $1\mu s$. The MDP performs message reception, buffering, and scheduling in hardware to eliminate the software overhead of $100\mu s$ or more associated with these functions. Task switches are performed quickly because the MDP is memory rather than register based. The MDP memory provides both associative and indexed access. The associative access is used to support a global virtual address space needed to support concurrent programming systems. The MDP provides very general hardware mechanisms that can support many different concurrent programming models including conventional message-passing [30], actors [1] [3], futures [15], communicating processes [16], and dataflow [13]. All of these programming models require the same execution mechanisms: communication, synchronization, and translation. Specializing a machine for a particular model of computation results in only a small increase in performance.

Concurrent programming is not difficult if suitable abstractions are used. Programmers should use the natural partition of the problem and not be concerned with placement. Synchronization can be performed by allowing the data flow of the program to sequence the required operations. As this technology matures, we expect to see abstractions for concurrency that will make concurrent programming no more difficult than sequential programming.

Many challenging problems in the design of hardware and software for concurrent computers remain. A major research area is the design of fault tolerant systems. While we can construct a 4K node machine with an MTBF of 2400 hours (4K chips at 100FITS), future machines may have MTBFs of only a few hours and will require architectures that can survive node and link failures without loss of data.

The mechanisms described here efficiently execute concurrency at a grain size of $5\mu s$. Many numerical programs, however, have potential concurrency at the level of single operations. Architectures must be developed that can exploit this concurrency without incurring the overhead of message delivery or synchronization.

Another critical problem is the development of (communication, processor, and memory) resource management policies for concurrent operating systems. It is quite easy to write a program with sufficient concurrency to swamp any concurrent machine. A concurrent operating system must provide a means to *throttle back* such massively concurrent applications to match the concurrency to the available resources.

Concurrent programming systems are still quite primitive. Abstractions for concurrency that express common patterns of computation while hiding the details of implementation are required [11]. Compilers should perform optimizations that expose concurrency in programs and automate the placement of objects onto processing nodes. Concurrent software technology must mature for these powerful machines to see widespread use.

Acknowledgement

The following MIT students have contributed to the work described here: Linda Chao, Andrew Chien, Stuart Fiske, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Michael Larivee, Paul Song, Brian Totty, and Scott Wills.

I thank Tom Knight, Gerry Sussman, Steve Ward, Dave Gifford, Tom Leighton, and Carl Hewitt of MIT, Chuck Seitz and Bill Athas of Caltech, and Adriaan Ligtenberg of AT&T Bell Laboratories for many valuable suggestions, comments, and advice.

References

[1] Agha, Gul A., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

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

[3] Athas, W.C., and Seitz, C.L., *Cantor Language Report*, Technical Report 5232:TR:86, Dept. of Computer Science, California Institute of Technology, 1986.

[4] BBN Advanced Computers, Inc., *Butterfly Parallel Processor Overview*, BBN Report No. 6148, March 1986.

[5] Dally, William J., *A VLSI Architecture for Concurrent Data Structures*, Kluwer, Hingham, MA, 1987.

[6] Dally, William J. and Seitz, Charles L., "The Torus Routing Chip," *J. Distributed Systems*, Vol. 1, No. 3, 1986, pp. 187-196.

[7] Dally, William J. "Wire Efficient VLSI Multiprocessor Communication Networks," *Proceedings Stanford Conference on Advanced Research in VLSI*, Paul Losleben, Ed., MIT Press, Cambridge, MA, March 1987, pp. 391-415.

[8] Dally, William J. and Seitz, Charles L., " Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, Vol. C-36, No. 5, May 1987, pp. 547-553.

[9] Dally, William J. et.al., "Architecture of a Message-Driven Processor," *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, June 1987, pp. 189-196..

[10] Dally, William J., and Song, Paul., "Design of a Self-Timed VLSI Multicomputer Communication Controller," *Proc. International Conference on Computer Design, ICCD-87*, 1987, pp. 230-234.

[11] Dally, William J., "Concurrent Data Structures," Chapter 7 in *Message-Passing Concurrent Computers: Their Architecture and Programming*, C.L. Seitz et. al., Addison-Wesley, Reading, MA, publication expected 1988.

[12] Dally, William J., "Concurrent Computer Architecture," *Proceedings of Symposium on Parallel Computations and Their Impact on Mechanics*, 1987.

[13] Dennis, Jack B., "Data Flow Supercomputers," *IEEE Computer*, Vol. 13, No. 11, Nov. 1980, pp. 48-56.

[14] Flaig, Charles, M., *VLSI Mesh Routing Systems*, Technical Report 5241:TR:87, Dept. of Computer Science, California Institute of Technology, 1987.

[15] Halstead, Robert H., "Parallel Symbolic Computation," *IEEE Computer*, Vol. 19, No. 8, Aug. 1986, pp. 35-43.

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

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

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

[19] Kermani, Parviz and Kleinrock, Leonard, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, Vol 3., 1979, pp. 267-286.

[20] Knight, Tom, and Krymm, Alex, "Self Terminating Low-Voltage Swing CMOS Output Driver," *Proc. Custom Integrated Circuits Conference*, 1987.

[21] Ligtenberg, Adriaan, Presentation at *1987 Princeton Workshop on Algorithm, Architecture, and Technology Issues in Models of Concurrent Computation*, October 1987.

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

[23] Mead, Carver A. and Conway, Lynn A., *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.

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

[25] Pfister, G.F. et. al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proc. International Conference on Parallel Processing, ICPP*, 1985, pp. 764-771.

[26] Seitz, Charles L., "System Timing" in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Addison-Wesley, 1980, Ch. 7.

[27] Seitz, Charles L., et al., *The Hypercube Communications Chip*, Display File 5182:DF:85, Dept. of Computer Science, California Institute of Technology, March 1985.

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

[29] Seitz, Charles L., Athas, William C., Dally, William J., Faucette, Reese, Martin, Alain J., Mattisson, Sven, Steele, Craig S., and Su, Wen-King, *Message-Passing Concurrent Computers: Their Architecture and Programming*, Addison-Wesley, publication expected 1988.

[30] Su, Wen-King, Faucette, Reese, and Seitz, Charles L., *C Programmer's Guide to the Cosmic Cube*, Technical Report 5203:TR:85, Dept. of Computer Science, California Institute of Technology, September 1985.