EFFICIENT MESSAGE SUBSYSTEM DESIGN

by

Whay Sing Lee

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1994

Author _____

Department of Electrical Engineering and Computer Science

Certified by _____

William J. Dally

Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by _____

Frederic R. Morgenthaler

Chairman, Departmental Committee on Graduate Students

EFFICIENT MESSAGE SUBSYSTEM DESIGN

by

Whay Sing Lee

Submitted to the

Department of Electrical Engineering and Computer Science

May 12, 1994

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science

Abstract

As more powerful processors and faster networks are developed, the message interface is becoming one of the bottlenecks in modern multicomputer systems, and the importance of more efficient message interfaces is gaining. The overall performance of the message interface is influenced not only by the efficiency of the processor/network coupling, but also by the efficiency of the system in performing other end-to-end tasks in a message operation, particularly *target name resolution*, *protection enforcement*, and *resource management*. To achieve optimum performance, it is necessary for the message subsystem to take into consideration all these concerns and attempt to provide balanced and complementary support for them. This thesis examines these issues, and develops an integrated message subsystem which incorporates efficient and flexible mechanisms working together in a mutually supportive manner, with a simple software-hardware interface.

Thesis Supervisor: William J. Dally

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgement

The message subsystem architecture described in this thesis represents the result of a combined effort by the members of the MIT *M-Machine* designers team consisting of William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang and Marco Fillo, in addition to the author.

The author wishes to register his gratitude to Professor William Dally for his guidance, to the M-Machiners for their cooperation and kind assistance, and to the members of the MIT Concurrent VLSI Architecture group for their inputs to this study. The author is particularly grateful to Ellen Spertus for providing a sample implementation of `ifetch/istore` on the J-Machine, and to Nate Osgood and Duke Xanthopoulos for their gracious help in proofreading this writing.

The research described in this thesis was supported in part by the Defense Advance Research Projects Agency of the Department of Defense under contract F19628-92C-0045.

Contents

List of Figures

Chapter 1

Introduction

With its ability to efficiently support most of the major programming paradigms [1], message-passing has been widely adopted as the underlying communication mechanism in many contemporary multicomputers, such as [2, 3, 4, 5, 6]. This thesis examines the important factors in the design of an efficient integrated subsystem for message-passing communication, focusing on the message interface itself, sans the network proper.

In the next section, we outline the purpose of this thesis. Section 1.2 discusses the background and related previous efforts. We state our assumptions and terminology in section 1.3, and a framework of the proposed architecture is briefly summarized in section 1.4. The organization of rest of this writing is included at the end of this chapter.

1.1 Purpose

As more powerful processors and faster networks are developed, the message interface is becoming one of the major bottlenecks in modern multicomputer systems. Consequently, the design of more efficient message interfaces are gaining importance.

To design an efficient message interface, several factors, in addition to fast processor/network coupling hardware, must be considered. From a user's perspective, the significant criteria are the effective cost and complexity of accomplishing a desired

remote operation. This cost and complexity are in turn influenced by the efficiency of the system in performing other end-to-end tasks in a message operation, particularly *target name resolution*, *protection enforcement*, and *resource management*.

Unfortunately, the need to provide adequate support for these functions, especially protection, can often conflict with the desire to maximize efficiency. To achieve optimum performance, it is necessary for the message subsystem designer to take into consideration all these concerns, and attempt to provide balanced and complementary support. The purpose of this study is to develop such an integrated message subsystem which incorporates efficient and flexible mechanisms working together in a mutually supportive manner, with a simple software-hardware interface. We will focus particularly on the three issues indicated above.

1.2 Background and Previous Efforts

The *message interface* is the mechanism and protocol which dictates the interaction between software threads and the physical words transported in a message. Generally, this interaction consists of the following phases:

- Composition: Creating the message.
- Sending: Initiating the message delivery process.
- Injection: Moving the data physically into the network/router.
- Transport: Moving the data across the network.
- Receiving: Extracting the data from the network/router.
- Signalling: Invoking the appropriate software thread to handle the message.
- Handling: Fulfilling the request carried in the message.

While these components need not constitute distinct and separate modules of the message subsystem, each represents a required step in a message-based communication, and each has an influence on the overall performance of the system. A balanced

message subsystem must therefore provide adequate support for each of these phases. Also note that the above-mentioned end-to-end functions should be considered as parts of the *composition* and *handling* steps.

As our focus is on message interface design, the study of the transport component is beyond the scope of this thesis.

A number of message interface studies have aimed at reducing overhead in the sending/injecting and the receiving/signalling of a message. In most cases, this is accomplished by exposing primitive mechanisms to the application programmer, in order to open opportunities for various optimizations [5, 6, 7]. At the sending point, the overhead of making a system call to inject a message after it is generated is typically avoided by providing direct network port access to the user. Some, like [8], also attempted to mask latency by overlapping communication with computation. At the receiving end, a software thread is usually invoked to handle the message from the network, per Message Driven [9] or Active Message [5, 10] styles, which have been shown to provide much flexibility. The result from these are simple interfaces with low overhead.

The issue of naming concerns the need to unambiguously locate an object regardless of its physical position, which may change dynamically due to migration, or statically due to load-time relocation. Protection on the other hand is concerned with restricting changes in the state of the machine that can be effected by each user. It has to be position-independent, given that the potential relocation of protected states. In [6], a software-manipulated *Global Virtual Address* is offered as a solution to the naming issue. Software implemented name resolution schemes however, generally incur unacceptable overhead or protection risk in a fine grain environment, as the system programmer decides between restricting the name resolution function to a protected but costly system call, or making it accessible as an unprotected but less costly library routine, for example. A hardware-assisted XLATE instruction using a software-managed name cache is provided in [7], offering a faster name translation functionality, but is nonetheless non-transparent to the user, nor does it enforce any restriction on the set of targets addressable by a thread.

Other recent studies have seemingly converged towards the approach of building the naming and/or protection functionalities on top of the virtual memory system, such

as in [5, 11, 12]. Such approaches exploit the protection mechanism already present in the memory system and extend the virtual memory mapping mechanism to indicate remote accesses. Others, like [4], have taken the stance that it is sufficient to protect only system-level states, and designed the hardware to exclude user access to those states. Our study extends the virtual memory mapping approach to provide a finer grained, user-transparent protection scheme.

The issue of resource management is concerned with avoiding over-commitment of critical resources required for message operations, which include buffering space, network bandwidth and processor thread slots. Not many systems have thus far provided *primitive* mechanisms to support this function, although an efficient mechanism for this function is needed to ensure balanced performance along the end-to-end message path. In this study, we build on the buffer management scheme implemented in the Cray T3D [12], which we show in chapter 4 to be sufficient for avoiding exhaustion of the resources above.

Further discussion of these efforts will be found in the corresponding chapters in the rest of this thesis.

1.3 Assumptions and Terminology

There are a number of assumptions made in this study about the environment in which the message subsystem will be functioning, other than the platform being a message-passing multicomputer. These assumptions are:

- Multi-user environment. We assume that multiple *independent* programs/jobs can share each processor concurrently to optimize machine utilization. The usual protected operating system schemes are expected to apply here.
- General register load-store architecture. Our study is geared toward machines of this dominant genre.
- Custom chips. We assume the designer has the freedom to design a custom chip, so that the message interface can be tightly integrated with the processor.

- Multiple hardware context. The design is optimized for fine grain multi-threading on a multiple hardware-context machine [13].

Throughout this writing, we use the term "*payload*" to refer to the data that is transferred in a message, to distinguish it from the "*request*", which we use to refer to the *operation* requested to be performed. The term "*receiver thread*" is used to refer to a thread that is actively *expecting* a particular message.

1.4 Proposed Architecture

We propose a message subsystem architecture that integrates tightly into the processor, and leverages off the system's other existing mechanisms [1]. To provide a position-independent naming environment where name resolution is fast, and transparent to the user, we abandon the more traditional practice of addressing messages to tuples of (*node*, *memory location*), in favor of a globally named message addressing scheme using a two-layer, hardware-cached name translation mechanism. Protection in this system is enforced by a combination of extending the protected pointer types from the M-Machine memory subsystem [14] and restricting the messages to invoke only privileged handlers at the destination. To minimize latency, we try to eliminate redundant buffering/copying in the message's path, by mapping the message composition space and the network port to the processor's general registers [15, 16, 17]. Finally, to prevent network congestion, a simple hardware/software solution is included to support T3D-style [12] message throttling, by inhibiting non-critical messages when buffering resource is near exhaustion.

The architecture proposed in this thesis is inspired by a number of previous studies and discussions with members of the MIT Concurrent VLSI Architecture group. It also builds on mechanisms originally developed in other domains of the MIT M-Machine[13]. The author's contribution is in the design of:

- A register-mapped, FIFO-based message input interface that exploits the M-

[1] The M-Machine multicomputer [13] provides the platform for the studies in this thesis.

Machine's *stall* mechanism [2] for fast dispatch.

- A register-mapped, dual-bank, compose-and-launch message output interface using SEND operations that permit independent specification of (recipient × request × payload) in each message.
- A protection scheme that exploits the M-Machine's protected pointer mechanism to provide control over the target objects addressable by a sender thread, as well as the remote operations it may invoke on the target object.

1.5 Outline

In Chapter 2, we introduce naming and protection schemes in the context of the message subsystem. It is followed by the description of the send/receive facility in Chapter 3. The description of the architecture comes to a completion with the message throttling mechanism in Chapter 4. Chapter 5 provides an evaluation and a skeleton implementation of the proposed message subsystem. We conclude this writing with some suggestions for further investigation in Chapter 6.

[2]The *stall* mechanism prevents an instruction from issuing until all its operands become *present* [13].

Chapter 2

Naming and Protection

In this chapter, we consider two properties that are crucial in any modern computing system, without which producing any deterministic program behavior would be exceedingly difficult:

- Given the name of an object, one should be able to unambiguously locate the object.
- One should be confident that pieces of critical state are protected from undesired/unexpected modification by unauthorized processes.

In a multicomputing environment, objects may potentially reside on different physical nodes at different times (*e.g.* due to load-time/dynamic relocation), while each object may further be distributed across a collection of physical nodes. The naming scheme must therefore be able to uniquely locate each interesting piece of an object regardless of its current physical position, and protection must be upheld across node boundaries. In the following sections, we present a scheme that provides this ability in the context of the message subsystem.

2.1 Naming and Messaging on the M-Machine

Given that code/data may not always reside on a single fixed physical node, a message addressing method using $(node, address)$ pairs is not quite suitable in a multicom-

puter. Therefore, from the point of view of a user-level programmer, M-Machine messages are always addressed to a target *object.* In this context, an object can be any software/hardware entity, such as data structures, threads and process groups, memory locations and nodes. Each object is identified by a unique global *virtual name/address*, implemented as a user-unforgeable data type (the address *pointer* data type on the M-Machine).

Using global virtual names in messages allows the programmer to reference an object in always exactly the same way, regardless of its relative position from the caller, and decoupled from the actual mapping between the virtual naming space and the physical resources. The system provides this position-independent addressing by putting the target object name of messages through a two level translation mechanism [1]:

- *Global translation* to identify the physical node currently hosting the interesting piece of the object named. This information is used by the low-level network hardware to deliver the message.
- *Local translation* at the destination node to further translate the virtual address of the target object (which is passed along to the destination within the message) into its corresponding physical address within the destination node, or to detect conditions requiring exceptional handling, such as in the case of a migrated target object.

To provide the translation services at minimum overhead without overwhelming hardware costs, the two translation layers are made up of translation tables with hardware look-aside buffers (TLBs). A *Global TLB (GTLB)* is used for the global translation layer, while local translation is facilitated by a *Local TLB* (LTLB). These TLBs and their associated translation tables are similar to the translation mechanism commonly used in conventional paging virtual addressing systems. Misses in these TLBs cause the system to trap into appropriate routines to respond accordingly. In either case, the entire translation scheme is completely hidden from user processes. A brief description of an implementation of the GTLB can be found in Chapter .5.

[1] This idea came out from a J-Machine evaluation study [18]

With this translation mechanism, state changes that are local to the node need only be reflected in the local translation layer, without being communicated to other parts of the machine. Even for dynamic remapping between names and host nodes, such as in an object migration, the change need only be registered in the global/local translation layers of the previous and current host nodes. No extra network traffic is generated until the object is indeed referenced again. When a message does eventually arrive for a relocated target object, it can be forwarded to the object's new location by the LTLB-miss trap routine described above, which may optionally also notify the sender to modify its GTLB accordingly. The system is therefore capable of providing a very flexible global naming environment. At the same time, the use of unforgeable object names in message addressing also facilitates the tight integration of system-wide protection, as described next.

2.2 Protection and the Message Subsystem

Protection refers to the systematic restriction of each user's ability to effect changes in the state of the machine. Where the message subsystem is concerned, this restriction can be achieved in one or both of the following ways:

- Enforce permission checking at the receiving end, before the request carried in the message is fulfilled. Under this approach, the receiver has finer, and more dynamic, control over what requests it wants to entertain. The disadvantage, however, is that unauthorized requests would have consumed network bandwidth by the time they are rejected.

- Enforce permission checking at the sending end, before the message is injected. Since the system must record the permission assignments concerned in this approach, it is really more suitable for permission assignments that change infrequently. It has the advantage, however, of being able to avoid unnecessary network traffic due to unauthorized remote accesses.

The primary protection mechanism of the M-Machine message subsystem falls into the latter category. The scheme heavily exploits the user-unforgeability of the *pointer*

data type mentioned earlier. On the M-Machine, each pointer contains three subfields [13]: the *Address* field which identifies the object being named by this pointer, the *Permissions* field, which specifies the operations permitted on the object, such as *read only*, *read/write* and *execute*, and the *Length* field, which limits the extent of the domain that can be derived from this pointer. In the message subsystem, the *Permissions* field provides the basic means for enforcing system-wide protection.

Two pointers are required in order to send a message from a user program, using a SEND operation. The first pointer, referred to as the *destination virtual address*, is the name of the target object as described earlier. This pointer is passed along to the destination, with its permission field preserved and honored at the destination node. Therefore, any access control scheme enforced via the pointer permission field is automatically extended to cover the entire system.

The second pointer required, known as the *dispatch instruction pointer* (dispatchIP), must have the *execute message* permission, which is mutated by the network interface into *execute* when the message is injected. This pointer names a code fragment at the destination called the message *handler*, which is to be invoked to operate on the body of this message at the destination upon its arrival. It should be noted that on the M-Machine, the handler runs concurrently with other threads within the processor, in its own dedicated hardware context. Therefore no interrupt is caused by a message arrival.

Failure to adhere to the above pointer conditions results in the system trapping into an exception routine.

From the fact that pointers are user-unforgeable, two properties can be inferred about the scheme described:

- Regardless of the physical location of an object, it can be accessed only by threads that possess a copy of the pointer to it, and the access may only be in the manners permitted by that pointer.
- The system has complete control over all message handlers, and a thread may invoke a remote operation only if it possesses a copy of the pointer to the corresponding handler.

From the above, we see that the permission field of the target object name alone provides sufficient global protection for simple read/write/execute operations on the named object. The *dispatch instruction pointer* further protects classes of critical remote operations from being invoked by unauthorized senders. For more sophisticated protection needs, the message handler, being a fully programmable code fragment, has the ability to enforce any receiver-end permission-checking as necessary.

In addition to the above forms of access control, the message subsystem also guarantees against hogging of message resources by any user thread. This is accomplished by making message injection an atomic operation:

- A message cannot be initiated until it is, in its entirety, ready for immediate injection into the network.
- A message cannot be interrupted once it has been initiated.
- Injection of a message into the network is guaranteed to complete within a fixed amount of time after it was initiated. The resources are released automatically after injection is complete.

The atomicity of message injection is enforced in hardware, without explicit software interlocks. The system is therefore protected from a user thread's failure to release control over the shared network resource after being granted permission to deliver a message. It also ensures that the message resource is allocated to a thread only for the actual duration the resource is needed by the thread, so that the resource is never unnecessarily tied up.

At the same time, we require each message handler at the receiving end to be a trusted thread, guaranteed to complete quickly without risk of a deadlock. This is enforceable since the system software has complete control over which threads may be specified as message handlers. For a more elaborate remote operation that may run for a long time, or involve dependencies on the network ports, the message handler schedules a separate thread to complete the operation in a regular thread slot (other than the message thread slot). This eliminates the possibility of the network input port becoming inaccessible.

The above two conditions guarantee that both the network input and output ports never become continuously inaccessible for an unbounded period of time. As far as the message subsystem is concerned, a user thread can thus never preclude other processes from accessing the network resources. Therefore, this arrangement serves to prevent user threads from causing deadlocks that compromise the operation of the entire system.

As will be discussed in Chapter 4, each user message generates an *Acknowledge* response when it is accepted by the destination node. This *Acknowledge* message must be returned to the original sender node for correct operation of the system. To prevent the sender from misdirecting the *Acknowledge* response to a different node, the network interface hardware attaches the node identification of the sender to each message. This provides a trusted return node ID to which the handler can deliver the Acknowledge message. A *count* value is also attached similarly, to provide a trusted account of the message length to the receiver for any necessary checking.

Finally, two independent message *priorities, 0/1,* are provided on decoupled network resources, where only priority 0 resources are accessible to user programs. By reserving the priority 1 resources for privileged threads, the system is protected from catastrophic conditions where system management messages are blocked / delayed due to user errors. The *Acknowledge* messages described above are generated by the message handler on priority 1.

It should also be pointed out that the system software is capable of constructing/mutating pointers, and can therefore bypass any of the pointer-related restrictions above.

2.3 Related Efforts

Traditional approaches to the protection issue include requiring explicit permission-checking in software, restricting message resource to be accessible only via system calls, and enforcing gang scheduling [6]. Unfortunately, without adequate primitive mechanisms, these approaches were often not reliable, incurred too much overhead, or provided only a coarse level of protection.

In the more recent generation of multicomputers however, efficient protection schemes have been attracting more attention. Many of these studies have converged towards exploiting/extending the virtual memory system to address the protection needs.

The CM-5 [5] for example directly maps the network port to memory locations, and a message is generated by directly writing the words to be delivered into these memory locations. On the Cray T3D [12], a message is implicitly generated by a cache-line write to a remote memory address, which is subsequently translated by hardware into a destination node identifier needed for routing. The user-level message interface in the Stanford FLASH [11] system is accessed via memory-mapped commands which invoke a *transfer handler* on its MAGIC controller chip to perform the message delivery. Given that the memory locations concerned can be mapped into protected pages of the memory address space, these systems can therefore restrict access to message operations and/or prevent messages from being sent to specific addresses.

The Alewife [19] system holds the philosophy that protection is enforced only to prevent errant user code from accessing system level operations (differentiated by the *major opcode* value). The message interface is based on a memory-mapped *output descriptor array*, which is used to describe the message before it is atomically launched. Hardware restriction is imposed such that system level message operations are not accessible to the user.

The approaches taken by these system are quite adequate for isolating critical system states from user processes. They differ from our protection model however in that the remote *request* in most of these systems is generally specified by an unprotected opcode. This implies the lack of full control over permissible combinations of (sender $\times$ recipient $\times$ request), a flexibility feature for enforcing fine grain protection. At the same time, our approach also deviates conceptually from most existing systems in that our messages request an operation to be performed on a target object, instead of asking a receiver thread to perform an operation. Finally, our two-level translation scheme provides additional flexibility in protection enforcement, as discussed in the previous section.

Chapter 3

Processor-Network Interface

In this chapter, we present the design of an efficient interface between the software and the hardware in the message subsystem. Our major goal is to minimize the overhead of sending and receiving messages while preserving system-wide process isolation. It is important that the primitives presented to the user for message sending/receiving are simple and efficient so that programmers are not driven to amortize the effort/cost by using only large, infrequent messages. The interface mechanisms must also offer enough flexibility to support development of higher level protocols on top of them.

3.1 Common Models of Network Interface

In this section, we begin by surveying the common models of network interface designs.

3.1.1 Network Output Interface

In terms of the network output unit, existing models can be broadly divided into two classes:

- Compose and Launch: The sender thread is required to first compose the complete message in some buffering space, and then launch the message into the network via some mechanism such as executing a dedicated instruction or accessing a special register/memory location, such as in [6, 8, 19, 20].

- Generate and Inject: The sender injects the message into the network on-the-fly as each word is produced [5, 7].

It may appear that the former approach is at a disadvantage since each word in the message must be copied into the buffer, and out again into the network. In contrast, in the latter category , no such copying takes place. However, such comparison does not take into account the protection issue. To ensure against interruption of an on-going message and splicing together of messages, the Generate-and-Inject model must rely on some semaphore scheme. This semaphore, whether implemented in hardware or software, guarantees an atomic region of execution in the sender thread while the message is being injected. Note that this implies a dependence on the sender thread to release the semaphore when it is done injecting the message. Such dependency risks deadlocks and resource-hogs when the sender is a user level thread.

Conceivably, a watchdog timer may be used to interrupt the processor after a period of inactivity, to avoid the above deadlock situation. There is a complication however, in that the network input must then be augmented to handle incomplete messages, *e.g.* when a thread is interrupted due to a timeout condition while in the middle of injecting of a message. This also implies that the message cannot be handled until it is received in its entirety, or the handler must take special care to be prepared for the case where the message it is handling turns out to be incomplete.

On the other hand, the compose-and-launch approach has a well-defined upper bound of time after the Launch operation during which the sender needs to be granted the network output resource. It is thus safe to provide direct network output access to user processes without deadlock risks, and without system software intervention.

A number of variations may further exist in the compose-and-launch category, that revolve around the mapping/location of the buffering space. This buffer space may be *addressed* as register(s), or memory location(s), or accessed via special instructions, while physically *residing* in dedicated registers, or main memory, or a special FIFO storage, for example. The resulting efficiency of the interface is influenced by both the logical and physical mappings of the buffer. The logical mapping dictates the manner in which the buffer is accessed (and hence the number of instruction cycles required to perform this access) while the physical mapping determines how fast the

buffer may be accessed (*i.e.* the latency up to the completion of the access). A study of the relative merits of each variation is found in [17], which suggests that the overhead can be minimized by mapping the buffer to processor register address space. As will be shown in the next few sections, by further mapping the buffer physically into the processor registers, the copying overhead objected to earlier can be almost eliminated. Taking into account the deadlock issue, compose-and-launch can therefore be the preferred message subsystem model in a multi-user system.

3.1.2 Network Input Interface

In the recent network input designs, there are generally the following approaches to message handling:

- Direct access: In this case, it is assumed that the message is *expected* by a certain receiver thread, at either the application or the system level, which will directly consume the message into its computation. This receiver thread is given direct access to the network interface so that it can actively extract message words from the network buffer via some mechanism such as special registers/instructions. An example is the *pathway* mechanism on the *iWarp* [16].

- Hardware intensive interface: The network interface controller itself interprets and performs the request in the message. It may sometimes trap into software to extend its capability. Two examples can be found in the *P-Machine* [21] and the *Network Interface Processor* [22].

- Software Handlers: Each message specifies a trusted handler to be invoked upon its arrival at the destination. The trusted handler has direct access to the message contents and performs the requested operation. Examples include the MIT *J-Machine* [7] and the Fujitsu *AP1000* [8].

The direct access model allows a receiver thread to have very fast access to the message, and is most efficient when the programming model is based on threads directly communicating with each other. It is however not appropriate for a multi-user system. Since user-level receiver threads will have access to the network resource,

in order to preserve process isolation, the sender/receiver threads must be scheduled in a synchronized fashion, such that the correct receiver is always monitoring the network interface when the corresponding message arrives. Alternatively the message arrival must raise a processor interrupt which then wakes up the right receiver to extract the message. (An interrupt may also be required when the incoming message has no well-defined and awaiting "receiver", *e.g.* a remote procedure call.) Being costly operations, both alternatives may well defeat attempts to exploit fine-grain parallelism.

The hardware intensive approach is most appropriate where a small collection of remote operations dominate the message traffic. These frequent operations could be wired-in into fast hardware, while other less frequent operations trap into software to complete. It is however not economical to implement complicated functions *entirely* in hardware, given that the marginal increase in performance is nonetheless limited by the availability of other shared resources in the system. Therefore, as we take into account the required end-to-end functions in a message operation, the performance/cost ratio of a hardware intensive interface is not likely to be substantially higher than a software-handler based interface.

On the other hand, software handlers offer a much higher degree of flexibility over the other approaches. In a multi-threading / multi-user environment, this flexibility becomes even more essential as we consider the increasing complexity in the operations required of a message. (Consider, for example, the need to manage semaphores, maintain various models of memory coherence and arbitrate for shared resources.) Adding the low hardware cost advantage, this approach is therefore our preferred choice.

3.2 The M-Machine Message Architecture

As discussed, the compose-and-launch and software handler message interface models are the appropriate choices for the environment of our study. In this section, we present the design of the interfaces in detail.

3.2.1 Network Output Interface

The network output interface adopts a *user-level* compose-and-launch mechanism, using a subset of the *general* registers directly as the compose buffer. These registers, called the Message Composition registers (*MC registers*), are in fact regular processor registers, which are used as source/destination in normal instructions, just like any other general register.

There are two independent *banks* of *MC* registers on each M-Machine *cluster* [13] corresponding to its two register files. Each *MC* register bank has 10 registers, plus an extra two which are accessible to system level threads for system management.

A non-blocking SEND instruction is used to launch the message from the appropriate bank of *MC* registers (from *MC#0* to *MC#length-1*) into the network. It also specifies the destination and the *request*, in the form of a *destination virtual address* and a *dispatchIP*, as described in the previous chapter:

```
SEND0 bank length dest dispatchIP ccreg
```

Injection of each message into the network output buffer is guaranteed to be an atomic operation. This is achieved by stalling the SEND operation until all of the *MC* registers involved are present [1]. Once the SEND operation is issued, the message words are non-interruptably extracted from the *MC* registers into the network output buffer. This atomicity guards against mutation of messages by a third party, and against deadlocks due to network resource hogging.

Each SEND instruction also specifies a Condition Code target register, `ccreg` (for predicated execution). This `ccreg` is marked *not-present* by the hardware when the message send is initiated, only to be is set to TRUE and present again by the hardware when the entire message has been extracted from the *MC* registers over a number of cycles. This is used as a barrier to stall attempts to mutate the *MC* registers before their contents have been extracted. All operations may proceed normally immediately after a SEND is executed, so long as they do not mutate the *MC* registers before `ccreg`

[1] The *presence bit* mechanism keeps track of the validity of the content in each register [13].

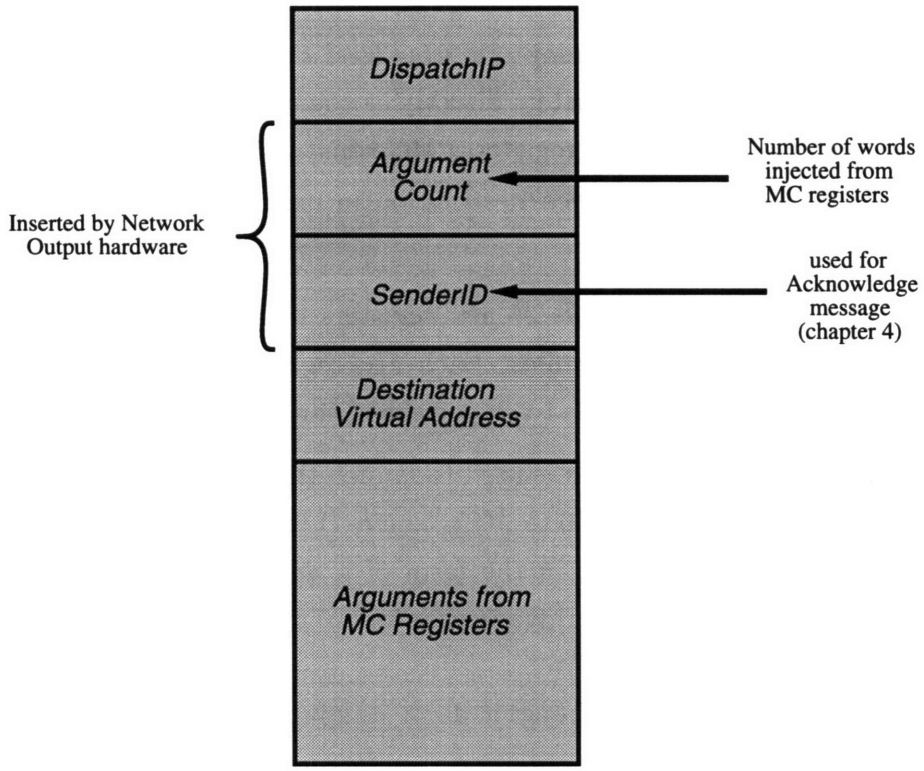

Figure 3.1: Message Format

becomes present. Using a *MC* register as a source before `ccreg` becomes present is however permitted. The presence bits of the *MC* registers are *never* affected by the network interface.

The delivered message format is shown in Figure 3.1. Note the addition of the *count* and *senderID* fields, as discussed in the previous chapter. These two fields could be packed into the message header and expanded into full words by the network input unit at the destination.

3.2.2 Network Input Interface

The M-Machine has a software-handler based network input interface with a register-mapped network input FIFO.

Upon a message's arrival from the network, it is streamed into a hardware FIFO, called the *message queue*. The message queue is tied to 2 special registers, R_{head} and R_{body}, which appear as two of the general registers of a dedicated message thread slot [2]. When it is read, the R_{head} register returns the `dispatchIP` specified in the *next* unread message in the FIFO. It also performs the side effect of flushing left over words, if any, from the incoming message before this `dispatchIP`, eliminating the security hole due to the user sending a message that is longer than expected by the handler.

Each subsequent read to the R_{body} then *pops* and returns the next word from the message at the front of the FIFO. Any attempt to further read R_{body} when the words from the current message have been exhausted will return an error value.

Note that R_{head} and R_{body} are accessible from a dedicated message thread slot only. Typically, a *dispatcher* thread runs in the message thread slot, branching to the `dispatchIP` software handler when a message arrives. The software handler then extracts the rest of the message via R_{body}, performs the requested operation, and then branches to the next message's `dispatchIP` when it is done [3].

Each of R_{head} and R_{body} has a *presence bit* similar to the general processor registers. Exploiting the stall mechanism mentioned above, the message thread is automatically stalled when it tries to read R_{head} or R_{body} in the absence of a pending incoming message. This allows the message thread to make progress as soon as the first word of next message word arrives, without resorting to expensive mechanisms such as busy-waiting for the next message word or taking an interrupt when the next word arrives.

3.3 Discussion

In this section we will examine how the proposed message architecture compares to the message handling mechanisms in other systems.

[2]The M-Machine supports up to 24 active thread slots simultaneously, each with its own hardware context.

[3]Running in the message thread slot, the message handlers on the M-Machine have the added advantage that they can quickly mutate any user thread's state, including its register contents, via the *configuration space* facility.

We begin with a summary of the strengths of the proposed architecture. At the network output end, the advantages can be described in the following:

- Decoupling of orthogonal components in a message. The target, request and payload can each be specified independently of the others, as *destination virtual address*, *dispatchIP* and *MC registers*, allowing any/all of these components to be reused in a subsequent message.
- Minimization of redundant copying. By arranging for the message to be generated (or loaded from memory) into the appropriate *MC* registers directly, no further copying is needed to prepare the message for launching.
- High computation-communication overlap. By alternately filling and launching from the two banks of *MC* registers, the system can easily achieve very high overlap ratio [4].

At the same time, the network input of the proposed architecture has the features described below:

- Low dispatch overhead. The message handler is invoked quickly after the *dispatchIP* arrives at the destination [5]. There is no need to wait for the rest of the message. Since the *dispatchIP* uniquely identifies the message handler, it involves no overhead to look up the appropriate action.
- No explicit buffering. The message is not buffered explicitly before it is given to the handler. Each word in the message becomes available to the handler immediately upon its arrival, without waiting for the rest of the message.
- Minimal copying overhead. For requests that can be fulfilled immediately, the R_{body} register can be used directly as an operand to regular instructions executing the request. There is no need to explicitly move the word into a register before hand.

[4]The SEND operation can even be executed in the same cycle as two other operations from the same thread on the M-Machine.

[5]By using R_{head} directly as the source to a JMP instruction, the handler thread can be started within one branch delay (3 cycles delay slot on the M-Machine) after the arrival of the dispatchIP.

To further understand the utility of mapping the interfaces to processor registers as described, we include here some architectural comparisons, in terms of a few common primitive message operations, against two contemporary models of message subsystem which also deliver messages from registers. To avoid being distracted by specific features of a particular machine that are not due to its message interface, we consider hypothetical machines *M, J, T*, each with a message interface resembling the M-Machine, J-Machine [9], and *T [6, 23] respectively. Each of these hypothetical machines will adhere to the following restrictions:

- Processor executes one instruction per cycle. Branch operations have a 3-cycle delay slot, with conditional (predicated) execution in the delay slot.
- Multiple hardware contexts with one reserved for message reception.
- Bandwidth between processor and memory is 1 word per cycle, fully pipelined. Startup latency for memory accesses is ignored.
- Bandwidth between processor and network is 1 word per cycle, fully pipelined.

System *M* has a message interface as described in this chapter. The relevant characteristics of the other hypothetical systems, as derived from the original machines, are as the following:

- System *J*: Injects messages by explicit SEND instructions, at up to two words per SEND instruction (1 cycle) from processor registers. Message is completed with a SENDE instruction. Message is dispatched at the destination by hardware (4 cycles) to a software handler in a reserved hardware context.
- System *T*: Composes/receives messages in a set of transmitter/receiver (tx/rx) registers, accessible with dedicated store/load instructions (*sttx/strx* (up to 4 words per cycle), and *ldrx/ldtx* (up to 2 words per cycle)). Messages can be launched from either the tx or the rx registers, and are launched with a dedicated instruction `stty.go` (1 cycle). Check for success is needed after each message launch (2 cycles). Message arrival is detected by polling (2 cycles).

Since the *T does not provide a comparable name resolution mechanism, and the J-Machine does not have a comparable protection scheme, we will assume that the destination specification and remote operation specifier/IP are already available in registers. So that this comparison of the architecture is not influenced by the actual register file size, we also make the number of rx and tx registers in system T equal to $2R$, same as the number of MC registers in system M. Listed below is the number of processor cycles required for some common message functions on each of the above systems:

No.	Remote Operation	# Processor Cycles Needed		
		System M	System J	System T
1	Send N words from processor registers	1	((N/2)+1)	((N/4)+3)
2	Generate & Send N words	(N+1)	(N+(N/2+1))	(N+(N/4+3))
3	Multicast N words from processor registers to D nodes	D	(D×((N/2)+1))	(N/4+(D×3))
4	Block Transfer N words words from memory	(N+N/R)	(N+(N/2)+1)	(N+((R/4)+3)×(N/R))
5	Receive N words to memory	(N+4)	(4+(2×N))	(5+((N/2)-3)+N)
6	Return N words from memory to sender	(6+N+1)	(4+2+((N+N/2)+1)))	(5+(N-3)+N/4+3)
7	Fetch & Add	(4+1+1+1+1+1) = 9	(4+4+1+1+1+1+1) = 13	(2+3+1+1+1+1+3) = 12

In row 1 of the table, we consider sending an N-word message, where the words are already present in user registers. In row 2, the message is first generated or loaded into registers before it is sent. We have assumed that care is taken to generate the message words in the right MC registers directly in system M.

In row 3, a particular message of N-words is delivered to D destinations. The entire message is assumed to be already residing in registers. We note that the message body is preserved on the sending end after launching on systems M and T, which allows it to be reused directly in the next message. (Name resolution, and the generation of the next destination specifier, are not accounted for here.) System J on the other hand needs to repeat the entire send sequence for each of the D messages.

In row 4, we are interested in how fast each system can transfer N words of data. N is assumed to be large, and we have ignored memory startup latency. The block is packetized into (N/R) messages for systems M and T. The two MC register banks

are used alternately for maximum overlap between message injection and memory accesses. In the case of system *J*, for top speed injection, we have assumed that the entire block is transferred as one monolithic message. (The thread must guarantee that no exceptional conditions will be encountered while in the middle of the message in this case. Otherwise, to packetize the block into messages of size R, it would take (N+(N/2)+N/R) processor cycles.)

Rows 5 and 6 show, respectively, the number of processor cycles needed to receive N words to memory, and to return N words from memory in response to the sender's request. The ((N/2)-3) and (N-3) terms for system *T* account for the ability to execute *ldrx* in the branch delay slot, and should be set to 0 if these terms turn out negative. In row 7, we add a value in message to a word in memory, store it back to memory, and return to sender the original word read from memory [6].

We observe that system *M* consistently consumes fewer processor cycles for each of the operations shown. The main contribution to this lowering of overhead is from the direct sharing of registers between the processor and the network ports, such that no extra cycles are spent moving message words between register files. In most cases, the majority of the words within a received message are used only once (*i.e.* immediately stored in memory, or operated upon once), the pop-after-read characteristic of R_{body} in system *M* is therefore a valuable optimization that eliminates the need to explicitly dequeue each word from the network input.

Notice that in one cycle, 4 words can be moved into the *tx* registers on system *T*, while only 2 words can be extracted from the *rx* registers. This results in an imbalance between the maximum message injection and extraction rates. Therefore, the sustainable message rate would be limited by the extraction rate. (If we adhered strictly to the assumptions listed earlier, the message rate would be further limited

[6]The memory latency is ignored. The cycle counts are derived from the following:

M	JMP (delay slot: get address) + load + add + store + get returnAddr + send
J	Dispatch + (get/setup address/value/returnAddr/returnIP from network input) + load + add + store + SEND(destination/IP) + SENDE(data)
T	Polling + (delay slot: get/setup address/value/returnAddr/returnIP from network input) + load + add + store + *strx* + launch

by the network bandwidth of one word per cycle.)

We note that the functions compared above are rather representative of the primitive building blocks for most common message operations (*e.g.* A remote procedure call is made up of operations similar to case 2 and case 6, while cases 1,4 and 5,6 are representative of most remote read/write operations.). It seems therefore that this lower overhead will have an impact on the efficiency on most message operations, and hence is likely to influence the overall performance of applications with non-negligible communication activities. As an attempt to estimate the extent of this influence, we refer next to a J-Machine/CM-5 performance evaluation study by Spertus *et. al.* [24].

3.3.1 Quantitative Analysis

In an evaluation study of mechanisms for fine-grain programming by Spertus *et. al.* [24], the effectiveness of the mechanisms in the J-Machine was compared against those of the CM-5, using the TAM (*Threaded Abstract Machine* [25]) execution model to abstract away from the specific implementation differences between the two machines. The comparisons were based on normalized models of the two actual systems, called *J'-Machine* and *CM-5'*. In this section, we compare the performance of our proposed mechanisms against that of the *J'-Machine* which have restrictions very similar to what we have imposed on the M and J systems described in the previous section.

A suite of six benchmark programs (QS, GAMTEB, PARAFFINS, SIMPLE, SPEECH, MMT) were used as the basis for comparison. Running under the TAM model, the communications component for these programs is predominantly due to *heap-messages* implemented as `ifetch / istore` messages. (A small portion of the communications component also comes from other messages, but we shall ignore them here due to the small percentage ($< 9\%$) of their contribution to the total communication cycles.) The `ifetch` operation reads an element by sending a message to the processor containing the data, which then returns the value to a reply *inlet* [25]. The `istore` operation writes a value to an element, and resumes any deferred readers.

Using the system M model from the previous section, we can implement the `ifetch / istore` operations to examine their costs. For the purpose of comparison, we have as-

sumed that a *cfutures* mechanism similar to that of the J-Machine [9] exists on system *M*. (We have avoided using the *presence bit* mechanism on the M-Machine to implement the futures functionality, in an attempt to isolate this comparison from being influenced by mechansisms that are not part of the message subsystem.) The reader is referred to the M-Machine instruction set manual [26] for the detailed definition of the instruction syntax used in the example code fragments.

The `ifetch` operation can be implemented as follows:

```
/* we use i_count, i_vaddr, i_senderID, i_target etc                */
/* as symbolic names for registers                                  */

_mesg_dispatch:
ialu jmp rhead;              /* Jumps directly to handler.          */
ialu mov rbody, i_count;     /* Use the jump delay slots to         */
ialu mov rbody, i_senderID;  /* read off some parameters            */
ialu mov rbody, i_vaddr;     /* from the message                    */
          .....

_ifetch:

/* message format (input):                                          */
/*              ifetch mesg (dispatchIP)                            */
/*              i-structure                                         */
/*              offset                                              */
/*              return address (implicit MMR & fp)                  */
/*              return inlet (dispatchIP)                           */
/*                                                                  */
/* message format (output):
/*              return msg                                          */
/*              return fp                                           */
/*              value                                               */

/* ''i-structure'' should be in the register i_vaddr by             */
/* the time control flow reaches here.  the next read to rbody      */
/* returns ''offset''                                               */

ialu lea i_vaddr, rbody, i_target;    /* target address             */
memu ld  i_target, #0, i_value        /* read the value             */
```

```
ialu mov rbody, i_reply;
ialu send #bank0, #count1, i_reply, rbody    ; /* send it back       */

_done_istore:
ialu jmp rhead;                          /* wait for next message      */

          .....

_reply:

/* 2nd phase of the ifetch operation which writes the returning      */
/* into the frame                                                    */

/* ''return fp'' should be in the register i_vaddr by                */
/* the time control flow reaches here.  the next read to rbody       */
/* returns ''value''                                                 */

memu st  rbody, #0, i_vaddr;          /* store the value             */

_done_reply:
ialu jmp rhead;                          /* wait for next message      */

          .....
```

The `istore` operation is implemented as the following:

```
/* we use i_count, i_vaddr, i_senderID, i_target etc                 */
/* as symbolic names for registers                                   */

_mesg_dispatch:
ialu jmp rhead;              /* Jumps directly to handler.           */
ialu mov rbody, i_count;     /* Use the jump delay slots to          */
ialu mov rbody, i_senderID;  /* read off some parameters             */
ialu mov rbody, i_vaddr;     /* from the message                     */
          .....

_istore:

/* message format:                                                   */
```

```
/*              i-store mesg (dispatchIP)                              */
/*              i-structure                                            */
/*              offset                                                 */
/*              value                                                  */
/*

/* ''i-structure'' is should be in the register i_vaddr by             */
/* the time control flow reaches here.  the next read to rbody         */
/* returns ''offset''                                                  */

ialu lea i_vaddr, rbody, i_target;    /* target address                */
memu ld  i_target, #0, i_deferred;    /* number of deferred reader     */
ialu lea i_target, #8, i_flag;        /* synchronization flag          */
ialu st  i0, #0, i_flag;              /* clear the flag                */
memu st  rbody, #0, i_target;         /* store the value               */
ialu ieq i_deferred, #0, cc0;         /* done if no deferred readers */
ialu cf cc0 br _istore_service_deferred;

_done_istore:
ialu jmp rhead;                       /* wait for next message         */
          .....

_istore_service_deferred:
ialu ieq i_deferred, #CFUT, cc1;       /* C_future expected            */
ialu cf cc1 br _bad;

ialu lea i_target, i_deferred, i_reply;
ialu mov reply_deferred_read, i_command;   /* assume the dispatchIP  */
                                           /* was saved away in some */
                                           /* register               */
ialu send #bank0, #count1, i_reply, i_command; /* send it back       */

_check_for_more_deferred_reader:
/* check for more deferred readers, and repeat                        */
/* _istore_service_deferred if necessary.                             */
/* (left out for brevity since the check does not rely on messaging   */
/*  mechanisms)                                                       */
```

The above code fragments are based on the `ifetch / istore` implementation for the actual J-Machine, used by Spertus in her study [7]. The cycle counts for the above

[7]The author is grateful to Ellen Spertus for showing her library functions.

TAM operation	cycles needed		
	J'-Machine	System *M*	% improvement
Ifetch message	24	13	45.8 %
Istore message	15	11	26.7 %

Table 3.1: `ifetch / istore` costs

code fragments and the cost for the `ifetch / istore` operations for the J'-Machine as quoted from [24], are shown in Table 3.1. The cycle count for `istore` is for the case where there is no deferred reader.

As can be seen, system *M* demonstrates a 26% $\sim$ 45% improvement over the J'-Machine in the `ifetch / istore` operations. Several factors contribute towards this improvement. We first note that the handler on system *M* is able to use `rbody` directly as an operand to its instructions, instead of having to move the value into a register first. Then we observe that by carefully mapping the reply values such as `i_value`, `i_deferred` *etc.* to the appropriate *MC* registers, the reply message can be delivered without additional copying. The virtual naming scheme also contributes to this improvement by allowing the return node number and frame pointer to be specified using a single address word. Finally, by using the JMP delay slot (`_mesg_dispatch`) to save away the `i-structure` address, the dispatch latency is turned into productive use.

For each of the six benchmarks involved in [24], the dynamic instruction mix statistics for different classes of operations were recorded. The percentage of total execution cycles due to *heap-messages* for each benchmark on the J'-Machine is reproduced in Table 3.2 (CPT = normalized *Cycles per TAM instruction*).

As shown in Table 3.2, more than 20% of the total execution cycles on average are due to heap-message operations. As a rough estimate, the improvement we saw earlier in Table 3.1 therefore translates into a 6.2% (`istore` dominates) to 10.5% (`ifetch` dominates) *overall* performance boost in these benchmark programs. We further suspect that this is an under-estimate, since the comparison above focuses on message handling, and we have yet to consider the utility of the *MC* registers to threads that are *initiating* the `ifetch / istore` messages, nor the savings due to

Benchmark	Fraction execution time due to heap-messages $\left(\frac{\text{CPT due to heap-messages}}{\text{total CPT}}\right)$
QS	$(2.0/12.25) \approx 16.3\%$
GAMTEB	$(1.7/11.5) \approx 14.8\%$
PARAFFINS	$(3.9/14.25) \approx 27.4\%$
SIMPLE	$(3.4/12.5) \approx 27.2\%$
SPEECH	$(4.1/12.25) \approx 32.8\%$
MMT	$(2.6/13.0) \approx 20.0\%$
AVERAGE	$\approx 23.1\%$

Table 3.2: Fractional Execution Time due to Head-Messages

automatic name translation, nor the savings due to the integrated protection scheme. When these other factors are taken into account, we therefore believe the proposed message interface will contribute an even higher performance gain.

Chapter 4

Message Throttling

Given that a message's functions are basically transferring data and causing further processing activity, we understand that the operation of the message subsystem relies on the availability of the following system resources:

- *Buffering space* for temporary storage of message words during composition and reception, and for buffering a message until it is processed.
- *Network bandwidth* for the actual transport of the message.
- *Thread slots* [1] for performing the additional processing caused by the message.

As all three are shared system resources, depletion of any of the above is likely to cause severe system-wide performance degradation, sometimes with hysteresis and domino behavior. The message subsystem must therefore be cautious not to over-commit the resources concerned. To maintain a sustainable consumption rate of the limited resources, the message subsystem must then ensure balanced rates of message creation/completion. In the next section, we will consider the need for low-level architecture primitives to efficiently support this message throttling task. We then study a *Return To Sender* throttling method, which keys message back-pressure to the exhaustion of local buffering space. Finally we present a primitive mechanism on the M-Machine that supports low-overhead and user-transparent message throttling.

[1] Here we refer to the hardware resources needed for running a thread.

4.1 Architecture Support

Message throttling implementations generally fall under two broad categories of approach:

- System-imposed throttling. In this approach, each sender must first obtain some *permit* [2] from the system before it is allowed to generate/inject new messages. By monitoring resource availability, the system is able to ration the permits and control the resource consumption rate accordingly.
- User self-imposed throttling. Here, each sender abides by some programming convention to cooperatively avoid excessive resource consumption. Usually a sender/receiver pair would agree on some handshake protocol (*e.g.* various message *window* protocols) to ensure that their outstanding messages take up no more than a certain amount of system resources at any time.

Notice that in either approach, the overhead can be substantial if throttling were implemented as an add-on to a conventional message subsystem. To enforce a permit policy without the benefit of low-level architecture support, the system software must intervene to validate every message injection, effectively turning each message SEND into a costly system call. And to adopt the latter approach, we must assume a trusted programming environment, which is not always possible in a multi-user environment. Overhead is also incurred as each user explicitly manages its share of resources and handshake protocols.

Considering we are interested in fine grain messages that are only a few (~ 10) words long, any protocol that adds more than a few cycles of overhead to the message delivery time is not reasonable. We are therefore motivated to consider including primitive mechanisms to support low overhead message throttling.

[2] By "permit" we merely mean any form of privilege without which a message injection would be blocked. The permit may or may not expire after some number of messages or some duration of time.

4.2 "Return To Sender" Throttling

Having identified earlier the three resources critical to the message subsystem, we note that they could be made to "spill over" into one another, such that the system could be kept stable as long as we can properly manage one of the three elements. One *example* of such spilling is shown in Figure 4.1. While the figure seems to suggest a vicious cycle when overhead is added along the way as work is spilled from one stage to the next, the chain can be broken by conditioning one of the transitions shown upon the availability of the other two resources from which the transition does not emanate. If each of the states could then continue to perform its function, the accumulated work in each state would eventually be consumed, and the corresponding resource would be freed up.

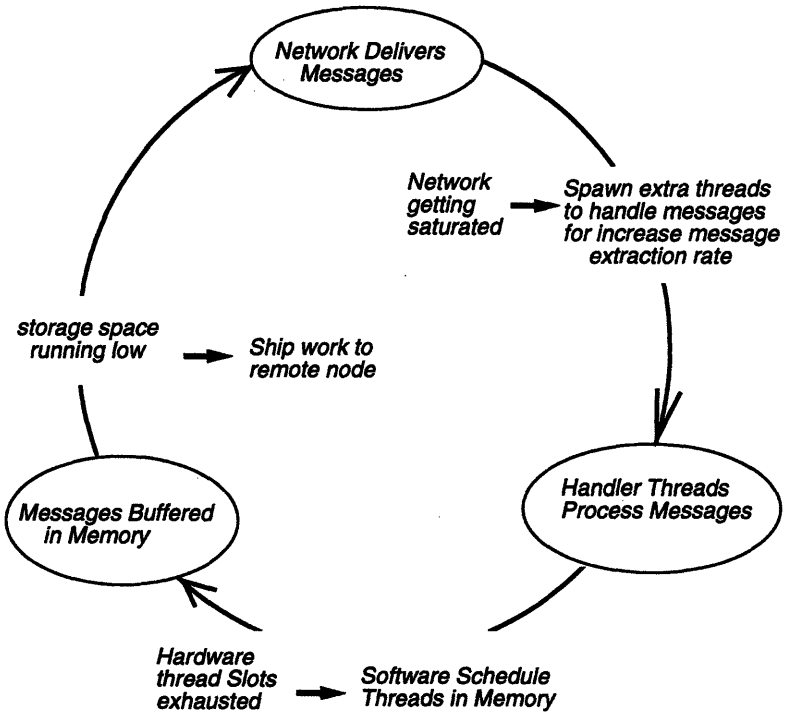

Figure 4.1: Message resources spilling

In particular, by inhibiting injection of new messages when available buffering space runs low, we can ensure that the network stands a chance to drain. The buffered messages are meanwhile being processed, and the buffering space eventually returned for reuse. The messaging capability can then be restored when sufficient buffering space is again available. A stable load could then be sustained in the cycle shown.

Ideally, injection of a message should be inhibited when its destination node is short of buffering storage, so that an injected message could always be accepted by its receiver. No network bandwidth is ever wasted that way. It however requires each and every potential sender to be constantly updated with the availability of buffers in all its potential receivers. Obviously this is not feasible without incurring large amounts of extra network traffic. Further more, as each sender must record the availability of buffers in all potential receiver nodes, the local memory for that purpose must scale linearly as we grow the machine size. The system then cannot be scaled by simple replication of nodes.

A compromise where only local information is tracked, is found in the Cray T3D [12]. In this approach, each sender keeps track of its local free buffers, instead of its receivers' available space. A buffer is reserved at the sending node before each message is injected, and the message is bounced back to its sender for buffering if the receiver is unable to handle the message quickly nor to buffer it locally. If the message is successfully received, an acknowledgment from the receiver causes the reserved buffer to be released.

The T3D approach effectively disallows any sender from flooding the system with excessively many messages, thereby providing the back-pressure desired for sustaining the cycle in Figure 4.1. In the following section, we describe how the we build upon the T3D approach to provide throttling on the M-Machine message subsystem.

4.3 Message Throttling Mechanism

The key features of the message throttling mechanism we propose are its transparency to user-level programs and its low overhead.

The system uses a hardware counter, called the Outstanding Message Buffer Counter

(OMBC), to keep track of the number of bounced messages that can be buffered in the available local space. At initialization, the system software reserves a local memory segment for buffering bounced-messages, and sets the OMBC value accordingly [3]. As each message is injected, the counter is automatically decremented by one. Hardware support is also provided for the OMBC to be incremented/decremented atomically by system software. When the counter reaches zero, SEND operations are inhibited from issuing.

The message handlers are written to respond to each user level message with an *Acknowledge* message. As each *Acknowledge* message is received back at the originating node, it invokes a message handler to increment the OMBC as necessary. If a message is bounced, the message handler at the originating node places it in the reserved space. Fig 4.2 shows a successful transaction example, where the acknowledgment is piggy-backed on the reply message.

Note that there is no need to statically associate each message with a particular buffer at its injection. When a message is indeed bounced, it can be placed in any unoccupied but reserved buffers. As long as every injected message is accounted for, each bounced message is guaranteed adequate buffering space.

There are two caveats to this scheme. Firstly, to guarantee freedom from deadlock, critical messages must be able to bypass the restriction imposed by the throttling mechanism. An *Acknowledge* message, for example, must always be allowed into the network in all cases. A reply generated by the message handler, such as that shown above in Figure 4.2, may also safely bypass the decrementing of its OMBC, since it is guaranteed to be accepted by the original sender who had reserved a buffer.

The M-Machine thus provides a subset of its SEND family of operations that permit *system level* messages to be impervious to the throttling mechanism. User messages are not allowed to bypass the throttling system. Unlike replies generated by message handlers, each user-level "reply" message must count as a new transaction that warrants the reservation of an additional buffer. This is because user-level threads are not guaranteed to complete quickly like message handlers, therefore treating user-

[3]The OMBC is set to the number of maximum-size messages that can be accomodated in the reserved segment. The actual management of the buffers is decided by the system software programmer, though a link list of buffers might make it easier to recycle them.

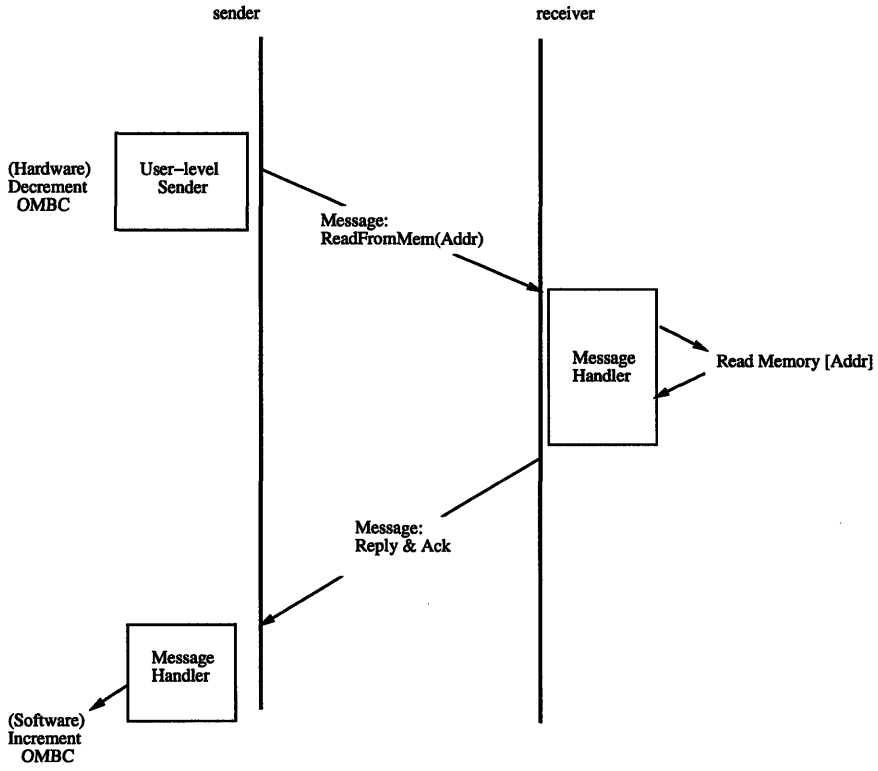

Figure 4.2: M-Machine Message Throttling

generated replies in the same fashion as handler-generated replies would unnecessarily tie up for a long time the buffer reserved at the originating node. An example is shown in Figure 4.3.

Secondly, the system must be able to swap out threads that are stalled by the throttling mechanism, to vacate their thread slots for use by message handlers that will process any buffered messages. This is needed in order to be consistent with the earlier assumption that each of the states in Figure 4.1 could continue to consume accumulated work in all instances. This guarantee is already provided on the M-Machine in the form of a timeout watchdog counter that causes a trap into system routines when a user thread has failed to make progress for a predetermined period [13]. The trap handler could then swap out the stalled thread.

In addition to that, a system event is raised whenever OMBC reaches zero. This

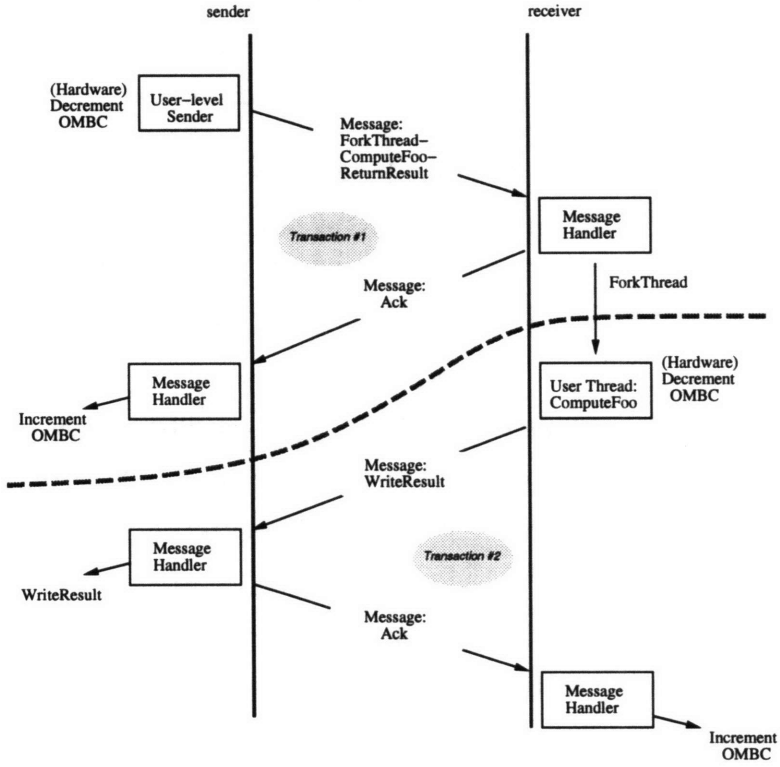

Figure 4.3: User-level Reply Message

notifies the system about the exhaustion of the allocated buffering space, so that it could:

- swap out the stalled thread quickly instead of idling until a timeout event.
- rearrange thread priorities such that any buffered bounced messages are handled first.
- dynamically reallocate more memory for buffering if necessary.

4.4 Advantages

The message throttling mechanism described above has several advantages:

- Transparency to user programs: The OMBC counter, stalling mechanism, message acknowledging/bouncing and buffering are all completely transparent to user threads, freeing the programmer to concentrate on the task at hand.

- Low overhead: Since the message is *not* copied into a buffer before injection, no latency at all is added to the message injection/delivery time. In the normal case where a message is accepted by the receiver, the total cost added by the throttling system is only 5 cycles [4] of processor time *after* the message is received.

- Dynamic control over network load: By dynamically resetting the value of the OMBC counter, and reallocating bounced-message buffer space if needed, the system on each node may independently adjust the load it imposes on the network.

- Scalability: Depending on the topology, the network bisection bandwidth, and therefore the saturation message rate, may grow either linearly or sub-linearly as the machine scales up in size. As a result, the average sustainable message rate for each node either remains the same or decreases. In either case, the bounced-message buffering requirement on each node does not increase. Therefore the throttling mechanism itself places no restriction on machine scalability.

Note that this mechanism does not restrict the system to one form of throttling technique. The system could well take advantage of the ability to dynamically disable user messages [5] in implementing various higher order protocols. If desired, the primitive throttling mechanism could even be effectively disabled by setting the OMBC to a large value and simply resetting it when it reaches zero.

[4] In the best case where the message handler slot keeps frequently-used values in live registers, it takes one cycle to send an Acknowledge message, and 4 cycles to invoke the message handler back at the originating node and increment OMBC.

[5] By setting OMBC to zero.

Chapter 5

Evaluation

In this chapter, we shall examine the result of having the mechanisms we have discussed thus far working together in one system. The reader is referred to [13] for a full implementation of this message subsystem. Here we shall keep our focus on the relative merits of the architecture, as compared to other similar architectures. A brief discussion of implementation issues is however included in this chapter.

5.1 Performance

It is difficult to quantify performance without a point of reference. To evaluate how our proposed architecture performs, while isolated from other unrelated features of specific machines, we therefore refer again to the hypothetical systems M, J and T developed in chapter 3 for our discussion in the following sections. As a reminder, system M represents our proposed message architecture, system J is similar to the J-Machine [9] and system T resembles the *T [6]. These machines provide fine grain message facilities closely resembling what we have discussed, delivering message words from registers. The example code fragments in this section are based on the M-Machine instruction set and its implementation of the message mechanisms [26].

5.1.1 Latency

The primary strategy for reducing latency is the elimination of redundant buffering and software overhead. We shall examine this characteristic through the following remote-write example [1].

```
/* at the sender */
_sendmesg:
    /* i_addr contains remote write address,                          */
    /* i_disp contains the dispatch IP for the remote write handler   */
    ialu add i10, i11, i4;              /* compute the word to be written */
    ialu send #bank0, #count1, i_addr, i_disp, cc0;
    ialu       ....                     /* continue with computation     */
               ....

/* at the receiver */
_dispatch:
    ialu jmp rhead;                     /* Jumps directly to handler.    */
    ialu mov rbody, i_count;            /* Use the jump delay slots to   */
    ialu mov rbody, i_senderID;         /* read off some parameters      */
    ialu mov rbody, i_vaddr;            /* from the message              */

               ....

/* message handler */
writewordhandler:
    memu st rbody, #0, i_vaddr;          /* store the word               */
    ialu send1p #bank0, #count0, i_senderID, i_ACK, cc0;
                                        /* send an acknowledge           */
    ialu br _dispatch;                  /* done, wait for next mesg      */
    ialu nop;
    ialu nop;
    ialu nop;
```

[1] A Remote Procedure Call is performed in very much the same manner, except the data would contain the procedure entry point and arguments, while a message handler to fork the requested thread would be specified in the SEND operation instead.

Notice that the word to be written (the result of `i10+i11`) is computed directly into a message composition register `i4`, with no intermediate copying before the message is launched. Similarly, at the receiver node, the data word is stored from `rbody` directly into its *final* destination by the handler (`i_vaddr` contains the same value as the sender's `i_addr`, passed in the message.), giving a total end-to-end latency of $((N+1)+$ Network Latency $+4+(N+$ Memory Latency $))$ cycles, where N is the number of words transferred. The significance of this is perhaps best illustrated in a comparative time line diagram, as shown in Figure 5.1.

Observe in the first panel that there is no need to buffer the message before it is injected, since the entire message is guaranteed to be present in the *MC* registers, and can be streamed out without interruption. The network is thus safe from undesired bubbles. Also note how the message can be dispatched as soon as its first word arrives at the destination. These account for the shorter latency compared to the other panels.

5.1.2 Protection

We must realize that the time line comparison in Figure 5.1 does *not* explicitly take into account any protection operations. The panel shown for the *M* message architecture however, already includes its primitive protection mechanisms. To support full control over the possible combinations of (sender×recipient ×request) for *each* message, the time line for our design need only be extended by a *one-time* system-call or memory-load to obtain the appropriate protected dispatch pointer. (*E.g.*, the system could statically allocate dispatch pointers to be placed in each thread's data segment/stack at its initialization, and then dynamically change the binding of the pointers to appropriate message handlers as necessary.) This privilege can further be dynamically, yet transparently to the user, modified by the system software via remapping of the virtual dispatch pointer to a different handler.

Much more however is required for the alternative architectures to be able to provide the same granularity of protection. For the J-Machine message architecture, there is

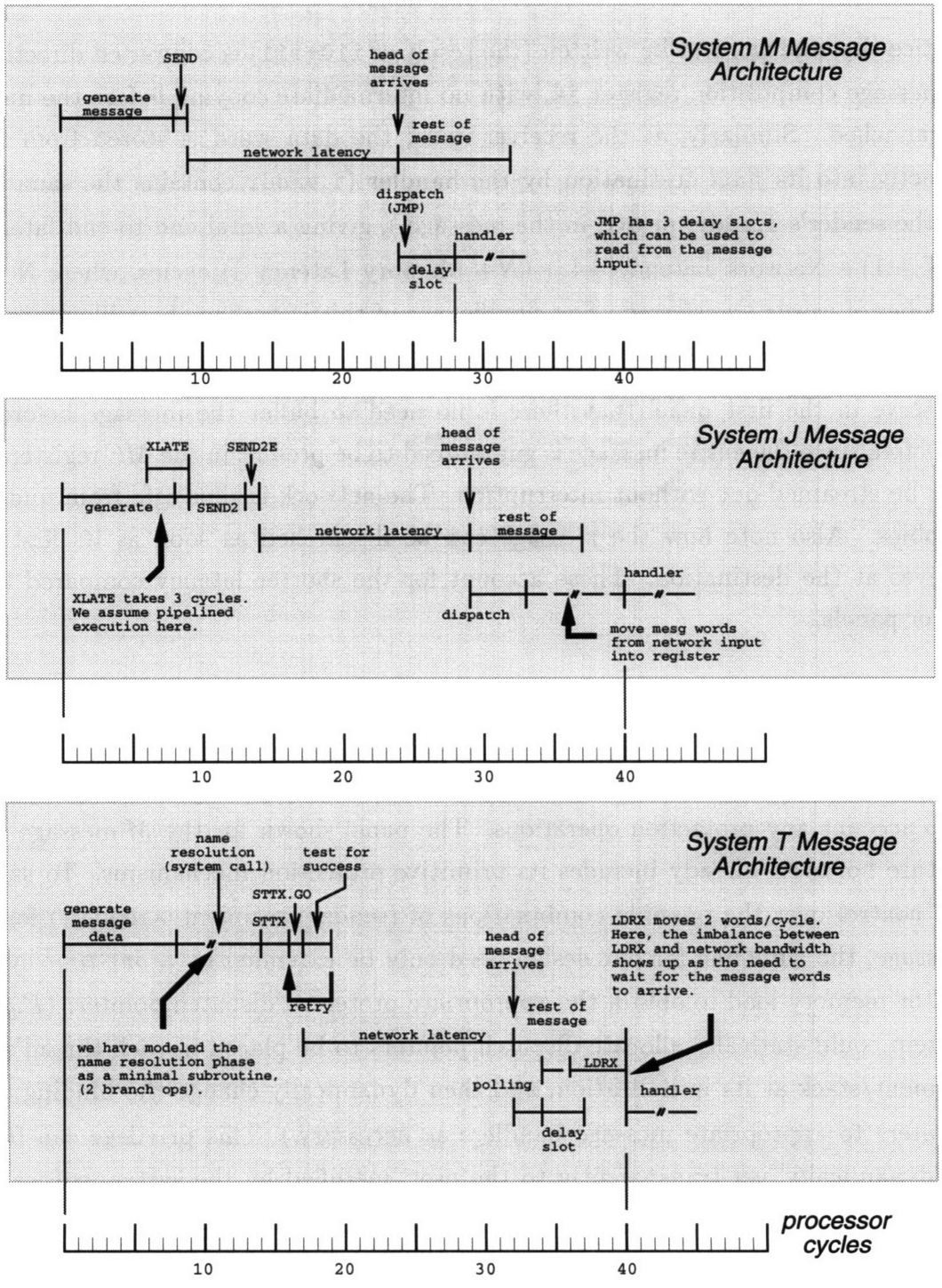

Figure 5.1: Latency

really no effective way to prevent a sender from making inappropriate remote requests, given that that sender may use any integer number as the destination specifier, and any physical address as the handler specifier. Therefore, to have any protection at all, its handlers would need to to explicitly check into *every* message to filter out unauthorized accesses. Meanwhile, the *T does have some hardware mechanisms to trap messages that are not meant for the currently active *job* on the receiver. Nonetheless it is unable to provide the flexible control over (sender× recipient×request) above without dropping into a system routine or enforcing gang scheduling. The proposed architecture therefore stands at a further advantage when protection enforcement is taken into account.

5.1.3 Communication/Computation Overlap

As the processor is not blocked after the SEND operation is executed, the proposed architecture also allows substantial overlap between the message injection time and computation/memory operations. The following large-block transfer example demonstrate this feature:

```
/* i_dest contains the destination , i_disp contains the dispatch IP,  */
/* i_addr contains the source address,                                  */
/* i_iter contains the number of iterations to loop                     */

    ialu mov #num_iter, i_iter;       /* number of iterations needed   */
                                      /* to complete the transfer      */
    ialu ieq i0, i0, cc2;             /* make cc2=T for 1st iteration  */

_loadbank1:
    memu ld i_addr 8, i4;             /* load up MC bank 0 (i4..i12)   */
    memu ld i_addr 8, i5;
         ....
    memu ld i_addr 8, i12;
    ialu ct cc2 send #bank0, #count10, i_dest, i_disp, cc1;
                                      /* launch it                     */
 _loadbank2:
```

```
    memu ld i_addr 8, f4;              /* load up MC bank 1 (f4..f12)   */
    memu ld i_addr 8, f5;
         ....
    memu ld i_addr 8, f12;
    ialu lea i_dest, #64, i_dest;      /* destination of the next block */
    ialu ct cc1 send #bank1, #count10, i_dest, i_disp, cc2;
    ialu igt i_iter, i0, cc0;
    ialu ct cc0 br _loadbank1;
    ialu sub i_iter, #1, i_iter;

_done:
    ialu cf cc0 exit;
    ialu nop
```

Notice in the given example how the two banks of *MC* register are loaded up alternately so that the next message is composed in parallel with the injection of the current message. Since the injection process is performed by the network output controller, independent of the processor, the overlap time is available for normal computation, so long as the critical *MC* registers are not overwritten until the injection completes. By comparison, the alternative architectures have a period of network port "deadtime" between subsequent messages, needed for moving the message words from/to the registers. Again, this is best illustrated with a time line diagram, Figure 5.2. Note that the loop operations have been left out from the diagram for clarity, since they do not contribute to differences between the normalized *M*, *J* and *T* systems.

Note that object relocation has not been taken into account in the given time lines. Therefore the name-resolution step is shown once in every panel. If object relocation were to be considered, then the name resolution should be performed for each individual message so that the most updated information is obtained. This addition has no effect on the time lines in the top panel since the translation is built into the subsystem. It will, however, extend the other two panels, which must explicitly translate each destination address.

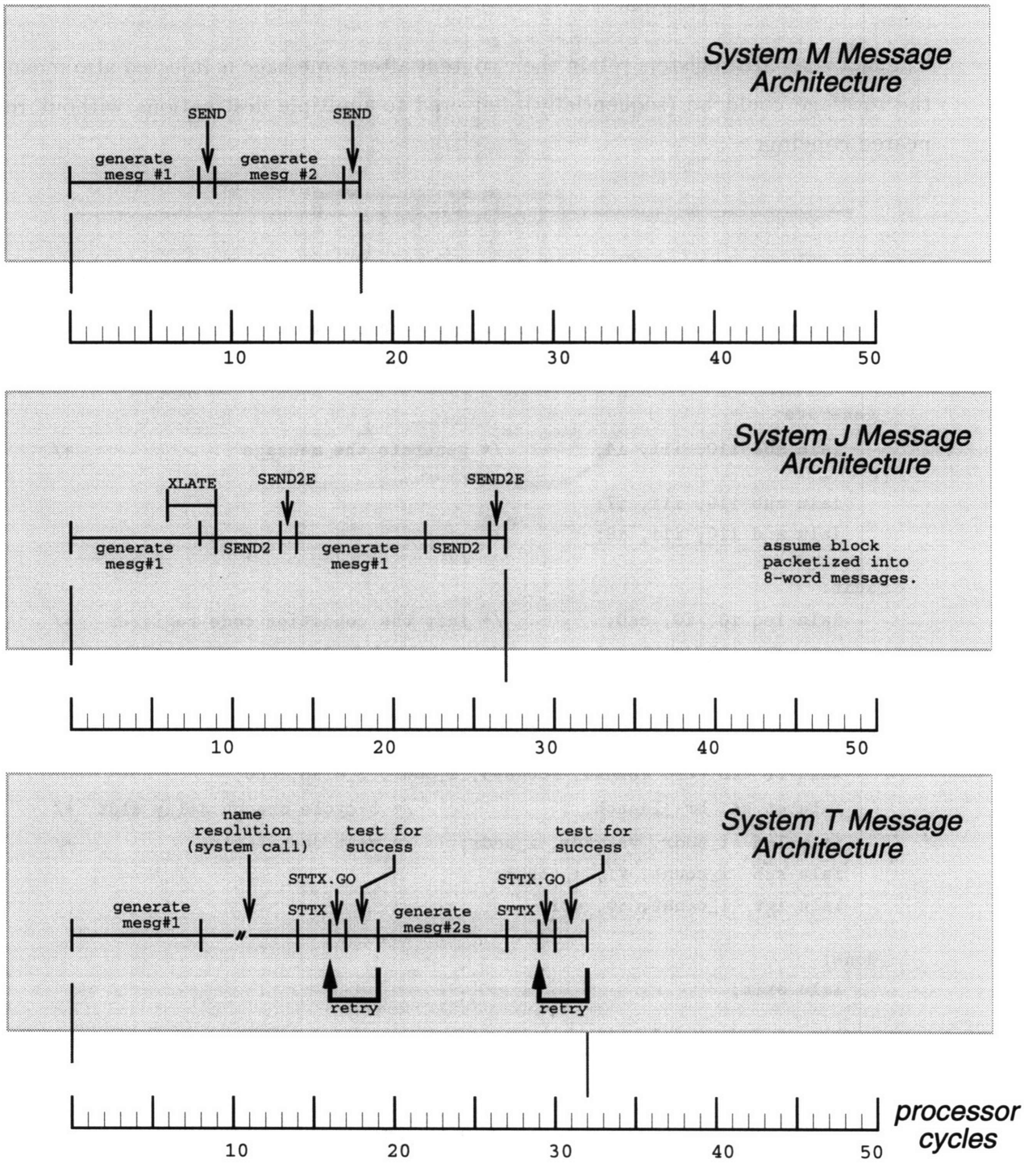

Figure 5.2: Communication/Computation Overlap

5.1.4 Multicasting

The fact that *MC* registers retain their content after a message is injected also means the message could be (sequentially) delivered to multiple destinations without repeated copying:

```
/* i_addr contains remote write address,                                  */
/* i_disp contains the dispatch IP for the remote write handler           */
/* i_count contains the number of messages to send                        */

_generate:
    ialu add i10, i11, i4;        /* generate the message                 */
             ....
    ialu sub i10, i11, i7;
    ialu and i10, i11, i8;

_init:
    ialu ieq i0, i0, cc0;         /* init the condition code register     */
    ialu ieq i0, i0, cc1;         /* init the condition code register     */
    ialu mov #numMsg i_count;     /* init number of messages to send      */

_launch:
    ialu ct cc0 send #bank0, #count5, i_addr, i_disp, cc0;
    ialu ct cc1 br _launch;                /* 3-cycle branch delay slot   */
    ialu lea  i_addr, #stride, i_addr;     /* next destination            */
    ialu sub  i_count, #1, i_count;
    ialu igt  i_count, i0, cc1;

_done:
    ialu exit;
```

No explicit synchronization/test needs to be performed to ensure the previous message has completed. Condition register `cc0` simply does not become *present* until the injection is complete, therefore the next message cannot be initiated due to the

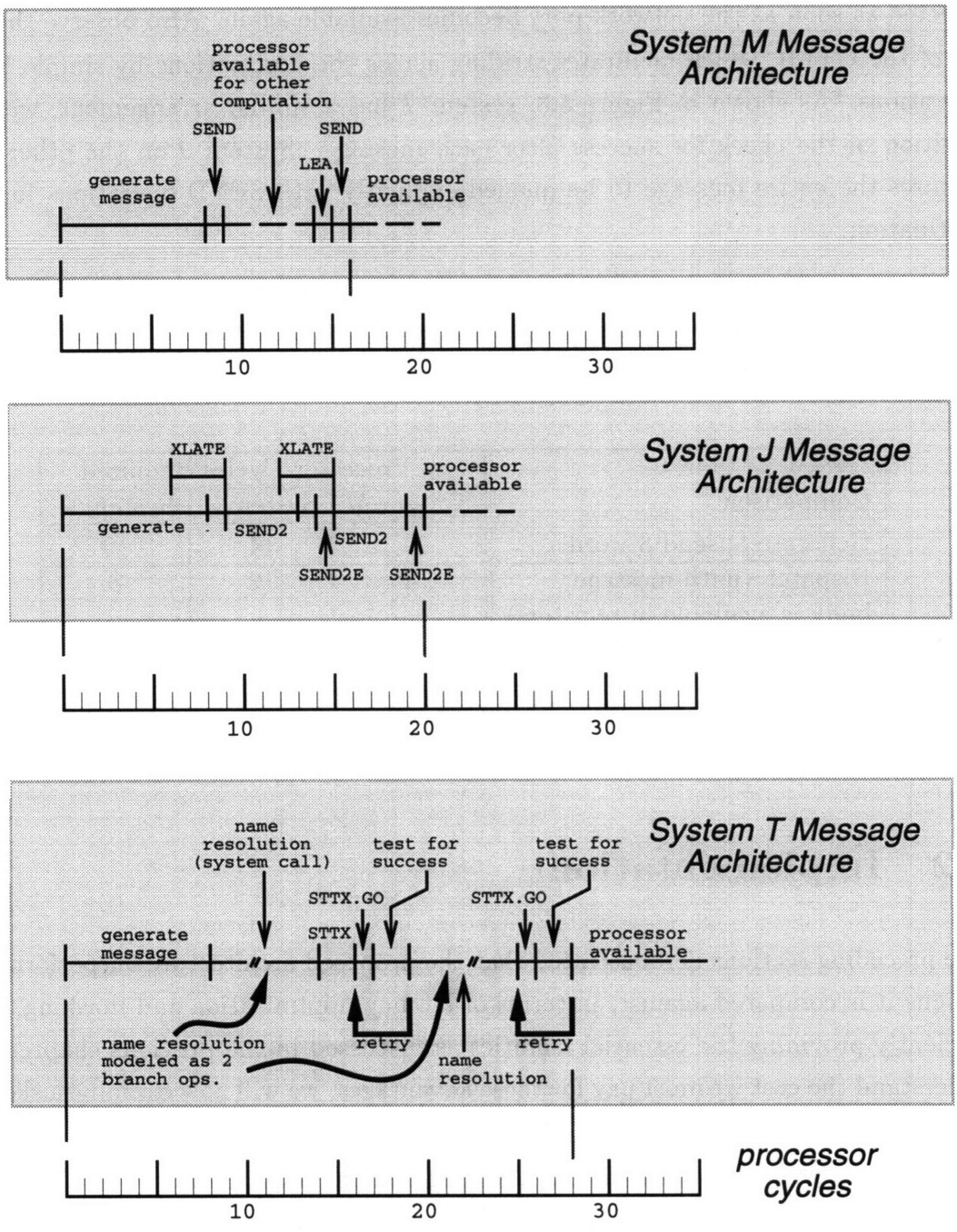

Figure 5.3: Multicast

implicit *stall* mechanism. If it does stall on the `cc0`, the message will nonetheless be injected as soon as the network port becomes available again. Also observe the utility of the GTLB, which facilitates striding across the destinations by simple LEA [2] operations. As shown in Figure 5.3, system T has a similar arrangement, with the addition of the check for success after each message. System J on the other hand requires the entire message to be queued explicitly with SEND operations for each destination.

The comparisons in Figures 5.1, 5.2 and 5.3 are summarized in the table below. The first 2 rows refer to Figure 5.1 (network latency ignored), while rows 3 and 4 refer to Figure 5.2 and Figure 5.3, respectively.

Comparison	# Processor Cycles Required		
	System M	System J	System T
Generate & send 8 words	9	14	19
Dispatch (until message body is available in registers)	4	12	8
Send 2 consecutive 8-word messages	18	27	32
Send one message to 2 destinations	16	20	28

5.2 Implementation

The preceding sections demonstrated that the proposed architecture outperforms the designs it is compared against, in terms of latency minimization and masking, while efficiently providing the essential facilities as discussed in the previous chapters. To understand the cost we must pay for these advantages, we will now examine a skeleton implementation of the architecture [3] based on the M-Machine system. We divide our discussion into two parts, the network output and the input units.

[2] Load Effective Address. Refer to section 5.2 for more descriptions on the GTLB.

[3] For this evaluation, we consider only a single-cluster implementation. The reader is referred to [27] for a four-cluster implementation of the architecture, where hardware SEND arbitration is discussed.

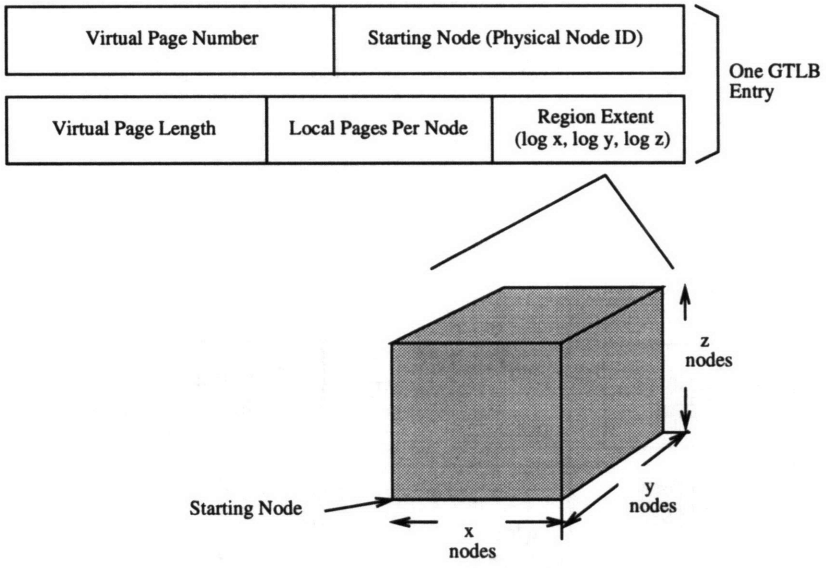

Figure 5.4: GTLB

5.2.1 Network Output Unit

The responsibilities of the network output unit include name resolution, pointer-type validation, and message injection. Given that type-checking is a mechanism common to systems that provide protected data types, we will not concern ourselves with its implementation here.

Name resolution depends on the GTLB/LTLB pair. Physically, the LTLB is no different from a regular TLB in a paging memory system. The difference is that the LTLB miss handler is enhanced to also look into the GTLB for the virtual address referenced.

A very flexible implementation of the GTLB [4] is described in [13]. In essence, it features an entry format that permits regions of the virtual address space to be distributed across prisms of nodes using one single GTLB entry, as shown in Figure 5.4. As a result, for regular address mapping patterns (*E.g.* dividing the machine into sub-regions for running different processes, each with a separate region of the address

[4]This design is attributed to Nicholas P. Carter

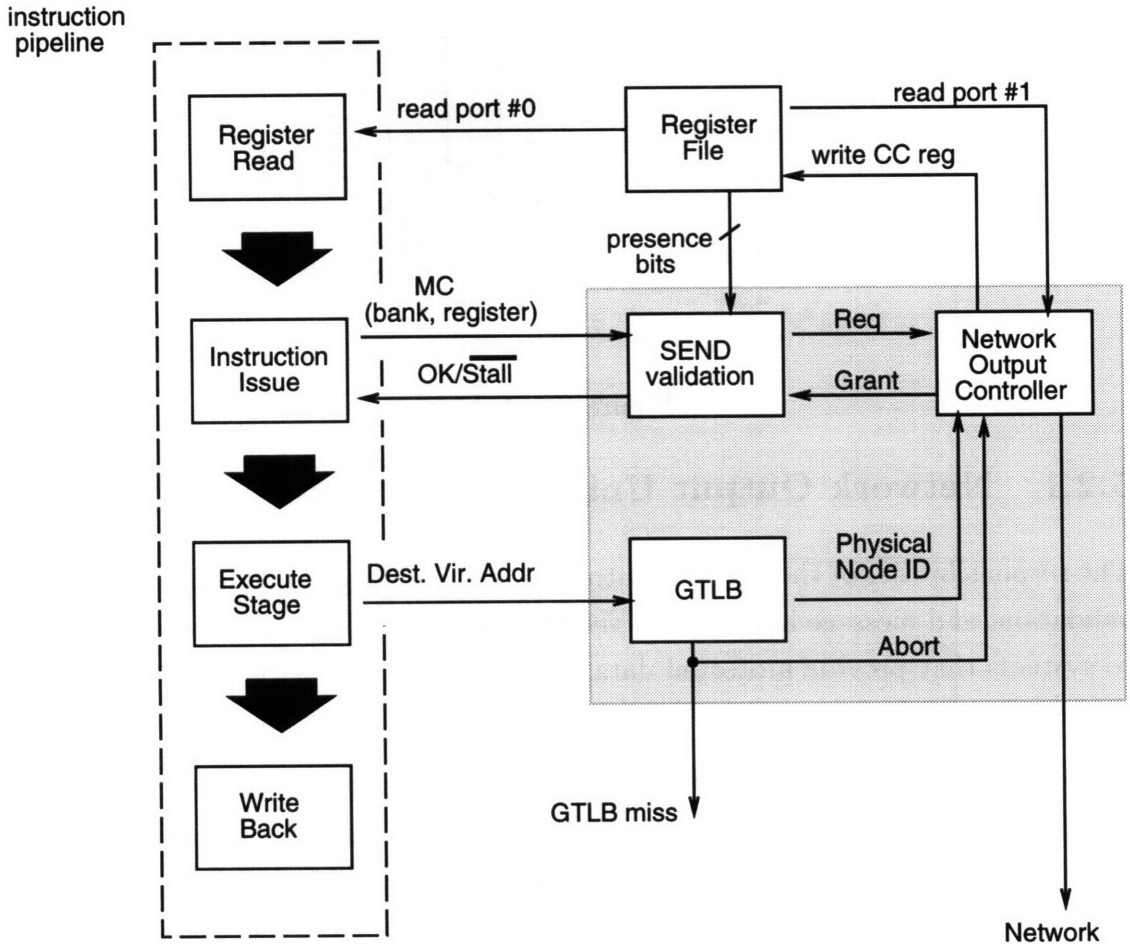

Figure 5.5: Network Output Interface

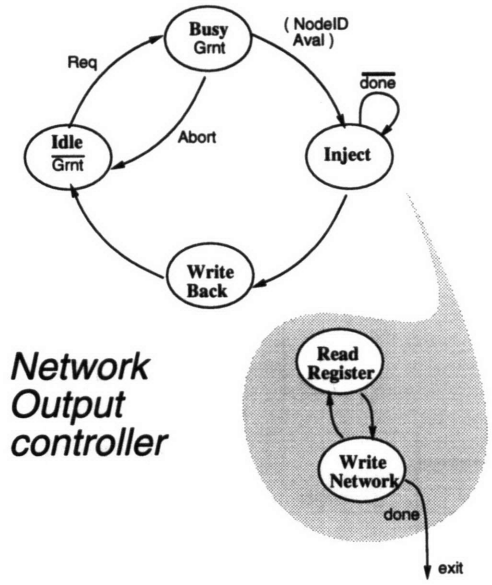

Figure 5.6: Network Output Control

space, or physically distributing pieces of a object across a sub-region of nodes.), GTLB misses can be made extremely rare, even with a very small GTLB cache. In the event of a GTLB miss, a handler simply performs a global page table lookup, in much the same way as the operation of the LTLB miss handler. In terms of the hardware, both the GTLB/LTLB can be implemented using regular cache/TLB technology.

The architecture prohibits a SEND operation from being issued unless all *MC* registers involved in the message are *present.* This restriction can be enforced by a simple SEND validation module as shown in Figure 5.5. The function of this module is to assert the *stall* signal that prevents the SEND operation from issuing, when the *MC* registers concerned are not present or the network resource is busy. This implementation relies on a *thread timeout* watchdog timer in the processor to avoid deadlock, in the case where an *MC* register is never validated due to program error. The network output controller can be implemented as a simple FSM, shown in Figure 5.6.

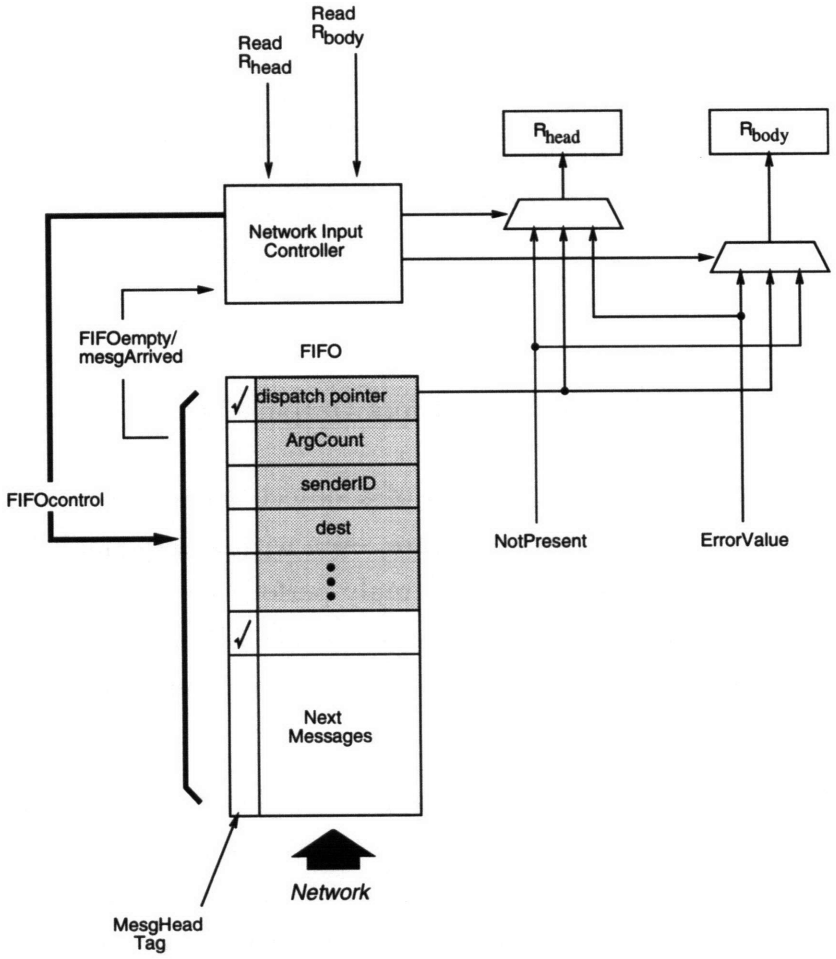

Figure 5.7: Network Input Interface

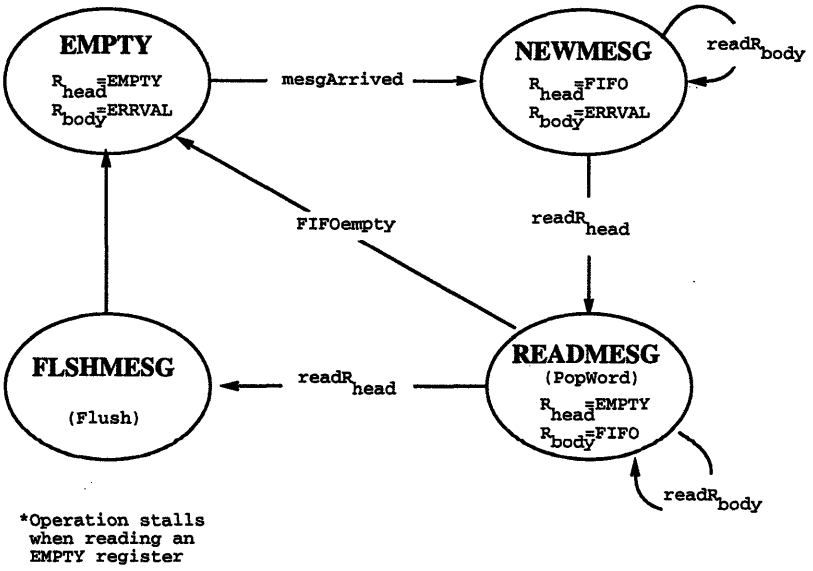

Figure 5.8: Network Input Control

5.2.2 Network Input

The network input unit is responsible for providing the R_{head}/R_{body} interface. This implementation assumes that a hardware context is reserved for a *message dispatcher* thread in a multiple context processor [13]. The dispatcher thread is responsible for executing a JMP operation to R_{head} (which causes the thread to jump to the specified message handler) when it is ready to accept the next message As shown in Figure 5.7, the network input unit manages the presence bits of R_{head}/R_{body} to stall the dispatcher/handler when no message word is available, or otherwise returns an error value when appropriate. Also note that an incoming FIFO is deployed so that messages can be extracted from the network quickly to avoid congestion.

The operation of the network input controller is shown in Figure 5.8. As can be seen, this function can be easily implemented as a simple FSM.

Chapter 6

Conclusion

We have presented in this thesis a message subsystem architecture featuring integrated primitive mechanisms that efficiently support the basic message-related operations.

The system provides a globally named, position-independent naming environment, with a completely user-transparent name resolution facility. Dynamic and transparent object relocation is supported via a two-level hardware-cached translation mechanism. To minimize message send/receive overhead, all redundant buffering is eliminated by mapping the network ports directly to the processor register space. A message-handler based network input interface is chosen for its flexibility. Also considered to be of critical importance is system-wide protection, where fine grain access control is accomplished via a combination of protected pointers and message handlers. Finally, as a low overhead measure to prevent network congestion, the system incorporates a primitive user-transparent message throttling mechanism.

Based on comparisons with existing architectures and a brief study of the implementation issues, we have also shown that this proposed architecture delivers improved performance and user transparency at only a cost comparable to conventional FSM and cache designs.

In conclusion, we demonstrated that balanced and efficient support for every phase of the *end-to-end* message path is essential for optimal overall performance of a message subsystem. To minimize overhead in a fine grain programming environment, this support should be integrated tightly into the processor, provided by efficient primitive

mechanisms, which were further shown can often be implemented by exploiting the mechanisms already existing in other domains of the system.

6.1 Future Studies

While we have shown that the architecture proposed here meets the needs of message-passing operations at a comparative advantage to existing models, there are at least two unresolved issues arising from this study.

Firstly, we note that the *MC* registers cannot be used as targets for instructions when the message is being injected into the network. Depending on the number of other registers available to the programmer during this period, this constraint may discount the usefulness of the ability to overlap computation with the injection time. One plausible solution to this problem is to provide *shadowed MC* registers, where two physical registers, *primary* and *secondary*, are mapped to each *MC* register address. The *primary* register is used for regular read/write operations to the associated register address. During a write to the *primary* register however, the instruction may *optionally* also deposit a copy of the datum into the *secondary* register. Messages are extracted from the *secondary* register set only. With this arrangement, the system will have the same no-copy launch feature we have described, yet the user is not required to explicitly avoid the *MC* registers during message injection. The reader is referred to [17] for an idea similar to this.

On the other hand, since the *secondary* register set must take up silicon area, there might be some flexibility gained by using this area to simply expand the regular register set, and amending the SEND operation to accept *MC* register specification in the form of $(begin, end)$, instead of $(bank, count)$ as we have described. The user/compiler would then have more freedom in optimizing the register allocation policy. The penalty however would be the need for a larger register address space, and a more complex validation logic for the *MC* register presence bits. The tradeoff between the two alternatives is as yet not clear.

Secondly, cache effects at the destination have not been considered in this study. Given that message handlers are likely to have a lower frequency of invocation compared to the other pieces of code running on a node, expectedly a cache miss is also

likely to accompany a message arrival. The effective dispatch time that we have assumed is therefore perhaps optimistic. Conceivably, this latency can be minimized by caching / hard-coding the handlers in specialized / reserved fast storage. It is nonetheless not clear how the utility of that approach compares to the flexibility of the arrangement described in this study, given that the former is likely to be restricted to a small number of handlers due to area considerations, while the latter provides arbitrary access to the virtual memory space. These alternatives, and perhaps others, should be explored to find a reasonable compromise between flexibility and efficiency.

Bibliography

[1] William J. Dally. A universal parallel computer architecture. In *FGCS'92*, pages 746–758. ICOT, June 1992. invited paper; Also appears in *New Generation Computing FGCS'92 Special Issue*, June 1993.

[2] Gary A. Schmidt. The Butterfly parallel processor. In *Proceedings of the Second International Conference on Supercomputing*, pages 362–364. International Supercomputing Institute, Inc., 1987.

[3] Jiun-Ming Hsu. A communication architecture for multicomputers. Techreport IBM RC 18178, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598., 1992.

[4] Anant Agarwal et al. The MIT Alewife machine: A large scale distributed-memory multiprocessor. Techreport MIT/LCS/TM-454, MIT Laboratory for Computer Science, Cambridge, MA., 1991.

[5] Charles E. Leiserson et al. The network architecture of the connection machine CM-5. In *Symposium on Parallel Architectures and Algorithms*, pages 272–285, San Diego, California, June 1992. ACM.

[6] Michael J. Beckerle. An overview of the START (*T) computer system. Techreport, Motorola Inc. Cambridge Research Center, One Kendall Square, Building 200, Cambridge, MA 02139, July 1992.

[7] William J. Dally, J.A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The Message-

Driven Processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[8] Toshiyuki Shumizu, Takeshi Horie, and Ishihata Hiroaki. Low-latency message communication support for the AP1000. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 289–297. ACM, May 1992.

[9] William J. Dally et al. The J-Machine: A fine-grain concurrent computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[10] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pages 256–266. IEEE, 1992.

[11] Jeffrey Kuskin et al. The stanford FLASH multicomputer. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 302–313. ACM, May 1994.

[12] Steve Nelson. Cray T3D massively parallel processing system. Distinguished Lecture Series, 6, University Video Communications, Stanford, CA., 1993.

[13] William J. Dally et al. M-Machine architecture v1.0. Techreport MIT Concurrent VLSI Architecture Memo 58, MIT Artificial Intelligence Laboratory, Cambridge, MA., 1994.

[14] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Protected pointers: Segmentation without descriptors. unpublished, 1993.

[15] P. Agrawal, W.J. Dally, A.K. Ezzat, W.C. Fischer, H.V. Jagadish, A.S. Krishnakumar, and A.E. Dunlop. Architecture and design of MARS, a microprogrammable accelerator for rapid simulations.

[16] Shekhar Borkar et al. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 70–81, May 1990.

[17] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 111–122. ACM, October 1992.

[18] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224–235, San Diego, California, May 1993. IEEE.

[19] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the Alewife multiprocessor. Techreport MIT/LCS/TM-498, MIT Laboratory for Computer Science, Cambridge, MA., 1993.

[20] Jiun-Ming Hsu and Prithviraj Banerjee. A message passing coprocessor for distributed memory multicomputers. In *Proceedings of the 5th International Conference on Supercomputing*, pages 720–729. International Supercomputing Institute, Inc., 1990.

[21] W.J. Jager and Loucks W.M. The P-Machine: A hardware message accelerator for a multiprocessor system. In *Proceedings, International Conference on Parallel Processing*, pages 600–609, 1987.

[22] Jeffrey Rothman. Vlsi design of a network interface processor. Techreport UCB/CSD/91/647, University of California, Berkeley, Computer Science Division., Berkeley, CA 94720., September 1991.

[23] G. M. Papadopoulos et al. *T: Integrated building blocks for parallel computing. In *Proceedings of the Eighth International Conference on Supercomputing*, pages 624–635. International Supercomputing Institute, Inc., 1993.

[24] Ellen Spertus et al. Evaluation of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 302–313. ACM, May 1993.

[25] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175. ACM, April 1991.

[26] William J. Dally et al. The MAP instruction set reference manual v1.1. Techreport MIT Concurrent VLSI Architecture Memo 59, MIT Artificial Intelligence Laboratory, Cambridge, MA., 1994.

[27] William J. Dally et al. M-Machine microarchitecture v1.0. Techreport MIT Concurrent VLSI Architecture Memo 60, MIT Artificial Intelligence Laboratory, Cambridge, MA., 1994.