Mechanisms for Efficient, Protected Messaging

by

Whay Sing Lee

S.B. Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992
S.M. Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1994

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy
in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
February 1999

Signature of Author _____

Department of Electrical Engineering and Computer Science
January 20, 1999

Certified by _____

Dr. William J. Dally
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by _____

Dr. Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Mechanisms for Efficient, Protected Messaging

by

Whay Sing Lee

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of
the requirements for the Degree of Doctor of Philosophy in
Electrical Engineering and Computer Science

Abstract

Fine-grain parallelism is the key to high performance muticomputing. By partitioning problems into small sub-tasks – grain-sizes as small as 70 cycles have been found in common benchmark programs – fine-grain parallelization accelerates existing applications beyond current limits, and promises efficient exploitation of multicomputers consisting of thousands of processors. However, contemporary multiprocessor architectures are not equipped to exploit parallelism at this level, due to high communication and synchronization costs that must be amortized over a large grain size. Operating system-managed message interfaces account for most of the high inefficiency in traditional systems. Conversely, in contemporary user-level network interfaces, fast hardware is defeated by software layers that are needed to provide safeguards against starvation and protection violation.

This thesis addresses both the efficiency and robustness issues in the message interface. I propose a design which features a processor-register mapped, atomic-injection, streaming-extraction message interface where handlers are dispatched in a dedicated hardware thread slot. Compared to a conventional interrupt-driven, memory-buffered interface, this design yields an order-of-magnitude performance improvement – of which 60% is due to the fast dispatch mechanism while 30% is due to *mapping* and *atomicity* choices. Hardware-support for *address translation* accounts for the remaining 10% overhead reduction.

Protection and starvation avoidance are achieved through bounded-time message injection from dedicated assembly buffers, and by using *guarded pointers* to regulate the data and remote operations accessible to each user thread – only non-blocking *trusted handlers* that honor the protection constrains are able to access the network port. With a flush mechanism, the interface can quickly return to a known-good state from unexpected error conditions, permitting flexible user-optimized message handlers, subject to simple restrictions that facilitate the certification of trusted properties by the compiler.

An analytical model is also developed to relate performance to the *latency* and *occupancy* of message operations, and application *grain-size*. Performance is found to be extremely sensitive to occupancy for a sample application with ~40 cycles thread length, but less so to latency, due to masking by slack. Results suggest that occupancy $\ll 50$ processor cycles and latency < 200 cycles are critical for fine-grain computing. At < 10 cycles null-message `SEND` occupancy, the M-Machine interface enables even a moderately-sized program (16K-Cell LIFE) to achieve $> 50\%$ efficiency while employing thousands of processors, and to reach a speedup of 5000 times.

Thesis Supervisor: Dr. William J. Dally
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Professor William Dally for being a most excellent mentor. His guidance is what made this work possible. It has been a most enriching and fulfilling experience for me, and I am grateful for the patience and understanding that he has had for me in the process. My appreciation also goes to Tom Knight and Anant Agarwal for their invaluable feedback towards this work, and to Charles Leiserson for his insights and advice over the years. For their constant encouragement, assistance and stimulating discussions, I would like to thank my esteemed colleagues in the M-Machine project, Steve Keckler, Andrew Chang, Nick Carter and Marco Fillo, and also other fellow members of the CVA group, especially Scoot Rixner. And to my family, I owe a debt of gratitude for their unconditional support during all this time.

The research in this thesis has been supported by the Defense Advanced Research Projects Agency monitored by the Air Force Electronic Systems Division under contract F19628-92-C-0045.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis develops a very low overhead multicomputer messaging system that makes no compromise on robustness. The design targets architectures that require an efficient communication system for high performance computing through the exploitation of fine-grain parallelism. It also recognizes that any software workaround needed to circumvent starvation and protection problems overlooked in the underlying architecture greatly impedes efficiency. The system thus minimizes latency and processor occupancy [1] while specifically preventing concurrently-executing user-level programs from accessing unauthorized domains or denying service to one another.

Efficiency is achieved through a collection of simple, complementary user-level mechanisms. Starvation by way of resource monopolization is prevented with a *bounded time lease* allocation scheme, while protection is accomplished through a lightweight capability system, using *guarded pointers.* To understand the tradeoffs, the thesis also quantifies the overhead due to each primitive design choice, and develops a *LOG model* – that relates performance to Latency, Occupancy and Grain size – for measuring the impact of communication overhead on large scale, fine-grain parallel computing.

The thesis is focused on the interface between user-level programs and the communication system hardware. The multithreaded, multi-ALU MIT M-Machine [1] is

[1] *Processor occupancy* refers to the amount of productive computation displaced by message-related operations.

used as the experimental platform for this study. In the resulting M-Machine system, only five cycles of processor occupancy are consumed in sending a null message. It thus demonstrates that a cost-effective design with occupancy $\ll 50$ processor cycles and latency < 200 cycles – which the LOG model analysis shows to be critical for fine-grain parallelization, at least in the LIFE program – is indeed practical.

1.1 Motivation

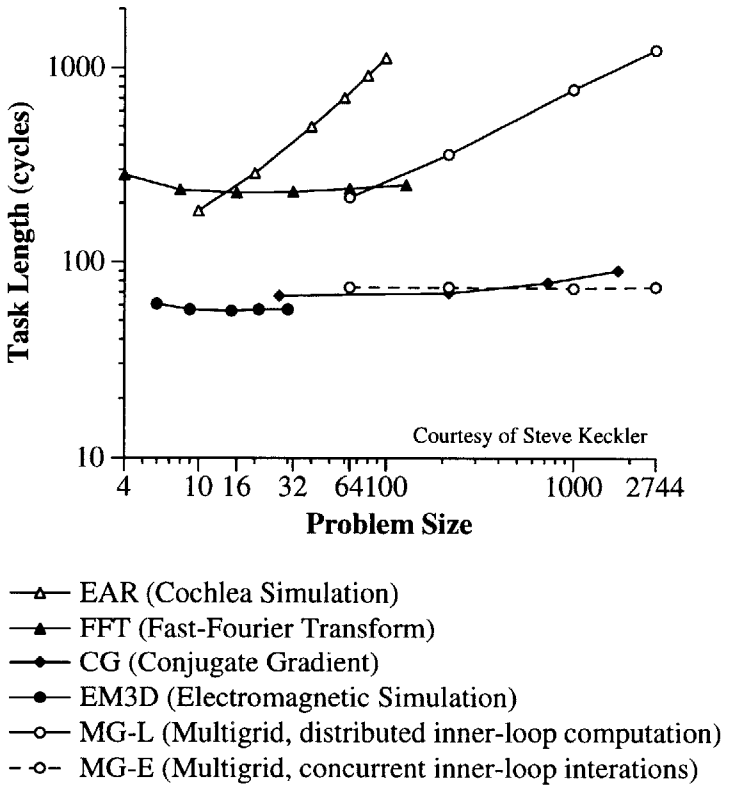

Figure 1-1: Fine Grain Parallelism Exists

Exploitation of fine-grain parallelism is the key to high performance computing, while protection is a basic requirement in many of today's applications. Rather than growing the problem sizes to amortize the large overhead, fine-grain computing promises faster and more detailed solutions to existing, fixed-size problems. This ability to extract performance at small granularities is in turn central to the success of large-scale multicomputing, which is fueled by the growth in VLSI density and

chip size, and the emergence of affordable integrated processor components [2, 3, 4, 5, 6, 7]. Such systems would not be economically viable if limited to exotic, huge-size problems.

Few existing architectures are able to take advantage of fine-grain parallelism, although a recent study [8] shows that it is readily available even in common benchmark programs. Figure 1-1 (reproduced from pp.113 [8], courtesy of Stephen Keckler) for example, shows that grain sizes as small as 70 cycles are accessible by parallelizing the applications at the inner-loop level. Nonetheless, most existing machines, burdened by communication overhead that ranges from many tens to many thousands of processor cycles, are too inefficient to take advantage of parallelism at this level. The few that are "lean-and-mean", such as [9, 10], lack robustness and lose their efficiency when software layers are used as a workaround. To effectively exploit fine-grain parallelism, an efficient *and* robust communication architecture is necessary.

1.2 Approach

This thesis focuses on the message interface. Conventional message interfaces – that take tens to thousands of cycles to send a message – present a bottleneck in the advent of advanced processors that are capable of generating multiple results on-chip every cycle (*e.g.* [1, 2, 3]), and high-speed signaling pads [11, 12] that are able to carry that data off-chip quickly. In terms of robustness, the message interface is also the weakest link in the communication system. As user programs share the resources such as message buffers, care must be taken to shield them from one another. The challenge is to minimize overhead, without compromising robustness.

Message interface overhead comes from three sources: (a) operating system involvement (b) software protocols and workarounds and (c) ad hoc interface mechanism choices. To avoid operating system overhead, the M-Machine message interface consists of user-level mechanisms. Software workarounds are minimized by making higher level communication needs an inherent design consideration. Primitive mechanisms are carefully chosen to complement one another, thereby avoiding redundancy

and loopholes in their functionality.

The M-Machine message interface is register-mapped. The user thread assembles a message in its regular register file, and launches it atomically into the network with a `SEND` instruction. To eliminate address translation overhead, the remote object to be referenced is specified with a virtual address, *destination*, which is translated into routing information by a small hardware translation table called the *Global Translation Lookaside Buffer* (GTLB). The `SEND` instruction also indicates a `HandlerIP` that directly specifies a message handler to be invoked at the destination processor. The hardware requires both *destination* and `HandlerIP` to be unforgeable *guarded pointers*, a lightweight capability mechanism [13]. As a result, the system has complete control over the remote data as well as the remote operations accessible to each user program.

The register-mapped message composition buffer eliminates buffer sharing, while the atomic injection interface allows the network port to be assigned to each user for only a bounded period of time. Therefore no user thread can cause starvation by holding on to the injection interface for prolonged periods. At the destination end, for flexibility and efficiency, the programmer may supply an optimized handler for each message. The guarded-pointer protected `HandlerIP` however enables the system to confine users to invoking verified *trusted handlers*, which always release the critical resources promptly. Verification may be done conceivably through human inspection or compiler analysis.

A streaming extraction interface is used so that a message can be dispatched upon arrival of the first word. Message handlers are invoked in a reserved hardware thread slot. Two registers known as `MsgHead` and `MsgBody` in the reserved slot are mapped to the incoming message queue. Reading `MsgBody` causes the next word in the *current* message to be popped and returned, while reading `MsgHead` causes any remnants from the current message to be flushed, and the `HandlerIP` from the *next* message to be popped and returned. This enables a message to be dispatched in a 3-cycle `jump` delay after the arrival of its `HandlerIP`. The flush mechanism associated with `MsgHead` can also be used to quickly return the message queue to a known good state after an error

is detected. In addition, `MsgBody` is automatically padded with null values when the end of the message is reached, to guard against malformed messages. To avoid wasting execution resources when the handler thread slot is idle, the availability of the `MsgHead` and `MsgBody` data is connected into the register scoreboard, which is consulted by the instruction-issue logic.

To efficiently support higher level communication services, the M-Machine message system provides user-selectable in-order delivery and a flow-control counter. When sending each message, the user may choose in-order delivery as needed by the algorithm, or allow out-of-order delivery and enjoy the full performance benefits of the four virtual channels in the network. The flow-control counter facilitates a simple *return-to-sender* protocol, by tracking the number of unacknowledged messages injected by each node, and disabling user-level message injection on the node when a pre-determined value is reached. In this system, each originating message is either consumed and acknowledged by the receiver, or bounced back to the sender. Nominally, the pre-determined value corresponds to the number of *bounce buffers* pre-allocated on the node, so that storage space is guaranteed for every bounced message. This scheme provides for efficient end-to-end flow control without explicitly allocating a buffer during each message injection.

To evaluate the tradeoffs in primitive mechanism choices, the design space is considered in three aspects: (a) mapping (b) atomicity, and (c) dispatch. The mapping of an interface defines how the messaging facility is presented to the software. Register-mapped mechanisms[2] are compared against memory-mapped interfaces, where messaging operations are performed by reading and writing reserved memory addresses. Atomicity refers to whether messages are buffered and transfered whole in uninterruptible blocks, or moved piece-wise in a streaming fashion as each word becomes available. The dispatch mechanism determines how the destination processor reacts to a incoming message. The dedicated thread slot mechanism in the M-Machine is contrasted against conventional interrupt-driven and polling interfaces. An explicit

[2]While there also exist instruction-mapped interfaces such as [10] where the instruction set contains specific messaging operations, I consider them only as a variant of register-mapped interfaces due to the similarity of their overhead characteristics.

context swap is required to dispatch each message handler in the last two cases.

An analytical model that relates application performance to message Latency, processor Occupancy and Grain size – the *LOG model* – is developed in this thesis to further the understanding of architectural tradeoffs. A simple program, Conway's Game of Life [14] (~40 cycles grain size) is used to demonstrate its construction. The impact of communication overhead on fine-grain computing is then measured from the model. While the effects of latency is partially masked by slack in the application, processor occupancy is found to be generally unmaskable, and is therefore the critical impediment to fine-grain performance.

1.3 Thesis Overview

The rest of this thesis is organized into seven chapters. To put this work in context, I briefly review some existing message system architectures, and discuss some related work in the next chapter. The MIT M-Machine multicomputer, its Multi-ALU Processor (MAP) chip, and the architecture and implementation of its message system are then described in Chapter 3. In this system, the mapping of message composition buffers into user register files results in a modular design that scales naturally with the number of threads of control incorporated in the processor chip. Further details concerning the microarchitecture and implementation of the M-Machine network interfaces are discussed in Chapter 4.

Chapter 5 details the complementary starvation avoidance and protection schemes in the M-Machine. The system is equipped to regulate both the data objects and remote handlers accessible to each user thread. The primitive design choices are evaluated in Chapter 6. Taken together, the M-Machine messaging mechanisms are found to provide an order of magnitude improvement over traditional interrupt-driven, memory-mapped, atomic interfaces. The dedicated-thread dispatch mechanism accounts for 60% of the improvement. Another 30% of the overhead reduction is due to the register-based mapping and atomicity choices in the M-Machine, while the GTLB provides the remaining 10% of the speedup. A round-trip message to the

nearest-neighbor processor takes only 38 cycles in the M-Machine.

In Chapter 7, the LOG model is constructed. Results from the sample application (LIFE) suggest that occupancy $\ll 50$ cycles – a very steep performance curve shows continuously increasing speedup as occupancy approaches zero – and latency < 200 cycles are crucial, enabling factors for efficient, large-scale parallelization. With such low overhead, massive multicomputing is shown to be capable of delivering very-high performance as well as efficiency. With the M-Machine mechanisms, 50% efficiency is easily achievable for a 16K-cell LIFE problem distributed over 1000s of processors. Employing one processor for each LIFE cell, a speedup of more than $5000\times$ can also be achieved if raw performance, rather than efficiency, is the primary concern.

Finally, Chapter 8 summarizes the study and the findings. I examine the drawbacks, explore the potential impacts and applications of the proposed design, and discuss several future research opportunities. In particular, while the LIFE example provides valuable insights into the behavior of fine-grain applications, generalized conclusions must not be drawn from its specific results. For a more thorough understanding of the general tradeoffs, many more LOG models for other common applications must be studied. The extremely-low overhead and solid robustness of the M-Machine mechanisms on the other hand are very well suited for the upcoming generation of highly integrated, massive multicomputers. A potential extension to the basic design for addressing the higher bandwidth requirement between the processor chip and the network fabric in such systems is also discussed.

1.4 Contributions

The major contributions of this research are :

- Identifying the robustness problems in existing designs and proposing a simple solution,
- Quantifying the overhead in the primitive mechanisms design space.

- Developing a *LOG* model for better understanding of the architectural issues in fine-grain computing,
- Demonstrating the viability and potential of fine-grain multicomputing,
- Presenting the M-Machine architecture as proof of concept and implementation example.

Chapter 2

Background

Traditional message interfaces do not have the efficiency required to support fine-grain computing. In older machines, the operating system (OS) serves as a intermediate layer between user programs and the communications hardware, and messaging operations are performed by trapping into system code. High overhead results from this heavy OS involvement, forcing the programmer to rely on various software techniques, such as grain-packing, unrolling, and prefetching to amortize the communication cost over larger grain sizes and messages. While latter designs have opted for more efficient user-level mechanisms, the inadequate robustness in many of these architectures ends up defeating the carefully tuned hardware by necessitating cumbersome software layers. The mapping and atomicity choices also affect performance. Memory-mapped interfaces force their messages to traverse the memory hierarchy before entering the network, and are generally less efficient compared to integrated messaging mechanisms that have a direct path. Streaming interfaces are faster than atomic ones, as the message head can *worm-hole* through the system without waiting for its tail. While the protection issue gained more specific attention in recent studies [15, 16, 17, 18]), careful treatment for the starvation problems, particularly those due to erroneous or malicious message handlers and exhaustion of physical-memory based message buffers, is still scarce.

2.1 Past and Present Architectures

To illustrate the prevalence of high communication overhead, the cost of a round-trip message in a number of multicomputers from the past and present are shown in Figure 2-1. The round trip cost, further explained in Table 2.1, is roughly based on a two-way null remote procedure call (RPC), or a *ping-pong* operation. It is obtained by doubling the reported value when only the one-way cost is provided in the literature [19, 20, 21, 15, 22, 23, 24, 25, 26, 27]. Since an actual implementation for [18] does not exist, the round trip cost is extrapolated from the specified overhead of assembling, sending and receiving a remote read message [1].

On the horizontal axis, Figure 2-1 also shows that the systems employ a variety of mechanisms for robustness. Protection in the network interface is enforced by limiting the user's accessibility to the messaging resources, or by explicitly matching up senders with legitimate receivers. The former category performs permission checking through the operating system or virtual-memory translation layers, while the latter uses gang-scheduling, process-identifier matching, and logically-insulated communication channels to ensure that messages are presented only to their intended receivers. Lacking a specific protection mechanism, the J-Machine [10] relies on the program's good behavior to avoid protection violations. For comparison, the M-Machine message system is also shown in Figure 2-1. For robustness, messages are injected atomically from dedicated message buffers, and trusted message handlers are enforced through guarded-pointers [13].

The high cost of operating system involvement in the message interface is evident in the original Intel iPSC/2, where 85% of the communication time for a short message is spent in software overhead [19]. Context switches between user and system mode alone account for 18% of that overhead. The overall latency is however reduced by almost 7× when a co-processor is added to relieve the operating system [19]. More recently, the Shrimp [25] system uses a user-level direct memory access (UDMA)

[1] Fifteen cycles are added to the total as an estimate for the network latency and dispatch overhead.

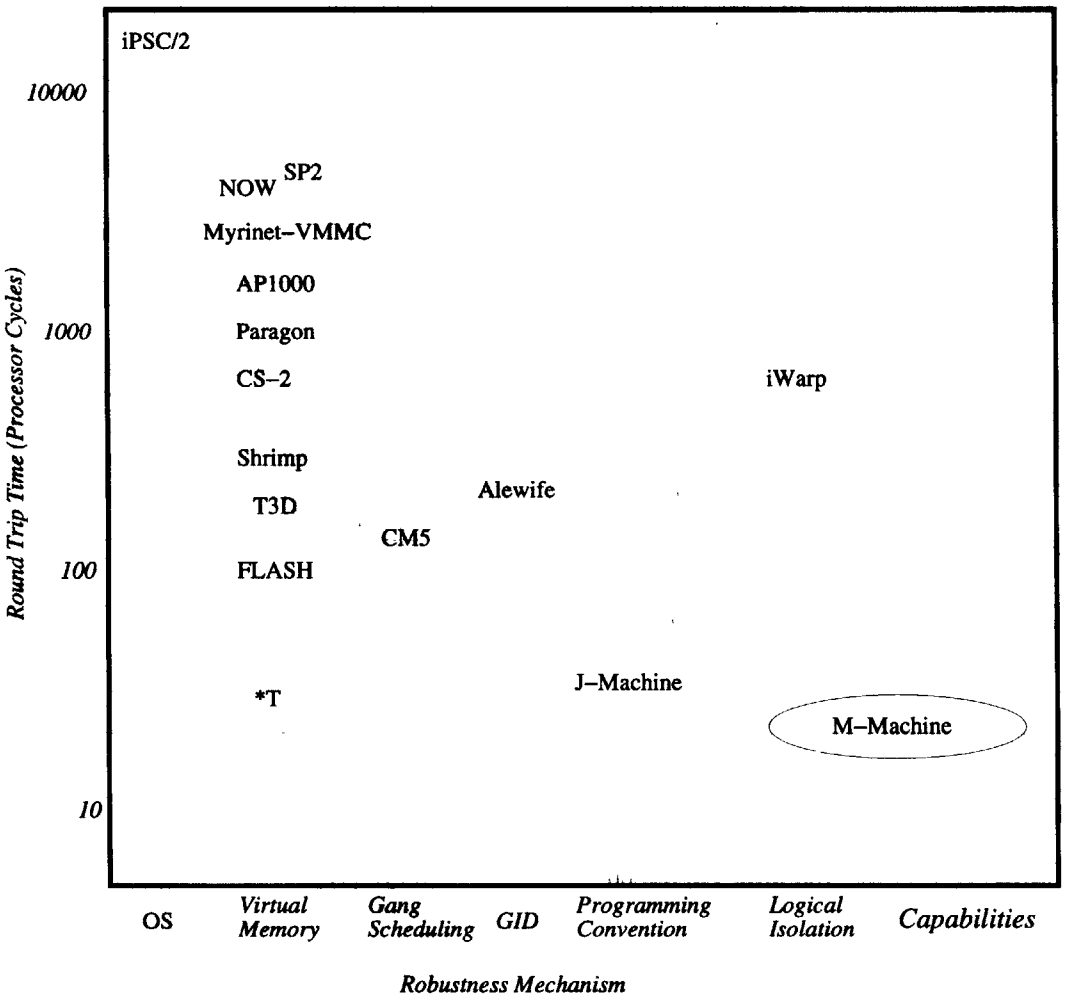

Figure 2-1: Past and Present Multicomputers

System	Processors	Round Trip	Remarks/Features/Issues
iPSC/2	20 Mhz 80386	710 μS[19] 100 byte message	85% sofware overhead (18% from context switches)
		110 μS[19] 100 byte message	with MPC (Communication Co-Processor)
CM-5	32 Mhz SPARC	143μS swap[9]	CMMD library
		3.4μS swap[9]	CMNF library
J-Machine	12.5 Mhz MDP 1024 max	43 cyc[10] round-trip null RPC	Streaming Injection
CS-2	40 Mhz SPARC	20μS[39]	Channel
		24.6μS[23] active message	DMA w/ Hardware Table-Lookup
		174μS[21] ping pong	PARMACS macros
		206μS[20] mesg exhange	mpsc library
T3D	150 Mhz 21064 2048 max	600nS[26] remote read	Shared Memory
		2.76μS [40] Fast Messages 16-byte Fetch and Increment	F&I Specific Hardware Support
		$\sim$ 120μS[26] User-Level Message	Interrupt-Driven Message Handler
*T	88100MP	dispatch + $\sim$ 20 cyc remote load [18]	microthreading
NOW	HP9000/735 125Mhz PA-RISC 7150	50μS [24] sockets on Active Message	LAN based cluster of workstations
Paragon	50 Mhz i860 1024 max	$\sim$20μS[38] Round Trip	Active Message LogP analysis
		116μS[20] mesg exhange	NX library
SP2	66.7 Mhz RS/6000	96μS[20] mesg exhange	MPI-F library
SHRIMP	60 Mhz Pentium	9.5μS [25]	User-Level DMA w/ Automatic Update
FLASH	100 Mhz T5/R4000	100 cyc [17] remote read	Shared Memory
		175 cyc [41] fetch-and-add	Active Message
AP1000	25 Mhz SPARC	65.6μS[22] ping pong	Line Sending & Buffering Receiving
Alewife	33 Mhz SPARCLE	14.8μS[15] round-trip null RPC	GID
Myrinet VMMC	166 Mhz Pentium	19.6μS[42] ping pong	LAN-based Multicomputer
iWarp	20 Mhz	$\sim$800 cycles [27] send-receive	Message Passing using FX Deposit Model

Table 2.1: Round Trip Communication Cost in Past and Present Multicomputers

mechanism that removes the operating system from the critical path when sending a message. However, a costly system-call is still required to setup the UDMA buffer mappings for each sender-receiver pair. As the UDMA system requires locking down physical memory pages at the destination end to receive asynchronous messages, a starvation risk is also created by the exhaustion of physical memory.

At the other end of the spectrum, the Thinking Machines CM-5 [28] and the MIT J-Machine [10] feature fast but unprotected message interfaces. To prevent a user program from intercepting messages destined for others, the CM-5 relies on a high-overhead *gang scheduling* technique, where the OS ensures that only threads from the same job are scheduled to run within each time-slice on all processors. The J-Machine features a streaming injection interface, to which a user thread may retain access indefinitely, thereby causing starvation to the others. It is up to the programmer to refrain from monopolizing the messaging resources. The same risk is present in the iWarp system, where messages are delivered through logical pathways. As each pathway cannot be reclaimed until it is voluntarily released by the user, programs are vulnerable to starvation.

The AP1000 [22] maps the message composition buffer into its data cache. Messages are launched into the network using the cache-line flush mechanism. Mukherjee *et.al.* [29] take the caching approach further and show that the performance of memory-mapped messaging interfaces can be improved by up to 88% by exploiting the cache-coherent mechanisms in the processor, and caching the network device registers as well as the message queues. The performance of these memory-mapped systems however are ultimately constrained by the memory hierarchy baggage. On the other hand, by integrating the message interface into a set of special registers, the *T [18] architecture provides an efficient interface optimized for short messages. Each message however has to be explicitly copied into and out of these registers during injection and reception. This copying overhead may be eliminated by making the message registers directly accessible to the entire instruction set, as in the M-Machine.

A global virtual address is used to specify the destination in a *T message, which goes through a translation layer that prevents the user from sending messages to

nodes not mapped into its protection domain. Similar mechanisms are also employed in other virtual-memory mapped interfaces, such as the FLASH and Shrimp systems. In *T, each message dispatches into a *microthread* at the destination. The successful dispatch of a new message however hinges on its handler being scheduled by the previously-executing microthread upon the latter's termination. Nonetheless, being user-level programs, microthreads do not necessarily behave in a cooperative way. Instead of scheduling the handler for the next message, a microthread may block and cause a deadlock, or access the message itself and violate the protection model. This problem is common across most implementations of handler-based interface that uses a user-specified handler to receive each message. In FLASH, the problem is avoided by renouncing support for user-level handlers.

Taking a different approach, the Alewife [15] messaging hardware stamps each message with the sender's process group identifier (GID). At the destination, the message is presented to the currently-running thread only if their process IDs match up. Otherwise the message is handed to the operating system, which is notified via an interrupt. To avoid starvation due to interrupt-masking, the technique is further extended with a *revocable interrupt-disable* mechanism in the FUGU architecture [30]. Nonetheless, good performance in this approach relies on an GID-match being the common case, making it suited mostly for a gang-scheduled machine.

Note also that software libraries are written for several systems in Table 2.1 to provide a standardized messaging interface, such as MPI [31] and NX [32], to the user. While it enhances program portability, this approach sacrifices efficiency for generality. In the Meiko CS-2 for example, a 10× degradation is observed when an `mpsc` library is layered on top of its primitive mechanisms. For high performance computing, the compiler needs to target the native communication interface itself. Software layers are also needed if the underlying architecture does not provide sufficiently flexible properties to meet common higher level communication needs. In particular, Schoinas and Hill [33] show the need for a flexible address translation mechanism, while Karamcheti and Chien [34] discover that 50%–70% of the software overhead can be eliminated by providing efficient primitive mechanisms for buffer

management, flow-control, and in-order, lossless delivery. To take full advantage of fine-grain computing, the message system has to be designed from the ground up from low-overhead mechanisms which work together to constitute a robust system that efficiently facilitates these high-level requirements.

2.2 Related Work

The M-Machine message interface architecture is influenced by a number of related research efforts, especially its predecessor, the MIT J-Machine [10]. The J-Machine has two thread slots on the processor chip, each of which is connected to an incoming message queue. Because both thread slots are fully accessible to unprotected user-level code, the J-Machine is susceptible to robustness problems, despite having one of the fastest message system in existence. The M-Machine inherits its predecessor's threaded fast dispatch mechanism, but avoids the robustness problems by reserving the critical message-handling thread slots for trusted programs. The message handler based reception interface is also promoted by Eicken *et.al.* through their work on the Active Message [35] model. Now in its second revision, Active Message systems have similar non-blocking, non-faulting and quick-completion requirements on their handlers as the *trusted handlers* on the M-Machine. However, while the guarded pointer facility in the M-Machine serves as a enforcement mechanism for these requirements, most Active Message implementations do not have comparable provisions. Instead, they must rely on cooperative behaviors in the software.

The register-mapped interface is inspired by Henry and Joerg [36], who describe an interface that *scrolls* a register-window's worth of message words into or out of the network. Variable-size messages are supported by way of successively scrolling the message words in or out, one window at a time. In contrast, for starvation avoidance, M-Machine messages are bounded-size, and message injection is made atomic so that no incomplete message can occupy a network channel indefinitely. Recognizing trusted handlers as starvation-free programs, and that most message

words are used exactly once by the handler [2], the M-Machine adopts a more efficient streaming extraction interface that is optimized to automatically remove each word from the message queue as it is consumed. Although it does not provide random access to the message body through a window as in [36], no negative effect is expected as most instructions cannot simultaneously use more than two operands anyway, while the arrival-order of message words can be re-arranged so that those needed first also arrive first.

The M-Machine network employs four virtual channels [37] for performance and deadlock avoidance. Depending on the channel arbitration policy, virtual channel systems usually do not guarantee ordered delivery. In the M-Machine implementation, the channel selection logic is augmented to support in-order messaging by forcing all messages tagged for ordered-delivery through one fixed channel [3].

The *LOG* analysis closely resembles the LogP model developed by Culler *et. al.* [38]. The latter characterizes the capability of multicomputers in terms of their communication system constraints – latency, occupancy, gap and available processors – and has more of a programmer's perspective. In this thesis, I am concerned with understanding the tradeoffs in defining a new architecture. Hence, the LOG model adopts an architect-centric view.

[2]For example, when processing a simple remote-write message, the handler needs the the data as well as the the memory address only once.

[3]A distinct in-order channel is used for each message priority.

Chapter 3

M-Machine Messaging Mechanisms

The M-Machine message system accomplishes high efficiency by exposing the interface to user programs and eliminating unnecessary copying of the messages. The interface is mapped into registers that are accessible directly as operands to regular instruction. In order to reduce message creation overhead, the translation of a virtual address into routing information is performed transparently by a small, on-chip hardware cache. A simple flow control mechanism is incorporated to help regulate the network traffic. For communication protocols where message ordering is crucial, the system supports optional in-order delivery between processor pairs. The system also particularly addresses the issues of protection and starvation avoidance.

A brief overview of the M-Machine and its multi-ALU processor chip is first presented in the next section. A more detail description of the overall M-Machine architecture can be found in [1]. The message interface architecture is described in Section 3.2. In Section 3.3, I discuss the address translation, flow-control and in-order delivery features. The prototype microarchitecture and implementation details are presented in the next chapter, while the discussion of protection and starvation issues are delayed until Chapter 5.

3.1 The MIT M-Machine

The MIT M-Machine [1] is designed to explore the architectural issues in high-performance, fine-grain multicomputing. It employs the *processor-coupling* technique [48] to efficiently exploit instruction level parallelism, and incorporates multithreading to improve resource utilization. The M-Machine is constructed of an array of up to 1024 Multi-ALU Processor (MAP) nodes, each of which contains three loosely-coupled execution clusters. Five independent *thread slots* are incorporated into each cluster. In every cycle, each cluster independently multiplexes one of its thread slots onto its function units, which consist of a floating-point ALU, an integer ALU, and a memory ALU. Nominally, the thread slots on the three clusters are grouped into 5 *v-threads*, each of which is assigned to the same problem, as shown in Figure 3-1. However, depending on its resource requirements and inherent grain size, a program may be distributed across the clusters, v-threads, and/or MAP chips.

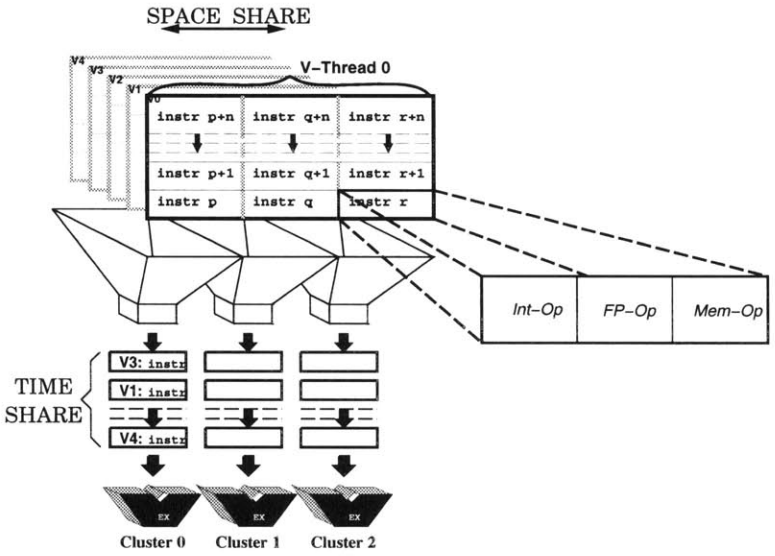

Figure 3-1: Multithreaded, Loosely-Couple Clusters

MAP chips communicate with one another through an integrated messaging network, while thread slots on the same chip communicate through local memory. Within each v-thread, the instructions streams on each cluster can communicate by writing

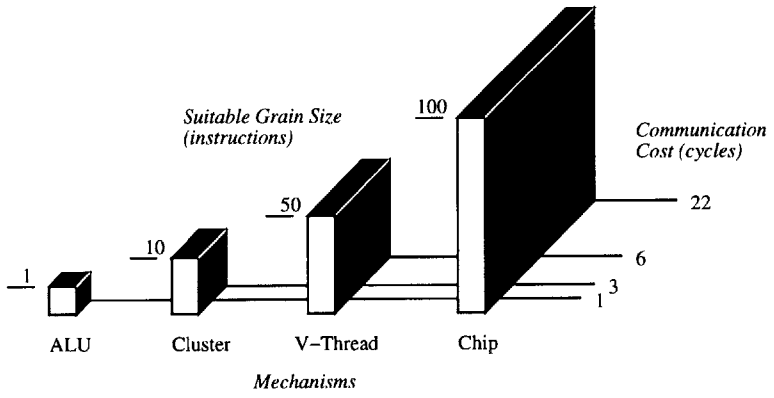

Figure 3-2: Exploiting All Levels of Parallelism

into each other's independent register files, while the ALUs on the same cluster simply share their register files [1]. The communication cost associated with each of the mechanisms, assuming a cache-hit in all cases, is recorded in Figure 3-2. A register written by an ALU is available to another ALU in the same cluster in the immediately following cycle. Writing within the same v-thread across clusters takes 3 cycles, and a producer-write followed by consumer-read to the local memory takes 6 cycles. A 19-cycle latency is incurred for a datum to be written into remote memory via a message while the consumer then takes another 3 cycles to read the datum. These very low and very gradually increasing communication costs enable the programmer to efficiently exploit parallelism at all levels, and transition smoothly across the granularities to take full advantage of the available resources.

3.1.1 The MAP Chip

A block diagram for the MAP chip, containing three execution clusters, a two-banked, unified cache, and an external memory interface is found in Figure 3-3. The network interface units and a two-dimensional router are also integrated into the chip. These components are interconnected by two crossbars, the M-Switch and the C-Switch.

[1]Since each cluster is multithreaded, five sets of registers files are incorporated into each cluster. Each 3-ALU instruction stream on the cluster can only access the register files – floating-point and integer – associated with the stream's own thread slot.

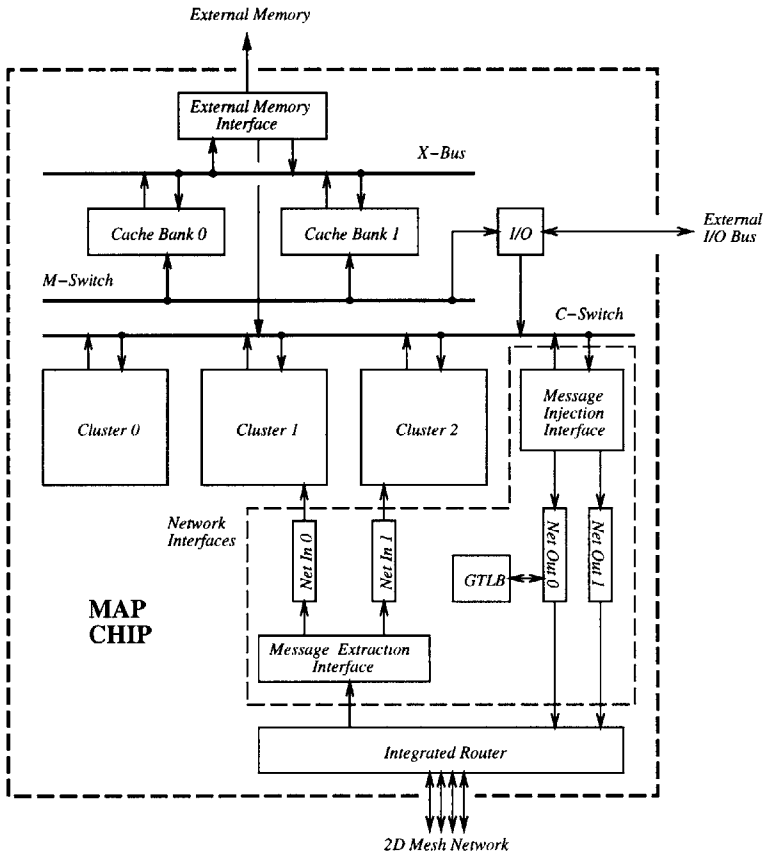

Figure 3-3: MAP Chip

Clusters make memory requests to the appropriate bank of the interleaved cache over the 3 × 2 M–Switch. The 7 × 3 C–Switch provides inter-cluster communication, returns data from the memory system [49], and connects the clusters to two outgoing message queues. Each cluster may transmit on the M-Switch and receive on the C-Switch one request per cycle. Each of the three clusters is a 64-bit, three-issue, pipelined processor with two integer ALUs – one augmented to execute memory instruction – a floating–point ALU, five sets of register files, and a 4KB instruction cache [2].

Each cluster implements cycle-by-cycle multithreading, with the register files and

[2]In the silicon implementation of the MAP architecture, only cluster 0 has a floating–point unit, due to chip area constraints. The simulation studies performed in this thesis include floating–point units for each of the three clusters.

pipeline registers at the top stages of the pipeline replicated for five independent *thread slots*. Each thread has 14 integer registers, 15 floating–point registers, and also 16 condition code (CC) registers for boolean values. Instructions from the threads are interleaved over the execution units on a cycle-by-cycle basis, with no pipeline stalls when switching between threads. A synchronization pipeline stage [8] selects the thread to issue based upon resource availability and data dependency, using a score-board to keep track of the validity of each register. An instruction is not considered for issue unless all of the resources needed are available, including the validity of its operand registers.

3.2 M-Machine Message Interface Architecture

The M-Machine message interfaces, illustrated logically in Figure 3-4, are completely mapped into the processor's general register name space. A buffered, atomic injection interface is paired with a streaming extraction interface. Messages are dispatched asynchronously upon arrival within a *jump* delay (3 cycles). Two message priorities (P0 and P1) are supported. Canonically, to assist in deadlock avoidance, the P0 logical network is used for originating request messages, while P1 is used for reply messages. The architecture is first described in abstract below. The details are included in the later parts of this chapter and the next chapter.

3.2.1 Injection

To send a message, a user thread first assembles the message body, up to 10 words in length, in either its integer or floating-point register-files, starting at register $i4$ or $f4$, respectively (Figure 3-4). A non-blocking *SEND* instruction then atomically injects the message into the network:

```
SEND <length>, <DestAddr>, <HandlerIP>, <Ack>
```

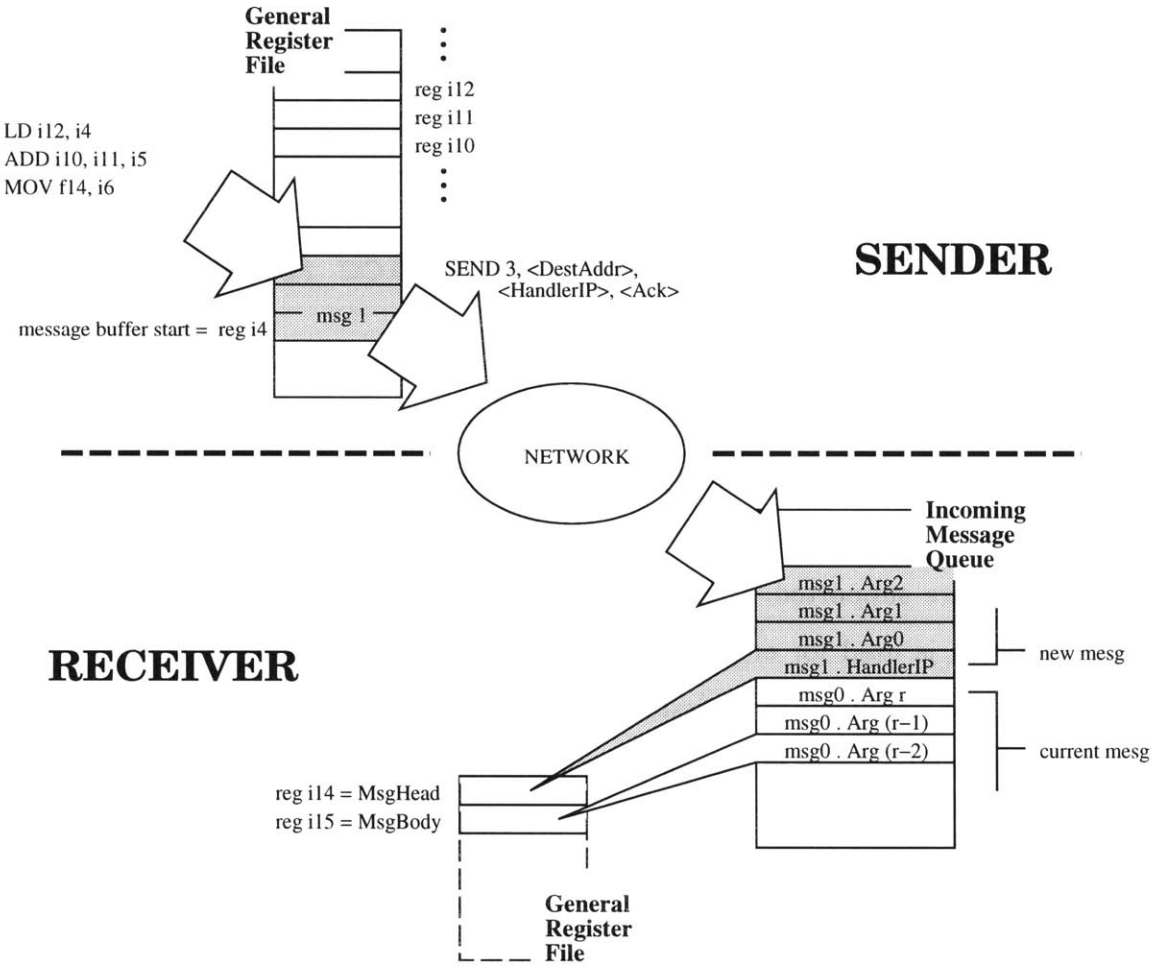

Figure 3-4: M-Machine Message Interfaces

A virtual memory pointer, `DestAddr`, specifies the destination. During injection, a small hardware cache, known as the *global translation look-aside buffer* (GTLB), translates `DestAddr` into physical routing information. The action at the receiving end is specified by `HandlerIP`, which is an instruction pointer to a *message handler* routine. The M-Machine requires `DestAddr` and `HandlerIP` to be unforgeable pointers [13], and aborts the SEND instruction with a protection violation exception if either is found to be invalid. `Ack` specifies a condition (CC) register to be validated after the network controller has retrieved the message words from the register file. As soon as the SEND instruction is issued, the program can thus proceed with further computation, as long as it avoids contaminating the message registers or getting swapped out before `Ack` is validated.

3.2.2 Extraction

The M-Machine reserves two thread slots for message reception, one each on cluster 1 and cluster 2, for P0 and P1 respectively. Integer registers *i14* (`MsgHead`) and *i15* (`MsgBody`) in each of these slots are mapped to its corresponding incoming message queue (Figure 3-4). Whenever `MsgHead` is read, the network hardware returns the `HandlerIP` of the *next* message, discarding any remaining words from the *current* message. Reading `MsgBody` returns the next word in the *current* message instead. In either case, the returned word is also removed from the queue. Thus, a sequence of reads to `MsgBody` returns the subsequent words in a message. After the tail of a message is consumed, the network interface unit pads further reads to `MsgBody` with a dummy value, until the next message is scrolled in by a read to `MsgHead`. Both `MsgHead` and `MsgBody` can be used directly as the source operand in any regular instruction.

3.2.3 Dispatch

For `MsgHead` and `MsgBody`, the corresponding register scoreboard presence bits are mapped to the presence of a new message and the availability of the next word

in the current message, respectively. Consequently, an instruction sourcing these registers does not issue until the corresponding message word is available. This allows a *message dispatcher* installed in the reserved thread slot to wait for message arrival without consuming any execution resource, yet remain able to activate immediately when the first message word arrives.

An Example: Fetch And Add An example, which performs a *Fetch and Add* operation, will be used to illustrate the simplicity of the M-Machine message interfaces. In this example, a value is to be added into a remote memory location. At the originating end, the sender computes the remote address and the addend into its register file, and then injects them into the network. Notice the simultaneous use of the memory unit (MEMU) and the integer unit (IALU) for generating two results in one cycle. The presence bit for the condition register `cc0` is cleared – set to *empty* – when the `SEND` instruction issues, blocking the instruction that accesses `cc0` until the presence bit is set to *full* again, by the network interface hardware when the message has been extracted from the register file.

```
/* Sender computes F&A Address into      */
/* register i4, and F&A Value into i5    */

_call_FetchAdd:

    instr MEMU lea i10, #16, i4      IALU sub i11, i12, i5;

/* Inject two words.                     */
/* Remote address in i4 automatically    */
/* translated into routing info by GTLB. */
/* _fetchAdd HandlerIP assume present in */
/* some register.                        */

    instr IALU send 2 i4, _fetchAdd, cc0;

/* Sender may proceed with further       */
/* computation, but must sync on cc0     */
/* before reusing i14 and i5             */

    ....
```

```
instr IALU ct cc0 and i13, i9, i4;

....
```

At the destination, the message dispatcher waits within the reserved thread slot for message arrival. Since the absence of a new message causes the presence bit of the `MsgHead` register to be marked empty, the dispatcher thread does not compete for execution resources until a new message arrives. However, upon message arrival, it immediately jumps to the code pointed to by the `HandlerIP` in `MsgHead`, which is `_fetchAdd` in this example:

```
/* Dispatcher jumps to HandlerIP */

_dispatch:

    instr IALU jmp MsgHead;
```

A simple `_fetchAdd` handler is shown below. It first loads the original value from the specified location, while also saving away the address for later use. The latter step is necessary as the address is removed from the message queue when `MsgBody` is read. Note that in the M-Machine, IALU and MEMU operations scheduled for the same cycle are always also issued in the same cycle, and will get the same value from `MsgBody` in that situation.

The new value is then added into the original value in `_fetchAdd`, and the result is written back into memory. At this point, the handler is finished, and it branches back to the dispatcher to wait for the next message:

```
/* Load from memory, stash away address */

_fetchAdd:

    instr MEMU ld MsgBody, i10     IALU mov MsgBody i11;
```

```
/* Add in new value                          */

    instr IALU add MsgBody, i10, i10;

/* store back to memory, done.               */

    instr MEMU st i10, i11          IALU br _dispatch;
```

The entire fetch and add operation takes as few as 16 cycles to traverse the injection and extraction interfaces. Notice that copying is minimized due to the convenient mapping of the interface into the processor register space.

3.3 More SEND Instructions

The SEND instruction has been described in the abstract thus far. In the M-Machine implementation, a family of SEND instructions are included to provide optional features, including transparent address translation, flow control and order-preserving delivery. These instructions are listed in Table 3.1.

Instruction		In-Order	Flow-Control	Message	Physical Addressed
IALU	FALU	Delivery	(Throttling)	Priority	(Privileged)
isnd0	fsnd0	×	√	0	×
isnd0o	fsnd0o	√	√	0	×
isnd0p	fsnd0p	×	√	0	√
isnd0po	fsnd0po	√	√	0	√
isnd0pnt	fsnd0pnt	×	×	0	√
isnd0pnto	fsnd0pnto	√	×	0	√
isnd1pnt	fsnd1pnt	×	×	1	√
isnd1pnto	fsnd1pnto	√	×	1	√

Table 3.1: SEND Instructions

The isnd instructions are used in the integer unit, while the fsnd instructions are for the floating-point unit. Only the isnd0 and isnd0o instructions and their floating-point unit counterparts are accessible to user-level threads. While in-order delivery

is user-selectable, all user-level messages are subject to the flow-control mechanism, and must be virtually-addressed.

The remaining `SEND` instructions are restricted to system-level threads, and would cause a permission violation exception if their use is attempted by a user thread. These privileged `SEND` instructions, used for configuration and other system tasks, are physically-addressed – a physical node identifier is expected in the `DestAddr` argument. Physically-addressed messages are not passed through the GTLB for automatic translation during injection. While system threads may also use the virtually-addressed `isnd0`, `isnd0o`, `fsnd0`, and `fsnd0o` instructions, the system programmer must be careful to prevent circular dependencies, as priority-0 messages may be blocked, for example, while a GTLB-miss is being serviced by system routines.

Note that while P0 messages may be rejected by the receiver under the flow-control scheme, P1 messages are always consumed at the destination. Canonically, the priority-0 network is used for originating requests, and only acknowledgments and replies are sent as priority-1 messages. This guarantees that the network eventually drains even under congested conditions, and helps avoid deadlock conditions.

3.3.1 The GTLB

The GTLB [49] translates virtual addresses into physical node identifiers (NID), which take the form of absolute cartesian coordinates $[x, y]$ within the M-Machine's 2D mesh network. Each of x and y is a 5-bit unsigned integer, allowing up to 2^{10} nodes to be connected together. The translation is done implicitly for all messages injected via the `isnd0`, `isnd0o`, `fsnd0o` and `fsnd0o` instructions. A *GTLB-Miss* event is generated if the corresponding mapping is currently not cached in the GTLB. The `<Ack>` condition register specified in the offending `SEND` instruction [3] is not validated in this case, to indicate that the user must leave the message body intact in its register file. The event is resolved in software by an event handler, which installs the appropriate mapping into the GTLB and retrieves the message body from the user's

[3] Recall that the `SEND` instruction format is
`SEND <length>, <DestAddr>, <HandlerIP>, <Ack>`.

register file for retransmission. The `<Ack>` register is validated by the event handler after the message has been successfully injected. While the GTLB-Miss event is being thus serviced, all `isnd0o` and `fsnd0o` instructions are blocked on the node to ensure that order-preservation is not violated.

A virtual address may also be translated explicitly using the `IGPRB` instruction. A condition code is used to indicate whether the translation is successful. If a match is found, the translated NID and the matching GTLB-entry number are returned. The GTLB is organized as a fully-associative, content-addressable cache with four software-managed entries. Despite the small GTLB size, GTLB-Miss events are expected to be rare, due to an efficient encoding scheme capable of mapping a large address space in each entry.

Each GTLB entry contains five fields: *base*, *ext*, *tag*, *mask* and *ppn*. The *base* and *ext* (extent) fields delineate a rectangular, $2^{x_{ext}} \times 2^{y_{ext}}$ region of nodes within the network, with the origin at $[x_{base}, y_{base}]$ (Figure 3-5). The *tag* and *mask* fields specify a portion of the virtual address space to be mapped on to the selected nodes. An input virtual address is masked by *mask* and then compared against *tag* to determined if it falls within the region covered by the entry. The *ppn* (pages-per-node) then determines the actual distribution – the selected address space is interleaved, with wrap around, onto the nodes in the rectangle, 2^{ppn} pages [4] at a time. Figure 3-6 shows how the NID is finally computed for the processor hosting a given virtual address. Note that with this encoding, large regions of, or even the entire address space can

[4] Each memory page is 2^{11} bytes on the M-Machine.

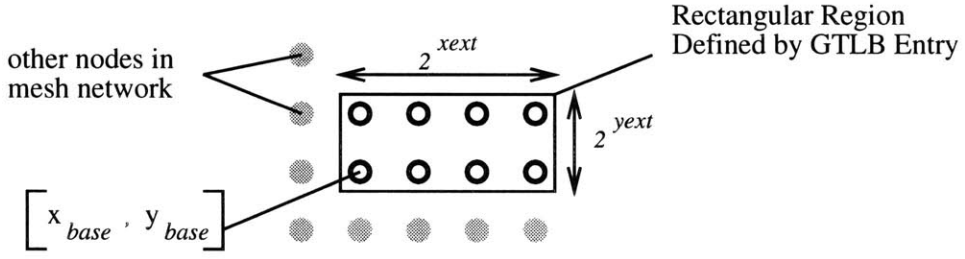

Figure 3-5: A Rectangular Region of Nodes Defined by a GTLB Entry

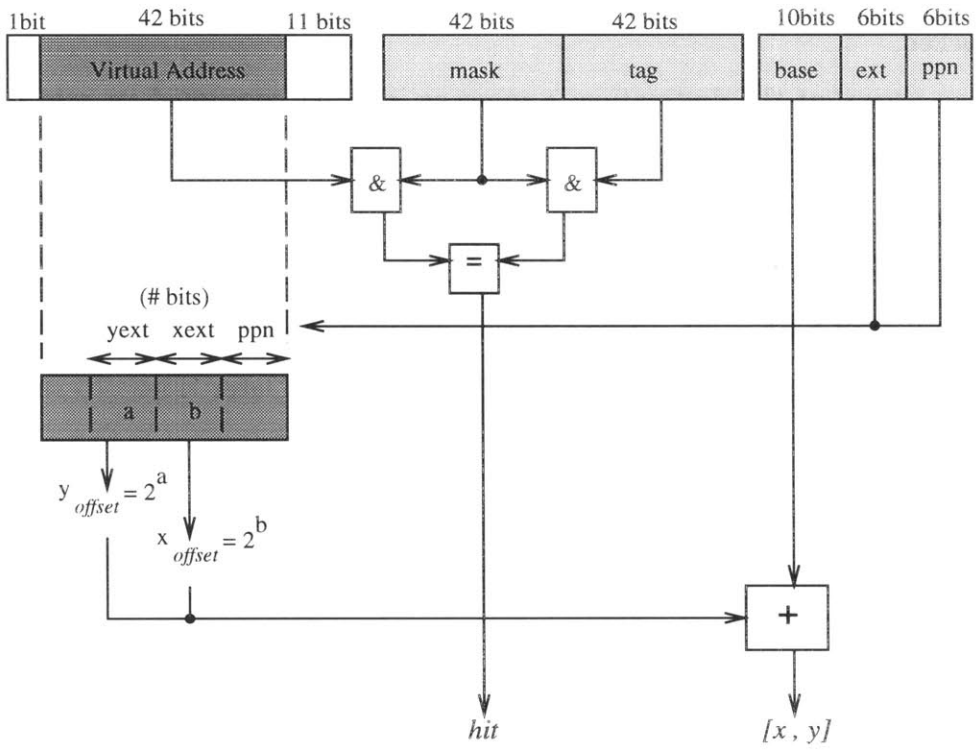

Figure 3-6: GTLB Translation

be mapped using a single entry.

3.3.2 Flow Control

A flow-control mechanism *throttles* message injection at the senders to prevent the receivers from being overwhelmed by incoming messages. In the M-Machine, a simple 10-bit counter in the P0 injection unit, known as the *outgoing message buffer counter* (OMBC), is used to implement a *return-to-sender* throttling protocol. In this scheme, a message is bounced back to the sender if the receiver is unable to process an incoming message immediately, due to local resource shortage, for example.

Nominally, the operating system reserves a number of *bounce buffers* in local memory on each node at initialization. The OMBC is initialized to the same number. Then, as each message – except those sent through a non-throttling `SEND` instruction – is injected, the OMBC on the sending node is automatically decremented by one.

When the counter reaches zero, SEND instructions, except the non-throttling variants, are blocked.

Upon arrival at the destination, a message is either consumed, or rejected if the destination node is unable to process nor buffer the message locally. If the message is consumed, the handler returns a result or acknowledge message to the original sender, which in response *increments* its counter. A rejected message on the other hand is bounced back in its entirety [5] to be deposited in a reserved bounce buffer on the sender node, pending retry or special handling at a later time. To simplify the design, message-bouncing and OMBC-increments are performed in software by the handlers. While not as transparent as hardware-only schemes such as that used in the Cray T3E [50], this approach affords more flexibility, to implement a hysteresis behavior, for example.

Effectively, the OMBC records the number of bounced messages that each node is able to absorb at any time. The simple accounting ensures that each injected message is backed by a bounce buffer, and guarantees the system's ability to eventually remove from the network every message it injects, without requiring the user to explicitly reserve a buffer when sending each message. To avoid double-counting, the result, acknowledge, and bounced messages are injected with non-throttling SEND instructions.

3.3.3 In-Order Delivery

The network router on the MAP chip features four [6] virtual channels [37] for each physical path (Figure 3-7). Depending on the channel allocation and switch arbitration policies, virtual channel routers do not usually preserve ordering – a message arriving later may overtake a previous message by being assigned to a channel that is faster-moving, *e.g.* because it happens to be carrying shorter messages. In the M-Machine however, the channel allocation policy is designed to provide user-selectable

[5] A special message up to 12 words in length is used in this case, to capture the original **HandlerIP** and **DestAddr** as well as the message body.

[6] To fit into the alloted chip area, only VC0 and VC1 are implemented in the prototype.

in-order delivery. The programmer may send certain messages in-order to simplify particular protocols and algorithms, while allowing the others to be delivered out-of-order to take advantage of the virtual channels for better performance.

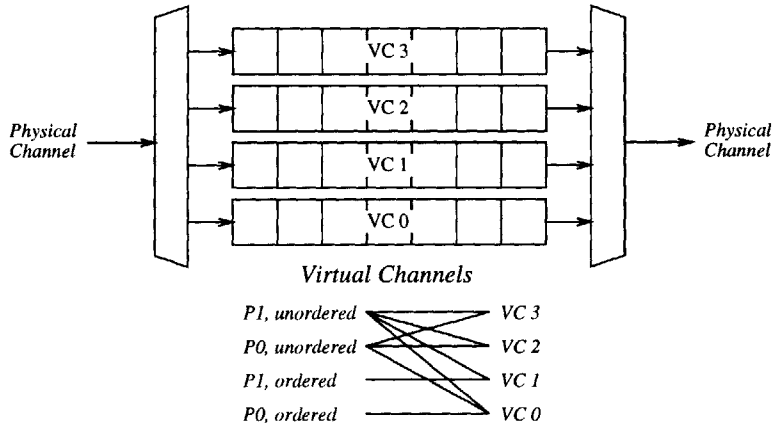

Figure 3-7: Virtual Channels

To avoid deadlocks, the VC1 virtual channel is reserved for priority-1 messages, so that there always exists a forward-progress path for P1 messages that is independent of P0 traffic. Under normal conditions, *non-ordered* P0 and P1 messages may use any of VC0, VC2 and VC3 whenever they are available, to maximize throughput. In addition, unordered P1 messages may also be routed on VC1. However, *ordered* P0 and *ordered* P1 messages are respectively routed strictly through VC0 and VC1 only. As a result, messages sent with the in-order delivery flag set from any particular node to any particular destination are guaranteed to arrive in the same order as their injection. To avoid overly complicating the design, in-order delivery nonetheless applies only to messages of the same type, *i.e.* no ordering is guaranteed between virtually-addressed and physically-addressed messages, nor between P0 and P1 messages.

The channel allocation decision is determined from the availability of channels downstream, and the type of message (*ordered*, *priority*) being routed. To simplify the allocation decision logic, the following default assignments, statically built into the router, are used in the absence of contention:

Sink VC number	*Default Assignment*	*Alternate Dynamic Assignment*
VC0	P0 Ordered	P1 Un-Ordered, or P0 Un-Ordered (decreasing preference)
VC1	P1 Ordered	P1 Un-Ordered
VC2	P0 Un-Ordered	P1 Un-Ordered
VC3	P1 Un-Ordered	P0 Un-Ordered

These default assignments are used whenever possible in the allocation process. If, and only if, no requests exist for a particular channel, that channel may be granted to an alternate request, whose default channel is currently not available. When choosing such an alternate assignment, the following preference orderings are used:

	P1 Un-Ordered	*P0 Un-Ordered*
↓	VC3 (default)	VC2 (default)
Decreasing	VC2 (yields to P0 Un-Ordered)	VC3 (yields to P1 Un-Ordered)
Preference	VC1	
	VC0	VC0 (yields to P1 Un-Ordered)

Finally, if a P1-unordered message fails to make progress for an excessive period of time – nominally 16 successive loses to P1-ordered messages – the default preference for VC1 is switched from P1-ordered to P1-unordered messages. This simple escape mechanism prevents a P1-unordered message from being blocked indefinitely in a router. Such a pathological scenario occurs when *all* of VC0/VC2/VC3 downstream are occupied by P0 messages that are also indefinitely stuck, *and* there is an *incessant* stream of new P1-ordered messages which always preempts the waiting P1-unordered message [7] from using the downstream VC1.

The default channel assignments guarantees that no one type of messages can be indefinitely starved out, even under full-capacity traffic conditions. This feature greatly simplifies the reasoning of the messaging system, such as when layering a software-implemented coherent shared memory model on top of the system.

[7] A deadlock condition may arise if we allowed P1 messages to be blocked behind P0 messages, given the request/reply relationship between P0 and P1 messages (P0 progress is dependent on P1 progress). Such is not the case here, as P1-unordered messages are only waiting on P1-ordered messages, and both are reply type messages that are guaranteed to be sunk by their receivers. The scheme is aimed at preventing unfair starvation conditions.

Chapter 4

Microarchitecture and Implementation

A prototype of the MAP chip has been fabricated in $0.7um$ (drawn gate-length) CMOS technology. It measures $18mm$ on a side and contains approximately 5 million transistors. A plot of the prototype chip is shown in Figure 4-1.

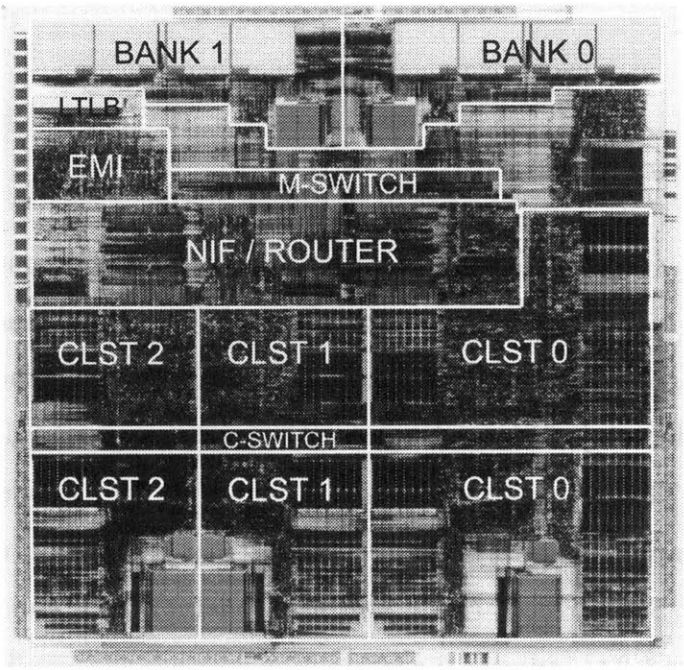

Figure 4-1: MAP Chip Prototype

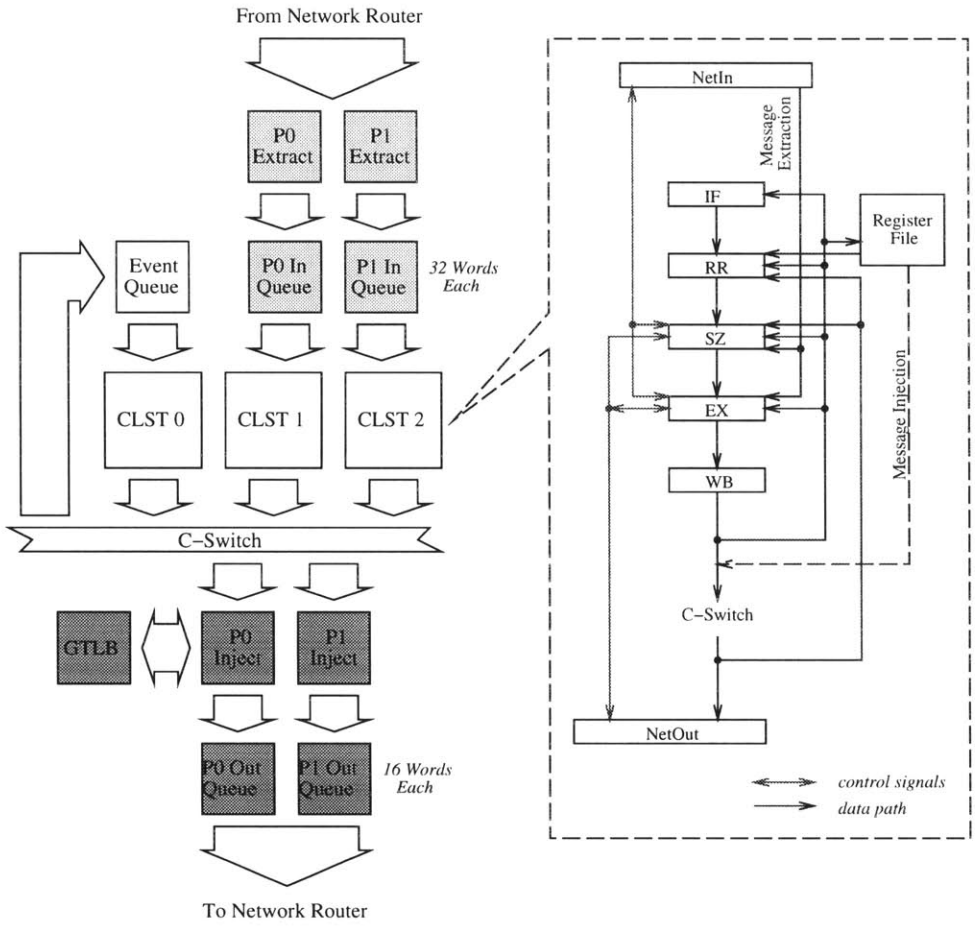

Figure 4-2: Network Interface Units

4.1 Overview

The message subsystem is made up of three major components: the network output (message injection) unit, the network input (message extraction) unit, and an integrated network router. Figure 4-2 shows the organization of the network interface units. Independent injection and extraction units are provided for each message priority. The GTLB is implemented as a module of the network output unit. A simplified illustration of the five-stage – *instruction-fetch (IF), register-read (RR), synchronization (SZ), execute (EX), and write-back (WB)* – cluster pipeline is also included in Figure 4-2 to indicate the interaction between the pipeline and the network interface units. Each of the network input (NETIN) and network output (NETOUT) units are connected to the synchronization and execute stages by resource availability,

arbitration, and commit handshake signals. Messages are injected through the C-Switch to the NETOUT unit [1], but are bypassed directly into the SZ and EX stages from the extraction unit.

Figure 4-2 also shows the MAP's *event queue* (EQ), which buffers non-blocking, asynchronous events within the chip, such as GTLB misses and memory synchronization failures. Functionally, it is rather similar to the network input unit, except the EQ is connected to cluster 0, and receives its entries from the C-switch instead of the network. Events are processed by a privileged event handler running in a reserved thread slot. The EQ does not respond to `MsgHd` operations, nor have the associated flush mechanism. The handler accesses the EQ by reading `MsgBody`, which also discards each word as it is consumed, but unlike the message queues, the EQ's entries are not padded with null values.

A 2-dimensional mesh network connects together up to 1024 MAP chips, each identified with its cartesian coordinates $[x, y]$ within the mesh. Inside the network, each message is represented as a sequence of flow control digits, or *flits*, which are 74-bit packets – 6 bits of control information, 68 bits of payload. A flit is the smallest unit on which flow control is performed in the router, and is transfered across chip boundaries in two 37-bit physical digits, or *phits*, per clock cycle.

The router performs dimension-order routing – messages are routed first in the x dimension, then the y dimension – and is made up of 2 mostly-identical *dimension* modules as shown in Figure 4-3. Each of the *dimension* modules is independently responsible for the buffering and routing of messages in one of the x, y dimensions. The inject and extract modules in the router core connect to the network interface units. The inject module formats the *flits* by computing the initial turn direction and a parity bit, while the extract module is responsible for verifying the parity bit of at the destination. Parity is not checked in the middle of the route.

As shown in Figure 4-3, a message enters each dimension by being deposited into

[1] To conserve chip area, messages are streamed out from the register file into the C-Switch by reversing the write-back bus in the prototype. This eliminates the need for a dedicated message injection bus. Due to the small size of M-Machine messages, the penalty of stalling the pipeline during message injection is not significant.

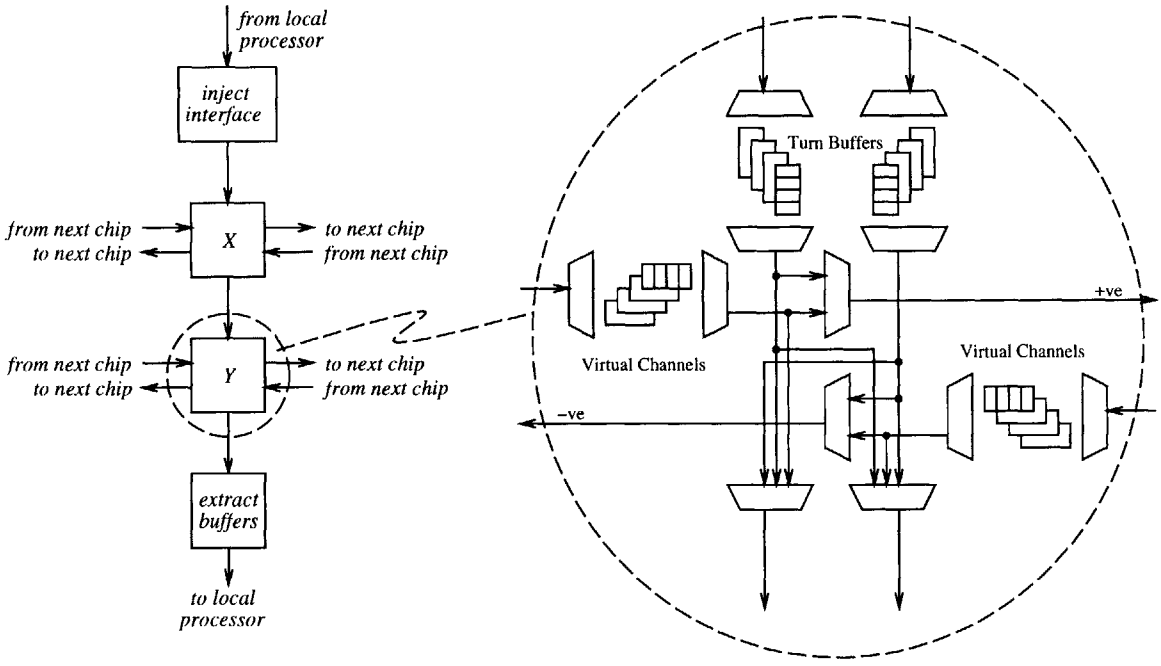

Figure 4-3: M-Machine Network Router

a *virtual channel* or a *turn buffer* [2], which are simple FIFO buffering elements. There are 4 virtual channels [3] associated with each of the +ve/-ve directions, and also 4 turn buffers going to each of the +ve/-ve direction. Messages leave a dimension module by being delivered to the next router chip (in the same direction), or being deposited into a turn buffer in the next dimension module or the extraction module.

4.2 Injection Interface

The netout units are accessed via the C-Switch crossbar. Each unit contains a 16-word outgoing message queue and an injection controller. The message queue serves as a buffer between the processor and the network, ensuring that an entire message

[2] Logically, turn buffers perform the same function as virtual channels. They are named differently to indicate that a message actually cross from one dimension to another when it is deposited into a turn buffer.

[3] Four of these FIFOs, VC3/VC2 and TB3/TB2, were eventually removed from the final prototype implementation to fit into the alloted chip area. The description in the rest of this chapter refers to the original architecture containing 4 virtual channels.

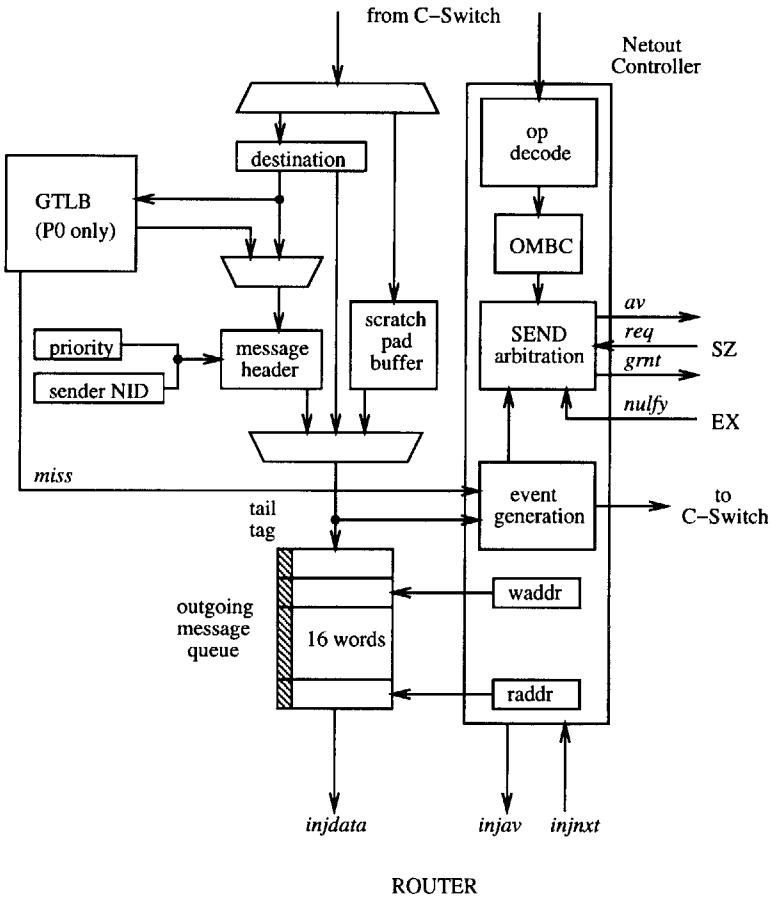

Figure 4-4: Network Output Unit

can be absorbed once a SEND instruction issues, even when the network is congested. This is crucial for providing the atomic semantic of the SEND instruction without causing prolonged blocking of the sender thread. To provide this guarantee, a SEND instruction is permitted to issue only when the corresponding message queue has vacancy for at least 13 words, which is the maximum length for a message, including the header. If the network router is able to accept the message immediately, the message is simply streamed out into the network through the queue, without waiting until the entire message is deposited in the queue.

4.2.1 Netout Controller

The message queue is managed through a pair or pointers, `raddr` and `wraddr`, which tracks the beginning and end of valid data within the FIFO structure. The netout controller monitors the vacancy in the queue and the OMBC (P0 only), and provides three signals, `msgav`, `thtav` and `ordav`, indicating the availability of the injection resources, to the synchronization stage which contains the instruction issue logic. The `msgav` signal indicates that the injection unit is not currently busy handling a message, and that necessary vacancy is present in the message queue. The `thtav` signal is cleared when the OMBC value is zero, indicating that throttling `SEND` operations should be blocked. To maintain the in-order delivery property, `ordav` is de-asserted whenever a GTLB-miss occurs, so that virtually-addressed, ordered `SEND` instructions are blocked while a GTLB miss event is being resolved by the system software. This prevents subsequent ordered messages [4] from leaving the node while the previous, GTLB-missed message is being retried.

The controller also contains an arbitrator which fairly assigns the injection resources to the three clusters. Each cluster must first present a request stating a message type, and be granted the use of the injection unit before it can issue a `SEND` instruction. A cluster that recently used the injection unit is given lower preference in the next round of arbitration, ensuring that each cluster gets an opportunity to send its message eventually. Since a `SEND` instruction is not considered for issue until the injection resources have been thus reserved, a predicated `SEND` operation must also release the reserved resources if it is nullified. This is done with a `nullify` signal from the EX stage.

4.2.2 The GTLB module

Once a SEND instruction issues, the message words are transfered across the C-Switch in a burst. A scratch pad buffer is used to hold the message words momentarily while

[4] Recall that in-order delivery applies only to messages of the same type, *i.e.* no ordering is guaranteed between virtually-addressed and physically-addressed messages.

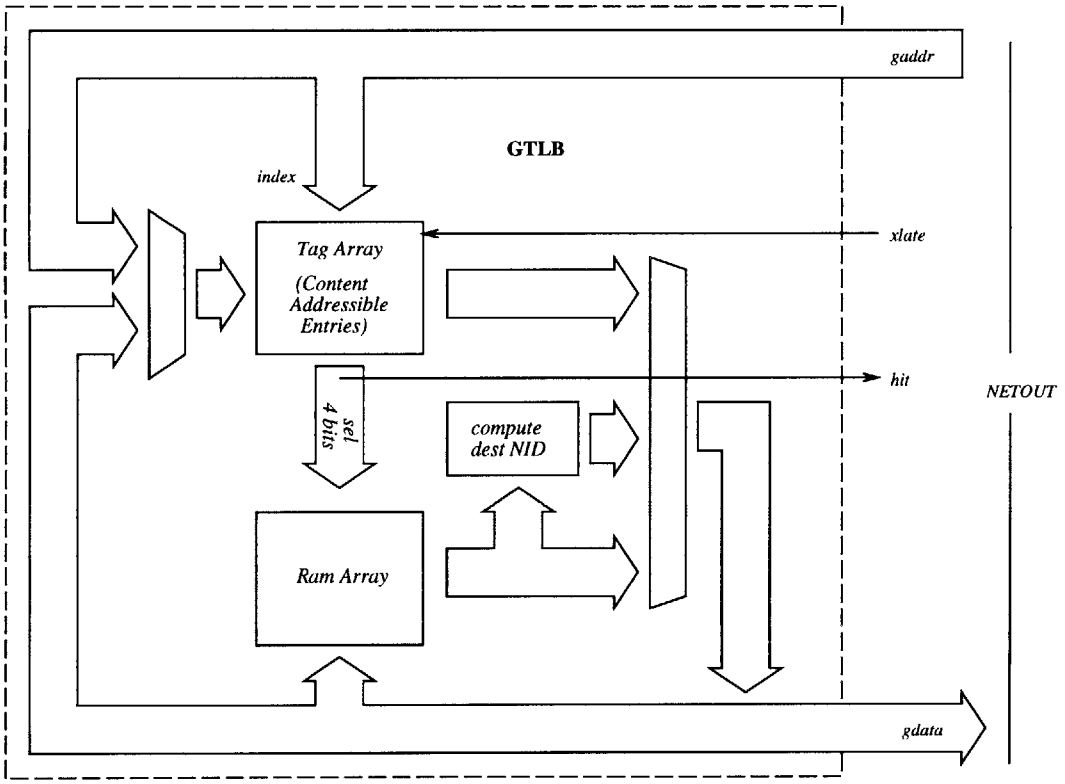

Figure 4-5: GTLB Data Path

address translation takes place, and the message header is being assembled. Address translation, performed by the GTLB module, is available to P0 messages only.

The GTLB consists of 4 entries which span two cache arrays. The *tag array* holds the *tag* and *mask* fields, while the *ram array* contains the *base*, *ext* and *ppn* information [5]. The first array is content-addressable by the input virtual address. It provides an index into the second array if a match is found. A *valid* bit is also associated with each GTLB entry, to indicate if the entry contains a valid mapping:

Tag Array			*Ram Array*		
Valid	*Tag*	*Mask*	*(Log) Pages Per Node*	*Extents*	*Base Node ID*
1 bit	42 bits	42 bits	6 bits	6 bits	10 bits

LSB

The GTLB connects to the rest of netout unit via a *GTLB Addr* bus and a *GTLB Data* bus, as shown in figure 4-5. It can be access in two ways – with a virtual

[5]The GTLB entry format is described in Section 3.3.1.

address, or with an entry index number. In the index-addressed mode, the low 2 bits of the *GTLB Addr* bus is used to directly select a GTLB entry. This mode is used by the `IGTRD` and `IGTWR` instructions, which directly read and write the GTLB arrays. The C-Switch packets returned by the `IGTRD` instruction, and expected by the `IGTWR` instruction, are shown below.

Bit 12 of Input Addr	*IGTRD C-Switch Data*			
0	(Word left-padded with 0's)	Entry Valid Bit (1 bit)	Base Node ID (10 bits, y/x)	GTLB Entry Tag (42 bits)
1	(Word left-padded with 0's)	(Log) Pages Per Node (6 bits)	Extents (6 bits, y/x)	GTLB Entry Mask (42 bits)
C-Switch Cycle	***IGTWR C-Switch Data***			
0	High bits Unused	GTLB Entry Index (6 bits)	Clear GTLB Miss Flag (1 bit)	Unused (12 bits)
1	High bits unused	Entry Valid Bit (1 bit)	Base Node ID) (10 bits, y/x)	GTLB Entry Tag (42 bits)
2	High bits unused	(Log) Pages Per Node (6 bits)	Extents (6 bits, y/x)	GTLB Entry Mask (42 bits)

LSB

The virtual-addressed mode is used in automatic address translation and to support the `IGPRB` instruction which performs an explicit translation. The input virtual address is compared against the entries in the tag array, masked by `mask`. If a match is found, the destination NID is computed using the contents of the corresponding entry in the ram array:

```
hit = valid & ((tag & mask) == (vaddr & mask))
yoffset = ((vaddr>>ppn)>>ext.x)[ext.y-1:0]
yoffset = (vaddr>>ppn)[ext.x-1:0]
dstnid.y = base.y + yoffset
dstnid.x = base.x + xoffset
```

4.2.3 Event Generation

If a SEND instruction is successful, the netout unit validates the `Ack` condition register after the final message word has been received over the C-switch, by writing back over

the C-switch itself [6]. If a virtually-addressed message fails to find a translation in the GTLB however, the validation write-back is suppressed, and the event generation finite-state machine (event FSM) is triggered. A *GTLB-Miss* flag is also raised in the netout unit, forcing `oxlav` to become low. The GTLB-miss flag stays asserted until it is cleared explicitly with an `IGTWR` instruction. The event FSM assembles a three-word event entry, and arbitrates for the C-switch bus to write it into the event queue. When the event has been successfully enqueued, the netout unit is reset to wait for the next SEND instruction. Ordered, virtually-addressed messages are blocked however, until the GTLB-miss flag is cleared by the event handler. The GTLB-miss event entry format, shown below, encodes the type of the offending message (ordered?, OMBC decremented?), the sender thread identifier (sender cluster, sender thread slot), the register file that contains the message body (rf), the message length (argc), and the `<Ack>` condition code register (rtnCC) to be eventually validated by the event handler.

Word 0	unused	1 bit ordered	1 bit decombc	2 bits sclst	3 bits stslot	1 bit rf	4 bits argc	1 bit xlate	4 bits rtnCC	4 bits (event type) "GTLB miss"
Word 1	65 bits HandlerIP (operand 2)									
Word 2	65 bits Message Destination (operand 1)									

4.2.4 Message Header

Every successfully injected message is prepended with a message header. Figure 4-6 shows the format of the header word. The `x` and `y` fields are the physical coordinates for the destination node. The `argcount` field records the number of words contained in the message body, not counting `DestAddr`, the `HandlerIP`, and the header itself. The `senderNID` field carries the node identifier for the sender. The `turns` field contains routing directives that indicate if the message should be routed to the +ve or -ve

[6]The netout unit is both a receiver and a writer on the C-switch. It starts arbitrating for the C-switch bus during the last cycle of the incoming burst, and can perform the write-back during the subsequent cycle if the bus request is granted.

direction – these are computed and inserted into the header by router's injection module as the message enters the network. The header also contains the flags for in-order delivery and message priority. The assembled header is placed into the outgoing message queue preceding the other message words, the last of which is tagged with a *tail* bit. Data from the queue is presented to the router on the `injdata` bus. The netout unit asserts an `injav` signal when a valid datum is available on the bus, while the router pulses an `injnxt` signal when it has consumed the word.

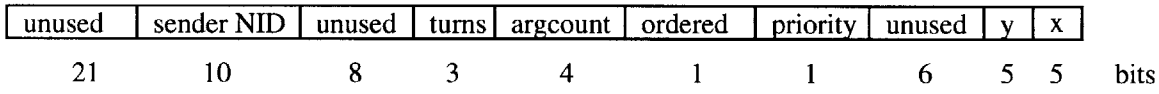

Figure 4-6: Message Header Format

4.3 Extraction Interface

The P0 and P1 extraction interfaces tie directly into the integer datapath on cluster 1 and cluster 2, respectively. Each extraction interface consists of an incoming message queue and an extraction controller. The message queue is 32 words long, as shown in Figure 4-7.

Five control signals go between the netin unit and the SZ and EX stages. Availability of the `MsgHead` and `MsgBody` data is indicated by the `headav` and `bodyav` signals, which are connected into the scoreboard. To request for the `MsgHead` word, the SZ stage asserts the `rhead` line. The `MsgBody` datum is returned otherwise. The `pop` signal is pulsed when an instruction sourcing `MsgHead` or `MsgBody` is issued. This causes the netout unit to optimistically advance the queue pointers to supply the subsequent word. However, if the instruction is squished, the `MsgHead` / `MsgBody` datum is not actually consumed. In this case, the EX unit can reverse the effect by asserting the `undo` line in the immediately following cycle.

The netin unit interfaces to the network router via four control signals and an `extdata` bus. The router asserts the `extav` signal to indicate that a valid word is

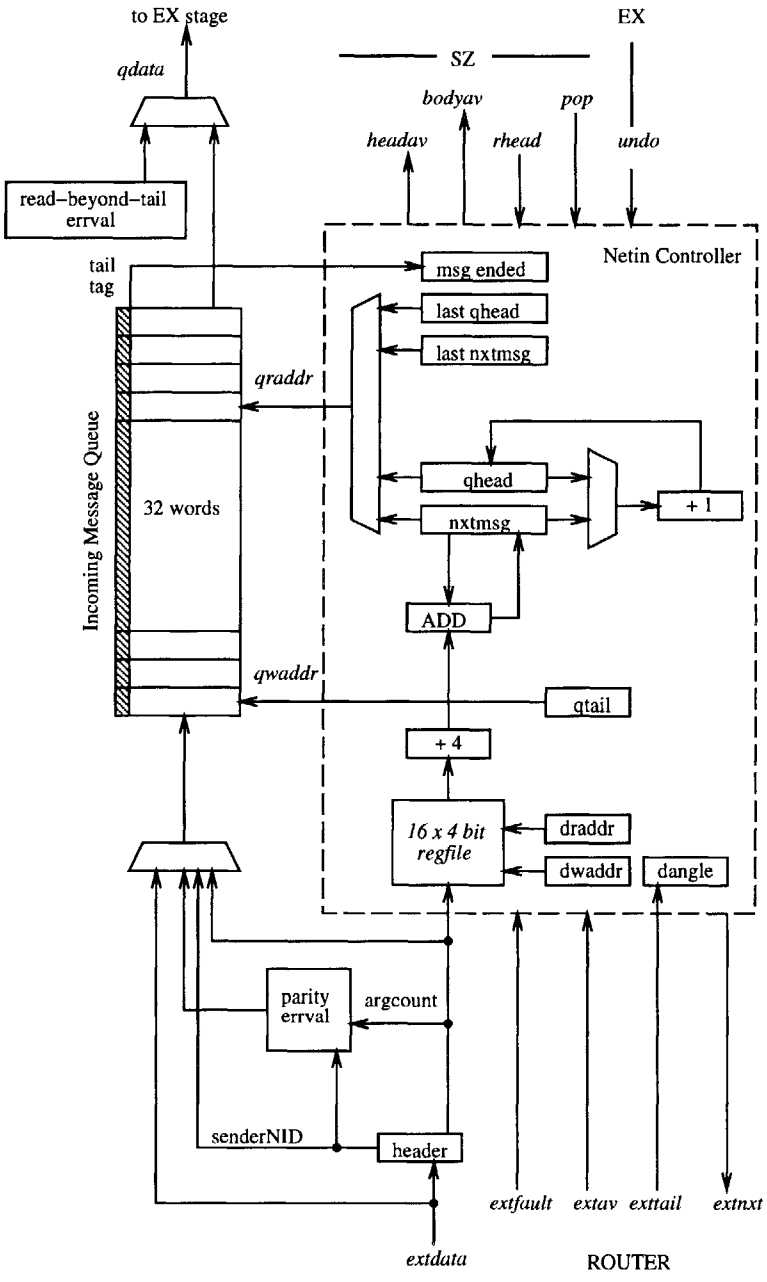

Figure 4-7: Network Input Unit

available on `extdata`, and uses `exttail` to indicate if it is the final word in a message. The netout unit pulses `extnxt` if it consumes the datum, causing the router to supply the next word in the following cycle. An `extfault` line is also provided by the router to signal a parity error, *i.e.* when the parity bit carried in a flit (tagged on by the router's inject module at the sender node) fails to match up with the parity computed by the extract module in the router core [7].

4.3.1 Parity Error

The netout unit responds to the `extfault` signal differently, depending on whether the parity error is detected in the message header, or one of the other message words. In the former case, the corrupted header may have caused the message to be misrouted. To avoid invoking a message handler on the wrong node, the `HandlerIP` is replaced with an *errval* dummy value [8], and the message is truncated into 4 words:

1.	handlerIP	Replaced with dummy value
2.	Message Argument Count	whatever that is salvaged
3.	Sender Node Indentifier	from the header flit.
4.	Destination Address	3rd flit from the message.

In the latter case, the netout unit simply replaces the corrupted word with an *errval* before it is placed into the incoming message queue. The rest of the message is enqueued normally. Note that the streaming extraction interface prevents the netout unit from discarding the entire message when a parity fault is detected, since a fault may occur towards the end of the message, after the first few words have been consumed by the handler. For diagnostic purposes, a count of the parity faults detected so far on the node is included in the *errval* encoding, shown below.

[7] The M-Machine network does not implement error checking/recovery within the routing path.

[8] An *errval* is a special type of *guarded pointers* that encodes fault information for later processing. Guarded pointers are described in Chapter 5.

Pointer Bit	*Pointer Type*	*Pointer Length Field*
1 bit	4 bit	6 bits
"1"	"errval"	"net parity fault"
	4 bits	6 bits

Pointer Address Field					
54 bits					
(unused)	Parity Fault Count	Priority	Ordered	ArgCount	Sender Node ID
	8 bits	1 bit	1 bit	4 bits	10 bits

LSB

4.3.2 Decoded Message Format

A successfully extracted message is placed into the incoming message queue. Front-end logic is provided to reorder the first two words in each message, and disassemble the message header into the sender's node identifier and the argument-count of the message. From the message handler's point of view, the message words arrive in the following sequence:

1.	HandlerIP	Word following the header flit
2.	Message Argument Count	Decoded from the
3.	Sender Node Indentifier	message header
4.	Destination Address	3rd flit from the message.
5 onwards.	Message Body	Remaining flits

4.3.3 Netin Controller

As in the netout unit, the incoming message queue is managed through a number of begin and end pointers. However, to implement the side-effects of `MsgHead` and `MsgBody`, the netin controller has to be a little more sophisticated. The appending end of the message queue is still tracked by a simple *qtail* pointer, but at the other end, the `rhead` signal from the SZ stage selects between the *nxtmsg* and *qhead* registers, which correspondingly point to the `MsgHead` and `MsgBody` words.

In addition to the main message queue, a small 4-entry queue is used to contain the size of the messages. This small *delta* queue is managed simply by a pair of *draddr* and *dwraddr* pointers. By adding 4 to the message size – to account for `HandlerIP`, `Destination Address`, and the two extra words extracted from the message header – a *delta* value is obtained, which can be added to the *nxtmsg* register to find the location of the next `MsgHead` word. This allows the message handler to jump to the next message instantly when it reads `MsgHead`.

As `MsgHead` is read, the *qhead* register is set to the location following *nxtmsg*. Subsequent reads to `MsgBody` then increment the *qhead* register to return subsequent words from the message. To implement the `undo` feature, the last *qhead* and *nxtmsg* values are always preserved in the *lastqhead* and *lastnxtmsg* registers, so that the side-effects can be reversed by simply copying back the old values.

A *dangle* register flags when a message is currently being extracted from the network. When the message queue is empty, *dangle* indicates that the next `MsgBody` word has yet to arrive, and the `bodyav` line should be de-asserted to stall accesses to `MsgBody`. Otherwise, `bodyav` should be high and `MsgBody` is padded with null values if the handler reads beyond the end of a message. To implement this, the end of each message is also tagged with a *tail* bit in the message queue. When a message tail is read, the *msgended* register is set. This causes all further reads to `MsgBody` to return an *errval* dummy value, until `MsgHead` is accessed again.

4.4 Network Router

The router core consists of two nearly-identical *dimension* modules, each of which is further made up of two mirrored halves. Each *half-dimension*, shown in Figure 4-8, manages one set of four *virtual channels* and one set of four *turn buffers*, and is responsible for routing in one direction. The virtual channels are 6-entry FIFOs, while the turn-buffers are 2-deep.

Two main functions are performed by the router unit: channel allocation and switch arbitration. Each message arrives into a half-dimension by being deposited

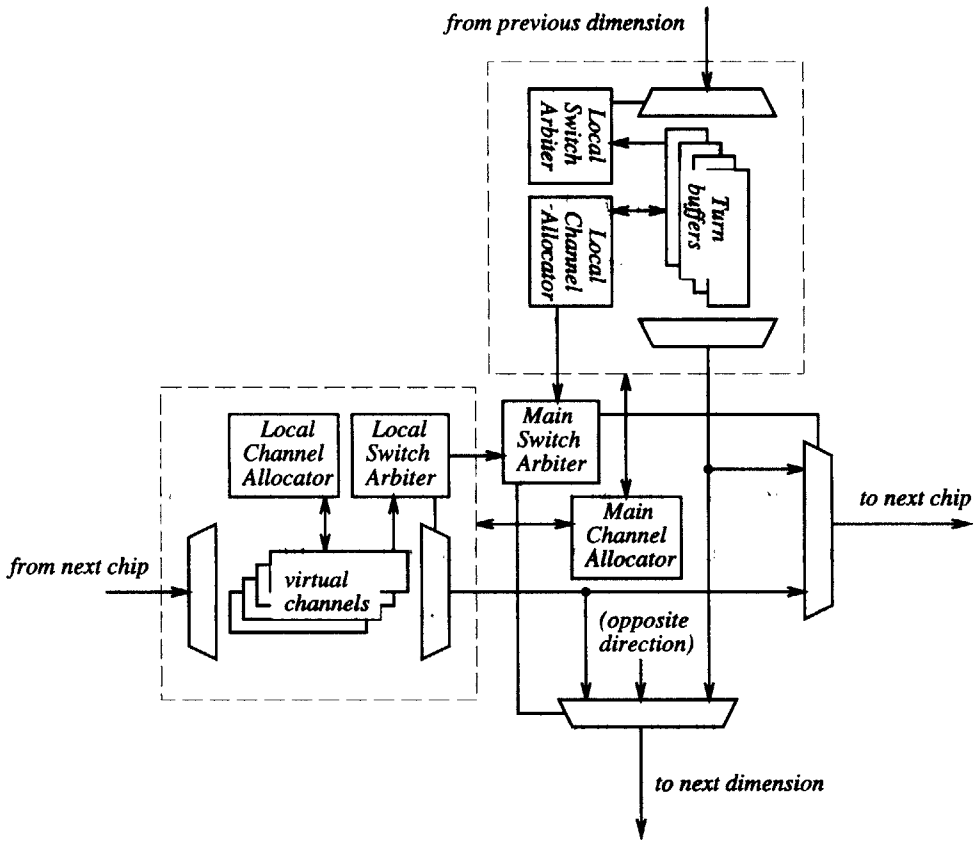

Figure 4-8: One Half of a Dimension Module in the Router Core

into a turn-buffer (TB) or a virtual-channel (VC). The router must allocate a a TB or VC downstream to accommodate each message, and multiplex the buffered messages onto the appropriate physical link, to be forwarded to the next dimension, the neighboring router, or the network input unit. The high-level allocation policy was described earlier in the discussion of in-order delivery in Section 3.3.3. The implementation and microarchitecture are described in the rest of this section.

4.4.1 Routing Decisions

Messages are routed in dimension-order, first in the x dimension, then in the y dimension. The routing decision – where to route the message next – is made in the dimension module on arrival of the header flit of a message. It is determined by comparing the subfield for the current dimension of the destination NID in the message to the local NID. If the two are *different*, the message is forwarded to the next router in the same direction. If they *match*, the message drops into a turn buffer in the next dimension.

To enable this decision to be made quickly, a redundant 3-bit *turn directions* field is pre-computed in the message inject module using the relative position of the receiver node from the sender, and the message priority. This information encodes the +ve/-ve direction to turn as the message reaches each terminal plane in the x and y dimensions, and which of the P0 and P1 netin unit to contact at the destination. As an convenience, a set of diagnostic-settable *flipdir* bits (one for each dimension) are also included, to swap the +ve/-ve direction of the appropriate dimensions in a router, so that neighboring MAP chips at the edge of the M-Machine can be assembled in a stacked fashion.

Once the next route direction has been determined, a buffer downstream of the route must be allocated to hold the message before any flits are forwarded. This allocation is performed in a 2-stage process, using a *main allocator* (one for each outgoing path) and a *local allocator* (one for each group of VCs or TBs). The main allocator chooses a group of VCs/TBs to grant a particular sink FIFO, while the local allocator determines which of the source FIFO within the chosen group gets

the resource. As discussed in Section 3.3.3, the allocation preference and decision are determined by the type of message (*ordered*, *priority*) being routed. The physical links to the neighboring routers, and the write ports into the TBs in the next dimension, are multiplexed amongst the source FIFOs using a 2-level arbitration process much like the allocators. One main arbiter is responsible for granting each output path from each dimension to one of the relevant source FIFO groups. Within each group, a local arbiter decides the final winner. No arbitration is needed in the y – *extract* boundary however, since the data paths are not multiplexed there – a separate extraction buffer is provided for each priority.

Once it is allocated a downstream FIFO, each VC/TB remembers its designated sink information, to be presented for switch arbitration in subsequent cycles. For newly arriving messages, switch arbitration is performed optimistically in parallel with channel allocation, so that straight-through traffic can be forwarded in the cycle immediately following its arrival, as shown in Figure 4-9. The timing diagram also shows the incoming phits being retimed from the remote clock domain to the local clock domain. This is described later in Section 4.4.3. To prevent a message from being forwarded if a sink FIFO cannot be allocated to it, grant signals from the switch arbiters are late-gated by the allocation results.

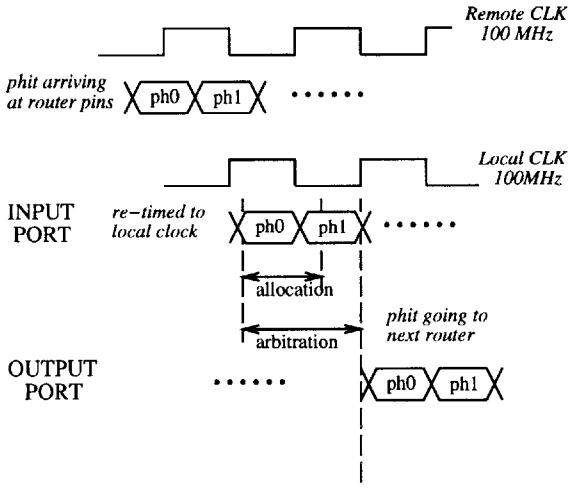

Figure 4-9: Router Timing for Straight-Through Traffic

4.4.2 Resource Management

A message may fail to secure a sink buffer if it loses in the allocation process, or if no suitable unused FIFOs are available downstream. Even after a sink buffer has been assigned, flits from a VC/TB still cannot be forwarded unless vacancy exists in the sink buffer. The VC availability and vacancy information is kept in 4 resource manager modules – one for each group of VCs downstream. Each VC resource manager contains 4 *free* flags, and 4 vacancy counters, one for each VC in the group it monitors. Similar resource managers are used for downstream TBs, except that the vacancy information is obtained directly from the TB modules instead of being kept in counters, since the TBs are conveniently located locally.

The corresponding *free* flag is cleared as each FIFO downstream is allocated, while the vacancy counter is decremented as flits are deposited into the sink. When flits are forwarded from a VC, the resource manager requests for the corresponding vacancy count in the *upstream* router to be incremented, by sending the information back in the *FVC* field [9] in each flit. After the message tail leaves the upstream router, the upstream resource manager waits for the downstream FIFO to become empty again, and then sets the *free* flag to indicate that the buffer has been de-allocated (Figure 4-10). To prevent buffer overflow, the allocator assigns a FIFO only if it is not already allocated, while the switch arbiter forwards a flit only if the vacancy for the corresponding FIFO downstream is not zero.

The sink FIFO for a flit being forwarded is identified in the the *DVC* field. Both the FVC and DVC fields are 3-bits wide – one valid bit, and two channel identifier bits [10]. The phit encodings are shown below. Two phits, *ph0* and *ph1*, make up a *flit*, which also contains a parity bit and a tail bit.

[9] The VC length is set at 6 flits to minimize bubbles in the pipeline, as the FVC information takes up to 5 cycles to affect the vacancy count upstream.

[10] In the prototype where VC2 and VC3 were removed, a 2-bit, one-hot vector is used.

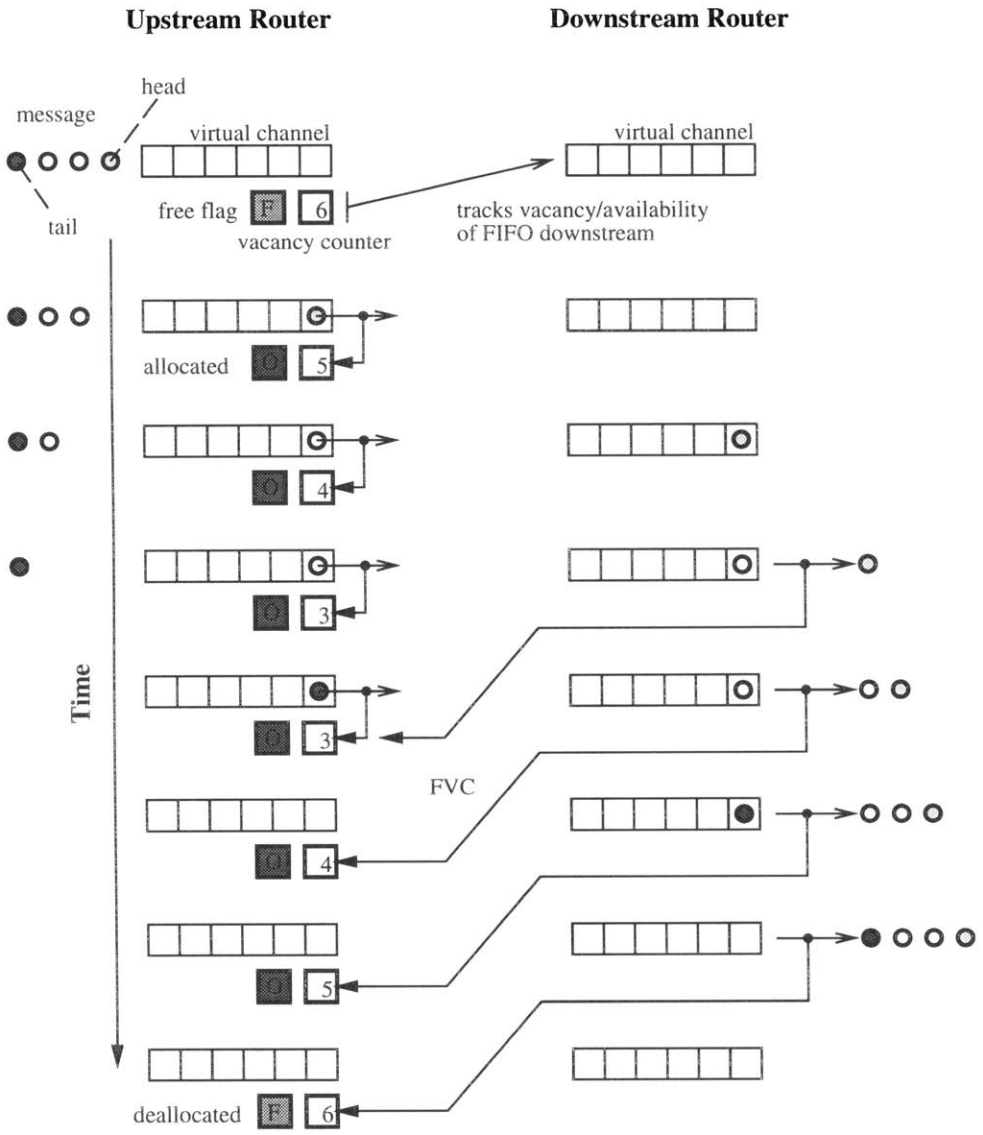

Figure 4-10: Vacancy Counter and Free Flag

Clock phase	*37-bit phit*		
	Control (3 bits)	*Data* (34 bits)	
ph0	*DVC*	*T* (1 bit)	*D0: Data 1 of 2*
ph1	*FVC*	*P* (1 bit)	*D1: Data 2 of 2*

MSB LSB

4.4.3 Mesochronous Router-to-Router Interface

The router interfaces to each of its four neighbors via a 39-pin port. Two of these pins carry clock signals and are driven uni-directionally – one in each way. The remaining 37 pins carry signals in both directions *simultaneously* [51]. Using both phases on the clock, two phits are transfered every cycle in each direction. To tolerate clock phase differences across routers, the inter-chip interface employs a *mesochronous* technique adapted from [52]. The system clock signal is assumed to be identical throughout the M-Machine, except for fixed phase differences due to varying distribution distances.

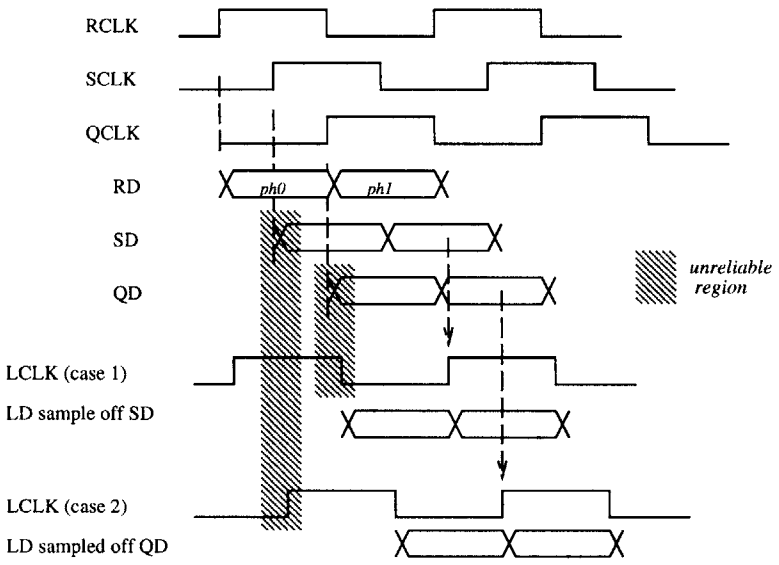

Figure 4-11: Mesochronous Interface

Each chip transmits its local clock signal along with the data. The remote clock signal RCLK, is used by the receiver pads [51] to derive two new clock signals, SCLK and QCLK, with a delay lock loop. SCLK trails RCLK by either 90° or 270°, while QCLK always trails SCLK by 90°. Depending on the phase difference between RCLK and the

local clock, LCLK, data latched off one of SCLK and QCLK can be sampled safely in the local clock domain (Figure 4-11).

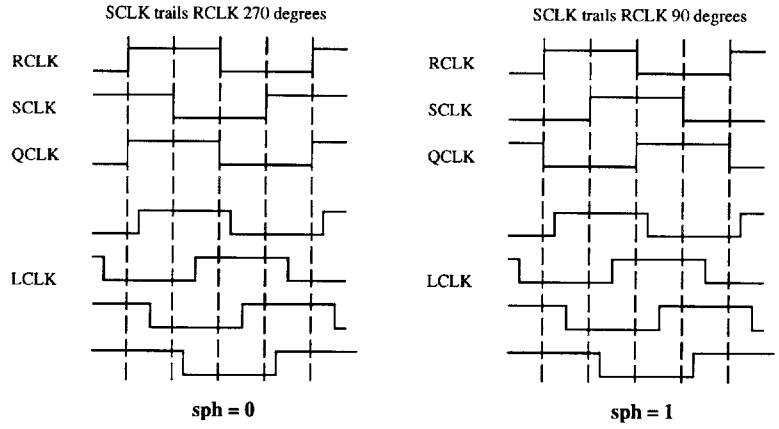

Figure 4-12: Phase Difference Between LCLK and RCLK

Figure 4-12 shows the 8 possible scenarios for the relationship between the clock signals. The sph signal, generated by latching RCLK with SCLK using a positive-edge flipflop, is 0 when SCLK trails RCLK by 270°. As shown, the rising edge of the local clock must fall within one of four quadrants of an RCLK cycle. The location of the LCLK edge relative to RCLK can be derived by sampling the SCLK and QCLK signals, as shown in Figure 4-13. With this information, it is possible to select one of SD and QD to be sampled by the local clock.

Note that if the edges of LCLK and SCLK or QCLK are too close together, the sampled ssclk or sqclk signals may be unreliable. However, recognizing that at least one of ssclk/sqclk must be correct, a careful selection can still be made to always provide enough margin in the data latches. For example, Figure 4-14 shows a situation where sqclk is unreliable because QCLK and LCLK are coincident. Depending on which value is obtained, the mesochronous unit may choose to latch SD0 or QD0 into LD0, assuming sph is zero in this example. However, in either case, LD0 has at least one quarter of a cycle to settle before it is latched again, off the opposite phase of the local clock, to provide a clean D0 signal. Note also that the latched data may also need to be delayed by 180° under half of the eight scenarios, so that the *ph0* phit is received

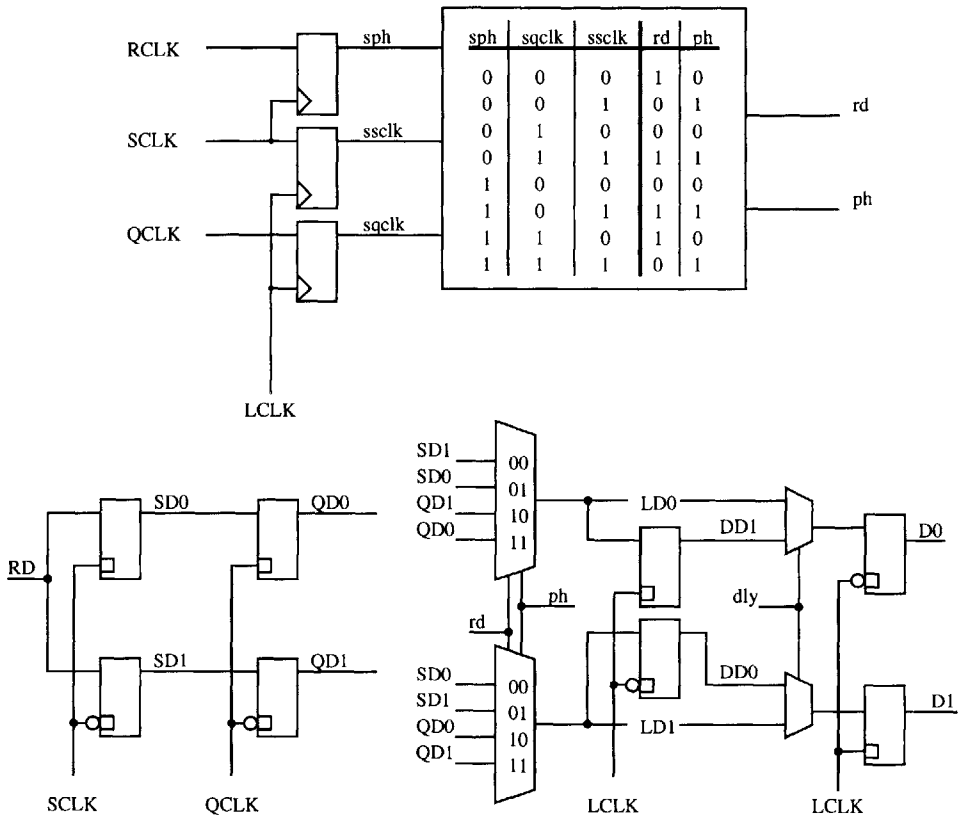

Figure 4-13: Selecting Between QD and SD Samples

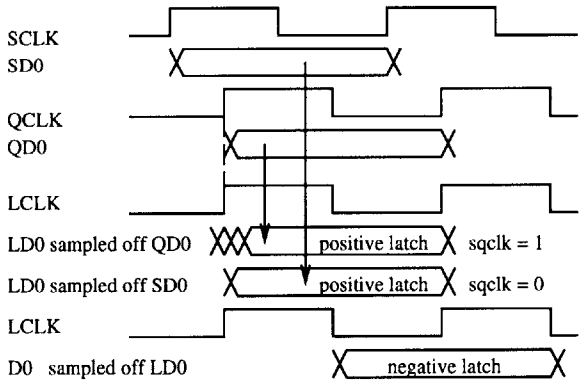

Figure 4-14: Unreliable `sqclk` Does Not Affect `D0`

during the high phase of `LCLK`. This is accomplished by selectively switching another set of latches into the data path.

Chapter 5

Protection and Starvation Avoidance

The network interface is a critical resource shared by all processes on each node in the multicomputer. This sharing calls for two crucial considerations in designing the message system. First, to avoid starvation and deadlocks, the system must guarantee that every messaging request is eventually serviced. Second, to protect message data from being intercepted or corrupted, either inadvertently or intentionally, processes must be insulated from one another within the message system.

These requirements are poorly addressed in many modern designs which have adopted user-level message interfaces in an attempt to avoid the prohibitive overhead of a system-software layer within the message system. Featuring direct user-level messaging mechanisms, these designs reduced the cost of sending a message dramatically, from many thousands of cycles [24, 22, 20], to as low as several tens of cycles [10, 18]. However, the direct exposure of critical, shared messaging resources to untrusted user-level processes also compromised protection and created starvation risks.

The inadequacy of the message system hardware in these designs is sometimes circumvented with scheduling protocols [28] and programming conventions [53]. Such remedial solutions however seriously defeat the raw performance of the underlying hardware, even when they are reliable. The performance penalty would be especially pronounced within the fine-grain, fast-context-switching environment of a system like

the M-Machine. In order to benefit from user-level messaging interfaces, specific robustness considerations must be incorporated inherently into the design,

In this chapter, I discuss the mutually complementary starvation avoidance and protection solutions in the M-Machine. Section 5.1 focuses on the starvation problem. Using a *bounded-time lease* allocation scheme, sender-side monopolization of messaging resources is prevented. The risk of starvation is eliminated when this scheme is coupled with *trusted handlers* at the receiving end. In conjunction with a *guarded pointer* [13] based capabilities system, these mechanisms also constitute the protection model in the M-Machine, which restricts the data objects as well as the remote operations accessible to each thread. The protection model is described in section 5.2. In section 5.3, some alternative approaches are considered in contrast to the M-Machine mechanisms which, despite their simplicity, are flexible, efficient, and reliable.

5.1 Starvation Avoidance

The fundamental cause of starvation is the *open-ended* allocation of critical, limited resources, such as the network ports and message channels or buffers, to untrusted processes. A shared resource thus allocated cannot be reclaimed until it is *voluntarily* released by the user [1]. Starvation ensues when processes fail to return the allocated resources, and the critical resources run out.

Figure 5-1 illustrates the problem as manifested in conventional messaging systems. A buffered injection interface, a streaming-channel injection interface, and their corresponding extraction models are shown in Figure 5-1A through 5-1D. Messages are transferred *en bloc* between buffers on the processor node and the network in a buffered interface, and *piece-meal* through a channel or conduit from the sender to the receiver in a streaming interface.

In the former case, two threads must each obtain a pair of send/receive message buffers from the system before they can communicate with each other. Communi-

[1] While the system may conceivably revoke allocated resources by force under certain conditions, it is unclear how it can properly clean up after a user thread that is actively using the buffer/channel/port being thus reclaimed.

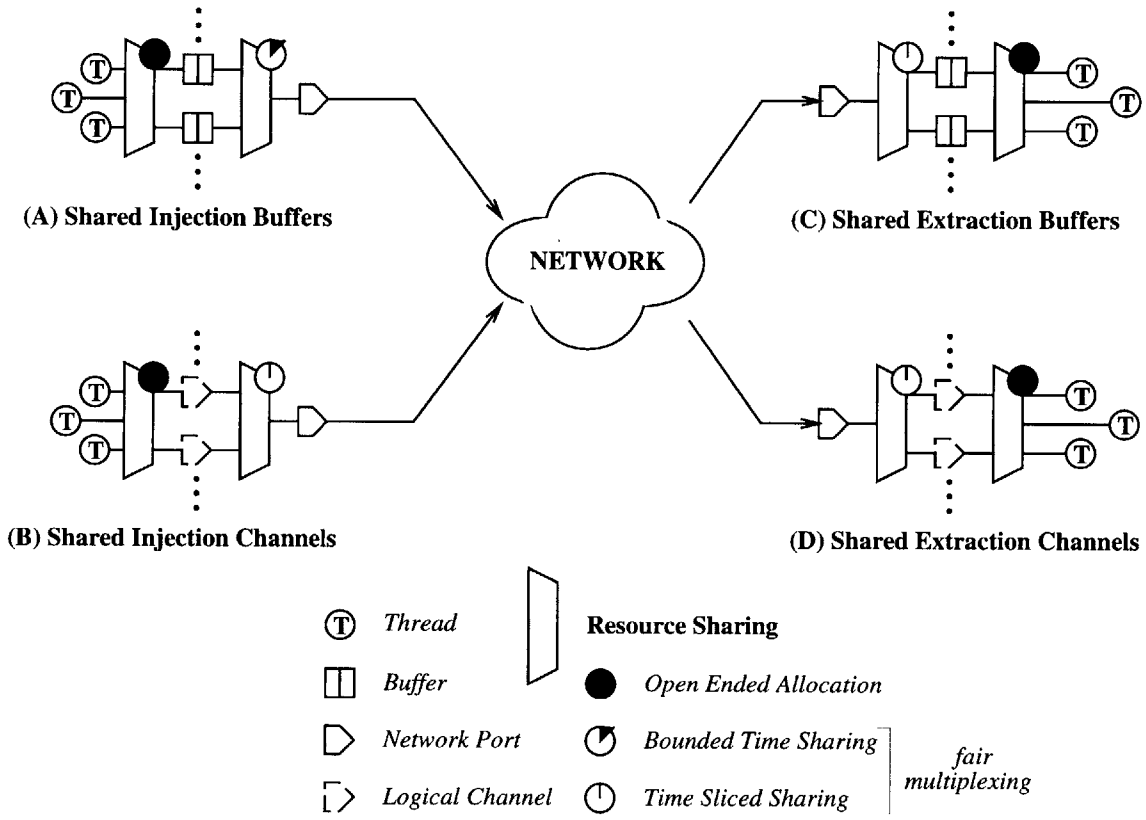

Figure 5-1: Conventional Message Interfaces

cation may thus be hampered if the system is unable to fulfill the buffer allocation requests, such as when the pool of available message buffers runs out. The user-level direct memory access (UDMA) mechanism in the Shrimp [16] system, for example, deposits messages directly into receive-buffers on the destination node. Because messages can arrive asynchronously, the receive-buffers – even though they can be virtual-memory mapped – must be locked down, or never evicted from physical memory, to guarantee that arriving messages can be accommodated. Therefore, unless users cooperate by relinquishing their message buffers promptly, starvation ensues when the limited physical memory pages are exhausted. The problem may become less pressing if messages were buffered in virtual memory [2], due to the much larger name space. It nonetheless complicates the interface, and is not immune to starvation as the virtual memory name space remains a limited resource.

The message size is unbounded in a streaming interface. Message words are written into a channel by the sender at one end, and read out by the receiver at the other end. Until the sender/receiver pair agrees to relinquish access to the channel, it cannot be reassigned. In the J-Machine [53] for example, the `SEND` instruction implicitly establishes such a channel, which is closed only by the `SENDE` instruction. Therefore, the code fragment between each `SEND` instruction and its corresponding `SENDE` instruction must be treated as a critical, exclusive region, within which no other thread may inject another message. Failure to adhere to this convention causes the independent messages to be spliced together. The situation is alleviated if multiple logical channels are multiplexed on a fairly time-sliced basis onto the physical network port (Figures 5-1C and 5-1D), such as in the iWarp [43] . By tagging each message word with its logical channel identifier, multiple messages can be transmitted or received concurrently on the same node without corruption. However, since the number of logical channels is limited, this approach is also not free from starvation risks.

The M-Machine solution is to make monopolization of resources impossible by

[2]The virtual memory based buffers used in such an approach must be backed by secondary storage that is local to the receiving node, or a separate memory paging network must be incorporated. Otherwise, the system risks a deadlock condition if a buffer has be paged in from a remote node through the same network.

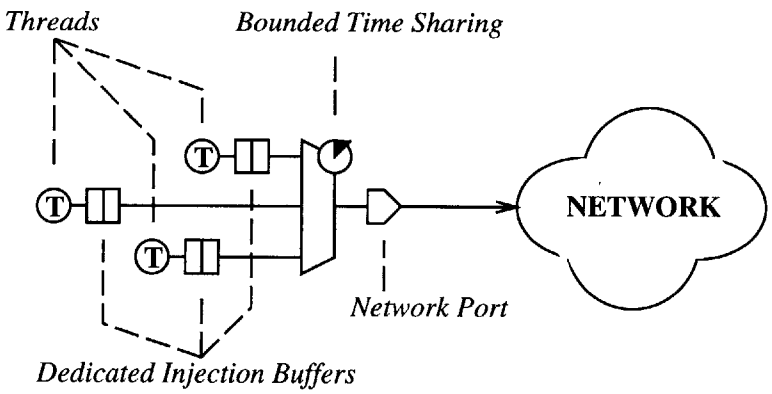

Figure 5-2: "Bounded Time Lease" Allocation

design. Figure 5-2 illustrates an atomic injection interface that permanently assigns a message buffer to each thread. Competition for message buffers is completely eliminated. Although the network port remains shared, it needs to be connected to a buffer for only a bounded period of time, during which the message is transferred into the network atomically. Given an appropriate fair arbitration module, this "bounded-time lease" scheme guarantees non-starvation on the sender side. The solution is amenable to both memory-based and on-chip buffering designs – a new thread is created only if a corresponding free message buffer is available. The system does not need to be concerned about fulfilling further requests from existing threads. In the M-Machine implementation, the message buffer is embedded into each thread's register files. By virtualizing the buffer, *i.e.* treating it as an integral part of the thread context that is swapped in and out along with the rest of thread state, the system is also able to migrate user threads freely between different physical thread slots. As more slots are added to the chip, the modularity of this design enables natural scalability.

The same approach is however impractical on the receiver side, since a newly arrived message risks corrupting a previous message if it asynchronously overwrites the recipient's dedicated buffer. Conversely, if it is made to wait, subsequent messages destined for other threads are blocked. A message handler based interface, shown in Figure 5-3, is adopted in the M-Machine instead. The incoming network port is exposed directly and exclusively to a dedicated thread slot. Each message specifies a

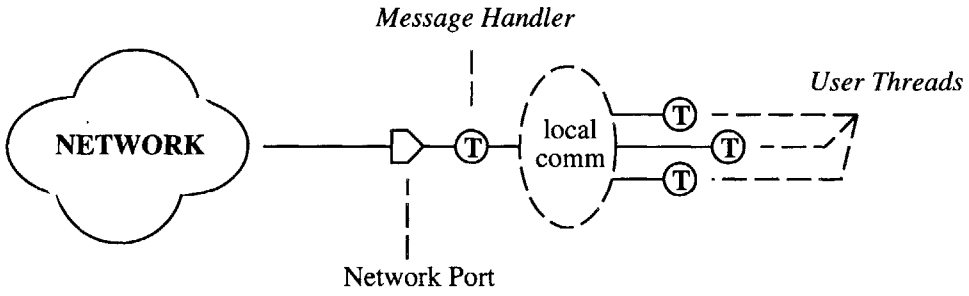

Figure 5-3: Message Handler

message handler which is invoked in this privileged slot upon arrival. To avoid abuse of access privileges to the network port, only trusted routines may be used as handlers. Otherwise, a handler may violate the protection model by accessing messages other than its own in the queue, and can cause starvation or a deadlock if it fails to complete quickly and dispatch the subsequent message properly. The enforcement of these *trusted handlers* in the M-Machine relies on its protection mechanisms, and is discussed further in section 5.2.3.

5.2 Protection

The challenge for a protection model that is both robust and efficient in a multicomputer comes from its distributed nature, as access control has to be enforced consistently across node boundaries with minimal authentication-related network traffic. The M-Machine system accomplishes this using a global addressing space that is accessed exclusively through *guarded pointers* [13]. Access permission is built into each globally-addressed guarded pointer, which refers back to the same object regardless of where the pointer is used.

The user thus cannot confuse the system by dereferencing a pointer at the wrong processor. In fact, the *Global Translation Lookaside Buffer* (GTLB) facility prevents the user from sending a message to the wrong processor at all. When a message arrives to access an object, the embedded protection information eliminates the need to query the originating node for permission checks or to authenticate the request

against any explicit access control table. In addition to restricting the accessibility of data objects, the protection system also uses guarded pointers to regulate the *trusted handlers* accessible to each process.

5.2.1 Guarded Pointers

The *guarded pointer* [13] is a primitive datatype that enables a fine grain access control scheme on the M-Machine, without relying on any lookup table. This is particular important in a multicomputer, as it is expensive to maintain an access-rights database that must be either queried with a message upon every access or replicated and synchronized on all processors.

A guarded pointer is distinguished from regular datatypes, such as integer and boolean, by a *pointer tag*, as shown in Figure 5-4. The pointer tag can be set only with a privileged instruction, making it impossible for guarded pointers to be created by user-level programs. This unforgeability enables a very efficient segmentation and capabilities system. Each pointer has a 6-bit *segment length* field and a 4-bit *type* field, in addition to a 54-bit *address* field. The length field defines a segment that is up to 2^{6+11} words in size, aligned to 2-Kword (2^{11} words) pages, and containing the address in the pointer. Except for the *ExecuteMessage*, *EnterUser* and *EnterSystem* types described below, address arithmetic may be performed by users on any pointer. However, any calculation that crosses the segment boundaries results in an *ErrVal*, which is a primitive datatype that encodes the error condition. The ErrVal triggers an exception when it is dereferenced, confining user access to within the prescribed segment. This results in a fine-grain (2KWord) segmentation system without any explicit lookup overhead for bounds-checking.

The type field indicates the permitted access mode of a pointer. Common types such as *ReadOnly* and *ReadWrite* are included, as well as executable *ExecuteUser* and *ExecuteSystem* types. The executable pointers define routines callable by user-level and system-level programs, respectively. In addition, the guarded pointer facilitates fast protection domain switching through *EnterUser* and *EnterSystem* types. When control flow is transferred by jumping to these executable pointers, the privileges

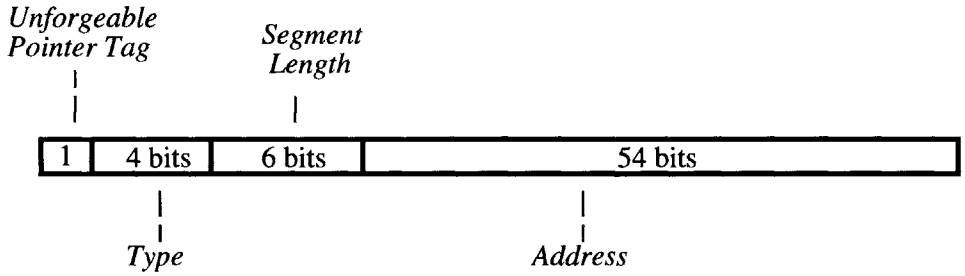

Figure 5-4: Guarded Pointer

switch automatically to user-level and system-level, correspondingly. An *ExecuteMessage* type is used for implementing trusted handlers. It is described in section 5.2.3.

The embedded protection information is checked by the hardware whenever a pointer is used in an instruction. Any attempt to use a pointer in an illegal manner results in an exception, which triggers an exception handler to remedy the condition in software. Efficiency is thus preserved without sacrificing robustness.

5.2.2 The GTLB and Virtually Addressed Messages

The GTLB (Section 3.3.1) prevents users from misdirecting their messages, either purposely or accidentally. Recall that the `SEND` instruction takes the form of `SEND <DestAddr> <HandlerIP>`, where `DestAddr` specifies the target object, and `HandlerIP` specifies the message handler. `DestAddr` is a virtual address pointer. The hardware flags a type exception and cancels the SEND operation if the user fails to provide a valid pointer in the `DestAddr` field. Otherwise, during message injection, the GTLB transparently translates `DestAddr` into a physical node identifier and other physical routing information, which then directs the message through the network. Since all user messages are virtually-addressed with transparently-translated, unforgeable pointers, stray messages are impossible. Meanwhile, the `DestAddr` pointer is faithfully delivered as part of the message. When `DestAddr` is referenced by the message handler at the destination, the embedded protection remains fully enforced by the hardware. The primitive capabilities, such as *ReadOnly* and *ReadWrite*, are thus

universal across node boundaries.

5.2.3 Trusted Handlers

.

The *ReadOnly* and *ReadWrite* capabilities encoded in the guarded pointer are adequate for simple access control, while the *EnnterUser* and *EnterSystem* types facilitate fast switching between protection domains within the processor. However, a more flexible scheme is desirable for remote operations in a multicomputer. Consider for instance the case when the request in a message cannot be serviced immediately, such as when the handler fails to secure a semaphore. Instead of blocking subsequent messages or rejecting the request outright and incurring the cost of a retry message, it may be more efficient for the handler to append the message to a retry-queue in memory and move on. In this example, the receiver must however be assured that the sender will not interfere with other requesters by corrupting the queue, even though it is allowed to write into the data structure. To permit such high-level remote operations without compromising robustness, *trusted handlers* are introduced in the M-Machine.

A trusted handler is a routine that is certified to be safe, in that it honors the protection model, never blocks indefinitely nor causes unrecoverable errors, and is generally "benign". Trusted handlers are enforced in the M-Machine with the guarded pointer system. The `HandlerIP` in every user message is required to be an instruction pointer of type *ExecuteMessage*. An exception is triggered if the hardware detects an invalid `HandlerIP`. The *ExecuteMessage* pointer specifies the entry-point to a trusted handler, but is otherwise not an executable pointer. The `HandlerIP` is mutated into a executable pointer only when it is injected into the network. This restricts trusted handlers to be accessible only via the message system, allowing simplifying assumptions, such as specific stack configurations, to be adopted by the handlers. Note that the unforgeability of guarded pointers enables the system to permit only starvation-free handlers, as well as *independently* control the remote operations accessible to *each* process. Nominally, the operating system provides several stock handlers to user

threads, such as

- `Fork` – spawns a new thread at the destination,
- `RemoteRead` – returns the datum from a memory location on the destination.
- `RemoteWrite` – updates a remote memory location on the destination.

The `Fork` handler may simply install the new thread directly into an unoccupied user slot if one is available, or place it into the scheduling queue for later execution. The `RemoteRead` and `RemoteWrite` handlers on the other hand must be written more carefully, to avoid undesirable interactions [3] with the coherent, distributed shared-memory (DSM) system that is implemented in software on the M-Machine [49]. Such interference can be prevented easily, however, by simply insulating the addressing domains for message-passing and shared-memory programs from each other.

While more sophisticated operations may also be provided by the operating system, the full advantage of a handler-based interface is realized through user-defined handlers that are optimized for specific tasks. Such user-level handlers may be permissible, as long as they also exhibit the properties required of a trusted handler. Although it is impossible to ascertain the "trustworthiness" of an arbitrary code fragment – the general halting problem is undecidable [55] – a user routine may be deemed trusted if it is carefully written to meet certain simple restrictions unambiguously, which can be checked by the compiler. Several examples of such sufficiency conditions are included below:

- The handler does not expose any protected data to the user.

  Protected data, be it owned by the system or an unrelated program, can be accessed only via a pointer. If all operations in the handler involve only constants and operands from the message body, and no pointer-creating `setptr` instructions are used, then no protection violation can occur, and the handler cannot

[3] For example, the virtual memory page being referenced by the handler may not be a shared copy with the appropriate access mode, or may not even be present on the destination node. A deadlock condition may arise if the message handler blocks while the DSM system attempts to bring in the memory page via the network.

corrupt any system-level data structures. Alternatively, software fault isolation techniques such as *sandboxing* [56] may also be used to contain the handler within its own domain, while secure procedures written carefully by the system programmer and entered through protected *EnterSystem* pointers, can be used to provide the handler with controlled access to critical data structures, such as the scheduling queue.

- The handler does not block indefinitely.

  In the M-Machine, an instruction stalls when referencing an *empty* register [4]. If all instructions can be shown to reference only registers that have previously been updated, no such stalling is possible. In order to prevent failed memory operations from blocking the handler, the compiler/assembler may insist that only special `LDSU` (synchronizing load, unfaulting) and `STSU` (synchronizing store, unfaulting) instructions [54] are used in a handler – these are special M-Machine memory instructions which return an error condition code instead of causing a fault if the operation cannot be completed successfully. To avoid complicated deadlock analysis, SEND operations should also be simply proxied to a non-privileged thread running in a separate thread slot. Meanwhile, the risk of infinite loops may be disregarded if a small, constant number of iterations can be clearly established for each `jump` and `branch` construct.

- The handler dutifully dispatches the next message.

  Once a handler is recognized to be non-blocking, a secure `DispatchNext` code fragment can be appended to process the next message upon termination of the handler. Naturally, to prevent handlers from illegally reading the next message, each handler must be allowed to access the `MsgHead` register exactly once, *i.e.* upon exit.

Note that plenty of flexibility remains even with these restrictions. Sophisticated

[4]The instruction also stalls if a needed resource such as the FALU, is busy. However, resource availability is beyond the control of the handler. In any case, under the fair arbitration scheme in the SZ unit, an instruction cannot be blocked indefinitely due to resource conflicts.

user-level handlers such as `Fetch&Add`, `Test&Set` and `Update&Forward` are conceivable, and the user may operate on its own data structures, as he should be. In fact, if the above properties are clearly established, the user-level handler does *not* even need to be unfaulting, since only the caller program can be affected by the exception condition, and the user can already sabotage himself without having access to the message system anyway. Any exception encountered in a message handler, such as divide-by-zero or illegal pointer reference, thus simply causes the routine to be killed, and the next message to be dispatched.

5.2.4 Malformed Messages

In addition to restricting the data objects and operations accessible to each process, the protection system must also guard against malformed messages, which can cause erroneous behaviors in an otherwise trusted message handler. To guard against messages that are shorter than unexpected, the M-Machine pads the message with an infinite stream of null values when `MsgBody` is read beyond the end of a message [5]. This prevents the handler from becoming blocked forever, waiting for the arrival of the non-existent arguments that it expects.

As a precaution against messages that are too long, the message interface discards the remaining words of the current message when `MsgHead` is read. The message handler therefore cannot be tricked into accessing the next message in the queue, nor using any stale data from the previous message. As a last defense, the flush feature associated with `MsgHead` also allows the system to easily reset the message queue to a known good state, should it become necessary to recover from a message handler exception.

[5] The message interface recognizes the end of a message by the *tail* tag in the last arriving word. Only reads beyond this last word are padded with null values. If the handler attempts to read `MsgBody` when the incoming message queue is empty, but the *tail* tag has not been encountered, the read operation is stalled until the next word arrives.

5.2.5 Limitation

Although the M-Machine constrains both the objects and remote operations accessible to each thread, it does not associate the permitted operations with particular objects. This creates a loophole for users to mix-and-match the data objects and operations accessible to them. For example, an ENQUEUE handler may be called with a data structure that is not a queue at all. Nonetheless, because the protection information embedded in the guarded pointers is honored universally, this limitation may be acceptable under most circumstances. In particular, if a user owns a *ReadWrite* pointer to an object, then it can already corrupt the object without any help from the message system. Conversely, if the user has only read permission, it would generally not be able to write to the data object through a handler that honors the protection model. The problem arises when the message handler has more powerful capabilities than its caller, such as the ability to mutate the type field in guarded pointers. In this case, the handler *must* be matched to the intended target object. While this stricter protection model is not implemented inherently by the guarded pointer system, a simple software solution is possible. For example, suppose handler **H** may only be invoked with data object **D**. Instead of granting access to **H**, the system may hand out handler **H'** and a a read-only pointer to the tuple **[H', D]**, which the user then sends along with the message. **H'** can then quickly authenticate itself and the data object specified in the message against the tuple before proceeding with its task.

5.3 Alternatives

Many systems have adopted user-level network interfaces. Few however provide the same level of efficiency, flexibility, and robustness afforded by the M-Machine. The very fast message systems in the CM-5 [28] and the J-Machine [10] feature a streaming interface which allows the user to tie up the network port indefinitely by initiating and then failing to complete a message injection. To avoid starvation, software conventions and workarounds are necessary in these designs. The CM-5 typically relies on a *gang-scheduling* scheme, which time-slices the machine and drains the network at the end

of every interval, while the J-Machine merely counts on the user to release critical resources promptly.

The more robust FUGU system [30] prevents message interception in its user-level network interface by identifying each message and thread with a hardware stamp. A message is presented to the current process at the destination only if their stamps match up, and is referred to the operating system otherwise. To prevent starvation, FUGU also allows the OS to revoke *interrupt-disables* which mask out message-arrival signals, and regain control if incoming messages are blocked for too long. Nonetheless, frequent operating system intervention may degrade performance unless the scheduling of communicating processes are carefully synchronized across nodes, due to the explicit send-receive messaging model. The *timeout-interrupt* solution can also be effective against resource monopolization in a more general sense, but only if the architecture provides suitable mechanisms for cleaning up after a misbehaving user process is forcibly displaced. It is fairly straight forward, for example, to interrupt a sender thread in a streaming injection interface if it fails to complete its message within a predetermined period of time. To safely reclaim the critical resources, however, the system must be equipped with the ability to explicitly reset the injection port and notify the receiving end of the exceptional condition. In addition, the sender thread must also be prevented from blindly accessing the resources to which it no longer owns, if it is not terminated out right when interrupted. Especially when used in a multithreaded architecture like the M-Machine, the timeout value must be chosen carefully, so that a thread is unlikely to be interrupted prematurely just because of the normal variabilities in its execution timings.

Flexibility is a major factor for the many implementations of the Active Message interface [35, 57, 23]. Robustness however is not. The protocol uses integer tags to match up messages with their intended destination nodes. This reduces the likelihood for an inadvertently misdelivered message to be accepted at the destination, but can give no guarantees due to the unprotected integer tags. In any case, the message interface privileges are open to abuse, as no facility is provided for regulating user-level message handlers. System control over message handlers are accomplished on

the FLASH system [41] via the virtual address translation layer in its virtual-memory mapped message interface. A handler is accessible to a user only if the corresponding entry point is mapped into the user's virtual memory domain. User-level handlers are however not supported.

In general, a simple protection model is easily obtained from the virtual-memory system by annotating the page tables with permissions such as *ReadWrite* and *Read-Only*. Illegal accesses can then be blocked in the address translation layer. However, this approach becomes complicated when the user may encapsulate an address within a message to be accessed in a remote processor. At the destination, before the legality of the access can even be determined, the system must fetch the relevant access control information, potentially from yet another processor. By embedding the permission into the guarded pointer, the M-Machine makes this overhead unnecessary.

In message interfaces that are mapped into the virtual-memory system, such as on SHRIMP [16], the system can remap critical network resources to fresh virtual addresses from the huge virtual memory space. This makes the message system less sensitive, but by no means immune, to starvation. In the end, such systems must rely on the user to release the allocated resources voluntarily. Even then, the system must ensure that the user genuinely no longer has access to a resource it has released. Otherwise, new messages that use the re-allocated resource would be open to corruption. The same problem is in fact inherent to message system that share the message buffer among processes. In the Alewife [15], a specialized fast buffer is provided for message assembly. When a process is swapped out, any data left in the message buffer becomes exposed to the next process that occupies the thread slot. This is however not an issue in the M-Machine, since the register files containing the message buffer is naturally overwritten in the process of swapping in a new thread.

5.4 Summary

The multicomputer system has to enforce protection consistently across node boundaries. In doing so, it should also minimize overhead and authentication-related net-

work traffic. These requirements are compromised in many user-level message interfaces, which expose critical shared resources to untrusted user processes in exchange for efficiency. In addition, starvation risks are created when users are allowed to monopolize the resources. The M-Machine however accomplishes a flexible and efficient user-level message system, without sacrificing robustness.

Based on the guarded pointer system, the M-Machine provides a fine-grain protection model, and facilitates fast domain switching using *EnterUser* and *EnterSystem* pointers. Primitive *ReadOnly/ReadWrite* capabilities embedded in the guarded pointer are honored universally, while high-level, user-defined remote operations can be supported through *trusted handlers*. The system prevents abuse of network interface privileges by creating *ExecuteMessage* pointers only for safe, starvation-free handlers. At the sending end, dedicated message buffers embedded into the register files of each thread ensure that messages are insulated from unrelated processes. Starvation is also avoided, since the network port is allocated to each thread for only a bounded period during atomic injection. With the transparent translation facility provided by the GTLB, users are also prevented from maliciously misdirecting the virtually-addressed messages. At the receiving end, the behaviors of the `MsgBody/MsgHead` interface are specifically designed to thwart all attempts to confuse the message handler using messages of the wrong size.

In short, by making robustness considerations an inherent part of the design, the M-Machine serves as a model for user-level message interfaces, which *can* be efficient and flexible, *and* remain protected.

Chapter 6

Network Interface Primitives and Communication Overhead

The message interface defines how programs interact with the messaging facility. Traditionally, messaging operations are managed by the operating system. User-level programs copy data into a memory buffer and make a system-call when they want to send a message. They then wait passively for the operating system to copy the message into the network hardware to be transported to the destination, to be eventually copied back into a memory buffer accessible to the users. This model is very inefficient due to three reasons – excessive copying, high context-switch overhead, and inflexible handling of message arrivals. For example, Burns *et.al.* [58] show that the conventional message passing protocol can be reduced from seven steps to one by avoiding copying the message from one buffer to another whenever possible, while Hsu and Banerjee [19] accounted that 18% of the 85% software overhead in sending an OS-managed short message in the Intel iPSC/2 is due to context switches. By exposing more of the primitive messaging facilities to the user and employing a more flexible handler-based Active Message dispatch model – which quickly incorporates the message into normal computation at the destination, Eicken *et.al.* [35] demonstrated that nearly an order-of-magnitude reduction in communication overhead can be achieved in the nCUBE/2. As a result, modern message systems have largely abandoned the old model, in favor of user-level mechanisms. Nonetheless, to actually

result in an efficient system, the messaging primitives must still be designed carefully to complement one another, so as to avoid overlaps and oversights in functionality.

This chapter is focused on such direct, user-level messaging interfaces that feature a handler-based dispatch model. The primitives design space for the modern message interface is considered along three axes: (a) mapping – how the interface is presented to the software, (b) atomicity – whether messages are transfered uninterruptibly, and (c) dispatch – how the message is incorporated into computation at the destination. When compared to a conventional interrupt-driven, fully-buffered, memory-mapped design, the M-Machine's register-mapped, atomic injection and dedicated-threadslot dispatched, streaming extraction interfaces are shown to be an order-of-magnitude faster. Results indicate that the dispatch mechanism choice contributes 60% of the overhead reduction, while the mapping and atomicity decisions together account for 30% of the improvement. The remaining 10% of the speedup is due to fast address translation in the GTLB.

The message interface models used in this study are described in the next section. The micro benchmark programs and experiments then follow in Sections 6.2 and 6.3. Section 6.4 records the raw results, which are then analyzed in detail throughout Section 6.5. The impact of each choice in the design space of the network interface is examined, exposing the high cost of sending and receiving messages through memory-mapped interfaces, the pitfall of register-pressure in register-mapped designs and the tradeoffs between buffered and streaming interfaces. The advantage of multithreaded dispatch mechanism over an interrupt-based or polled system, and the impact of an efficient address translation mechanism are also discussed. In section 6.6, the effects of combining the features used in the M-Machine are examined. A summary is presented as the chapter ends in Section 6.7.

6.1 Message Interface Models

In general, the design space for these modern message interfaces can considered from three aspects: (a) *mapping*, (b) *atomicity* and (c) *dispatch* (Figure 6-1). The

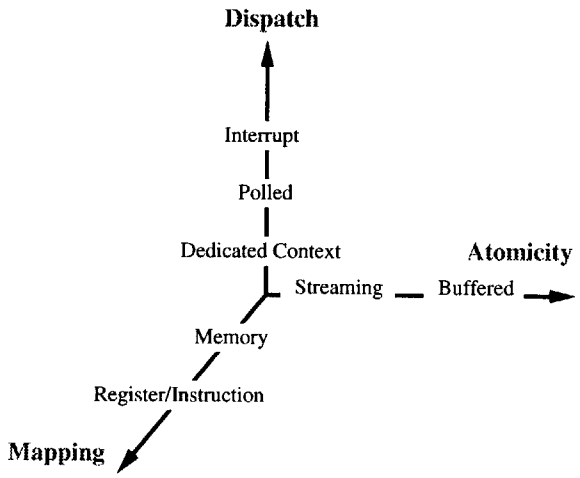

Figure 6-1: Messaging Primitives Design Space

mapping of an interface determines how it is accessed by user programs. An interface may be mapped into the *memory*, such that messaging operations are performed by reading and writing certain reserved regions of the memory address space. Alternatively, messages may be sent and received by accessing *network-device registers*, or using specialized *instructions* native to the architecture. The former is popular among designs based on off-the-shelf components, since it is easily implemented by connecting up a network controller to snoop on the memory bus. Messaging operations in such designs however suffer from the overhead of traversing increasingly-complex on-chip cache hierarchies. The latter category of interfaces are more efficient, but must be integrated tightly with the processor, and are therefore only feasible in custom-designed chips. For this study, I consider instruction-mapped designs as only a variant of the device-register mapped interface, since both consume processor cycles in moving the message words between the network device and processor-registers, *i.e.* the regular registers that serve as operands to all instructions. The M-Machine interface on the other hand is processor-register mapped. Its network-connected registers can be used as the source or sink in any of its instructions.

In terms of atomicity, a message can be injected or extracted either in a piece-wise fashion, or as an un-interruptible unit. A piece-meal, *streaming* interface allows the first word of the message to "worm-hole" through the message system without waiting

for the rest of the message. Conversely, a *buffered* interface requires each message to be completely stored into a buffer before it is injected or made available for extraction. Due to this waiting, atomic interfaces tend to be slower, but are also more robust since a message is either completely delivered, or not. In a streaming interface, the network may become blocked if the sender or receiver stops in the middle of injecting or extracting a message.

The dispatch mechanism determines how the processor is notified of message arrivals. Conventionally, an *interrupt* mechanism asynchronously and forcibly displaces the current program with an interrupt handler upon message arrival. The resident program may also periodically *poll* the network interface, and jump to a message handler if a message has arrived. In either case however, a very high overhead is incurred as the resident program is swapped out to make room for a handler. This can be avoided by providing *dedicated* hardware resources, such as a co-processor in the case of [19], for the message handler. The usually low utilization of the handler context however may not justify a full-featured processor. Thus, in the M-Machine, only a thread slot on cluster 1 and cluster 2 are reserved for the message system.

A set of micro benchmarks are run on a number of network interface models that represent different points in the design space. In addition, the efficacy of the global translation lookaside table is also quantified in the these experiments. The models are listed in Table 6.1. Since they each explore a particular aspect, these models may not correspond exactly to existing architectures. However, familiar points of reference include the CM-5 [28] (memory-mapped streaming injection and extraction), the *T [18] (register-mapped buffered injection and extraction), the J-Machine [10] (register-mapped streaming injection [1] and extraction), and SHRIMP [16] (memory-mapped buffered injection and extraction). The M-Machine message interface architecture is represented as MM.

The experiments are conducted on *msim* [59], a cycle-accurate simulator of the M-Machine architecture written in C. Experiments on the IRS and IMS models are

[1]The J-Machine uses a SEND instruction to move each message word into the network. Effectively, this is equivalent to writing each message word into a special network-mapped register.

Simulation Model	Differs from MM Architecture	Mapping		Atomicity	
		Memory	Register	Buffered	Streaming
IMB	injection	√		√	
IMS	injection	√			√
IRB	injection		√	√	
IRS	injection		√		√
EMB	extraction	√		√	
EMS	extraction	√			√
ERB	extraction		√	√	
ERS	extraction		√		√
SWX	translation	Software-based address translation			
INT	dispatch	Interrupt-driven dispatch			
POL	dispatch	Polled dispatch			

Table 6.1: Basic Network Interface Models

made possible by a special feature incorporated into *msim* so that it can optionally emulate a streaming injection interface that is mapped to register *i15* of the user program's register file. For IMS and EMS, extra cycles are added to the benchmark programs to account for the overhead associated with memory-mapped messaging operations. Similar instrumentation is done to the code for IRB and ERB to account for buffering delay. Meanwhile, to emulate the SWX model, additional code is added to the programs to perform explicit address translation using a 4-entry lookup table in memory.

In experiments involving POL and INT, a simple counter program is installed in the message handler thread slot in *msim*, alongside with the benchmark program. This counter program represents the user application that must be displaced by the message handler whenever a message arrives in a polled or interrupt-driven dispatch mechanism. In POL, the counter program is augmented with a short polling routine that is invoked once every 120 cycles, for a ~15% null-poll overhead. When a message arrival is detected, the volatile state in the counter program (6 registers in these experiments) is spilled, and a message dispatcher is swapped in. The dispatcher then jumps to the message handler specified by the message, and eventually restores the counter program when the handler completes. To emulate the INT model, an infinite-

Model	Translation	Injection	Extraction	Dispatch
	mechanisms similar to			
SWX_IMB_EMB_INT	SWX	IMB	EMB	**INT**
SWX_IMB_EMB_POL	SWX	IMB	EMB	**POL**
SWX_IMB_EMB	SWX	IMB	EMB	**MM**
SWX_IMB_EMS	SWX	IMB	**EMS**	MM
SWX_IMS_EMS	SWX	**IMS**	EMS	MM
SWX_IMS	SWX	IMS	**MM**	MM

Table 6.2: Incrementally-Enhanced Models (Memory-Mapped)

Model	Translation	Injection	Extraction	Dispatch
	mechanisms similar to			
SWX_IRB_ERB_INT	SWX	IRB	ERB	**INT**
SWX_IRB_ERB_POL	SWX	IRB	ERB	**POL**
SWX_IRB_ERB	SWX	IRB	ERB	**MM**
SWX_IRB_ERS	SWX	IRB	**ERS**	MM
SWX_IRS_ERS	SWX	**IRS**	ERS	MM
SWX_IRS	SWX	IRS	**MM**	MM

Table 6.3: Incrementally-Enhanced Models (Register-Mapped)

loop routine, in addition to the counter program above, is installed in one of the *msim* hardware thread slots otherwise unused by the benchmark program. This "daemon" routine emulates interrupt-detection hardware by continuously checking for message arrival. When a new message is detected, this daemon simulates a system interrupt by saving the entire state of the counter program and swapping in an interrupt handler in its place. The interrupt handler then dispatches to the appropriate message handler, and eventually restores the counter program when the message handler is done. Care is taken to discount the extra cycles introduced by the daemon thread in the final results.

In a second set of micro benchmark experiments, starting with the MM model, a single feature is removed and measurements are taken for each of the programs. The procedure is repeated until the last experiment is run on a classic buffered, interrupt-based interface. When the results are considered in reverse order, this set

of experiments provide a measurement of the incremental benefit that each of the described mechanism brings to the MM architecture. The models involved in these experiments are listed in Table 6.2 and Table 6.3.

6.2 Benchmark Programs

Four micro benchmarks are devised for the experiments, namely PING, RPC, DIST, and BLKW. Each benchmark program has distinct characteristics that capture certain message operation patterns.

- PING

  The PING benchmark involves two nodes and is characteristic of communication patterns with very short messages. It is a simple *ping-pong* program, where an empty *ping* message from node A is sent to node B, causing the latter to respond with an empty *pong* message to node A.

- RPC

  The RPC benchmark involves two nodes, and represents a typical remote procedure call operations. In this benchmark, an *rpc* message is sent from node A to spawn a new user thread on node B. Eight arguments are passed in the *rpc* message, in addition to the initial instruction pointer (IP) of the code to be spawned.

- DIST

  The DIST benchmark involves the distribution of eight *rpc* messages to eight distinct nodes. Each message carries 8 arguments in addition to the IP to be spawned. In this benchmark, four of the arguments are identical across all eight messages, while four others are different, so as to model the distribution of parallel threads across a number of processors with a mix of similar initial data and different seeds that direct the computation path of each child thread. Since the receivers are able to overlap the extraction of their corresponding messages

with one another, this benchmark is less sensitive to changes in the mechanisms at the receiving end.

- BLKW

  The BLKW benchmark models a large memory-to-memory transfer between two nodes. A total of 1024 words are moved from node A to node B, explicitly packetized into small (8 to 10-word) messages when appropriate.

6.3 Experiments

Experiments are conducted for each of the programs on a spectrum of network interface models, and network interface overhead is measured, in terms of *latency* and *processor occupancy*. For these simulations, a matched bandwidth of 1 word/cycle is assumed for both the network and the memory system.

Latency is the measure of how much time it takes to propagate the relevant information through the message system. This includes the time to format the message, to effect the transfer, and then to receive and respond to the message at the other end. To isolate the effects of the network fabric, end-to-end latency is measured in these experiments using a zero-latency network model. In PING, latency is measured from the initial creation of the *ping* message, to the time its corresponding reply is written to memory in the originating node, while for RPC, it is measured from the sender's initial assembly of the *rpc* message, to the first instruction execution in the newly created thread on the neighboring processor. It is measured from the initial call to DIST to the first instruction execution in the last child thread, and from the initial call to BLKW to the time when the last transferred word is written into memory on the destination.

Processor occupancy is the number of instruction issue opportunities consumed by the message operations, which is otherwise available for productive computation. Occupancy comes from the execution of instructions needed explicitly to format, inject, dispatch and extract the message, and other message-related functions such as address translation, flow control and protection enforcement. Measurements of

processor occupancy are taken from each benchmark by counting instruction-issue opportunities that are consumed or otherwise made unavailable by all message-related operations, at both the sender and the receiver nodes.

Latency and occupancy are related but distinct metrics of network interface overhead. Occupancy may be reduced, without changing the latency, if the network interface features special hardware to offload some messaging functions from the processor On the other hand, a less efficient queuing unit in the network interface may lengthen the overall latency, without having an impact on occupancy. While successful in masking latency, pre-fetching and other similar techniques are generally ineffective against processor occupancy. In any case, both forms of overhead are severely affected by any additional software layers imposed on the message sender/receiver, such as in architectures where the interface between the user program and the message facility is accessed by trapping into the operating system.

6.4 Results

The results for each of the experiments are recorded in Table 6.4. The latency results are measured end-to-end, while the occupancy numbers account for all operations issue-slots consumed by message-related operations at both injection and extraction interfaces. The results are optimistic for SWX and POL. For SWX, since the table is small (4 entries), only a simple sequential lookup algorithm is used. The results shown are based on the assumption that the matching translation is found in the first entry of the lookup table. Each missed entry costs an additional 16 cycles of latency and 22 cycles of occupancy in this lookup process [2]. The POL results shown are collected for the optimistic scenario where message arrival occurs exactly when the polling is done. A penalty of 121 cycles of latency is incurred whenever a message barely misses a poll and must wait for the next poll to be dispatched.

[2]Since multiple function units are used in the benchmark programs, it is possible for the the occupancy (the number of operations consumed) to be greater than the latency.

Interface Models	Latency (cycles)				Occupancy (cycles)			
	PING	RPC	DIST	BLKW	PING	RPC	DIST	BLKW
MM	38	44	185	2665	28	42	223	4691
IRS	35	38	209	2621	30	52	303	5069
IRB	43	53	167	3415	33	51	260	5684
IMS	117	68	492	7068	46	59	317	5790
IMB	103	99	595	9253	47	59	281	5882
ERS	39	47	198	3116	31	54	349	5716
ERB	55	62	203	6292	29	51	325	5612
EMS	87	77	225	6840	31	54	349	5818
EMB	109	90	231	10094	33	55	357	5819
SWX (entry 0)	79	85	535	3051	74	90	622	6875
POL (optimistic)	76	66	211	2684 [a]	–	–	–	–
INT	292	189	333	2792 [b]	–	–	–	–
SWX_IMB_EMB_INT	476	341	943	–	–	–	–	–
SWX_IMB_EMB_POL	254	251	855	–	–	–	–	–
SWX_IMB_EMB	219	198	939	–	–	–	–	–
SWX_IMB_EMS	207	188	885	–	–	–	–	–
SWX_IMS_EMS	199	142	867	–	–	–	–	–
SWX_IMS	156	109	844	–	–	–	–	–
SWX_IRB_ERB_INT	350	248	697	–	–	–	–	–
SWX_IRB_ERB_POL	142	139	560	–	–	–	–	–
SWX_IRB_ERB	100	114	537	–	–	–	–	–
SWX_IRB_ERS	85	97	520	–	–	–	–	–
SWX_IRS_ERS	82	82	555	–	–	–	–	–
SWX_IRS	76	79	552	–	–	–	–	–

Table 6.4: Micro Benchmark Results

[a] Derived from PING_{MM}, PING_{POL} and BLKW_{MM}

[b] Derived from PING_{MM}, PING_{INT} and BLKW_{MM}

6.5 Impact of Individual Design Choices

To understand the effects of each particular design choice, the micro benchmark results are compared in terms of (a) mapping, (b) atomicity, (c) address translation facility and (d) dispatch mechanisms.

6.5.1 Interface Mapping

The effects of mapping choices for the network interface include different levels of memory overhead, redundant copying and register pressure. The end-to-end latency and occupancy results from above are graphed in Figures 6-2, 6-4, 6-3 and 6-5 for interfaces with different mapping options. A program communicating through memory-mapped interfaces must generate/retrieve the appropriate addresses before it can perform any messaging operations. To prevent resource contention with other programs, it usually must also secure semaphores that are typically implemented as data structures in memory. These extra operations consume execution cycles on the processor, resulting in processor occupancy that is up to 1.6× that of the MM reference model (Figures 6-4 and 6-5). Since each message word has to traverse the on-chip memory hierarchy in a memory-mapped interface, which can take 10 – 30 cycles in aggressively pipelined systems [3], the resulting latency is also significantly higher, up to about 3.5× when compared with corresponding integrated register-based mechanisms (Figures 6-2 and 6-3).

Although an integrated, instruction/register-mapped interface can be accessed directly without address-creation overhead, its performance is limited as the typical implementation requires each message word to be copied explicitly between specialized network interface registers and general registers, the latter being usable as operands to regular instructions. This copying overhead is avoided in the MM architecture, where the injection buffer, as well as both `MsgHead` and `MsgBody`, are accessible through the general register name space. The message words are bypassed directly from the

[3]For example, in implementing Active Messages on the CM-5, Eicken *et.al.* [35] found that the largest fraction of send/receive time is spent accessing the network interface across the memory bus.

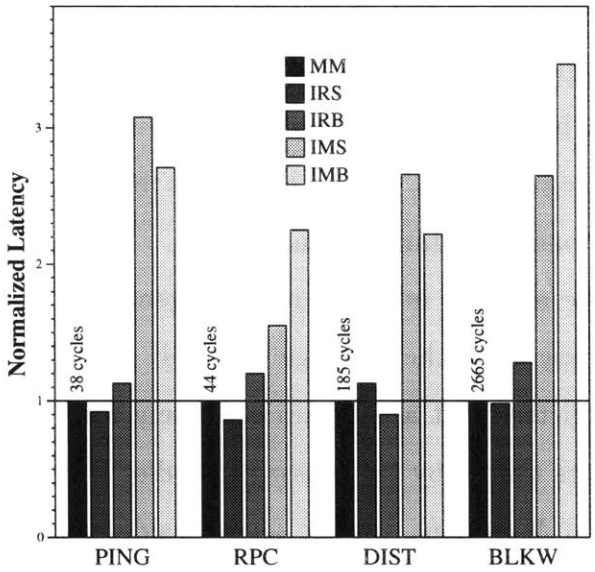

Figure 6-2: Injection Mechanisms: End-to-End Latency

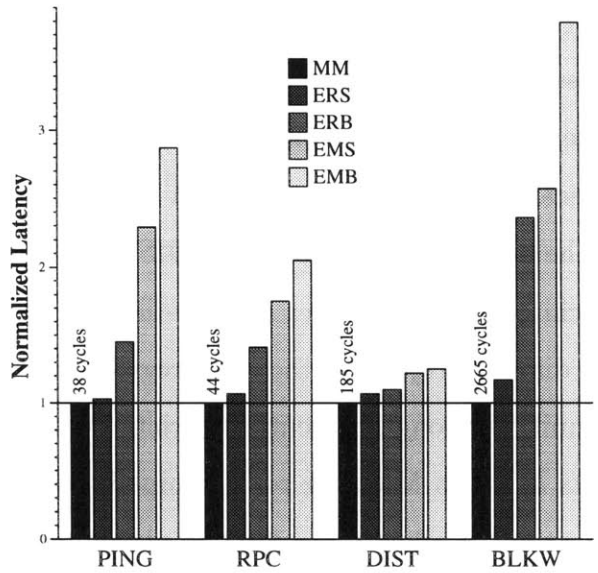

Figure 6-3: Extraction Mechanisms: End-to-End Latency

message queue into the execution unit. As a result, it commands an advantage over the other corresponding register-mapped models (IRB, ERS) in Figure 6-2 through 6-5. Because this advantage is proportional to the message length, it does not show up as significant differences in the charts due to the short messages used in these benchmarks. Although some, such as ERS, achieve a latency that is comparable to MM, they rely on having multiple function units for overlapping the copying of message words with computation. These interfaces must however pay for that with higher processor occupancy (Figure 6-4 and 6-5).

While mapping the network interface into the general-register name space eliminates unproductive copying of the message words, it effectively reduces the number of general-purpose registers that a program can freely use. In the MM architecture, the number of registers usable by the message handler is reduced by two to accommodate $MsgHead$ and $MsgBody$. On the sender's side, mapping the message buffer into the general register file in fact causes appreciable register pressure in, for example, the DIST benchmark, due to the small register file in the MM architecture. In this benchmark, four arguments have to be regenerated for each message. With the current message occupying 10 registers, too few registers are left to overlap that computation with message injection. Although MM can pipeline messages by sending them from the integer and floating-point register files alternately, this capability is limited in DIST because the floating-point unit does not support all of the instructions needed by the message-update phase in DIST. This latter shortcoming can be alleviated in an architecture with larger register-files, by allowing messages to be pipeline-injected from different *regions* of the same register-file that is attached to the appropriate function unit. Although currently restricted by the fixed message buffer mapping (always starts at $i4$) in MM, the revised pipelining method can be enabled with a simple modification to allow more flexible placement of the message buffer within each register-file, *e.g.*:

```
SEND <begin>, <length>, <DestAddr>, <HandlerIP>, <Ack>
```

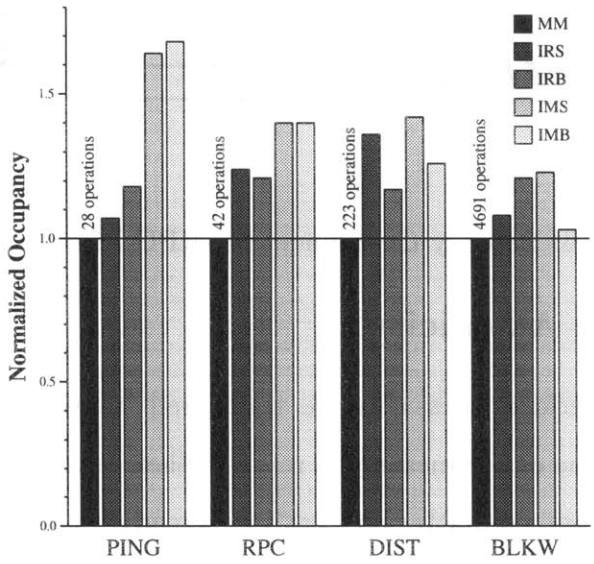

Figure 6-4: Injection Mechanisms: Processor Occupancy

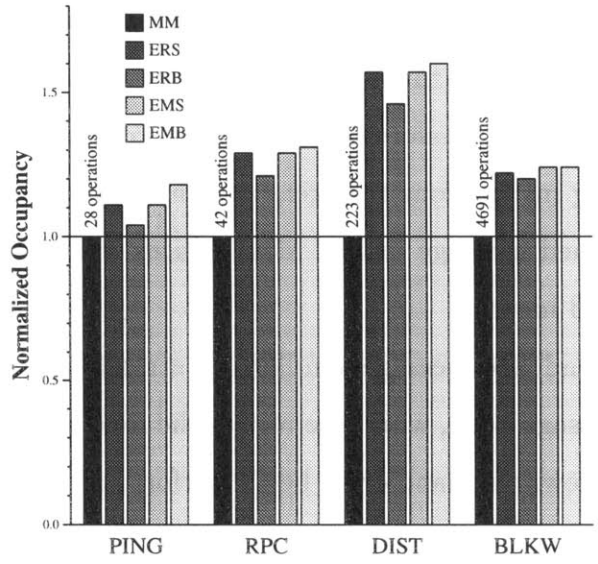

Figure 6-5: Extraction Mechanisms: Processor Occupancy

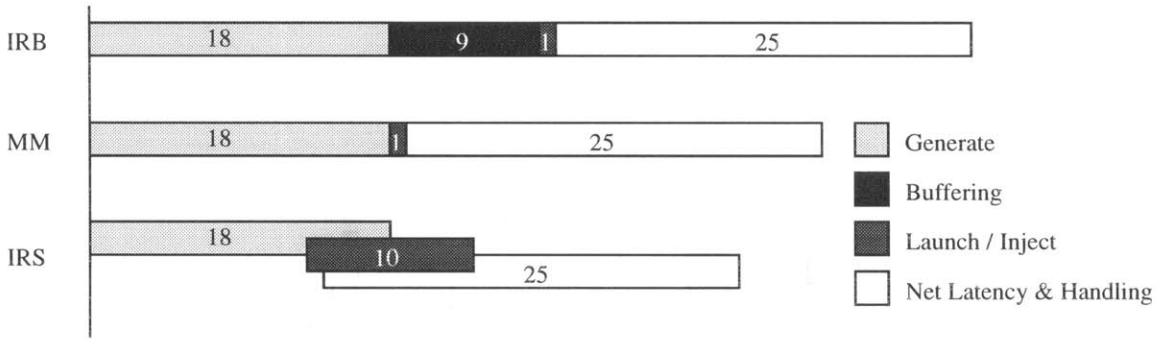

Figure 6-6: Latency Components in RPC Message Injection

6.5.2 Message Atomicity

In a buffered interface, messages are transfered between the network and the program only when the entire message is ready. When sending a message, this forces the head of the message, no matter how early it is ready, to delay its progress through the network by a duration that is the cumulation of all latencies incurred in the generation of each word. While it is usually collapsed via pipelining techniques, this delay can remain significant in the presence of cache-misses and other long latency events. Such an effect is illustrated in Figure 6-6, where the RPC benchmark latency is shown in detail for the the three integrated interfaces. Both MM and IRB are buffered interfaces, and are therefore penalized when compared to the streaming IRS. However, by targeting instructions that generate the message words to write directly into its register-mapped message buffer, MM reduces the overhead by avoiding the copying cost that IRB incurs.

On the receiving end, the message handler in a streaming interface activates upon the arrival of the message head, while the rest of the message slowly dribbles in from the network. This allows the handler to begin performing the requested task immediately if the necessary code and data are already cached, or at least starts bringing them into the cache. As a result, streaming extraction interfaces have a latency advantage over their buffered counterparts. The streaming extraction interface in MM also adds the side-effect of removing a message word from the incoming message queue as it is used in an instruction. Since most of the payload in a message is used only

once, this optimization streamlines the extraction process, further reducing processor occupancy (Figure 6-5).

There are however conditions under which streaming interfaces do not fare well. With very short messages, the fixed overhead for opening a streaming channel into the network dominates the per-word buffering overhead, as demonstrated in the memory-mapped PING results in Figure 6-2. A streaming interface is also unable to exploit message reuse. Therefore when the message assembly time can be amortized over several messages such as in DIST, IRS become slower than MM and IRB. In BLKW, the latency results are similar among the three tightly-integrated designs, as a result of aggressive software-pipelining compensating for the buffering delay in very large transfers.

The performance difference between buffered interfaces and streaming interfaces is proportional to the message length. For fine-grain messages such as those used in the micro benchmarks above, atomic interfaces incur only a small amount – as low as 10% – of extra overhead. Considering atomicity tends to simplify not only correctness reasoning but also many practical programming issues, buffered interfaces do have their rightful place in the message system, especially when robustness concerns are involved.

6.5.3 Address Translation Facility

In classic message-passing systems, remote objects are often referenced using `(nodeID, localAddress)` tuples. This requires the sender program to have up-to-date knowledge of the physical placement of its remote objects. As a result, the system must either forego dynamic object placement and migration (for purposes such as load-balancing), or require the user programs to confirm/update their object-location database before sending messages. The former approach is too restrictive for the efficient use of parallel systems, while the latter can be costly and cumbersome without the proper supporting mechanisms.

The MM architecture remedies the translation issue with the GTLB, which is a 4-entry fully-associative hardware cache that translates a virtual address into routing

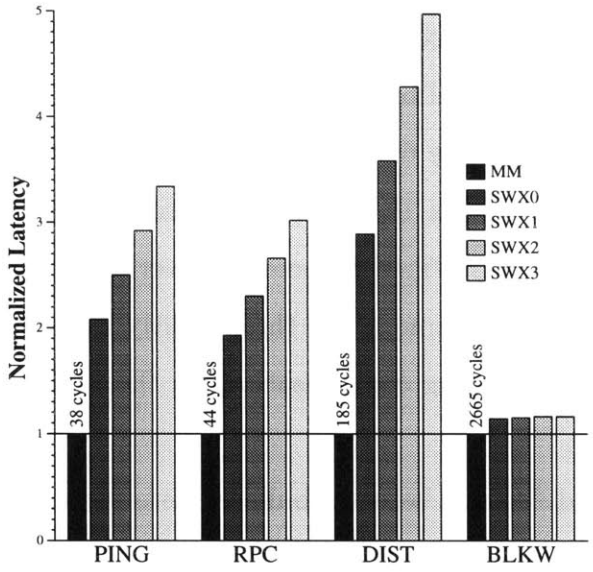

Figure 6-7: GTLB vs Software Address Translation : End-to-End Latency

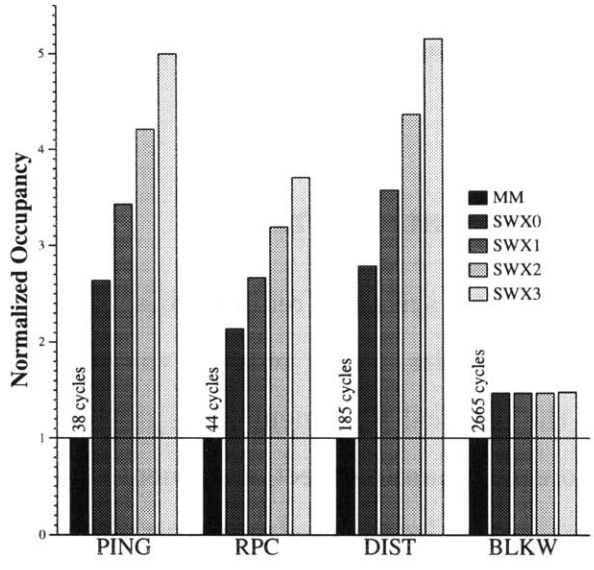

Figure 6-8: GTLB vs Software Address Translation : Processor Occupancy

information. In Figures 6-7 and 6-8, the MM model is compared to SWX, which implements the same functionality in software, without the benefit of the GTLB. Each bar labeled SWXi represents the latency/occupancy when the sequential-search SWX implementation finds a match in the i^{th} entry of the lookup table.

Since the overhead in SWX comes from executing the extra instructions in the lookup subroutine, which incur latency as well as consume processing resources, the similarity between the two charts is expected. Both figures show that the MM mechanism reduces the overhead dramatically, down to as low as $\frac{1}{5}\times$. For SWX, although only a moderate incremental overhead (16 cycles latency, 22 cycles occupancy) is added for each mismatched entry, a large cost is incurred going from MM to SWX0. This abrupt jump in overhead is in part due to the lookup instructions being implemented as a subroutine instead of being inlined in SWX (and therefore incurs procedure call overhead), but mostly due to register-pressure, which forces the program to spill a number of registers to make way for the lookup instructions.

Although the user program may save the translation result and thereby amortize the translation cost over a number of messages, it accomplishes that only by incurring the additional overhead of explicitly managing the saved result, and the cost of cleaning up after an exception should the address-location mapping become obsolete. In general therefore, the high cost of software-based translation forms a strong disincentives for fine-grain messaging.

6.5.4 Dispatch Mechanisms

A multi threaded processor, such as the MAP in the M-Machine, allows multiple threads of control to be simultaneously installed in hardware, ready for execution on very short notice. The MM network interface exploits this feature for fast message dispatch, installing in one of the hardware slots a *message dispatcher* thread that activates instantly upon message arrival. Upon activation, the message dispatcher jumps immediately to a message handler specified by the incoming message. By contrast, in conventional processors, the currently-running program must be saved away to make way for the message handler. This thread-swap process constitutes

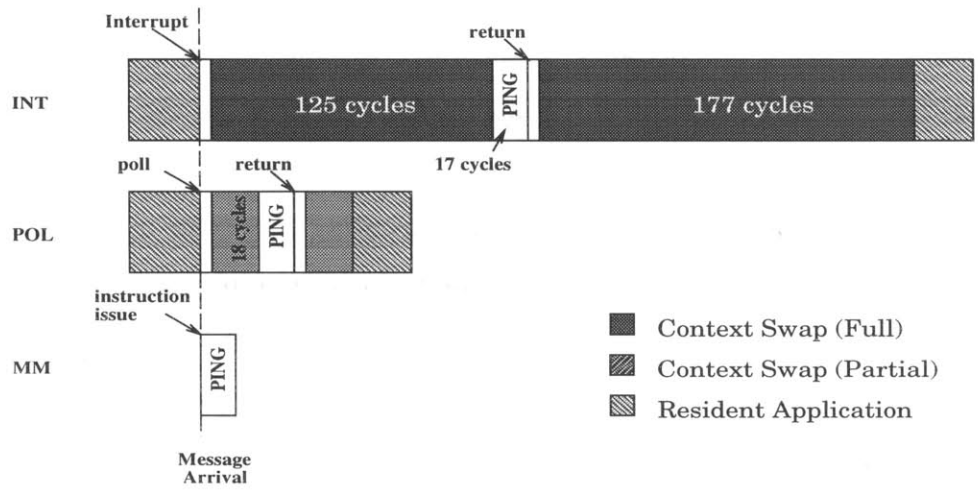

Figure 6-9: PING: Latency Components in Message Dispatch

most of the dispatch overhead in such systems.

In a polled dispatch system, the polling program, which is being displaced, needs to save away only the contents of its live registers that cannot be regenerated easily (6 registers are saved in the experiments). For an interrupt-driven architecture, however, since the system does not know exactly which of the registers are live, all of them must be saved away (32 in the experiments), resulting in a very large overhead. The end-to-end PING latency for the interrupt-driven (INT) and polled (POL) systems are shown in detail in Figure 6-9 alongside the MM model. Note that for INT, the thread swap overhead is nearly 18× the time it takes to actually service the PING request. The polled model, is less inefficient, but still results in a PING response time that is more than 3× that of MM. Nonetheless, the dispatch speed in a polled system is variable. Only the best case scenario is represented in Figure 6-9, where the message arrives exactly when the polling takes place. In the worst case, when message arrival just misses the poll by one cycle, the message would be dispatched only 120 cycles later – assuming a poll-frequency of 1 per 120 cycles.

In Figure 6-10, which shows the latency results for each of the benchmarks. The POL_MIN bar indicates the best-case scenario while POL_MAX represents the worst-case. For the INT model, the interrupt handler is designed to check for new messages

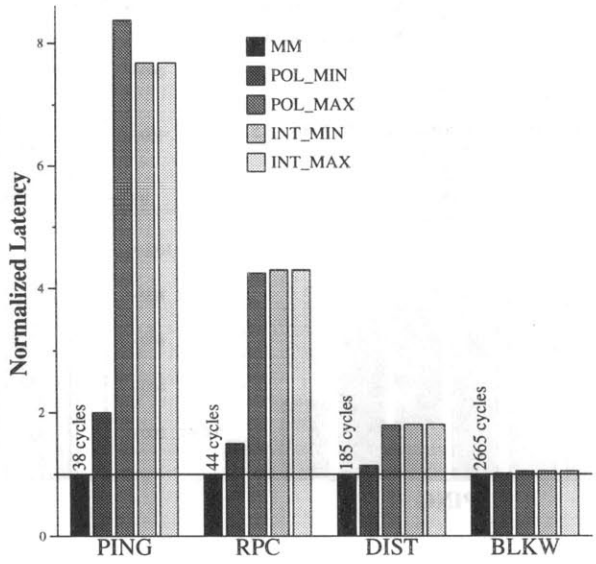

Figure 6-10: Dispatch Mechanisms: End-to-End Latency

before returning to the displaced program. The INT_MIN and INT_MAX bars in Figure 6-10 represent the best-case and worst-case scenarios where subsequent message arrivals happen to barely hit and miss this check, respectively. Naturally, the difference between INT_MIN and INT_MAX is only relevant when multiple messages are received in rapid succession, as in BLKW benchmark. It should be noted however that the INT_MIN result for BLKW is deceptively optimistic, as it achieves such good performance only by shutting out the displaced program for an extended period of time, until all 103 messages in BLKW have been received. In effect, the benchmark pays for the interrupt overhead only once, which is not a realistic scenario normal network loads, and not a desirable condition as far as the application programs at the receiving node are concerned.

The disparity between POL_MIN and POL_MAX illustrates the high variability in dispatch speed that can be expected in a polled interface, while the results for both INT_MIN and INT_MAX demonstrate the very high overhead in an interrupt-driven system. In contrast, the handler designated by the message is able to activate almost instantly in the MM interface, showing that very low overhead dispatch is very amenable with multi threaded hardware, without the higher cost of dedicated

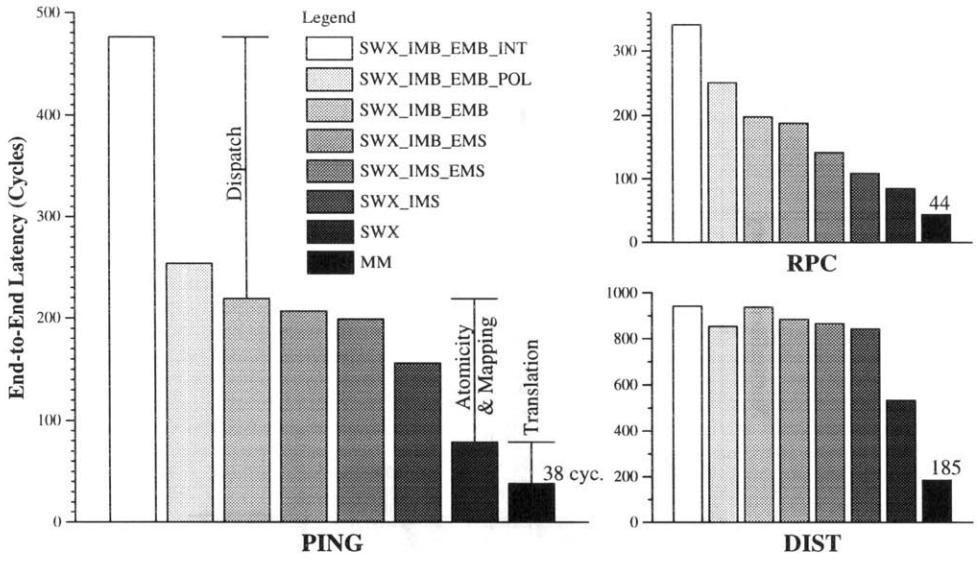

Figure 6-11: Incrementally Enhanced (Memory-Mapped) Interfaces

message handling co-processors.

6.6 Combined Effect of MM Mechanisms

The effects of incrementally combining the mechanisms in the MM architecture are shown in Figures 6-11 and 6-12. For PING and RPC, the results show that a significant portion of latency reduction comes from eliminating the context swap overhead upon message arrival. In particular, in Figure 6-11, about 60% of the improvement in PING is accomplished when interface switches from an interrupt-driven mechanism to a polled protocol, and then to the multi threaded MM architecture. Because of the small size of the PING messages, the various combinations of atomicity options do not have much impact on this benchmark, until the interface departs from the memory-mapped model for the register-mapped MM model. This change in mapping accounts for almost 30% of the performance boost. Finally, the remaining latency reduction is attributed to the fast address translation mechanism in the GTLB. In sum, the MM architecture shortens the PING latency by approximately an order of magnitude, even without considering software layers often present between user threads and the network in more classic systems, over a interrupt-driven, fully-buffered, memory-mapped

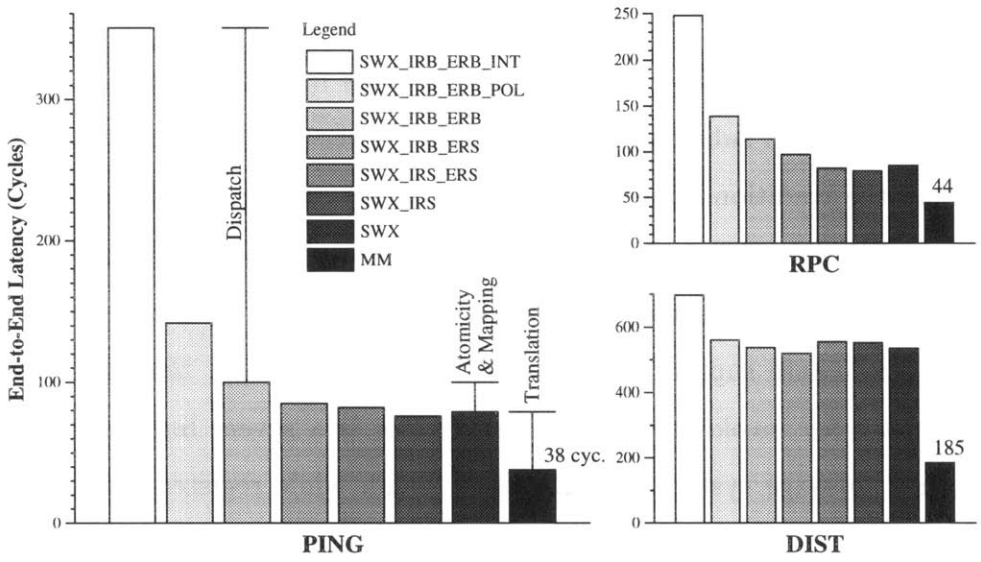

Figure 6-12: Incrementally Enhanced (Register-Mapped) Interfaces

interface with no hardware address-translation support.

The above trend is quite evident for PING and RPC in both the memory-mapped and register-mapped lineages. The results for DIST are however slightly different. As described earlier, the DIST benchmark is not very effective in highlighting the impact of receiver-side mechanisms because the eight receivers in the benchmark are able to overlap the extraction of their corresponding messages. Therefore the effects of the dispatch and extraction mechanisms do not show up as clearly in Figures 6-11 and 6-12. On the other hand, as DIST delivers messages to multiple targets, the impact of the translation facility is much more pronounced.

Common across all these results however is the fact that for fine-grain messaging with user-level mechanisms, the actual transfer of the message words between the program and the network accounts for a relative small part of the overhead. The dominant costs are due to address-translation at the sending side, and message-dispatch at the receiving side. Optimizing those two components should therefore yield substantial performance improvements. The choice of atomicity models is significant only when these two costs are minimized to the extent they are in the MM architecture.

6.7 Summary

The micro benchmarks show that the mechanisms in the M-Machine architecture collectively provide a substantial performance boost – up to an order-of-magnitude better than the traditional interrupt-drive, memory-mapped, buffered design – even when no additional software layer is present between the program and the network. Results show that the memory-mapped interface is slow, address translation with insufficient hardware support is a bottleneck, and above all, the very high thread swap overhead for invoking the message handler is a severe limiting factor in the conventional message system. On the other hand, while register-mapped interfaces are more efficient, the benefits may be diminished due to register-pressure if the register file is too small. In terms of atomicity, the additional delay incurred in buffered interfaces appears to be insignificant for small messages.

The interface models discussed in this study require message injection to be explicitly managed by the user program. Their performance however is not necessarily inferior to systems that incorporate a communication co-processor or DMA (Direct Memory Access) engine to offload the messaging operations from the main processor. In particular, for fine-grain messaging, the potential performance improvement in the latter is defeated by the overhead of initializing a DMA engine or passing parameters to a co-processor. For large transfers, as in the case of the BLKW benchmark, the lower occupancy factor in DMA-based systems may put it in a more favorable position. But in any case, given the 1 word per cycle memory/network bandwidth in these simulations, end to end latency is lower-bounded at 1024 cycles for the 1 Kword transfer in BLKW, compared to which the software-packetized MM model is only 2.6× slower. Therefore, for a system targeted primarily at fine-grain computation, such co-processing/DMA hardware features may not be cost-effective. Their inclusion into the architecture is however in no way precluded by MM and the other models used here.

Chapter 7

Communication Overhead and Fine Grain Parallelization

Fine grain parallelization is the key to very-high-performance computing. On one hand, a recent study [8] shows that many common applications have grain sizes as small as 70 cycles. This implies that even small problems can be accelerated through parallelization. On the other hand, increasing chip density and current research directions [2, 3, 4, 5, 6, 7] suggest that inexpensive components will be readily available for constructing affordable massive machines with many thousands of processors. To effectively take advantage of such machines, existing applications have to be broken down into many more, much smaller tasks. Both scenarios thus rely on the successful exploitation of fine grain parallelism. Unfortunately, these opportunities are hampered in current multicomputers that have 100s – 1000s cycles of messaging overhead, which negates the gains from fine grain parallelization. To amortize the high communication cost, the programmer is confined to using run lengths of tens of thousands of cycles.

To exploit fine grain parallelism, a more efficient messaging system is necessary. But how low does the overhead have to be? Design effort, system robustness, and programming complexity must all be balanced with the desire to support smaller grain sizes. To help understand the tradeoffs, this chapter focuses on how performance, communication overhead and grain size relate to one another. The study may

be reminiscent of the LogP model proposed by Culler *et.al.* [60]. The LogP model is aimed at characterizing the capability of multicomputers in a small number of parameters – latency, processor occupancy, minimum messaging gap, and the number of available processors – and understanding how parallel programs may be structured to take most advantage of the existing platforms under those constraints. In contrast, the analysis here adopts an architect-centric view, and is more concerned with understanding the requirements imposed on the communication system under fine-grain computing.

Grain size is the unit of parallelization, or the size of self-contained parcels of work being assigned to the processors. Each such parcel of computation is performed by a single thread, and runs to completion from an initial state without requiring further synchronization or updates from other threads. The *run length* is the amount of time, in terms of processor cycles, needed to complete this computation. As an example, in a particle simulation problem, the grain size may be expressed in terms of the number of particles allocated to each parallel thread. The new state of each group of particles can be computed independently, and the updated values are propagated to the appropriate neighboring processors, where they are needed only in the next phase. The time it takes for a thread to complete one such iteration of computation over all its particles is the *run length*, during which multiple messages may be sent.

The minimum grain size is limited by the complexity of managing the increased parallelism itself, *e.g.* it would be counter-productive to split the computation for a single particle onto multiple processors in example above. This smallest, *natural grain size*, G_o, is thus a basic limit in fine grain parallelization. The maximum performance at G_o is seldom accessible, as the speedup is negated by communication and synchronization overhead. In addition, diminishing returns force the programmer to use a larger *implementation grain size* [1] and fewer processors, in order to maintain an acceptable level of resource usage efficiency. Parallelization is also constrained by the amount of computational work available to go around within a given problem,

[1] In other words, *natural grain size* is inherent to the problem/algorithm, and *implementation grain size* is chosen by the programmer. In this chapter, when not specifically distinguished, *grain size* refers to the latter.

and the number of processors available for use. In this study, I assume that processors are plentiful, the data set size D is fixed [2], and focus on developing a performance model that expresses execution time T_x as a function of grain size G and messaging overhead in the forms of latency and occupancy.

While the shape of the T_x function is specific to each application, the effects of latency and occupancy are commonly influenced by *slack* and message traffic. Slack is the time elapsed from the creation of a critical datum till it is needed by the consumer. It is known to mask the deleterious effects of latency. Message traffic refers to the communication pattern and the ratio of the number of messages to computation. Increased traffic accentuates the effects of occupancy. These behaviors are discussed in Section 7.1. In Section 7.2, a sample performance model is constructed, using a simple LIFE program with a natural grain size of 38 cycles, as example. A few simulation experiments are also conducted, which validate the model. The performance model enables the sensitivity analysis in section 7.3 for determining the impact of communication overhead on accessible grain size, and consequently performance and efficiency. Both processor occupancy and latency are found to be limiting factors in fine-grain applications, although the latter is partially hidden by slack. Results show that performance and efficiency degrade rapidly when communication overhead dominates the run length.

For this study, I concentrate only on recurring costs, and ignore initialization and termination overhead which correspond to the initial distribution of work and data to, and the eventual merging of results from, the processors. Although they do tend to grow with the degree of parallelization, these one-time initialization and termination costs have limited impact in most reasonably sized applications.

[2]Since the objective is a faster solution for the given application, the problem size must not be scaled up arbitrarily just to compensate for falling efficiency.

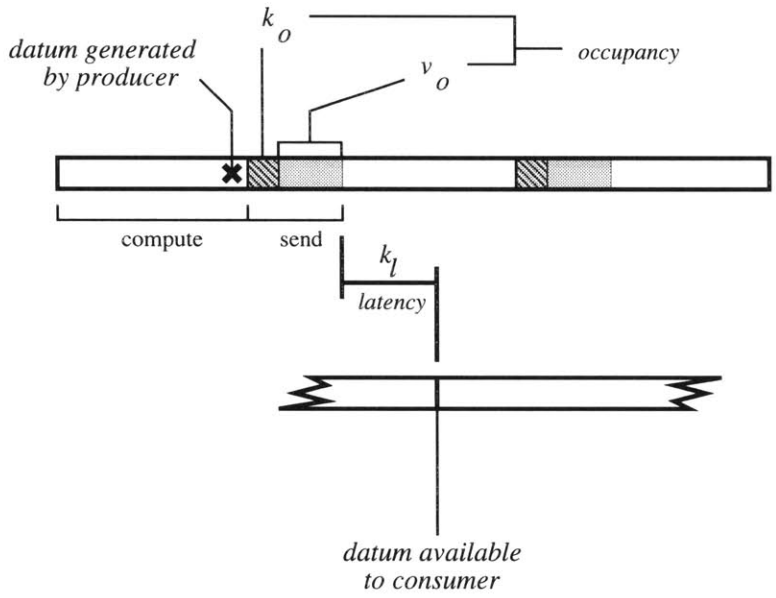

Figure 7-1: Communication Overhead

7.1 Overhead and Performance

Communication overhead consists of latency and processor occupancy, as illustrated in Figure 7-1. The terms k_o and v_o represent the fixed and variable components of processor occupancy – the former is always incurred for every message, and the later is proportional to the message length. Latency is represented as the constant k_l in the diagram. The absence of a variable latency component is due to the assumption of a wormhole-routed, streaming reception interface in this study. An atomic injection interface is assumed, so there is no overlap between k_l and v_o. Note that receiver side occupancy is not explicitly represented in Figure 7-1. As parallel applications typically have recurring phases and every message received must have been sent during some previous phase, it is sufficient to account for the receive overhead at the next injection point [3]. This approximation simplifies the model greatly.

[3]Even in cases where many messages sent in parallel from different senders are serialized at one receiver, the hot-spot receiver eventually has to send out the responses serially too.

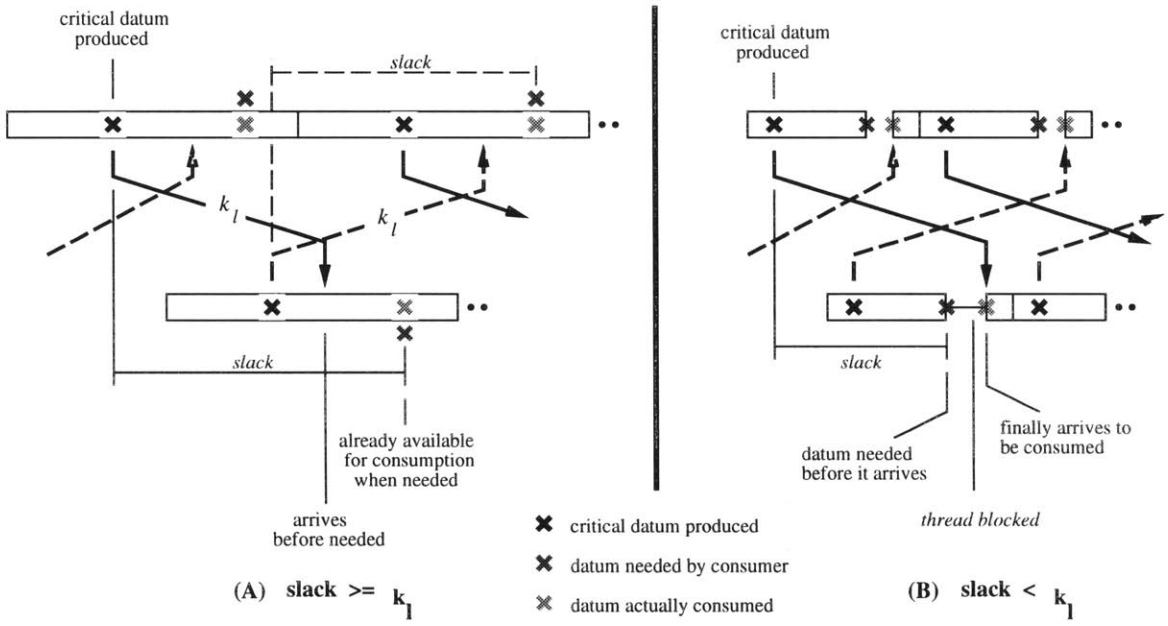

Figure 7-2: Slack in Tightly Synchronized Applications

7.1.1 Slack and Latency

Slack is the time from the creation of a critical datum, to the point of it being needed at the destination, as shown in Figure 7-2. Processor occupancy has been omitted for clarity in the diagram, which depicts a pair of producer-consumer threads. As illustrated in Figure 7-2A, up to the size of the slack, latency has no effect on the execution time, because the message arrives before it is needed. However, if the latency exceeds the slack, the consumer has to idle its resources and wait for the datum it needs, as shown in Figure 7-2B.

The above scenario represents a program that is tightly synchronized, where each thread must stall if the update message has not arrived when it is needed. A different behavior is found in more loosely synchronized applications, where the program is allowed to skip an iteration or reuse an old value if an update has not been received. This behavior, illustrated in Figure 7-3A, can be seen in search problems or incremental algorithms where an intermediate, approximate result is gradually improved over many phases using currently-best information.

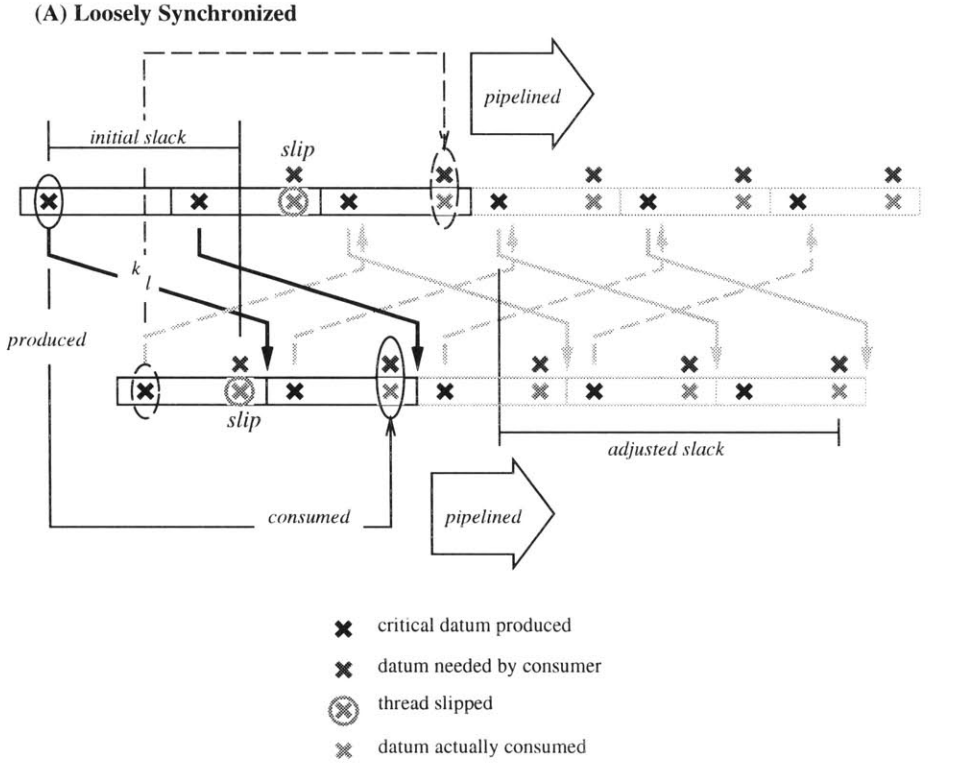

Figure 7-3: Slip in Loosely Synchronized Applications

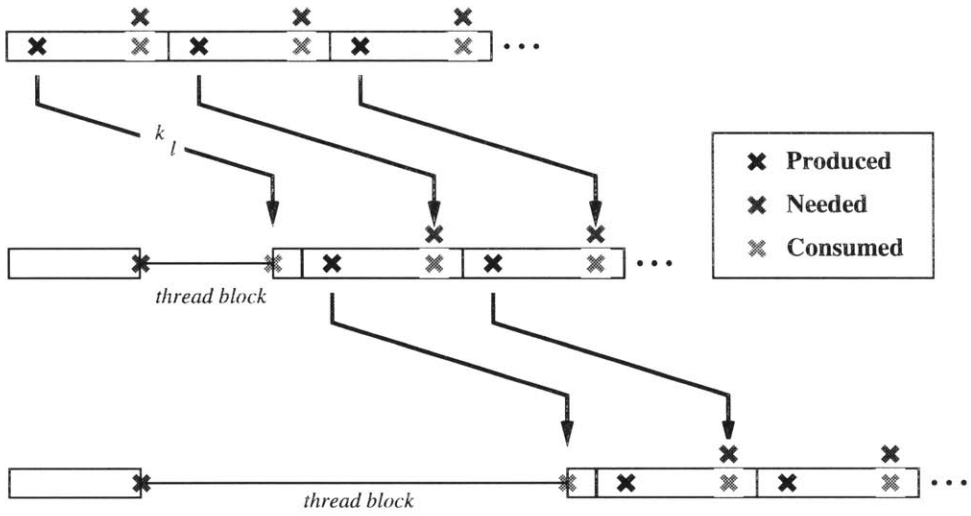

Figure 7-4: Pipelined Applications

In such programs, the threads are able to *slip* relative to one another, which in effect changes the slack dynamically. Slippage tends to gradually create a pipelined pattern among the threads. Eventually fresh data becomes always available for consumption before it is needed. As a result, save for a few iterations right after cold-start and before termination, the effects of messaging latency are virtually eliminated. Figure 7-3B provides a contrast, where tightly-synchronized threads which are unable to slip continue to block periodically throughout their entire execution.

Similar latency masking effects can also be observed in applications with an inherently pipelined dependency pattern. Figure 7-4 illustrates an example where each thread produces data only for the next thread, a characteristic often found in graphics manipulation and signal processing applications. Each thread is blocked until its first update message arrives, but they all merge into a pipeline eventually, and message latency is masked.

For sufficiently long-running programs, network latency can be left out of the performance model in the last two categories. Only a few iterations are needed to prime the pipeline the former case. In the latter case, it may take as many iterations as the number of threads. For strictly synchronized applications with a tight dependency loop, the latency effect must be considered carefully. Ignoring the initial offset, the

execution time for the fragments shown in Figure 7-2, for instance, can be modeled as $[\ i \times (C + max(0, k_l - s))]$, where i is the number of iterations, C is the per-iteration computation time for the corresponding grain size, and s represents the slack. Note that slack may in fact have a negative value, if the datum is generated way too late.

7.1.2 Message Traffic and Occupancy

Interestingly, it may seem at first that occupancy is also subject to masking, in that it adds to the overall execution time only if it displaces productive computation. Figure 7-5 for example shows an otherwise idle "elastic zone" which apparently allows a much higher message-induced occupancy without affecting the runtime. In the larger picture however, this is seldom the case, for the elastic zone most likely exists in the first place because the thread is stalled waiting for an incoming message. With higher occupancy, the message would arrive even later. The overhead is only shifted, not masked. Each additional cycle of processor occupancy on the critical path is essentially a cycle taken away from productive computation.

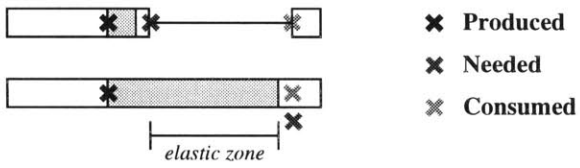

Figure 7-5: Supposed Elastic Zone

Multiple messages with the same destination are often bundled together in coarse-grain applications to incur fewer instances of the k_o occupancy, as illustrated in Figure 7-6. This is however ineffective in systems with small k_o and short run lengths, as the savings are defeated by the overhead in reorganizing the communications. Furthermore, message bundling changes the slack and may add to stalling, as shown in Figure 7-7. Therefore, the potential gain from message bundling is limited in fine-grain systems, and will be ignored in the performance model. To cap the benefits of bundling under large values of k_o, the execution time for a message-bundled pro-

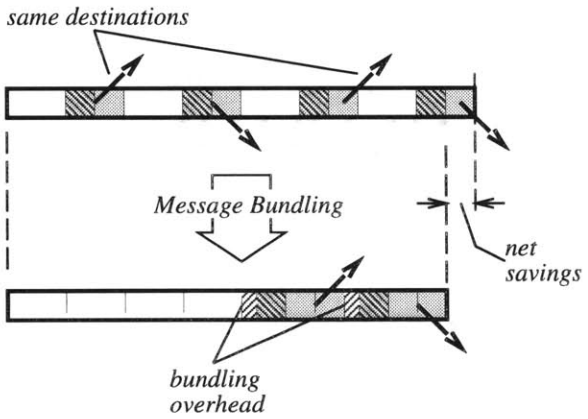

Figure 7-6: Message Bundling

gram can be approximated by factoring out the appropriate instances of k_o from the simpler model. For a thread that sends m messages of size w, the runtime can thus be modeled simply as $C + (m \times (k_o + w \cdot v_o))$ – recall that k_o is incurred for every message, and v_o is incurred by each message word.

7.1.3 Grain Size, Slack, and Message Traffic

The T_x model is instrumental for understanding the important fine grain architectural issues – What amount of overhead is acceptable? What size multicomputers are useful? How fast must the network be? To answer these questions, the overall execution time under different degrees of parallelization and overhead can be obtained by expressing C, s, w and m, and the number of phases in terms of grain size. For instance, in a matrix application with a *many-to-many* traffic pattern, as the problem is distributed to more processors, the number of messages sent and received by each thread also increases. On the other hand, as grain size shrinks, the slack also tends to shrink, as less computation is performed between messages. Naturally, the relationship is specific to each applications, and the architect has to understand the applications he wishes to support. In the rest of this chapter, I construct the T_x model for a sample application and show how it is used to understand how the program responses to communication overhead.

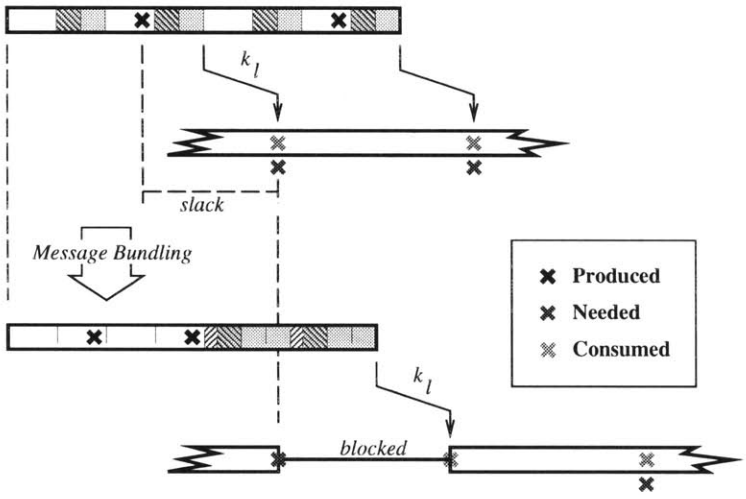

Figure 7-7: Message Bundling and Slack

7.2 A Sample Application

A simple application, LIFE is used to demonstrate the construction and analysis of a performance model. It uses a phased algorithm, and has a natural grain size of about 38 cycles. A nearest-neighbor messaging pattern is involved in LIFE, which has strict dependencies among threads between phases that precludes unrolling and similar software techniques. This example is simple to understand and analyze, yet bears a close resemblance to a broad class of real applications based on relaxation-like algorithms.

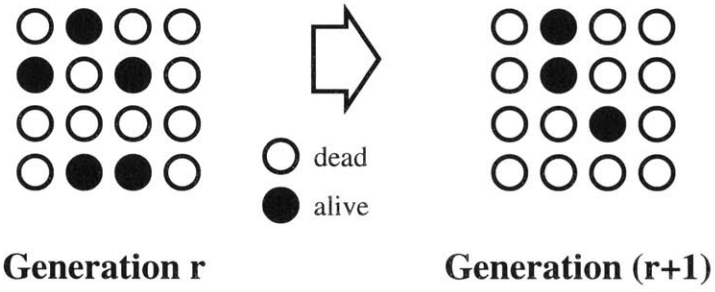

Figure 7-8: 16-Cell Game of LIFE

The LIFE program is an implementation of Conway's *Game of Life* [14] simula-

tion. It models a 2-dimensional, toroidal array of *cells*. Each cell is surrounded by 8 others, and contains a value of either *0* or *1*, representing respectively the absence and presence of a living entity in the cell. The game is played in terms of *generations*, with an arbitrary pattern of 0s and 1s occupying the array during the initial, 0^{th} generation. At the end of each generation, the status of each cell changes according to the following *birth-death* rules. A 16-cell example is shown in Figure 7-8 [4].

- Birth

  If a *dead* cell (containing the value 0) has exactly 3 *live* neighbors (containing the value 1), a new being is born. resulting in the cell having a value of 1 in the next generation.

- Death

  If a live cell is surrounded by 4 or more other live neighbors, it *dies* of overcrowding and gets the value 0 in the next generation. A live cell that has fewer than 2 live neighbors also dies, but of *loneliness*.

- Survival

  If a live cell has exactly two or three live neighbors, it survives and remains unchanged. A dead cell also remains unchanged if it has 2 or fewer live neighbors.

In this implementation, each processor is responsible for a rectangular section of the the LIFE environment, containing $\sqrt{\frac{N}{P}} \times \sqrt{\frac{N}{P}}$ cells (Figure 7-9), where N and P represent the number of cells in the system and the number of processors, respectively. During each phase, every processor iterates over all the cells it hosts, and sends their updated status to the relevant neighbors. As an example, the messages that must be sent by processor 2 are shown in Figure 7-9 (the edges wrap around to form a torus). Each thread blocks at the beginning of each phase until all expected updates have been received. No redundant messages are ever sent to the same processor. Message bundling is not attempted in this implementation.

[4]For simplicity, the toroidal array is represented as a rectangle in the illustrations.

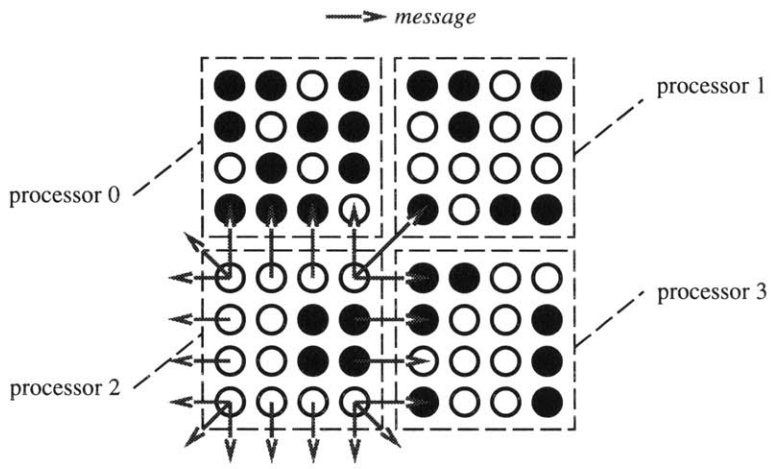

Figure 7-9: 64-Cell LIFE: Distribution of Cells onto 4 Processors

7.2.1 Constructing the Model

For concreteness, I consider the execution time for 1000 generations of an 8×8-cell LIFE system, using P = 1, 4, 16 or 64 processors in $\sqrt{P} \times \sqrt{P}$ configurations, corresponding to grain sizes of 64, 16, 4, and 1 cell-per-node. A fully connected network is assumed, and contention within the network fabric is not modeled.

The LIFE program involves a tightly synchronized algorithm with a nearest-neighbor traffic pattern. Each node hosts a number of cells and computes their new values serially during each phase. Figure 7-10 shows LIFE_{64} as it is parallelized to four cells per node (16 nodes), and to one cell per node (64 nodes). Each processor sends 2-argument update messages to its neighbors as soon as it finishes computing the new value for each cell. In the critical path, the last-generated value on some processors also happens to be the first-needed value on some of their neighboring nodes [5]. This gives the worse-case slack, where the update information is needed about 6 cycles after the consumer finishes sending its own messages in the current

[5]The slack may be lengthened by making each processor update its internal cells (cells whose neighbors all reside on the same processor) last. This optimization is not implemented as it is not applicable at the small grain-sizes shown in Figure 7-10 – there are no internal cells. The only configuration with internal nodes in LIFE_{64} is the one-processor configuration, where slack does not matter. For larger problem sizes, the effect of such an optimization can be captured by expressing the slack in term of the number of internal cells: $s = (\sqrt{n} - 4)^2 \cdot C_{cell}$.

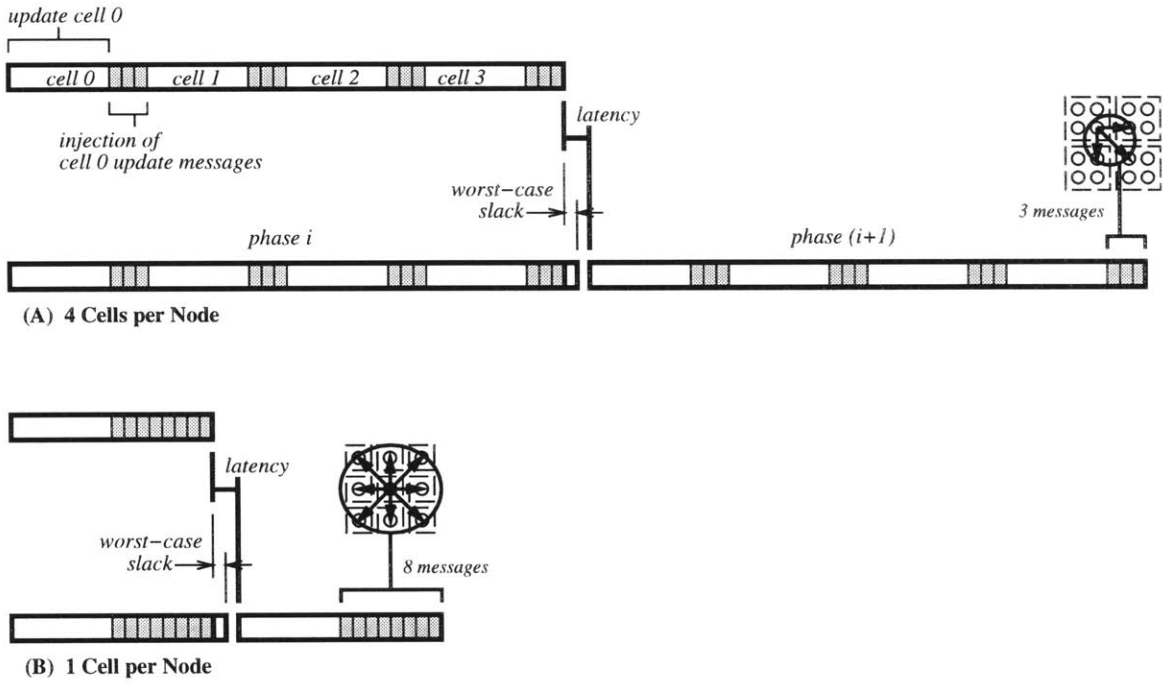

Figure 7-10: Fine-Grain Parallelization of LIFE

phase. For the one-cell per node configuration, this yields a slack that is equivalent to 6 cycles beyond the injection overhead of 8 messages, for a total of about 78 cycles (Figure 7-10). For all other configurations, the slack is about 33 cycles, corresponding to 6 cycles beyond 3 message injections, which is the maximum number of update messages for any one newly computed value in those configuration.

Since no bundling is attempted, the absolute number of messages sent by each processor in each generation is reduced as the grain size shrinks in LIFE. In every generation, the one-cell/node configuration requires 8 update messages for each value computed, the four-cells/node configuration uses 12 messages for 4 values, and the 16-cells/node configuration sends 20 messages in the course of computing 16 new values. In general, $4 \cdot (1 + \sqrt{n})$ messages are required for an n-cells/node configuration. The execution time for one thousand generations of LIFE parallelized to have $\sqrt{n} \times \sqrt{n}$ cells hosted on each processor is thus captured in the following equation:

$$T_x^{LIFE} = 1000 \times [n \cdot C_{cell} + 4 \cdot (1 + \sqrt{n}) \cdot (k_o + 2 \cdot v_o) + max(0, k_l - s)]$$

The time for computing a new cell value, C_{cell}, is approximately 35 cycles in this implementation. Notice that the runtime is minimized when the application is maximally-parallelized. However, since both the occupancy and latency stand to contribute significantly to the overall runtime when n is small, fine-grain parallelization is rendered inefficient if either is significant compared to C_{cell}.

7.2.2 Experimental Results

To test the correctness of the model, the 64-cell LIFE program is hand-coded in M-Machine assembly language for P = 1, 4, 16 and 64 processors, and executed on the *msim* simulator [59]. Execution time is measured for 1000 generations of LIFE. The initialization and termination costs are factored out. The experiments are conducted first under the base overhead of the M-Machine (MM) message system, and then with additional cycles of latency and processor occupancy. The base latency in the M-Machine is 11 cycles [6], while the occupancy costs k_o and v_o are 5 cycles and 1 cycle, correspondingly. The results are found to be in agreement with the theoretical model.

Figure 7-11 shows the execution time for LIFE, charted against latency. The measured results are shown as the solid lines while the predicted execution times are represented by the dotted lines. As expected, LIFE is sensitive to latency when the slack is exceeded, with a slope of approximately 1000, corresponding to the number of generations executed. In the measured results, a shorter slack of about 21 cycles is observed – bear in mind the base MM architecture has 11 cycles of latency. The discrepancy between the predicted and measured results, in terms of the observed slack and total execution time, is likely due to an inaccurate estimate for k_l. Under zero-contention, k_l is 11 cycles at the origin of the chart in Figure 7-11. However, when multiple messages share the network ports, the base latency increases due to contention.

[6] This includes both the routing latency in the network fabric, and the message handling time on the receiver processor. As the M-Machine dedicates a thread slot to message handling, the receiver side occupancy does not affect the user thread at all.

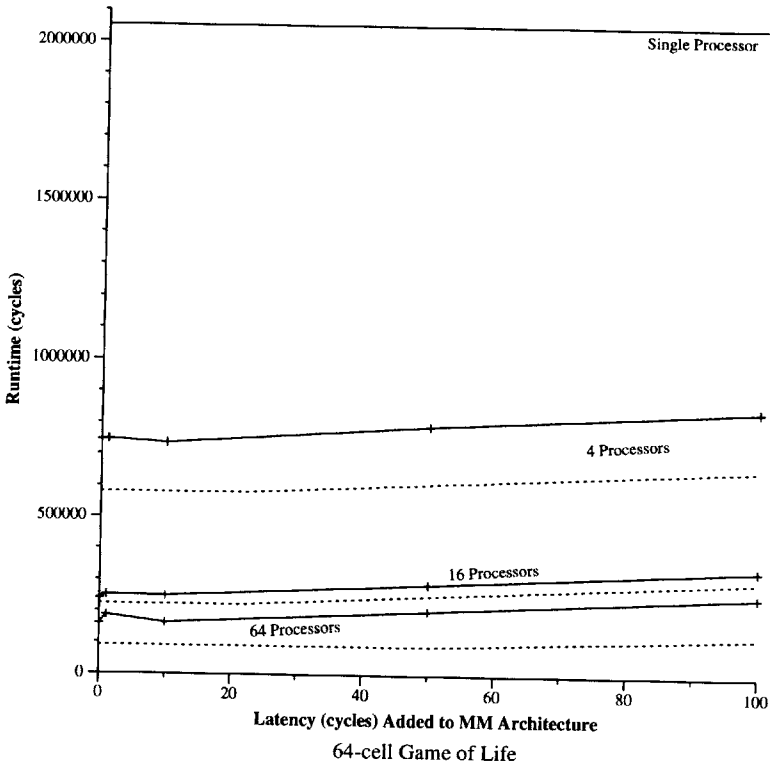

Figure 7-11: LIFE_{64}: Runtime vs Latency

The results also suggest that fine grain computing is indeed very viable, given an efficient communication system. Using the low overhead message system in the M-Machine, very good speed up is observed at 4 processors (about 4×), 16 processors (more than 8×), and even down to the very fine grain size of 1 cell-per-node – 12× speedup at 64 processors, The speedup curves under different latency and occupancy values are shown in Figure 7-12.

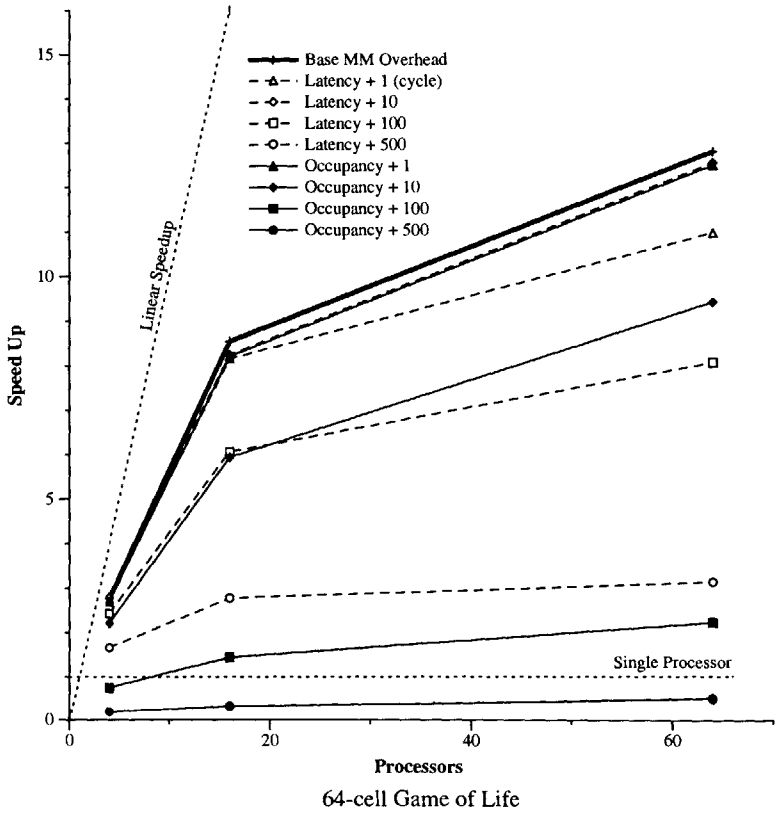

Figure 7-12: $LIFE_{64}$: Speedup

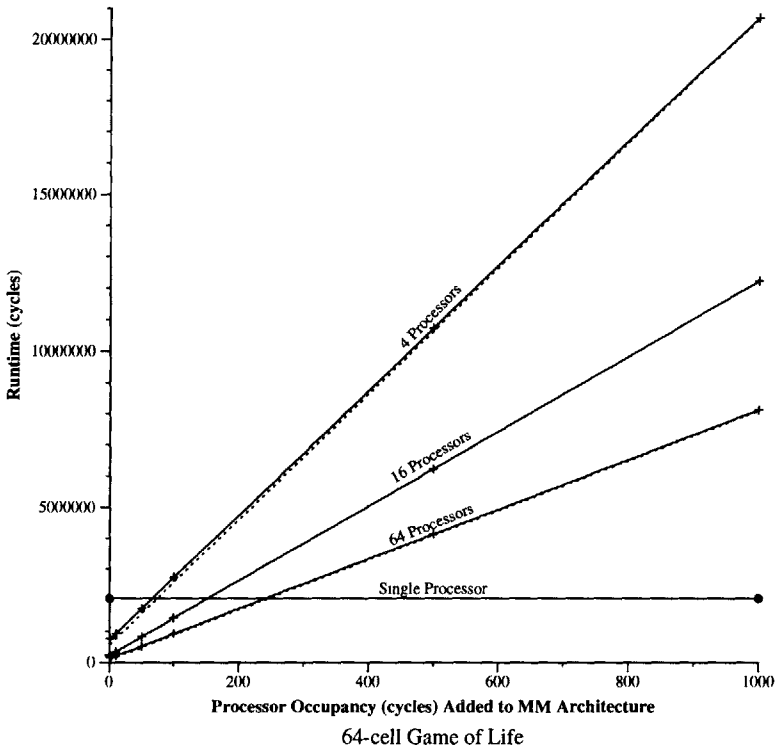

Figure 7-13: LIFE_{64}: Runtime vs Occupancy

The impact of processor occupancy on the execution time is also shown in Figure 7-13. Again, the measured results are plotted in solid lines, while the predicted values are shown in dotted lines. Reflecting the $4 \cdot (1 + \sqrt{n})$ coefficient for k_o discussed earlier, the sensitivity towards occupancy goes down as more processors are used. As expected, processor occupancy is a severe limitation for parallelization of LIFE. At about only 225 cycles, parallelization of LIFE_{64} becomes counter productive for every one of the implemented grain sizes. It should be pointed out that the experiment methodology plays a role in the accuracy of the predicted results in Figure 7-13. In the experiments, occupancy is varied by calling on a feature of the *msim* simulator to explicitly stall the pipeline for the desired number of extra cycles after each `SEND` instruction is issued, but before the message is injected. All three function units (IALU, FALU, and MEMU) are made unavailable for this duration to simulate occupancy by message-related operations. In reality, such operations may occupy only a subset of the function units, and productive computation can be interleaved with the

message operations. Nonetheless, this artifact is not expected to significantly impact the results, since all three function units are needed in each iteration of the LIFE program. It cannot make progress beyond an iteration if any one of the units is busy with message-related operations.

7.3 Sensitivity Analysis

By varying the various parameters in the T_x expression, much can be learned about the general fine-grain behavior of the target application. In the case of LIFE, both performance in terms of speedup of total execution time, and efficiency – measured as $\frac{speedup}{P}$ – are extremely sensitive to processor occupancy. It is found that $(k_o + w \cdot v_o) \ll 50$ and $k_l < 200$ are highly desirable. In particular, very low occupancy (< 10 cycles) is a critical factor for large scale multicomputing, enabling the high efficiency ($> 50\%$) necessary for 1000-processor machines to be economically viable, and providing thousand-times speedups when performance is the main goal. Since w is small for LIFE, and k_o is the bulk of the communication overhead in most existing system, the effects of k_o and v_o are not separately discussed.

Figure 7-14 plots the best achievable speedup for LIFE, against communication overhead. The chart is normalized to the linear speedup computed at the grain size of 1 cell per processor. This represents the best possible parallel performance under the given communication overhead, as a fraction of the theoretical maximum. The curves are obtained by selecting the best results among all different grain-size configurations of LIFE, assuming that an infinite number of processors are available. Since the perfect speedup scales with problem size – theoretically, a larger problem allows the use of more processors for a higher speedup – the curves are independent of problem size. With zero overhead in the communication system, 60% of linear speedup can be achieved [7]. The performance however tails off extremely quickly as processor occupancy increases, dropping to 34% at 5 cycles of occupancy, and less than 4% at

[7] Linear speedup is not achieved even with zero overhead, because extra instructions are added to coordinate the parallelized computation.

100 cycles of occupancy. The subsequent region of the occupancy curve suggests that efforts to reduce processor occupancy do not pay off as far as fine grain performance is concerned, until it is brought well under a hundred cycles. On the other hand, the steep slope from $k_o = 100 \rightarrow 0$ indicates that very much can be gained if occupancy is further lowered, to zero ideally. The latency curve shows a less severe impact, and reflects the slack effect. At least for this application, it appears that it is sufficient to keep latency under about 200 cycles, which is within the capability of most existing multicomputer networks.

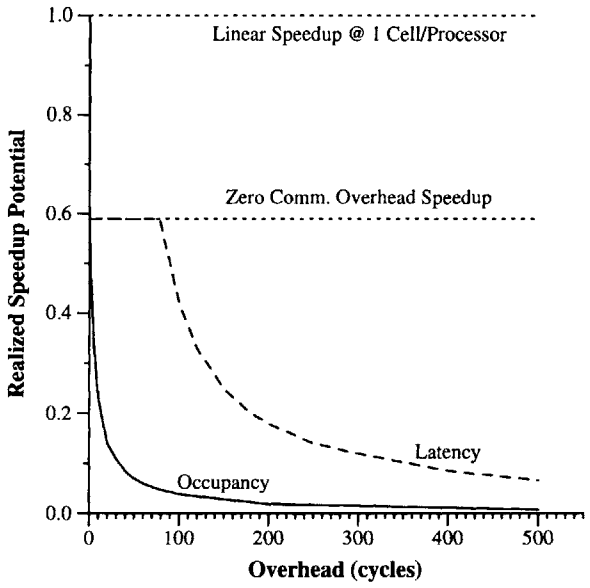

Figure 7-14: Overhead Limits Performance

The curves in Figure 7-14 also suggest that the very fine grain parallelism of LIFE is impractical in existing multicomputers, where the communication overhead ranges from many tens to many thousands of cycles. To maintain an acceptable level of efficiency, these architectures must rely on much larger grain sizes to amortize the communication cost, compromising performance in the process. In particular, Figure 7-15 shows the speedup of a 16K-cell LIFE program under different grain sizes when an efficiency level of at least 30% is imposed as a constraint. While the M-Machine easily supports the smallest grain size to achieve a speedup of more than 5000×, the maximum speedup drops to less than 100× when occupancy rises to only

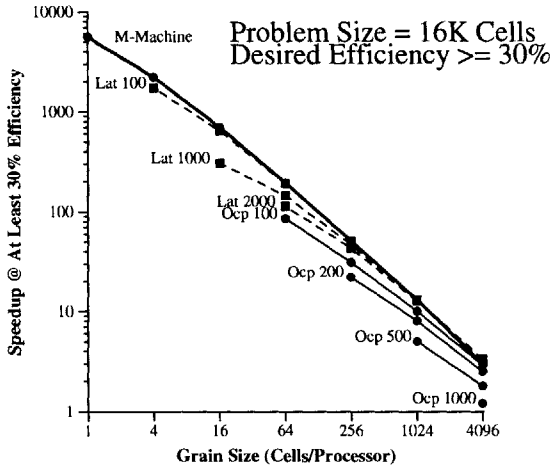

Figure 7-15: Overhead Limits Speedup

a hundred cycles.

This larger grain size requirement in conventional designs also constrains the utility of massive multicomputers which have a large number of processors. For instance, the maximum number of processors that can be deployed without going below 30% efficiency is shown in Figure 7-16, for problem sizes ranging from 64 cells to 16384 cells. The quantized behavior of the curves is simply due to LIFE being distributed into even power-of-2 cells per node in this implementation. As overhead increases, a larger problem size is needed to justify a particular machine size. For instance, a 256-processor machine hosts a 256-cell problem efficiently if the occupancy is no more than 7 cycles. When the overhead exceeds 120 cycles however, a much larger 16384-cell problem is needed to make good use of the processors. This suggests that very-low occupancy is a necessary enabling factor for economical, large scale parallelization. Without it, the utility of large machines is limited to less common, huge sized problems.

Conversely, with adequately low overhead, not just huge problems, but even moderately sized ones can employ a large number of processors very efficiently. Figure 7-17 shows a set of "iso-efficiency" curves. With implementation grain size on the vertical axis, these curves are independent to problem size. Given the M-Machine system ($k_o \leq 7$) for example, 50% efficiency is easily obtainable for grain sizes down to 4 cells

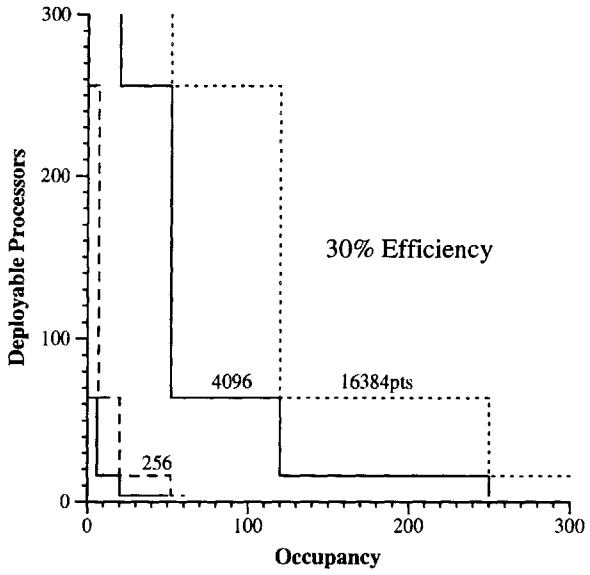

Figure 7-16: Overhead Limits Utility of Massive Multicomputing

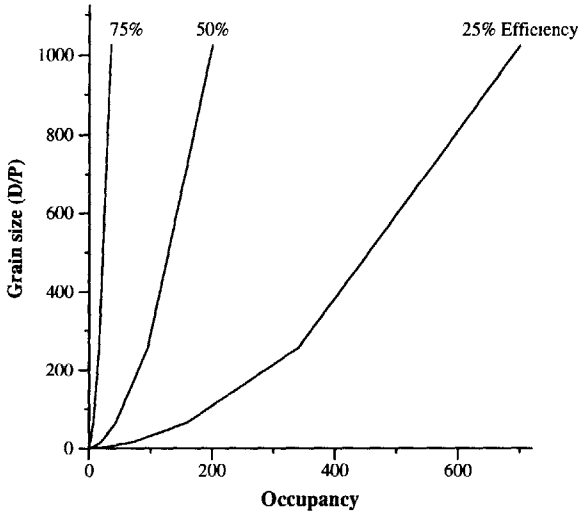

Figure 7-17: Overhead Limits Efficiency

per node, and even 75% efficiency is attainable, with a 64 cells grain size. In other words, up to 256 processors can run at 50% efficiency with a problem containing just 1024 cells, and reach 75% returns if a large 16K-cells problem is available.

While these analyses are specific to the LIFE program, they are useful for understanding the fine-grain behaviors of other relaxation-like algorithms exhibiting a nearest-neighbor traffic pattern. The performance/efficiency response to communication overhead can be obtained for such applications easily, by varying the C_{cell} and s parameters value in T_x accordingly. For other applications with more complex communication patterns, separate models will have to be constructed.

7.4 Conclusion

Fine grain parallelization with thread lengths of around 100 cycles, used whether for accelerating a small application or exploiting a massive multicomputer, relies on an efficient communication system. Without it, the performance gain is negated by messaging latency and occupancy that dominates the short run lengths.

Although latency is sometimes masked by slack, occupancy displaces productive computation from the processing resources, and is generally not maskable. To understand the architectural issues in supporting a fine grain application, it is necessary to understand how its performance relates to grain size, occupancy and latency. The construction and analysis of such a model is demonstrated using LIFE as a simple example. Under fine grain parallelization, the performance response to occupancy is found to be extremely steep under 100 cycles, but is largely insensitive beyond that region. Latency has a similar but less severe impact, but only takes effect when the slack is exceeded. It appears that $(k_o + w \cdot v_o) \ll 50$ and $k_l < 200$ are extremely desirable. At these levels, efficiency levels of $> 50\%$ are easily attainable with a number of processors that is just an order of magnitude lower than the problem size itself, *e.g.* 1000s of processors for the 16K-cell LIFE. The maximum achievable speedup is also only an order of magnitude below the problem size numerically – several 1000× for the 16K-cell example.

The reader is cautioned against drawing generalized conclusions from these specific results. Nonetheless, with appropriate parameter substitutions, the analysis readily applies to the broad class of applications that share the traffic pattern of LIFE. More importantly, the example demonstrates the viability and potential of large scale, fine grain machines, and shows how the *Latency, Occupancy and Grain Size* model can be used to understand the important architectural issues in fine grain computing. Many more such models should be constructed for other classes of common applications, for a full understanding of the general tradeoffs.

Chapter 8

Conclusions

High performance fine-grain multicomputing demands a communication system that is both efficient and robust. In this thesis, I developed an analytical model to better understand the architectural issues in designing the messaging interface. It demonstrates that large-scale, fine-grain multicomputers are a viable concept that can achieve very high performance if communication overhead is kept low. In particular, in an example program, LIFE, speedup of several 1000× is observed when processor occupancy is kept to below 10 cycles.

In addition, I quantified the overhead incurred by different primitive mechanisms in the design space of modern message interfaces, and found that the high context-switch cost in dispatching message handlers is the critical factor. I also addressed the robustness problem plaguing many existing designs. Recognizing open-ended allocation as a fundamental cause of starvation hazards that necessitate expensive software workarounds in existing machines, I demonstrated that a simple solution can be found in *bounded time lease* schemes. Taking advantage of the *guarded pointer* light-weight capabilities system and carefully integrating it into the messaging facility, I devised an enforcement mechanism for *trusted handlers*, thereby closing a major protection loophole in existing message-handler based interfaces.

The concepts discussed in this thesis are validated in the M-Machine communication system. It serves not just as an example for future designs, but also as an affirmation to the importance of incorporating complementary mechanisms into the

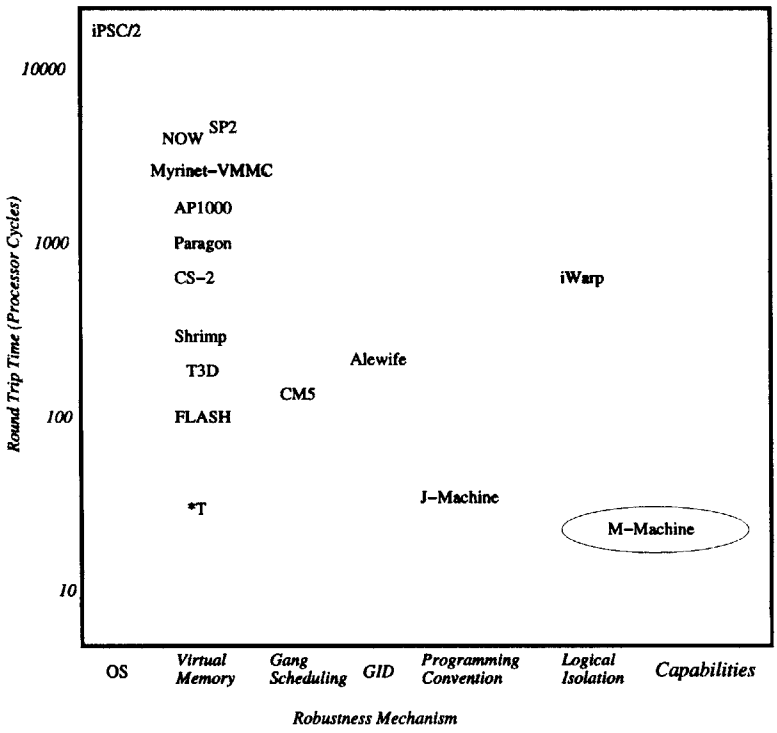

Figure 8-1: M-Machine, Relative to Past and Present Multicomputers

system. The simple M-Machine mechanisms – carefully chosen to supplement one another – constitute one of the fastest message interfaces ever proposed (Figure 8-1) when taken together, without sacrificing cost effectiveness, starvation-freedom, nor protection.

8.1 Fine Grain Computing

Exploiting fine-grain parallelism is the key to high performance computing. By breaking down problems down into short threads that are no more than a few hundred instructions long, fine-grain parallelization makes it possible to accelerate a fixed size application beyond current limits, and to take advantage of large-scale multicomputers made affordable by emerging highly integrated components [2, 3, 4, 5, 6, 7]. However, although task lengths as short as 70 cycles have be found in common benchmark applications [8], programmers in general are still unable to take advantage of

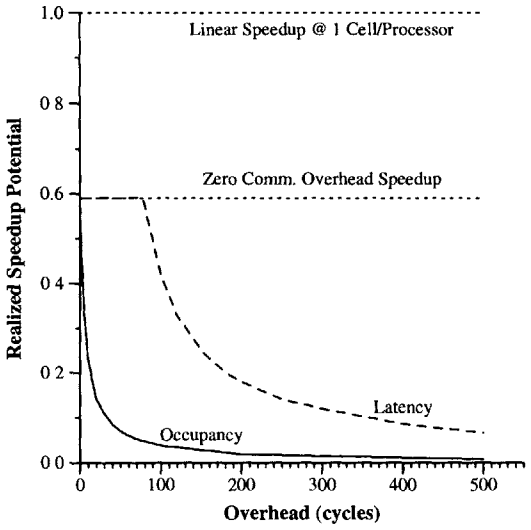

Figure 8-2: Fine Grain Performance and Communication Overhead

parallelism at that level, due to the high communication overhead in current machines.

To understand the impact of messaging overhead, an analytical model relating performance to Latency, processor Occupancy, and Grain size – the LOG model – is developed for a sample application, LIFE, with a thread length of ~40 cycles. It is found that while latency is sometimes masked by slack, occupancy is generally unmaskable. Results (Figure 8-2) suggest that performance is extremely sensitive in the region where occupancy < 50 processor cycles and latency < 200 cycles. On the other hand, optimization efforts that tune the communication overhead in the range of 100s – 1000s of cycles appear to be ineffectual for improving performance in very fine-grain programs, such as the implementation of LIFE in this thesis. They are beneficial only if more computation is packed into each thread, to increase the grain size to many thousands of instructions.

The fine grain model is indeed very viable and capable under sufficiently low communication overhead. In particular, with occupancy < 10 cycles, efficiency levels of $> 50\%$ are easily achievable, even when the problem size is only a order of magnitude larger than the number of processors (*e.g.* 1000s of processors for 16K-cell LIFE), while speedup as high as just an order of magnitude below the problem size is also possible (several 1000× for 16K-cell LIFE).

8.2 Communication Overhead

Communication overhead stems from three sources: (a) operating system involvement, (b) software workarounds and (c) incompatible primitive mechanisms. The operating system has traditionally been used to abstract the messaging facility from user programs. By requiring user-level messaging operations to be managed by the OS, high-level properties such as order-preserving delivery, end-to-end flow-control, error-correction, transparent address-translation and access-control can be provided. However, while this presents a convenient interface, the overhead of switching between user-level and system-level domains is prohibitive to fine-grain computing.

Having abandoned the OS-managed model in favor of more efficient user-level interfaces, most modern systems are inadequate in supporting higher level communication needs. Consequently, software workarounds are needed to make up for the missing high level properties, which account for up to 70% overhead [34]. Many designs are vulnerable to starvation problems due to their open-ended allocation of critical resources. It is shown that *bounded-time lease* schemes that assign resources to users for only bounded periods of time make a good defense against starvation risks. In message-handler based dispatch systems – a model widely-adopted due to its flexibility in allowing the user to specify an optimized handler for each message – protection risks due to malicious or erroneous handlers are abound. As a solution, the *guarded pointers*, a lightweight capability mechanism described in [13], is shown to integrate well with the message system, allowing the system to regulate the data as well as the handlers accessible to each user program.

Primitive mechanisms in the message interface must be complementary to avoid redundancy and oversights in functionality. Their design space (Figure 8-3) is considered in three aspects: (a) mapping – memory *vs.* register-mapped, (b) atomicity – buffering *vs.* streaming, and (c) dispatch – interrupt-driven, polled, or dedicated hardware. Microbenchmark studies (Figure 8-4) show that interface overhead is reduced by an order of magnitude when a traditional interrupt-driven, fully buffered, memory-mapped interface is replaced by a processor-register mapped, atomic injec-

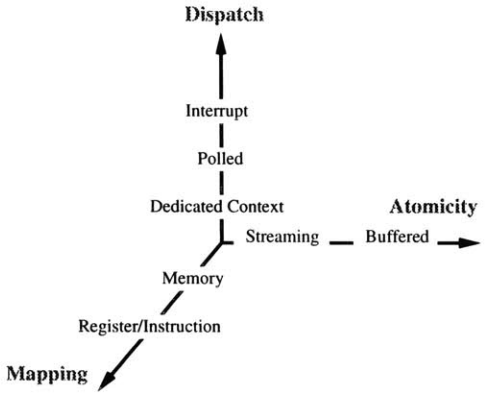

Figure 8-3: Primitive Mechanisms Design Space

tion, streaming extraction interface that also incorporates a hardware address translation facility and dispatches messages in a dedicated hardware thread slot. The elimination of context-switching upon message arrival accounts for 60% of the overhead reduction, while the mapping and atomicity choices together contribute 30% of the improvement. The fast address translation facility accounts for the remaining 10% of the speedup.

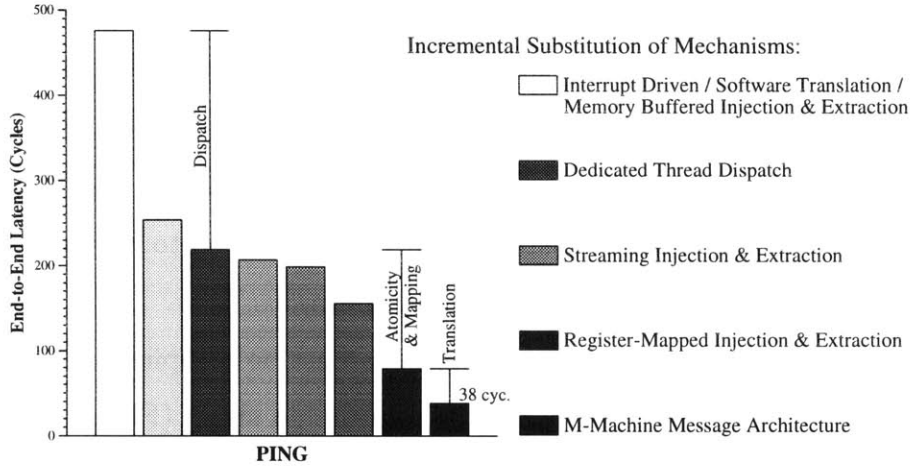

Figure 8-4: Microbenchmark Results – some labels omitted to reduce clutter

8.3 The M-Machine Message System

The M-Machine message interfaces (Figure 8-5) are completely mapped into the processor's general register name space. Messages are injected from within the user's register files using a `SEND` instruction, which specifies a `DestAddr`, a `HandlerIP` and the number of registers, starting from either *i4* or *f4*, to be injected as the message body. Only five cycles of processor occupancy is incurred to send a null message. Both `DestAddr` and `HandlerIP` are required to be unforgeable guarded pointers. `DestAddr` is translated into routing information by a small hardware translation table called the *Global Translation Lookaside Buffer* (GTLB), which prevents messages from being sent outside the user's mapped domain. The `handlerIP` is of pointer-type *ExecuteMessage* and points to a non-faulting, non-blocking, *trusted handler* that honors all protection restrictions at the destination. The `HandlerIP` is mutated into an executable pointer during injection, making trusted handlers accessible only through the message system, and thus allowing hard-coded assumptions and optimizations in them.

A bounded-time, atomic injection interface is chosen to prevent users from monopolizing the injection port, while mapping the message buffer into the user's register file eliminates starvation-inducing resource conflicts. At the receiving end, the ability to enforce *trusted handlers* permits a more efficient streaming extraction interface that dispatches upon arrival of the message head. Registers *i14* and *i15* – also called `MsgHead` and `MsgBody` – in two reserved thread slots, one each for priority 0 and priority 1, are mapped to the incoming message queues. Reading `MsgBody` pops and returns the next word in the message *currently* being processed, while reading `MsgHead` flushes any remnants from the current message, and then pops and returns the `HandlerIP` in the *next* message. Both `MsgHead` and `MsgBody` can be used directly as the source operand in any regular instruction. The reserved thread slots enable messages to be dispatched within a 3-cycle `jump` delay, yet because the availability of the *MsgHead* and `MsgBody` data is tied into the scoreboard consulted by the instruction-issue logic, no execution resources are consumed when the handler threads slots idle.

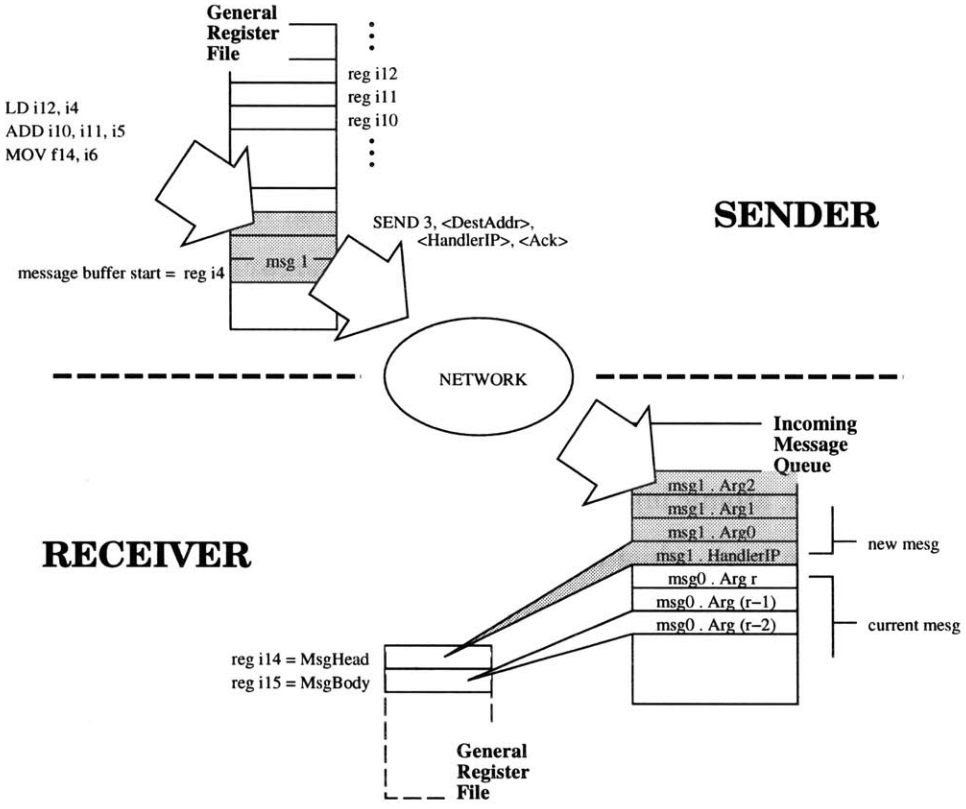

Figure 8-5: M-Machine Message Interfaces

To guard against malformed messages, `MsgBody` is padded with null data when the user reads beyond the end of the message, while the flush mechanism associated with `MsgHead` guarantees to return the message queue to a sane state when recovering from error conditions. In addition, the M-Machine system also supports user-selectable pair-wise in-order delivery, and features a flow-control counter that can be used to efficiently implement *return-to-sender* or other higher-level protocols.

8.4 Future Research

The M-Machine opens many opportunities for future research. Is this the lowest we can go in reducing messaging overhead without compromising robustness? What can we actually do in a thread with a 50-cycle run-length? How do we extract parallelism at this level from real applications? Of course, more LOG models will also need to be constructed to understand other classes of common applications.

More immediately however, a most important issue is designing the compiler to take full advantage of the register-mapped message buffer. To avoid copying, the compiler should ensure that message words are deposited *directly* into the appropriate message registers by the instructions that generate them. The complexity this adds to the register-allocation scheme in the compiler has to be studied. To keep the finite state machine that transfers the message from the register file to the network simple, the message words are assembled in a contiguous region of the register file. This constraint however complicates register-renaming mechanisms in both the hardware and the compiler. To simplify their interactions, the designer may consider adopting a more general SEND instruction format, such as with a bit vector, for specifying the location of message words within the register file.

Cache effects were ignored in this study. In reality, message handlers have little locality if message arrivals are sparsely distributed in time, and conversely may pollute the cache if they arrive too frequently. To ensure fast handler dispatch while minimizing the impact to other resident threads, smarter cache management schemes are desired. A more crucial issue is the resolution of references to non-resident mem-

ory pages from within the message handler. The *ExecuteMessage* guarded-pointer type enables the system to confine the user to calling only message handler code that is resident on the destination node. However, the system has little control over the memory addresses that the handler references. In the M-Machine, trusted handlers use special non-faulting memory operations that return a negative condition code upon a page fault instead of trapping into the machine's software-based page-fault handling system. Under this condition, the trusted handler is responsible for vacating itself – together with its message – out into a software scheduling queue, and then calling the coherent memory manager to import the missing page. The memory manager subsequently reactivates the dormant message handler after installing the affected memory page. This convoluted solution is necessary because user messages share the the priority 0 network with the response-messages used by the memory manager to transfer cache lines between nodes. A deadlock condition would result if the response message is blocked by the message handler that needs it. Future designs might explore simpler schemes, such as providing independent logical networks for user and system level messages.

A drawback of the guarded pointers – and other similar capability-based protection schemes – is the difficulty in revoking privileges. In the short term, the system may "destroy" the privileges associated with a pointer by removing the corresponding mapping from the virtual address translation layer. However, once a pointer is granted to the user, it can be copied, shared, or even saved into long term storage media. The system thus cannot safely recycle the "destroyed" virtual addresses without performing a full garbage collection by scanning all user-accessible domains. Taking a lesson from Chapter 5, the designer may attempt to replace the open-ended allocation of guarded pointers with a *bounded-time lease* scheme. Nonetheless, such a system would have to distinguish one time slice from another, and is ultimately faced with the more general problem of recycling a bounded name space, for which an efficient solution is yet unknown.

As more simultaneous threads of control are incorporated into the processor chip, the M-Machine interface scales naturally with the additional thread slots and their

register files. The basic architecture however does not address the higher bandwidth requirement that comes with the higher aggregate computation power of the chip. To relieve contention for the network ports, the architecture may be extended with multiple handler thread slots and message queues – possibly a number of logical queues being multiplexed onto a high bandwidth physical off-chip link – so that several messages may be extracted and injected concurrently. As the network will also be competing with the memory system for pin bandwidth, the requirements and tradeoffs in each case must be characterized so that a balance can be achieved, bearing in mind that on-chip locality may ease some of the off-chip bandwidth problems.

8.5 Impact

By fabricating an actual prototype system, the M-Machine design team not only creates an experimental platform for fine-grain computing, but also provides an implementation example for future designs. The proposed *Latency, Occupancy and Grain Size* performance model will also serve as a valuable tool in achieving a thorough understanding of the general tradeoffs in fine-grain multicomputing.

More importantly, the high efficiency and robustness in this architecture paves the way for an upcoming breed of massive multicomputers that incorporate thousands of processors. The excellent potential of such machines are already reflected in the performance/overhead sensitivity analysis in this study, while their economical viability are evident from the rapid emergence of affordable, highly-integrated components [2, 3, 4, 5, 6]. These machines are the future of high-performance computing, and the ability to exploit fine-grain parallelism – so that they are readily applicable to existing problems and not just exotic, huge-size programs – is the key to their success.

Specifically, the current research trends point to the emergence of three broad classes of high-performance processor chips: (a) multiprocessors-on-a-chip [61, 4], (b) memory-integrated processors [53, 62, 6], and (c) multithreaded (superscalar, multi-) processors [63, 1, 2, 3, 64]. All of these are aimed at exploiting the ever-growing chip

area more efficiently, without relying on the diminishing improvements provided by the currently-popular, but increasingly-complex, superscalar architectures.

The first category integrates multiple independent processors on to each chip. In current designs [61, 4], the geographical locality is not exploited to provide direct processor-to-processor communication – they communicate via a shared L2 cache or other on-chip memory. More recently however, the idea of an on-chip routing network is proposed [7]. While this gives rise to an extremely fast network fabric, the performance would be hindered if a conventional network interface is employed. To this end, the very efficient M-Machine messaging interface, or parts of it, can provide the necessary solution.

By augmenting the processor chip with random-accessed memory, or vice-versa, the second category of integrated components avoid the bandwidth and latency limitations of the external memory bus. Such architectures however accentuate the discrepancy between local-memory and remote-memory access times. Since the on-chip memory is finite, performance must plunge quickly when the frequency of references beyond the chip boundary increases due to a large data size, myriad sharing patterns or the simple lack of locality in the application. In such cases, the extremely-low overhead in the M-Machine messaging system allows a gradual transition between single-chip and multi-chip computation.

The M-Machine itself falls into the third category, where multiple hardware contexts or thread slots are incorporated into each processing pipeline. Although the designers of each system tend to experiment with a different collection of specific features [63, 1, 2, 3, 64], they all share a common ground – the critical resources are time-shared by many different threads-of-control on these chips. Protection and starvation avoidance are therefore important concerns in designing a communication system for such machines. These issues are well-addressed by the M-Machine mechanisms.

It is foreseeable, therefore, that the M-Machine message architecture, and its derivatives, will play an important role in the future of high-performance multicomputing.

Bibliography

[1] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, Whay S. Lee, "The M-Machine Multicomputer," in *Proceedings of the 28th Annual International Symposium on Microarchitecture, 1995.* pp. 104–114.

[2] Dean M. Tullsen, Susan J. Eggers, Henry M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", in *ISCA, 1995.*

[3] Gurindar S. Sohi, Scott E. Breach, T. N. Vijaykumar, "Multiscalar Processors", in *ISCA, 1995.*

[4] Tadaaki Yamauchi, Lance Hammond, Kunle Olukotun, "A Single Chip Multiprocessor Integrated with DRAM", in *ISCA, 1997.*

[5] Kimberly Keeton, Remzi Arpaci-Dusseau, David Patterson, "IRAM and SmartSIMM: Overcoming the I/O Bus Buttleneck", in *ISCA 1997.*

[6] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, Katherine Yelick, "A Case for Intelligent RAM: IRAM", in *IEEE Micro, 1997.*

[7] Mark Horowitz, "Smart Memories: A Universal Computing Element", Project Proposal, DARPA Contract Number MDA904-98-C-A933, Stanford University, 1998.

[8] Steve Keckler, "Fast Thread Communication and Synchronization Mechanisms for a Scalable Single Chip Multiprocessor", PhD. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

[9] Mengjou Lin, Rose Tsang, David H. C, Du, Alan E. Klietz, Stephen Saroff, "Performance Evaluation of the CM-5 Interconnection Network", in *COMPCON 1993*, pp. 189–197.

[10] Michael D. Noakes, Deborah A. Wallach, William J. Dally, "The J-Machine Multicomputer: A Architectural Evaluation", *ISCA 1993*, pp. 224–235.

[11] William J. Dally, Ming-Ju Edward Lee, Fu-Tai An, John Poulton, Steve Tell, "High-Performance Electrical Signaling", in *the Proceedings of the Fifth International Conference on Massively Parallel Processing Using Optical Interconnections, 1998.*

[12] Mark Horowitz, Chih-Kong Ken Yang, Stefanos Sidiropoulos, "High-speed Electrical Signaling: Overview and Limitations", in *IEEE Micro, January/February 1998*, pp.12–24.

[13] Nicholas P. Carter, Stephen W. Keckler, William J. Dally, "Hardware Support for Fast Capability-Based Addressing", in *ASPLOS VI, 1998.*

[14] Martin Gardner, "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game life", in *Scientific American, October, 1970*, pp. 120–123.

[15] John Kubiatowicz, Anant Agarwal, "Anatomy of a Message in the Alewife Multicomputer", in *ICS 1993.*

[16] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer", *ISCA 1994*, pp. 142–153.

[17] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, "The Stanford FLASH Multicomputer", in *ISCA 1994*, pp. 302–313.

[18] Michael Beckerle,"Overview of the START (*T) Multithreaded Computer", in *COMPCON 1993*, pp. 148–156.

[19] Jiun-Ming Hsu, Prithviraj Banerjee, "A Message Passing Coprocessor for Distributed Memory Multicomputers", in *SuperComputing 1990*, pp. 720–729.

[20] Shahid H. Bokhari, "Communication Overhead on the Intel Paragon, IBM SP2 and Meiko CS-2", *NASA Contractor Report 192811, 1995.*

[21] Roger W. Hockney, "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2", Parallel Computing 1994, pp. 389–398.

[22] Toshiyuki Shimizu, Takeshi Horie, Hiroaki Ishihata, "Low-Latency Message Communication Support for the AP1000", in *ISCA 1992*, pp. 288–297.

[23] Klaus E. Shauser, Chris J. Scheiman, "Experience with Active Messages on the Meiko CS-2" in *IPPS 1995*, pp pp. 140–149.

[24] Thomas E. Anderson, David E. Culler, David A. Patterson, "The Berkeley Networks of Workstations (NOW) Project", in *COMPCON 1995*, pp. 322–326.

[25] Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Liviu Iftode, Kai Li, "Early Experience with Message-Passing on the SHRIMP Multicomputer", *ISCA 1996*, pp. 296–307.

[26] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve G. Steinberg, Katherine Yelick, "Empirical Evaluation of the CRAY-T3D: A Compiler Perspective", *ISCA 1995*, pp. 142–153.

[27] Susan Hinrichs, Corey Kosak, David R. O'Hallaron, Thomas M. Stricker, Richiro Take, "An Architecture for Optimal All-to-All Personalized Communication", in *SPAA 1994*, pp. 310–319.

[28] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul,

Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, Robert Zak, "The Network Architecture of the Connection Machine CM-5", in SPAA 1992. pp. 272–285.

[29] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, David A. Wood, "Coherent Network Interfaces for Fine-Grain Communication", in *ISCA 1996.* pp. 247–258.

[30] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Walter Lee, Victor Lee, Anant Agarwal, M. Frans Kasshoek, "Exploiting Two-Case Delivery for Fast Protected Messaging", in *HPCA 1998.* pp. 231–242.

[31] The MPI Forum, "MPI: a message passing interface", in Supercomputing '93, pp. 878–883.

[32] Paul Pierce, "The NX/2 operating system", in Proceedings of the 3rd Conference on Hypercube concurrent Computers and Applications, 1988. pp. 384–390.

[33] Ioannis Schoinas, Mark D. Hill, "Address Translation Mechanisms in Network Interfaces", in *HPCA 1998*, pp. 219–230.

[34] Vijay Karamcheti, Andrew A. Chien, "Software Overhead in Messaging Layers: Where Does the Time Go?", in *ASPLOS VI, 1994*, pp. 51–60.

[35] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Shauser, "Active Messages: a Mechanism for Integrated Communication and Computation", in *ISCA 1992.*

[36] Dana S. Henry, Christopher F. Joerg, "A Tightly-Coupled Processor Network Interface", in *ASPLOS V, 1992*, pp. 111–122.

[37] William J. Dally, "Virtual Channel Flow Control", in *ISCA 1990.* pp. 60–68.

[38] David E. Culler, Lok Tin Liu, Richard P. Martin, Chad Yoshikawa, "LogP Performance Assessment of Fast Network Interfaces". in *IEEE Micro 1996.*

[39] Eric Barton, James Cownie, Moray McLaren, "Message Passing on the Meiko CS-2", in *Parallel Computing 1994*, pp. 97–507.

[40] Vijay Karamcheti, Andrew A. Chien, "A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D", in *ISCA 1995*, pp. 298–308.

[41] John Heinlein, Kourosh Gharachorloo, Scott Dresser, Anoop Gupta, "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor", in *ASPLOS VI, 1994*, pp. 38–50.

[42] Cezary Dubnicki, Angelos Bilas, Kai Li, James Philbin, "Design and Implementation of Virtual Memory-Mapped Communication on Myrinet", in *IPPS 1997*, pp 388–396.

[43] Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H. T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, Jon Webb, "Supporting Systolic and Memory Communication in iWarp", in *ISCA 1990*, pp. 70–81.

[44] T. D. Wagner, E. Smirni, A. W. Apon, M. Madhukar, L. W. Dowdy, "The Effects of Thread Placement on the KSR1", in *IPPS 1994*, pp 618–624.

[45] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, Yoshinori Yamaguchi, "Experience with Executing Shard Memory Programs Using Fine-Grain Communication and Multithreading in EM-4", in *IPPS 1994*, pp pp. 630–636.

[46] James Laudon, Daniel Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server", *http://www.sgi.com*, Silicon Graphics Inc.

[47] Daniel Lenoski, James Laudon, Truman Joe, Dvaid Nakahira, Luis Stevens, Anoop Gupta, John Hennessy, "The DASH Prototype, Implementation and Performance", in *ISCA 1992*, pp 92–103.

[48] Stephen W. Keckler, William J. Dally, "Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism", in *ISCA, 1992*, pp. 202–213.

[49] Nicholas Carter, "Hardware Mechanisms for Efficient, Flexible, Shared Memory", PhD. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1999.

[50] Steven L. Scott, "Synchronization and Communicationin the T3E Multiprocessor", in *ASPLOS VII. 1996.* pp. 26–30.

[51] Jeff Bowers, "The Design and Implementation of the Bidirectional Pads", CVA Memo 102, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1996.

[52] William J. Dally, Larry Dennison, David Harris, Kinhong Kan, Thucydides Xanthopoulos, "Architecture and Implementation of the Reliable Router", in *Proceedings of Hot Interconnects, 1004.* pp. 122–133.

[53] William J. Dally, Linad Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, Scott Wills, "Architecture of a Message-Driven Processor", in *ISCA, 1987.* pp. 189–196.

[54] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, Whay S. Lee, "The MAP Instruction Set Reference Manual v1.55", CVA Memo 59, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1997.

[55] John E. Hopcroft, Jeffrey D. Ullman, "Introduction to Automata Theory, Languages and Computation", Addison-Wesley Publishing Company, Inc. 1979.

[56] Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham, "Efficient Software-Based Fault Isolation", in *SIGOPS, 1993*, pp. 203–216.

[57] Lok T. Liu, David E. Culler, "Evaluation of the Intel Paragon on Active Message Communication", in *Proceedings of Intel Supercomputer users Group Conference, 1995.*

[58] Charles M. Burns, Robert H. Kuhn, Eric J. Werme, "Low Copy Message Passing on the Alliant CAMPUS/800", in *Supercomputing, 1992.* pp. 760–769.

[59] Steve Keckler, Whay S. Lee, Nick Carter, "MSIM UsersGuide", CVA Memo 63, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1994.

[60] David E. Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, Thorsten von Eicken, "LogP: Towards a Realistic Model of Parallel Computing" in *Proceedings of the* 4^{th} *Symposium on Principles and Practices of Parallel Programming, 1993.*

[61] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, Kunyung Chang, "The Case for a Single-Chip Multiprocessor", in ASPLOS VII, 1996.

[62] Ashley Saulsbury, Fong Pong, Andreas Nowatzyk, "Missing the Memory Wall: the Case for Processor/Memory Integration", in *ISCA, 1996.* pp. 90–101.

[63] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Proterfield, Burton Smith, "The Tera Computer System", in Supercomputing, 1990. pp. 1–6.

[64] Bernard Goossens, Duc Thang Vu, "On-Chip Multiprocessing", in *the Proceedings of the* 2^{nd} *International Euro-Par Conference 1996.* pp. 789–796.