An Evaluation of Fugu's Network Deadlock Avoidance Solution

by

Victor Lee

B.S. Electrical Engineering
University of Washington, June 1994

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

June 1996

Author .....
Department of Electrical Engineering and Computer Science
May 10, 1996

Certified by .....
Anant Agarwal
Associate Professor of Computer Science and Electrical Engineering
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

An Evaluation of Fugu's Network Deadlock Avoidance Solution

by

Victor Lee

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

In modern multiprocessor systems, interprocessor communication is extremely important. An anti-deadlock strategy is an essential part of a quality communication service. It guarantees that the system can always make forward progress, and it provides the system with tolerance over misbehaving applications.

The design of the anti-deadlock strategy has significant impact on the system cost and performance. This thesis presents the Fugu strategy which is an end-to-end anti-deadlock strategy. It achieves good performance with minimal architectural support. The Fugu strategy prevents deadlocks by emulating an infinite network resource and by preventing any application from blocking the network for an extended period of time. The infinite network resource is achieved in two steps with different performance trade-offs. First, the system converts the network resource consumption into memory consumption by emptying the network into memory when a possible deadlock is detected. Second, memory on each node is backed up by external I/O devices. The Fugu strategy provides a dedicated system network to ensure that each node can always reach the external resources. In our implementation, a deadlock-free network layer provides message transmission reliability, and a system timer is used to detect deadlocks. Our implementation further optimizes the system performance by tightly integrating the anti-deadlock strategy with the operating system scheduler. By allocating part of the second network's bandwidth to the flow-control process, network performance is drastically improved. This is because the flow-control process provides feed-back to allow the scheduler to effectively schedule processes that optimize the network usage.

Analyses and Simulations show that a moderate second network design is sufficient to support a relatively large multiprocessor system. We also find that the Fugu strategy scales relatively well and performs better than two other common anti-deadlock strategies.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Electrical Engineering

Acknowledgments

Writing a master's thesis is no small feat. I would like to take this opportunity to thank all who have helped me along the way to complete this thesis.

First, I would like to thank all members of the Fugu project. Without them, the Fugu project would not be possible. Special thanks goes to my advisor, Anant Agarwal, for giving me the opportunity to explore the Fugu project. Without his guidance and keen insights, I would have been lost in my research. I am greatly indebted to Ken Mackenzie for his generous help and for teaching me the various aspects in research. Many thanks to Rob Bedichek who built the basic T2 simulator for us. I also want to thank Matt Frank, John Kubiatowicz, Walter Lee, Jon Michelson and Don Yeung for their help and encouragement.

I would also like to acknowledge all my friends Francis, Tony, Tony, Rebecca, Simon, Walter, Patrick, Ken, Don, Ernie, Elaine, Kitkat, Matt, Vivien, and Herman for their cheers and encouragement. Many thanks to Walter and Tom for proofreading my thesis drafts on very short notice.

Last but not least, I would like to reserve the biggest thanks to Mom, Dad and my sister Sonya who have supported and encouraged me with unfailing confidence and understanding. They have given me the chance to learn and explore and for that I will always be grateful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In concurrent computing, communication is important. Whether it is the communication between two separated uniprocessors or the communication between two nodes in a massively parallel supercomputer, it is essential to have a communication service that guarantees forward progress in delivering messages and the system would not be crashed by misbehaved applications.

Over the years, the research community has developed different implementations for this type of service (See figure 1-1). At one end of the spectrum, multiple software layers are used to provide these guarantees in an implementation environment where the network hardware and software are not reliable. In this environment, the hardware or the device software are allowed to drop messages to ensure forward progress or to prevent deadlocks. The TCP protocol is an example of the solutions in this end. At the other end of the spectrum, the network hardware and software are made to behave in a reliable way. In this environment, the network layer guarantees to deliver the messages it accepts. The service guarantees in this case are fulfilled by a system level anti-deadlock strategy. Most modern Massively Parallel Processors (MPPs) and Symmetric Multiprocessors (SMPs) use this approach.

In recent years, many advances have been made in microprocessor development. Nowadays, systems are built with much tighter integration. With the new technologies, multiprocessor architectures can finally realize their potential to out-perform uniprocessor architectures. In a multiprocessor system where internal networks are

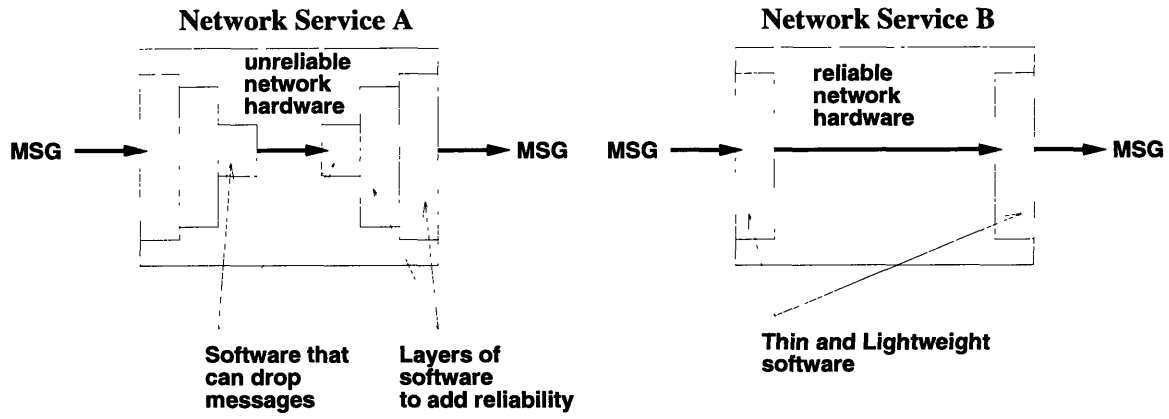


Figure 1-1: The Solutions for Reliable Network Service

built in a reliable way, the anti-deadlock strategy is an essential part of the design. It can significantly affect the system performance.

Designing an anti-deadlock strategy for multiprocessors is challenging. A complete anti-deadlock strategy must address all possible deadlocks. In a system, deadlocks are caused by circular dependencies. Circular dependencies can exist in any communication layer or in a combination of multiple layers. We have identified circular dependencies internal to three communication layers: the network layer, the protocol layer and the application layer.

Since most previous-generation multiprocessors only support a single user or at best a crude form of multiprocessing with a simple scheduling scheme, earlier researches mainly focused on solving deadlocks in the network layer and the protocol layer. Only a few application level anti-deadlock solutions have been proposed. Previous solutions to deal with application related deadlocks include: 1) transmitting all user level messages through a guarded operating system kernel layer [29, 31, 28, 17, 26, 19, 24], 2) using a time-out mechanism to preempt applications that block the network for an extended period of time [1, 8], and 3) providing a separate network buffer for each application and context-switching the network buffers when one application finishes its current time-slice or when it is deadlocked [6, 34].

The current generation of multiprocessors begins to explore the multiuser domain. In this domain, application caused deadlocks become more prominent. The simple strategies used in previous generation systems no longer work, or they do not satisfy

the performance requirements. To achieve high performance and to prevent any deadlock from interfering with other applications, many new systems use elaborate networking schemes and dedicate separate network resources for each application. For example, one proposal provides one virtual channel for each application on the system. However, these schemes are often complex and costly to build. This motivates the development of a cost-effective anti-deadlock strategy in the Fugu research project.

The Fugu project aims at designing a high performance multi-user, multiprocessor machine. The Fugu architecture is greatly influenced by the Alewife architecture. [2, 25] It inherits most of the Alewife features such as cache coherent non-uniform memory access (CC-NUMA) and user-level access to underlying network hardware. The Fugu architecture also has many new features such as multi-user support, shared virtual memory, a new process scheduling scheme, and a new anti-deadlock strategy.

The goal of the Fugu anti-deadlock strategy is to provide an anti-deadlock scheme that requires minimal architectural support and does not degrade the system performance. By recognizing that it is overly specified to provide a separate set of network resources for each application, the Fugu strategy only provides one set of network resources for all applications. To prevent the limited network resources from becoming the source of deadlocks, the Fugu strategy converts the consumption of scarce network resources into consumption of more plentiful memory resources. The Fugu strategy achieves this by emptying the network into memory when a possible deadlock is detected. The Fugu strategy further ensures there is always enough memory to handle network overflow by reserving a channel from every node to its backing store.

Our implementation of the Fugu strategy uses a system timer to time out any application that blocks the network for an extended period of time. When this happens, the operating system begins to buffer the subsequent messages in the network. If too many messages are required to be buffered and the application's message handler is slow to handle each message, the application's network buffer inevitably will run out of space. In this situation, the operating system initiates a throttling process to stop the senders and assigns a temporary overflow buffer to the application. If during the

throttling process, the node runs out of memory, the operating system has to page out a part of the application's message buffer. The Fugu strategy provides a system-only second network to ensure that overflow paging is always possible. When the overflow condition disappears, the operating system re-schedules the sender processes in the overflowed application. As an optimization, the anti-deadlock strategy is tightly integrated with the operating system's scheduler. The flow-control messages are being sent via the second network to ensure a timely action. This allows the scheduler to use network overflow condition to effectively schedule processes that optimize the network usage. This in turn leads to improved network performance.

Analysis and simulation results in this thesis show that the requirements for the second network are very minimal. With a moderate second network design, the Fugu strategy provides a low-cost high-performance solution to resolve deadlocks in multiprocessor communication systems. We have implemented a serial token ring second network in the Fugu simulator and it can support a machine design up to 32 nodes without much performance degradation. The rest of this thesis presents and evaluates the Fugu anti-deadlock strategy.

1.1 Contribution

This thesis makes two contributions. First, it presents a multiprocessor anti-deadlock strategy that requires minimal architectural support. A detailed analysis and the performance evaluation presented in this thesis enable system designers to gain insights in how this strategy achieves a good trade-off between cost and performance. Second, it systematically identifies and groups deadlocks and their solutions. This approach allows the system designers to clearly identify the deadlock problem and to apply appropriate solution at different communication layers.

1.2 Overview

Following this introduction, chapter 2 examines the different kinds of deadlocks in a multiprocessor network. It provides an overview of the anti-deadlock solutions suggested by previous researches and summarizes the anti-deadlock strategies used in different multiprocessors. Chapter 3 describes the Fugu anti-deadlock strategy. It develops a framework for analytical study and implementation. A justification of the strategy's correctness and functionality is also presented. Chapter 4 describes the analytical study of the Fugu strategy. Chapter 5 presents an implementation of the Fugu strategy based on the numerical results from the analytical study. Chapter 6 presents the simulation and results of the experiments run to validate the Fugu strategy. Chapter 7 concludes the thesis.

Chapter 2

Deadlocks and Anti-Deadlock Strategies for Multiprocessor Networks

In a multiprocessor network, deadlock occurs when no message can advance toward its destination. Deadlocks are caused by circular dependencies in the system, and a circular dependency can involve a single communication layer or multiple communication layers.

This chapter examines the different kinds of deadlocks that occur at the different communication layers and studies the solutions for these situations. The first section discusses the deadlocks and their solutions at various communication layers. The second section summarizes the anti-deadlock strategies used by the different multiprocessors.

2.1 Deadlocks and their solutions at Different Communication Layers

In [4], Bertsekas and Gallager use a layering approach to analyze a communication system. In the layering view, each layer abstracts away the internal details of the

implementation and exposes only the functionalities and the interface. Each layer consists of all the sub-layers below and provides the abstracted view to the layer above. In a multiprocessor, a circular dependency inside any layer or across multiple layers can cause the system to deadlock. We have identified three layers where circular dependencies can occur. These layers are the network layer, the protocol layer and the application layer. This section discusses the deadlocks at these layers and studies the solutions proposed for them.

2.1.1 Deadlocks at the Network Layer

The network layer consists of the physical layer, the link layer and the network layer itself. Its main function is to transport packets from one node in the network to another.

At the network layer, deadlock can occur if the following two conditions are met. First, the network allows packets whose size is larger than the total hardware buffer existing between the source and the destination. Second, a circular loop exists in the packet routing. For example, in a 2D mesh network using worm-hole routing, if a large packet is routed with four left turns in the same channel, the head of the packet will run into the middle of itself and the network will deadlock.

Dimension ordered deterministic routing is the simplest strategy used in worm-hole routed networks to prevent deadlocks at the network layer. The idea is to send packets as far as possible in one dimension of the network before switching to another dimension. This method is exceedingly simple and provides low latency and high bandwidth; however, deterministic routers do not perform well when there is any skew in the network load, and they have poor fault tolerance.

Recent researches show that adaptive routing schemes have better performance and fault tolerance than deterministic routing strategy. Adaptive routing schemes allow the routers to make decision to misroute a packet (to route a packet along a non-shortest path) in order to avoid congested or faulty segments of the network. However, fully adaptive routers are complex to build. It is not clear whether the performance increases in routing flexibility can offset the decrease in the speed of the

router [7].

Dally and Seitz were the first to propose an inexpensive scheme for adaptive routing by using adaptive virtual channels [12]. The idea of virtual channels is to divide up the physical channels into smaller but more versatile virtual channels. Dally showed in [10, 11] that using virtual channels for adaptive routing can increase the channel utilization and the overall routing performance. He also showed that adaptive routing with virtual channels is deadlock free if the allocation of the virtual channels is strictly based on the number of dimension reversals [11].

Earlier research by Linder and Harden [27] showed that 2^{n-1} virtual channels per physical channel are required to support full adaptivity in n-dimensional meshes. Later, Chien and Kim [23] proposed the planar-adaptive routing (PAR) scheme which reduces the cross bar size and the virtual channels required per physical channel. But PAR is only a partial adaptive routing algorithm. It limits the adaptive routing to only two dimensions at a time. At about the same time, Glass and Ni [16] introduced another partial adaptive solution: the Turn model. The Turn model does not require adding physical or virtual channels to resolve deadlocks. It analyzes the directions in which packets can turn. By prohibiting just enough turns, it can provide a routing algorithm that is deadlock-free for a given network. Recently, Boura and Das proposed a new fully adaptive routing algorithm that uses their direction restriction model [5]. Boura and Das's algorithm requires only two virtual channels per physical channel for n-dimensional meshes. Other researches in adaptive routing can be found in [18, 33, 13, 37].

2.1.2 Deadlocks at the Protocol Layer

The protocol layer provides programmers with the communication abstraction in terms of messaging protocols. Using these protocols, programmers do not need to be concerned with how the messages are transported. In addition, these protocols are intended to help the programmers write deadlock-free applications. Because of this, the protocols themselves must be deadlock-free. Two common messaging protocols in use are: the request and reply messaging protocol and the shared memory protocol.

The request and reply protocol provides a deadlock-free messaging environment but requires two separate networks to implement. The request and reply protocol breaks all communications down into two classes: the "request" and the "reply". The "request" handler can only send zero or one message (the "reply"), and the "reply" handler can only extract data from the network and cannot send any message. To form a circular loop under this protocol, it must involve at least one "request" message and one "reply" message. But this protocol dictates that these two classes of messages must be transmitted in two separate networks. Therefore, no circular loop is possible in this protocol, and it is deadlock-free. The disadvantage of this protocol is that it is extremely difficult to identify the relative bandwidth requirements for the two networks. As a result, most implementations use two identical networks.

The shared memory protocol provides an abstract view of a single globally shared memory despite the physical memory being statically distributed in the machine. Messages in this protocol are explicitly managed by the system. The applications do not need to send messages to communicate. Rather, they read and write to a commonly known location in the memory to communicate and to exchange data. Recent researches have shown that this programming model makes programming parallel applications much easier and it is generally preferred. The main disadvantage of this protocol is performance. Some hardware implementations claim good performance at the cost of more complex system design. Moreover, modern processors require caches to attain their performance. Maintaining cache coherence in a shared-memory multiprocessor is not an easy task. [26, 1, 19]. In particular, the shared memory coherence protocol can contain circular dependencies and is prone to deadlock.

To make the shared memory protocol deadlock-free, multiprocessors generally implement their shared memory on top of a request and reply protocol. Implementations of such design can be found in many modern multiprocessors [26, 34, 19, 8]. However, this design adds more cost to the system. The MIT Alewife machine [2] uses a different solution for solving deadlocks in the cache coherence protocol. It allows the system to back-off from a protocol deadlock by buffering network messages in the system memory when the system detects the network is blocked for an extended

period of time. A system timer associated with the network output queue is used for detection. The Alewife strategy takes advantage of the fact that memory locations are generally shared by only a few nodes. Therefore, under most circumstance, allowing the network to overflow protocol messages into the memory will not cause any node to run out of memory. However, the danger exists that in a pathological case, the system can run out of memory and crash. In actuality, experimental results show that the Alewife strategy achieves a nearly deadlock-free solution [25].

2.1.3 Deadlocks at the Application Layer

The application layer abstracts away the implementation details of a system. The application layer encompasses the entire system including all possible user applications. Since there is no control over the user application, there is no guarantee a user program will not deadlock a network. Protecting the system's integrity and allowing all applications to make forward progress on the machine become the primary concern in the application level anti-deadlock strategy.

Two approaches are commonly used to deal with deadlocks in the application layer. They are the deadlock avoidance approach and the deadlock recovery approach. The deadlock avoidance approach tries to avoid deadlocks at all cost. The deadlock recovery approach allows deadlocks to happen and attempts to recover from them. Newer multiprocessors are beginning to use a combined approach to relax the system requirements inherent in both extremes.

A way to implement the deadlock avoidance approach is to make all user messages transmit via a guarded operating system kernel layer. In a fully protected operating system, a typical message transmission starts with a system call to the operating system to set up a communication channel (we will call this the heavy-weight approach). The operating system then checks for protection violations and performs the buffer management. When all are done, the operating system will either transmit the message provided by the user, or it will hand over a dedicated communication channel to the user process for a single transmission. The Intel NX/2 operating system [31] that runs on the Intel Delta and Paragon multiprocessors [28, 17] is an example of

this type of operating system. The main disadvantage of this strategy is poor performance. Poor performance can be attributed to large transmission protocol overhead - each user message requires at least two system messages and one data transfer. Typical send and receive operations cost thousands of cycles. Moreover, the overhead is uniformly applied to all user messages.

An alternative implementation for software deadlock avoidance approach is the software reservation strategy. It is suggested in [24] and has been implemented on a number of multiprocessors including the CM-5 [6]. This strategy ensures that at most a bounded number of messages are outstanding between any two nodes in the network at any time. Each node is given a fixed quota for each receiver at the beginning. The quota is consumed every time the node sends a message. When the quota runs out, the node will not attempt to send any more message until the quota is renewed. This strategy improves the per message overhead cost but suffers from an inflexible use of network resources. Each node must allocate a large memory buffer to accommodate all possible senders. Furthermore, the unused portion of these buffers cannot be used by others who need them. The Princeton SHRIMP project [29] uses a variation of this strategy. It provides one network buffer for all senders in an application, and the buffer space is allocated dynamically. Before transmitting a group of messages, the sender requests enough buffer space to accommodate its messages. By grouping a series of user communications together, the SHRIMP protocol applies one setup cost to a number of transmissions. This way the expensive setup cost is amortized. The SHRIMP protocol is more flexible than the reservation strategy. However, it still suffers from poor performance because the setup cost is expensive.

The Stanford DASH and FLASH [26, 19] machines take a different approach to the protected operating system strategy. In these machine, the operating system only checks to see if the user messages conform to the pre-programmed messaging protocols (we will call this the light-weight approach). In the light-weight approach, user communications can avoid most of the operating system overhead if they conform to the system messaging protocols. If they are not, the operating system imposes the heavy-weight protocol on them. Although this strategy provides significant im-

provement in performance for the common case, it requires substantial hardware and operating system assistance, and it places restrictions on the programming model.

The deadlock recovery approach lifts the restriction on the programming model by allowing the applications to deadlock the network. The deadlock recovery approach typically uses some dedicated resources to ensure that a communication channel always exists for the operating system to recover the network. Cm* [32] is the first machine that uses a notion of deadlock recovery approach to resolve deadlocks. Although in the Cm* system, the only backup resource the system has is an interrupt system to trap all processors and to cause them to run the operating system code, this is enough to resolve simple deadlocks. Thinking Machine Corporation's CM-5 [6] is a commercial multiprocessor that uses a more complex system network. The primary purpose of the CM-5 system network is to provide user isolation and fault tolerance, but it is also the primary resource used for a deadlock recovery. The MIT Alewife [2] and the Wisconsin Typhoon [34] use a time-out mechanism to detect and preempt any application that blocks the network for an unreasonable amount of time.

2.2 Anti-Deadlock Strategies used by different Multiprocessors

This section summarizes the anti-deadlock strategies used by different multiprocessors to deal with deadlocks at different communication layers.

Alewife [2] uses the dimensional ordered routing solution to resolve the network layer deadlock problem. It provides a shared memory protocol but it is not entirely deadlock-free. The Alewife system guarantees the system would not deadlock by using a timeout mechanism to detect protocol and application deadlocks. In case of a protocol deadlock, the system tries to buffer all the network messages in system memory and hopes the deadlock resolves itself before the system runs out of memory. Otherwise, the system would crash. In case of an application deadlock, the system preempts the application that causes the deadlock and frees the network.

CM-5 [6] uses a fat-tree network topology to get rid of any circular dependency

in the network layer. CM-5 provides the request/reply protocol to the user for direct deadlock-free communications. It also allows user applications to deadlock the machine. CM-5 ensures it can reverse an application level deadlock deadlock by using a dedicated system control network and a separate buffer for each application.

DASH [26] uses the dimensional ordered routing to avoid network layer deadlocks. DASH only provides the shared memory protocol at the protocol layer. The shared memory is implemented on top of the deadlock-free request/reply protocol. The operating system dictates that applications must use the shared memory protocol to communicate; therefore, there is no deadlock issue at the application layer.

FLASH [19] provides the request/reply protocol for message passing applications and the shared memory protocol for shared memory applications. At the application layer, all user messages are processed by a light weight kernel in a special protocol processor. The protocol processor ensures user messages conform to the deadlock-free protocols.

M-Machine [30] uses the adaptive virtual channel technique to solve network layer deadlock problems. In the protocol layer, the request/reply protocol is used to provide deadlock-free communication. An implicit buffer management scheme is used to control the rate at which messages are injected into the network to avoid application layer deadlocks. A software shared memory coherence protocol is also available for shared memory programming model.

Paragon [17] uses dimensional ordered routing in the network layer to resolve deadlocks. At the application layer, user messaging is performed under a fully protected operating system.

Star-T NG [8] also uses a fat-tree topology that cannot deadlock its network layer. Its router uses virtual cut-through routing with large internal buffer to avoid deadlocks. The capability to support virtual channels in the router chip allows Star-T NG to provides the deadlock-free request/reply protocol at the protocol layer. At the application layer, it uses the deadlock recovery approach by implementing a system network in the high-priority channel in the network. A software implemented shared memory coherence protocol is also available.

Typhoon's hardware [34] provides support for implementing request/reply deadlock-free protocol. At the application layer, deadlocks are isolated by using a separate network buffer for each application.

Table 2.1 summarizes the anti-deadlock strategies in the multiprocessors mentioned in this section.

Multiprocessor	Anti-deadlock mechanism/approach used at Network Layer	Protocol Layer	Application Layer
Alewife	Uses the dimensional ordered routing scheme to eliminate deadlocks	Uses a hardware assisted shared memory protocol which uses a network overflow mechanism	Uses a timeout mechanism to preempt applications that blocks the network
CM-5	Uses the Fat-Tree topology which does not deadlock.	Provides the request/reply protocol.	Uses separate context switch buffers to isolate application deadlock. Also uses a dedicated system control network to recover the system from deadlocks.
DASH	Uses the dimensional ordered routing scheme to eliminate deadlocks	Provides the shared memory protocol which is implemented on the request/reply protocol.	Forces applications to use the shared memory protocol; therefore, no deadlock exists here.
FLASH	N/A	Provides the request/reply protocol for message passing, and the shared memory protocol.	Uses a protocol processor to check and ensure all user messages conform to the deadlock-free protocols.
M-Machine	Uses an adaptive routing scheme to eliminate deadlocks	Provides the request/reply protocol. Also provides a software shared memory protocol is available.	Uses an implicit buffer management scheme for flow control, also uses a high-priority channel in the network as a system network for deadlock recovery.
Paragon	Uses the dimensional ordered routing scheme to eliminate deadlocks	N/A	Processes all user messages in a fully protected kernel.
Star-T NG	Uses the virtual cut-through routing with large internal buffer to avoid deadlocks	Provides the request/reply protocol. Also provides a software shared memory protocol.	Uses a timeout mechanism to detect deadlocks and uses a system network to recover from deadlock.
Typhoon	N/A	Provides the request/reply protocol. Also provides a software shared memory protocol.	Uses separate context switch buffers to isolate application deadlock and uses a timeout mechanism to detect deadlocks.

Table 2.1: Summary of the anti-deadlock strategies on different multiprocessors

Chapter 3

Fugu's Anti-Deadlock Strategy

Chapter 2 examines the anti-deadlock solutions employed by various multiprocessors at different communication layers. This chapter describes the Fugu anti-deadlock strategy and provides a proof of functionality. In developing the Fugu strategy, our goals are to design an anti-deadlock strategy that requires minimum system support and does not degrade the system performance.

From the previous chapter, we learn that it is fundamental to have a deadlock-free network layer. In the protocol layer, deadlock free communication protocols can aid in developing deadlock-free applications. However, the programmers still have the freedom to choose not to use these protocols. Therefore, providing deadlock-free protocols does not guarantee that the system is deadlock-free. As a result, an application level anti-deadlock strategy is necessary if we want to provide the guarantees we stated at the beginning of this thesis: *Network services should guarantee forward progress in delivering messages and should guarantee a misbehaved application cannot crash the system.*

The first section describes the operating environment for which the Fugu strategy is designed. It describes the assumptions on the target machine, the deadlock condition which we are trying to solve, and the goal of the Fugu anti-deadlock strategy. The second section describes the Fugu strategy and develops a framework for analysis and implementation. The third section provides a proof of the Fugu strategy's functionality. The last section compares the Fugu anti-deadlock strategy with other

multiprocessors' strategies.

3.1 The Operating Environment

This section describes the multiprocessor environment for which the Fugu anti-deadlock strategy is developed. The first part states the assumptions on the operating environment. The second part describes some possible deadlock situations. The last part states the goal and the functionality for the development of the Fugu's strategy.

3.1.1 Assumptions

For the multiprocessor of interest, we assume the followings in the operation environment:

Assumption 1: The multiprocessor has only one network which is accessible to the user applications. The network (including both the hardware and the software) is assumed to be reliable and deadlock-free. We use the term "reliable" to mean that the network would not drop any packet or message. The term "deadlock-free" means the network routing algorithm will not deadlock itself. Together, both terms bind the network to provide a service contract which states that if the network accepts a packet from the processor, it guarantees its delivery to its destination. However, the network can choose not to accept new packets when it has no more room for any new message. For example, the network can be blocked by the flow-control mechanism to limit the rate at which the messages are sent.

Assumption 2: The multiprocessor supports a concurrent multiuser environment. This means multiple user processes can be active on the machine at the same time.

Assumption 3: A finite amount of internal memory is present at each node. External I/O devices are attached to some (but not all) nodes in the machine. External I/O devices are assumed to have an infinite capacity.

Assumption 4: The user applications can deadlock the machine. We simply assume the user applications can violate any deadlock-free protocol.

3.1.2 The Deadlock Situation

In this section, we first consider a simple deadlock situation and then consider a more complicated deadlock situation.

A Simple Deadlock situation

Consider a multiprocessor with only one user application. In node N1, process A sends requests to node N2. At the same time, process B on node N2 sends requests to node N1 for data. Suppose that both process A and process B need the data replies before they can continue. Now both node N1 and N2 are deadlocked. Figure 3-1 illustrates the simple deadlock situation.

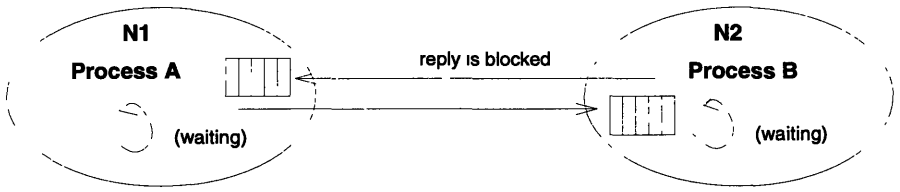


Figure 3-1: A Simple Deadlock Situation.

One observation from this situation is that if there is only one central network resource, circular dependence can arise and deadlock can happen. To prevent deadlocks in this situation, we either provide more network resources or provide a way for the system to back out from a deadlock.

A More Complicated Deadlock Situation

Consider in a multiuser multiprocess environment, user $U1$'s process A in node $N1$ sends some requests to its counterpart in node $N2$ and spin-waits for the reply. At the same time, user $U1$'s process B on a third node ($N3$) sends data messages to process A in $N1$. Then, after some computation process B waits for process A to send data. In addition to user $U1$, the machine has a second user $U2$. While these

events are happening, user $U2$'s process C on node $N4$ also sends request messages to its counterpart in $N2$ and blocks itself waiting for the replies. Suppose that both process A and process B have launched enough messages to fill up the input and the output hardware queues in $N1$ and $N2$. The network is now deadlocked.

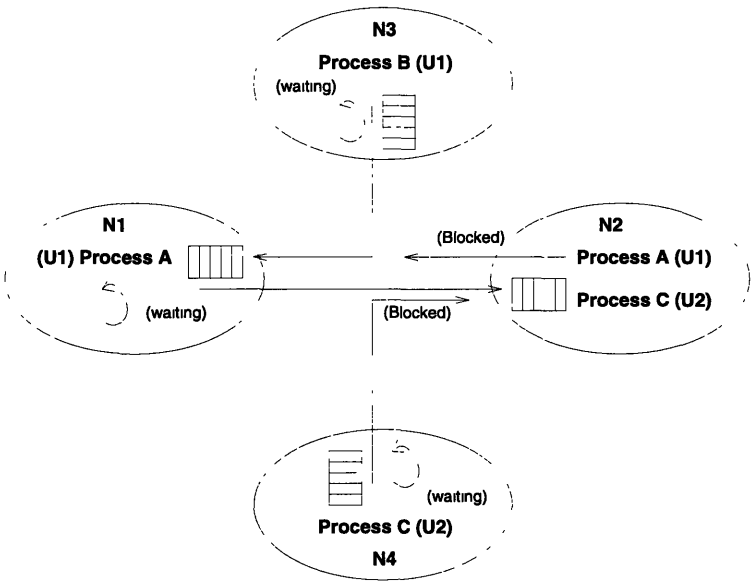


Figure 3-2: A More Complicated Deadlock Situation.

Figure 3-2 graphically illustrates the deadlock situation described. Process A, B and C cannot make any forward progress. Process A in $N1$ is blocked waiting for replies from $N2$, but the replies are blocked behind the messages from process B. Process A cannot process any message from process B. Therefore, process B cannot move forward. Process C cannot make any progress either because its messages cannot reach $N2$.

From our deadlock scenario, there are two important observations. First, process A is prone to deadlock by construct. Even if the system employs some deadlock-free protocols such as the request and reply protocol, process A would still cause the system to deadlock because its design does not conform to any deadlock-free protocol. Second, process C may be a deadlock-free process, but it simply runs at the wrong time and is caught in the network deadlock.

3.1.3 The Goal for the Anti-Deadlock Strategy

The goal of the application level anti-deadlock strategy is to solve the deadlock problem. We want to solve the problem in such a way that implements the service guarantees stated in the beginning of the thesis. We re-state them here: *Network services should guarantee forward progress in delivering messages and should guarantee the system would not crash because of a misbehaved application.*

3.2 A Description of the Fugu's Strategy

The Fugu anti-deadlock strategy provides an end-to-end strategy for a multi-user multiprocessor environment. It extends the Alewife strategy of timeout and buffering with the ability to create extra buffer space by paging out over a second network to a node with external I/O resources. In our implementation, we use a timeout mechanism to detect deadlocks. We also use a dedicated system network to transmit flow-control messages and to perform overflow paging. We believe this is an anti-deadlock strategy that requires minimum system resources. Next, we will describe a framework which we can use for analyzing and implementing the Fugu strategy.

A Framework for Analysis and Implementation

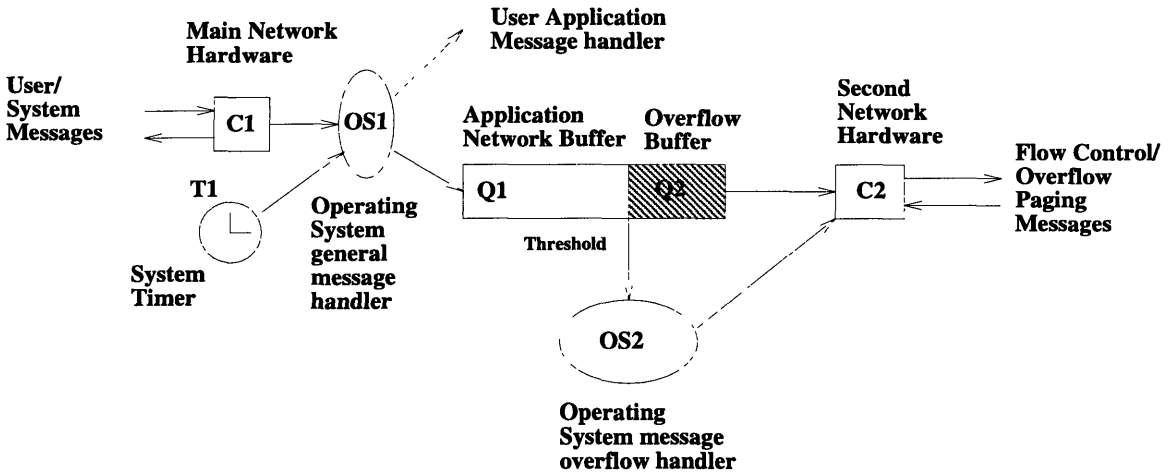


Figure 3-3: The Components in the Fugu strategy and its Data Flow Diagram.

Figure 3-3 shows a proposed framework for the Fugu's strategy. In this framework, the network layer implementation is assumed to be deadlock-free. The application layer anti-deadlock strategy is integrated with the overall communication system. Its main components are: 1) a network buffer ("Q1") associated with each application, 2) an overflow buffer ("Q2") that is dynamically allocated for the overflowed application, 3) an operating system handler ("OS1") that handles message deliveries, 4) an operating system handler ("OS2") that resolves congestion/deadlock situations, 5) a main network ("C1") that is accessible to the user applications, 6) a system-only second network ("C2") that is used for transmitting system messages, and 7) a system timer ("T1") that detects network blockage.

Under normal conditions, most user/system messages enter a processing node through C1, from the left-side of figure 3-3. The hardware controller in C1 first buffers these messages and then interrupts the processor. The processor then invokes OS1 to examine the message arrived. If the message is for the operating system, the operating system consumes it. If the message is for the currently scheduled process, OS1 delivers it to the process's message handler and starts the system timer T1. If the message is for a non-currently scheduled process, OS1 stores it in the non-scheduled application's Q1 of the inactive process.

If T1 runs out before the application's message handler finishes handling the current message, it signals that the current process has blocked the network for too long. The operating system temporarily suspends the process's message handler and informs OS1 to buffer all subsequent messages for this application.

Should an application's Q1 becomes full when OS1 tries to buffer an incoming message, the operating system invokes OS2 to examine and to correct the situation. When it is first called, OS2 allocates a temporary overflow buffer Q2 for the overflowed application, and it sends a flow-control message to each sender in the overflowed application. The flow-control messages tell the schedulers on these nodes to deschedule the sender processes. The senders are descheduled until the overflow situation is cleared.

If Q2 runs out of space before all the senders are descheduled, the operating

system invokes OS2 again. This time, OS2 has to page out part of the message buffer before the node can continue to function. The overflow paging mechanism is broken down into two phases: allocation of a remote page for overflow paging and the actual transfer of a page through the second network. The heuristic for selecting a candidate for overflow paging is first by randomly picking one of the sender nodes in the offending application. If the selected node does not have a free-page to allocate for overflow paging and it does not have an external I/O attached, it then forwards the request to another node at random. If the node does not have a free-page but has an external I/O device attached, it pages out one of its used pages to generate a free-page to accommodate the overflow paging request. The allocation process continues until the paging request reaches a node in the system that has a free-page or that is attached to an external I/O devices. This ability of some nodes to page out to external I/O devices provides the baseline guarantee for forward progress. The actual paging mechanism depends on the implementation of the second network. When overflow paging is completed, OS2 has freed up one page to be return to the system's free-page list. The system is then able to continue buffering other messages. If the system runs out of memory again, OS2 attempts to do overflow paging again. The operating system keeps track of the number of times overflow paging occurs in an application. It then uses this information to determine when an application should be terminated.

In the Fugu strategy the flow-control messages and the overflow paging are performed via a dedicated system-only second network. The implementation of the second network is flexible. It can be a virtual channel or a physical network. But the second network and its messaging protocol must be deadlock-free. When a flow-control message enters a node from C2 (the second network), the C2 hardware controller first buffers and examines the message. Then, it interrupts the processor and invokes the system routine specified in the message to carry out the flow-control action.

3.3 A Proof of the Functionality

This section provides a justification of the Fugu anti-deadlock strategy and proves that it is deadlock-free. This section begins by stating the system requirements for the machine. Then, it proceeds to the proof.

3.3.1 System Requirements

The system requirements necessary to make the Fugu strategy work are listed here:

1. All messages in the system must have a finite size.
2. There must be a system timer.
3. At least one node in the system must be attached to an external I/O device.
4. The external I/O devices must always accept data from the system. The external I/O devices are assumed to have infinite capacity (Assumption 3 in Section 3.1.1).
5. The second network hardware must provide a deadlock-free transmission network layer. (i.e. it must deliver all messages it accepts.)
6. The second network must provide some feedback, so the sender knows whether it needs to retransmitted the messages that are failed to be received. For example, our implement provides an ACK signal or an NACK signal to the sender for each message transmitted.
7. The second network interrupt must have the highest priority in the overall interrupt hierarchy.
8. The second network message handlers must have a finite execution time and must be run in an atomic session.

3.3.2 The Proof

This section presents a proof that the Fugu anti-deadlock strategy can provide a deadlock-free guarantee to the system. The assumptions used in this section are listed in Section 3.1.1. The requirements are listed in Section 3.3.1.

0. Establishing the necessary conditions for a system to be deadlock-free:

The first step in our proof is to identify the fundamental constituents of a deadlock-free system. Studying the deadlock situations in Section 3.1.2, we observe that the system with underlying reliable networks (*Assumption 1*) deadlocks because the system runs out of network resources when a receiver node is blocked by one of its processes. A careful analysis reveals the two fundamental constituents of a deadlock-free system. We summarize the necessary conditions required to construct a deadlock-free system in the following statement:

Statement 1: *A system is deadlock-free if it can ensure no process can block the network indefinitely and it has infinite network resources.*

Next, we need to show that the Fugu strategy provides the necessary conditions to satisfy the two constituents of a deadlock-free system outlined in *Statement 1.* The Fugu strategy achieves this by two mechanisms: timeout and buffering with an ability to page out to external I/O devices to create extra buffer space when the system runs out of physical memory.

1. The timeout mechanism guarantees no application can block the network indefinitely:

In a system where all messages have finite size (*Requirement 1*), a normal application does not block the network for an extended period of time to send a message. The Fugu strategy uses a system timer (*Requirement 2*) to detect any application that has blocked the network for a long time. When a time-out occurs, the operating system begins to buffer messages from the network.

2. Buffering allows a system to create a pseudo infinite network buffer by extending the network resource into the memory subsystem:

In the Fugu strategy, the infinite network resource is achieved in two steps with different performance trade-offs. First, the system converts the network resource consumption into memory consumption by emptying the network into memory when the system timer expires and the operating system begins to buffer messages from the network. Second, memory on each node is backed up by a channel to reach its backing store externally. The ability to reach an external I/O devices provides the baseline guarantee for forward progress in the system. The Fugu strategy uses a dedicated deadlock-free second network to provide a physical channel for each node to communicate with the nodes that are attached to external I/O devices. *Requirement 3* guarantees that at least one node in the system is attached to an external I/O device. *Requirement 4* guarantees that there is always room on the external I/O device to accommodate overflow paging.

Basically, we have shown that the Fugu strategy provides the two constituents for a deadlock-free system. What is left for us to show is that the second network is deadlock-free.

3. The second network is deadlock-free by construct:

Next, we are going to show that the second network is deadlock-free by construct. *Requirements 5* provides a deadlock-free messaging layer for the second network. Therefore, the second network guarantees deliveries of all messages it accepts. *Requirement 6* ensures the system transmits every messages that are intended to be transmitted. The second network we constructed so far provides a reliable network service and it is deadlock-free if every node in this network guarantees that it will eventually accept all messages sent to it. *Requirement 7* guarantees processor time on each node for handling the second network messages. *Requirements 4 and 8* ensure that each second network message can always run to completion. *Requirement 8* guarantees that the second network message handlers will not block the second network indefinitely. *Requirement 4* guarantees space for the second network handlers to complete their

tasks. Together, the system requirements 4, 5, 6, 7 and 8 enable us to construct a deadlock-free second network.

In summary, we have shown that the two fundamental constituents for a deadlock-free system are: 1) no process can block the network indefinitely and 2) the system can provide infinite network resources. We prove that the Fugu strategy provides sufficient conditions to satisfy the two constituents by using a timeout mechanism and by buffering with an ability to page out to an external backing store over a dedicated deadlock-free second network. We also show that given the system requirements we outlined in Section 3.3.1 the second network is deadlock-free by construct.

3.4 A Comparison with Other Anti-Deadlock Strategies

The Fugu's anti-deadlock strategy is similar to the Alewife strategy in that it uses the temporary overflow buffer, and it also resembles the multiple network strategy in CM-5 and Star-T NG. Taking advantage of the underlying exokernel operating system design [14, 15], the Fugu strategy allows the programmer (the person most knowledgeable of the application) to choose how to handle network overflow problems. The programmer can choose between using general purpose system library functions or writing a custom handler specifically for the application to maximize the network problem handling efficiency. Furthermore, by using a software oriented approach, the operating system can be programmed to adjust the system parameters dynamically. The Fugu strategy avoids the defects in other software approaches by not imposing deadlock avoidance overhead on regular network traffic in the main network and by efficient scheduling. Only the flow-control messages are made to bear the additional protocol overhead, which is imposed by the deadlock-free design of the second network. The system cost is made lower than that of the other implementations because of the use of small fixed-size network buffers and the use of a simple second network. In conclusion, we believe the Fugu strategy offers good performance at a relatively low cost. The Fugu strategy can also be realized on a system that provides deadlock-free

protocols. We have only shown a minimal design in this thesis.

Chapter 4

Analytical Study of the Fugu Anti-Deadlock Strategy

In the previous chapter, we have shown that the minimum architectural support required for an anti-deadlock strategy in a multiprocessor includes a system timer, a deadlock-free second network and a path to the backing store over the second network. Under the Fugu strategy, there are two usages of the second network: to send flow-control messages and to page out the overflowed software buffer . The requirements for the second network are very different under these two conditions. When sending the short flow-control messages, the transmission latency per message is important. When paging through the second network, the bandwidth is important. It is not clear what sort of features and performance requirements the second network should have. This chapter describes an analytical study to establish a mathematical model that relates the second network requirements to other system parameters. In particular, the transmission latency per message requirement and the bandwidth requirement for the second network are studied.

The first section describes the scenario we used to study the Fugu strategy. The second section describes the assumptions and the definitions we used in the analysis. The third section derives the second network transmission latency requirement. The last section derives the second network bandwidth requirement.

4.1 Scenario for Analytical Study

To study the effect of the Fugu strategy, we assume a scenario where a number of senders send messages to one receiver simultaneously and the receiver takes a long time to handle each message. As a result, the hardware buffer in the receiver is filled and the network is blocked by the hardware flow-control mechanism. When the network is blocked over an extended period of time, the system timer runs out and the Fugu anti-deadlock strategy is activated. The Fugu strategy first buffers the network messages into the application's network buffer. When the network buffer is filled, the Fugu strategy begins to send messages to deschedule the senders, and it also allocates a temporary overflow buffer for the application to continue buffering its messages. If during this deschedule process the receiver node runs out of buffer space in the overflow buffer, the Fugu strategy begins overflow paging to send part of the message buffer to another node.

Figure 3-3 in Chapter 3 shows the relationship between the various components in the Fugu anti-deadlock strategy. We have extracted the critical components in figure 4-1 to facilitate the development of the analysis.

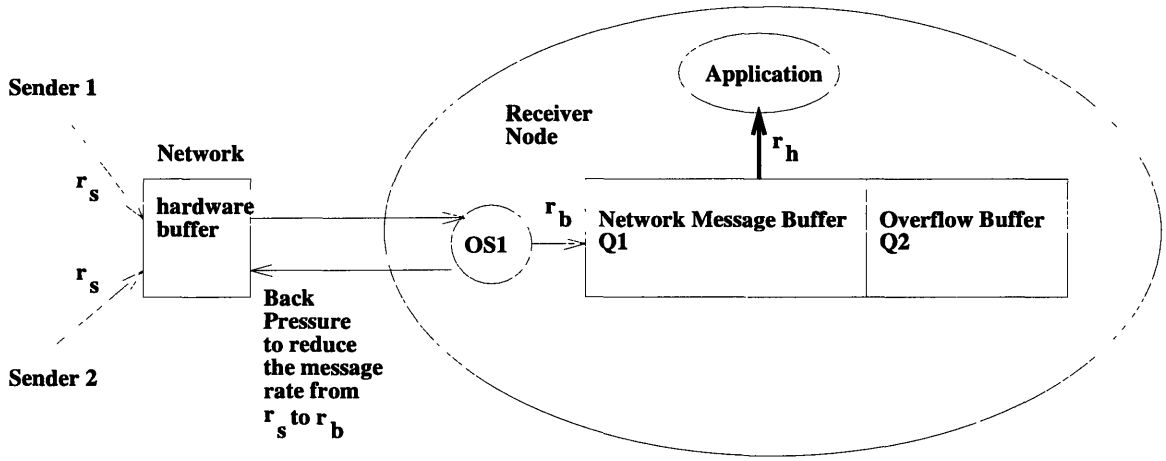


Figure 4-1: The Components in the Fugu Strategy.

4.2 Assumptions and Definitions

This section describes the assumptions and defines the notations we are going to use in the following discussion. These assumptions are made to simplify the analysis. After showing that the basic model works, we can relax the assumptions and introduce more complexity.

1. We assume all messages in the main network are unit size.

2. We assume there is no hardware buffer. Each message arrived at a node waits in the network until it is consumed or buffered. We further assume the system always buffers messages. (i.e. The system timer is set to expire when a message arrives at the input port of the network interface.)

3. We assume the rate at which each sender can send messages is r_s messages per cycle, the rate at which the application's message handler can handle messages is r_h messages per cycle, and the rate at which the operating system (OS1) can buffer messages is r_b messages per cycle. Suppose the system that we are studying has N processors, the maximum rate at which the messages can arrive at the receiver is $(N-1) \cdot r_s$ (We assume that except the receiver cannot send messages to itself every node in the system is sending messages to the receiver). In the scenario that we are interested, $(N-1) \cdot r_s > r_b > r_h$. Using the principle of conservation, we can derive the rate at which the software buffers (the application's network buffer, $Q1$, and the temporary overflow buffer, $Q2$) are filled. It is simply:

$$r_{fill} = min(N \cdot r_s, r_b) - r_h \tag{4.1}$$

$$= r_b - r_h \tag{4.2}$$

4. For the second network, we assume the software latency is small compared to the physical transportation latency. Therefore, in the following analysis, we ignore the software overhead in the message transmission latency model.

5. We further assume the second network hardware buffer only has room for one message and each second network message has the same size as the second network hardware buffer. If we let the size of the second network buffer be K, then the size of second network message is also K.

6. Finally, we assume that there is a fixed per message transmission overhead for each second network message. This overhead can be attributed to the design and the implementation of the second network. Some network designs inherently have a larger per message transmission overhead than the others. If we let this per message hardware overhead be T_{HW}, then we can derive a simple model for the second network transmission latency based on all the assumptions in the second network described. We have,

$$T_{SN} = T_{HW} + \frac{K}{B} \tag{4.3}$$

where T_{SN} is the second network per message transmission latency, and B is the second network bandwidth.

4.3 Deriving the Transmission Latency Requirement for the Second Network

The second network transmission latency is critical for transmitting flow-control messages in the Fugu anti-deadlock strategy. The transmission latency requirement is directly influenced by the size of the overflow buffer and the rate at which the application's message handler handles messages. This section derives the second network transmission latency requirement to prevent overflow paging.

When the application's message buffer reaches the buffer threshold, the operating system activates the process flow-control and allocates a temporary overflow buffer for the application. The flow-control process then sends flow-control messages to the rest of the system to deschedule the offending senders.

Assume the machine under study has N processors. The upper bound for the latency required to send flow-control messages to rest of the system is simply the

number of processors, N, times the second network per message latency, T_{SN}. Suppose the system provides an overflow buffer of size Q_{OVF} and it is filled at the rate, r_{fill} defined in Section 4.2. We can then relate these parameters and arrive at the second network transmission latency requirement to prevent overflow paging. That is,

$$T_{SN} \leq \frac{Q_{OVF}}{N \cdot r_{fill}} \tag{4.4}$$

This equation describes the maximum second network latency the system can tolerate for a given Q_{OVF} and r_{fill} before the system requires overflow paging.

Analysis:

In figure 4-2, we plot the overflow buffer size versus the second network per message latency required to prevent overflow paging based on equation 4.4. This figure shows that the second network latency requirement and the overflow buffer size have a linear relationship. The slope of the plot is equal to the number of processors in the system, N, times the rate at which the software buffer is filled, r_{fill}.

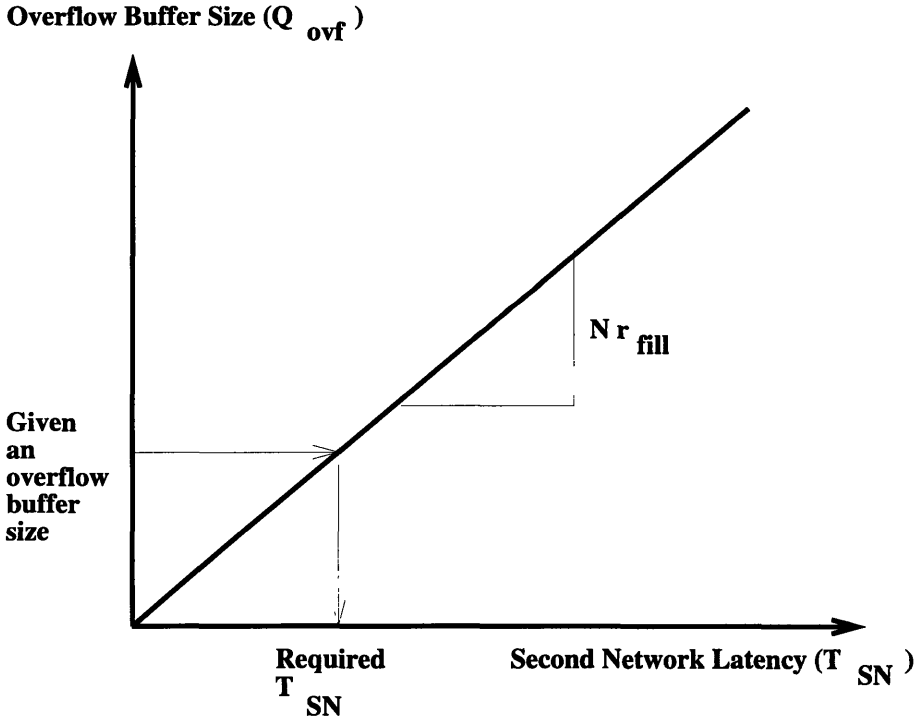


Figure 4-2: The Components in the Fugu Strategy.

Equation 4.4 also provides us a gauge to measure whether a certain second network design is sufficient for a particular system. For example, if the system we are interested in building has 32 nodes, the rate at which the operating system can buffer messages is one message per 100 cycle, the slowest application message handling rate the system can tolerate is one message per 1000 cycles, and the overflow buffer size is 100 messages, then from equation 4.4 we find that the maximum second network transmission latency the system can tolerate is 347 cycles before it requires overflow paging. For example, if a particular second network design requires 350 cycles to send a message in a 32-node configuration, then this network design should not be used in the system under consideration. However, in actuality, the 347 cycles requirement to send one messages in a 32-node configuration is a very minimal requirement. Even using a serial token ring design, we can easily achieve a per message latency less than 145 cycles in a 32-node configuration (assuming each flow control message is one word or 32-bit long).

4.4 Deriving the Bandwidth Requirement for the Second Network

The second network bandwidth becomes an important factor for network performance when overflow paging is required by the Fugu strategy. It determines how long the main network is blocked when the system is paging out a part of the overflowed message buffer. From previous section, we learn that overflow paging becomes necessary when the system runs out of physical memory trying to handle an overflow situation. There are two conditions which can cause the Fugu system to reach the limit of its software buffer: 1) the second network transmission latency is larger than the tolerance defined by equation 4.4, and 2) the physical memory in the system is very small.

Assume the unit of overflow paging is one page in the system that we are interested in, and each page has a size of P. Since the second network in our implementation does not directly support page size transfers. To perform overflow paging, we have to

break down the page to be transferred into small messages and send these messages out one by one. The number of messages required can be computed by dividing the second network message size, K, into the size of a page, P. Suppose that each message requires T_{SN} cycles to send in the second network. The total paging latency is:

$$T_{Page} = \frac{P}{K} \cdot T_{SN}$$

If we expand T_{SN} using equation 4.3 in Section 4.2, we have:

$$T_{Page} = \frac{P}{K} \cdot T_{HW} + \frac{P}{B} \tag{4.5}$$

where T_{HW} is the hardware overhead per message based on the network design and B is the second network bandwidth.

From this equation, we can study the effects of the various second network parameters on overflow paging performance.

Analysis:

From equation 4.5, we recognize the three second network parameters that affect the overflow paging performance are: the second network hardware overhead per message (T_{HW}), the size of the second network buffer or the size of the second network messages (K), and the second network bandwidth (B). After careful analysis, we realize the effect of the hardware overhead parameter and that of the buffer size parameter can be combined into a single parameter ($\frac{T_{HW}}{K}$).

In figure 4-3, we plot the overflow paging latency (which corresponds to the overflow paging performance) versus the second network bandwidth for three different network designs. In generating this plot, we assume the system page size is 100 messages and each message is 8-word or 256-bit long. The second network bandwidth is varied from a serial design (1 $bit/cycle$) to a 100 $bits/cycle$ design. The hardware overhead and the buffer size ratio is varied from 10 $cycles/message$ to 1 $cycle/message$ (a factor of 10 reduction).

The plot also shows that the effect of the second network bandwidth diminishes as the bandwidth is increased. We observed that the change in the bandwidth effect is

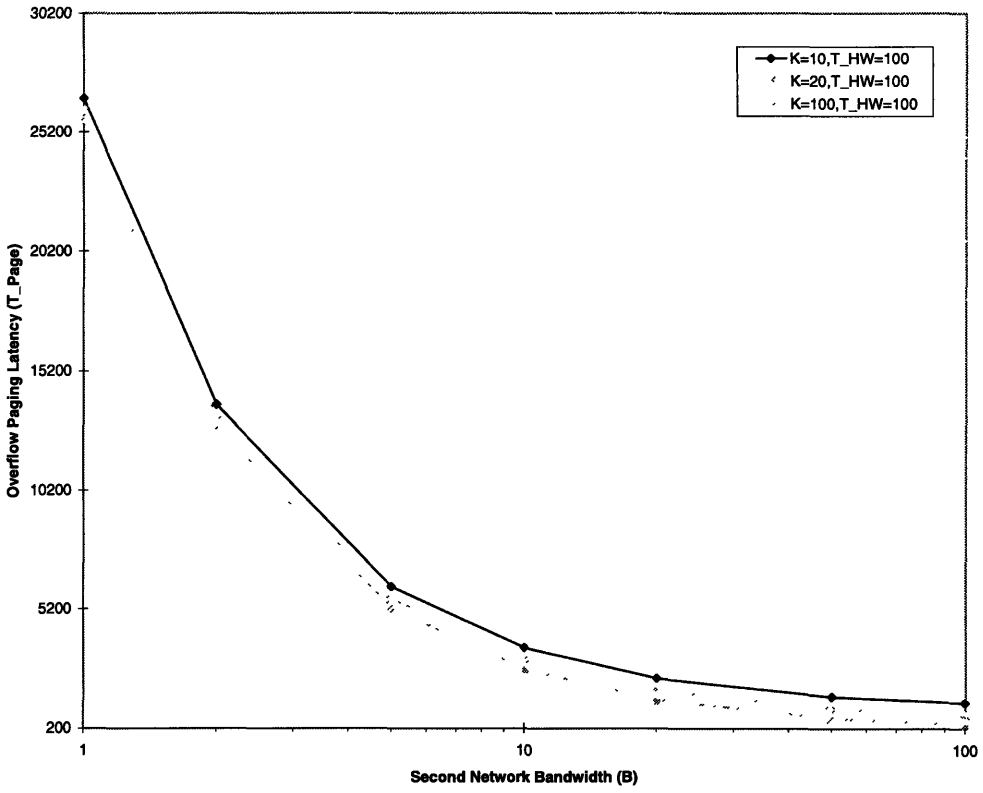


Figure 4-3: The overflow paging latency versus the second network bandwidth under different network configurations

minimal when the second network bandwidth increases beyond 5 *bit/cycle*. We also noticed that increasing the size of the second network hardware buffer or decreasing the per message hardware overhead has minimal effect (less than 1000 cycles in the largest case).

Chapter 5

Fugu's Implementation of the Anti-Deadlock Strategy

This chapter describes an implementation of the Fugu anti-deadlock strategy. The implementation consists of an operating system extension and a second network. The operating system extension detects deadlocks and generates the "corrective" flow-control action. The second network provides the physical layer to transport the flow-control messages to their destinations. The first section in this chapter provides a brief overview of the Fugu multiuser multiprocessor. The second section describes the operating system extension. The third section presents an overview of the second network interface and the hardware controller design.

5.1 The Fugu System

The Fugu system is an experimental multiuser multiprocessor under construction [22]. Its design is based on the Alewife single-user physically-addressed multiprocessor architecture [1]. The Alewife architecture, as shown in Figure 5-1, uses single-processor nodes to build a scalable multiprocessor machine. Each Alewife node consists of a Sparcle (SPARC V7 derived) processor, a sprac-compatible floating point processor (FPU), a custom Communications and Memory Management Unit (CMMU), 64K bytes of direct-mapped cache, 8M bytes of DRAM main memory and an Elko-series

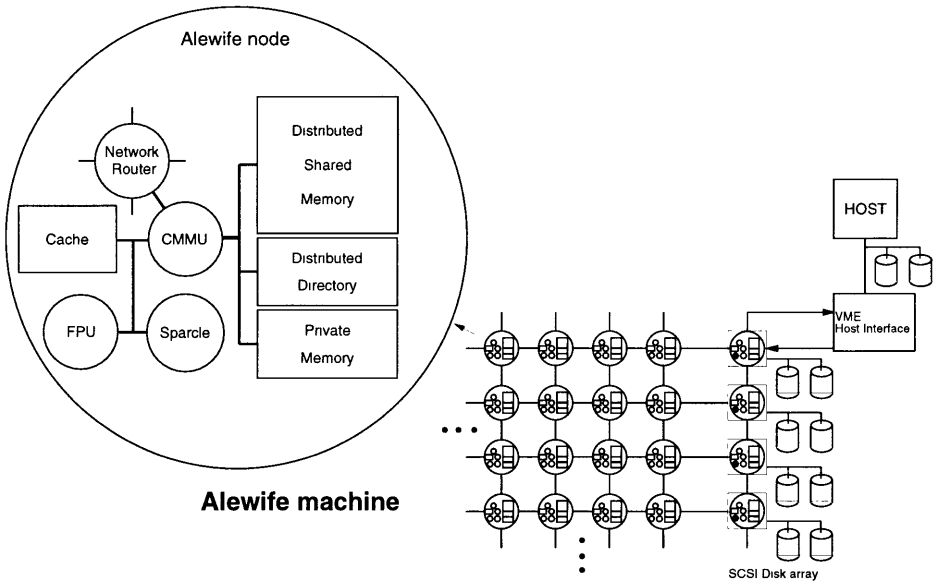


Figure 5-1: A View of the Alewife Architecture.

mesh routing chip (EMRC) from Caltech. The nodes communicate via messages through a direct network [35] with a mesh topology using worm-hole routing [9].

Fugu is similar to Alewife, but the two differ in a few architectural aspects: 1) Fugu is a multiuser machine. It supports multiple user processes concurrently running on the same hardware. Thus, it needs to provide protection mechanism to isolate individual processes; 2) Fugu provides virtual memory support for user programs to enhance programmability; 3) Fugu uses a two-network deadlock avoidance solution proposed and studied in this thesis.

A Fugu system is built by connecting many processor-nodes via a network system. Figure 5-2 shows the architecture of a Fugu node. Each Fugu node consists of all the integrated circuit components used in an Alewife node and a new User Communication Unit (UCU). The UCU is integrated on a single VLSI chip, and it consists of all the added features (such as the virtual memory support hardware - the TLB, the process protection check hardware and the second network controller). Currently, Fugu is under construction and the first version of the UCU is implemented on a Xilinx FPGA.

Fugu also has a new custom multiuser operating system based on the Aegis Exokernel [15]. The operating system supports multiprogramming, virtual memory,

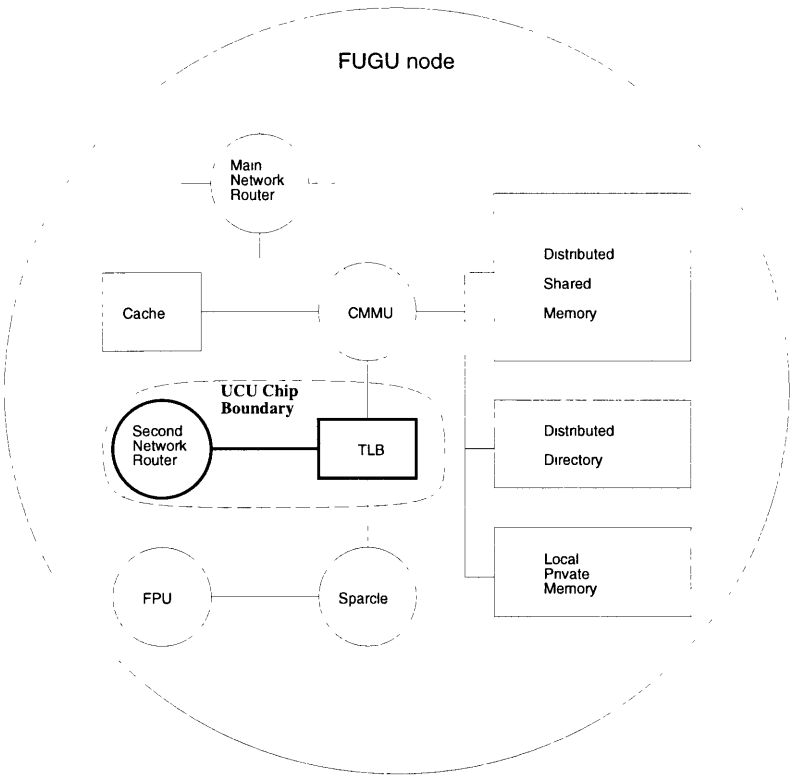


Figure 5-2: A View of the Fugu Architecture.

user-level messages, and user-level threads. It also implements a User Direct Messaging model that allows user-level, processor-to-processor messaging to coexist with general multiprogramming and virtual memory [21].

5.2 The Operating System Extension

The operating system extension for the Fugu anti-deadlock strategy consists of three parts. The first part is an embedded kernel routine to monitor the software buffer usage. The second part is an overflow paging mechanism. The third part is a flow-control message handler that interacts with the local scheduler.

5.2.1 The Monitor Routine

The monitor routine is embedded in the user kernel's message handling module. It monitors the buffering of messages into the application's network buffer. A software

counter is used to keep track of how many pages are allocated to the message buffer. Every time a new page is allocated for the message buffer, the monitor increases the counter; each time a page is de-allocated, the monitor decreases the counter. When the counter reaches a pre-set threshold, the buffer is considered to be overflowed.

When the message buffer overflows, the monitor routine generates throttling messages to deschedule the sender processes. The monitor routine continues to monitor the message buffer. Once the total size of the message buffer recedes below the reset threshold, the monitor routine sends reactivation messages to reschedule the sender processes.

The flow-control messages (the throttling messages and the reactivation messages) are transmitted via the second network using the "re-transmit on failure" messaging style described in Chapter 3. This ensures the flow-control messages will be delivered to their destinations (the offending sender processes in the overflowed application). The throttling message causes the scheduler to deschedule the sender process and to correct the overflow condition. The reactivation message causes the scheduler to reschedule the sender process.

5.2.2 The Overflow Paging Mechanism

Under the Fugu strategy, it is possible that a node runs out of memory while buffering the overflowed messages. In this situation, the operating system needs to page out part of the message queue to another node.

The overflow paging mechanism is embedded in the user kernel's virtual memory management module. When the virtual memory management module detects that there is no free page left, it invokes the overflow paging routine. The node that initiates the overflow paging (the requesting node) first sends a request to set up overflow paging. The heuristic for selecting a paging candidate is to pick one of the senders in the overflowed application first, if all the senders are full then it will pick a node with an external I/O device attached. At the receiving candidate node, the request message causes the page allocation handler to look for a free page. If a free page is found, the handler allocates the page for overflow paging, and sends a reply

back to the requesting node with a pointer to this page. If no free-page is found and the node does not have an external I/O device attached, the handler forwards the request to another node. If no free-page is found but there is an external I/O device attached to the node, then the page allocation handler pages out one of its used pages to the external device to create room for overflow paging. (We assume this is very infrequent.) Once the requesting node receives the pointer to the allocated page, it begins the actual transfer of the page. At the sender end, the overflow paging routine breaks the page down into small blocks. Then it sends the blocks out one at a time through the second network. At the receiver end, the first data message causes a receiving thread to start. The receiver thread examines the header of subsequent messages and places the data in the page until the page is filled. When the page transfer is completed, the overflow paging routine returns the page to the system's free-page list. Then, the virtual memory manager can reuse the page.

5.2.3 The Flow-Control Handler

The flow-control handler is a system level message handler. It is invoked when a flow-control message arrives from the second network. There are two types of flow-control messages. Depending on the type of the flow-control message arrived, the handler either deschedules or reschedules a user process.

5.3 The Second Network

As a separate independent system network, the second network provides a guaranteed communication channel to the operating system. In the Fugu anti-deadlock strategy, it is used for sending flow-control messages and for overflow paging under critical conditions.

The second network uses a two-phase (describe and launch) interface, similar to the main network. The hardware resources such as the input/output buffers, the machine info register and the status/command registers are memory mapped into the processor's address space. The processor can access these resources through special

load and store instructions.

The second network controller (SNC) provides a high degree of protection. It ensures that messages are transmitted atomically, and that the delivery of each message is acknowledged. It is designed and integrated as part of the Fugu User Communication Unit (UCU). It shares the processor's cache memory bus with other co-processors (the Communication and Memory Management Unit and the Floating Point Unit) in the system. See figure 5-2 for the relationship of the UCU with other core chips in the Fugu architecture.

5.3.1 An Overview of the Second Network's Operation

The high level view of the second network is a memory window in each processor that maps into other processors' memory spaces. Messages are being transmitted from processor to processor by writing to and reading from the special memory windows. Accesses to the special memory windows are performed via "colored" load and store instructions (*lda* and *sta*) in the Alewife Assembly Instruction Set. (See Appendix A for details of using these special loads and stores.)

In a lower level view of the secondary network, data is transported among processors in messages. A Fugu's second network message consists of one to eight 32-bit words. The first word of the message must be the header. The format of a message is specified in section 5.3.3.

Processors communicate with each other in this network via messages. Before sending a message, the processor first composes the message and stores it in the second network's output message buffer. When it is finished with composing, the processor writes a "send" command to the Status/Command register. This action triggers the hardware controller to send the message. The hardware controller takes over from this point. It first arbitrates for the network and then transmits the message to the destination processor. Once the processor commits the second network controller to send a message, the transmission cannot be interrupted. This provides the atomicity of a message transmission. The processor can either poll the Status/Command register to monitor the sending process of its message, or it can wait for the hardware

controller to interrupt when the message transmission is completed. At the destination end, the hardware controller accepts an incoming message only when its receiver is enabled. It is possible for a controller to refuse messages for a finite period of time. This is done to protect the input message buffer from being accidentally overwritten before the processor has a chance to examine its content (See the protection issues in Section 5.3.10). When a reception is completed, the receiving controller sends an acknowledgment (ACK) back to the sender. The receiver also causes an asynchronous interrupt to the local processor. (Polling from the receiving side is also possible. In this case, the processor just continuously monitors the Status/Command register.) When interrupted, the interrupt handler pulls the message off the hardware input message buffer and processes the message. In the case, the receiver does not accept a message, it will send a negative acknowledgment (NACK) back to the sender.

5.3.2 The Programmer's Interface to the Second Network

The programmer's interface to the second network is through a special two-phase process - *Describe* and *Launch*. In this process, a message is first described in the second network's hardware output buffer. When the message composition is completed, the message is launched into the second network by a launch command. This is the same interface used in the main network for transmitting regular messages. The advantage of this interface is that it allows a higher priority task/process to interrupt another process' message composition phase and to transmit a more urgent message. The routine that interrupts the message composition is responsible for saving and restoring the original data in the output message buffer.

A message consists of one to eight 32-bit words. The first word is the header, which contains the message type, the destination and other information for the message. Next section describes the second network message format in detail.

5.3.3 The Message Format

An anatomic view of the message in the secondary network is presented here. A message in the second network consists of 1) a 32-bit header word and 2) zero to seven 32-bit data word(s) (limited only by hardware resources). Figure 5-3 provides a detailed view of a second network message.

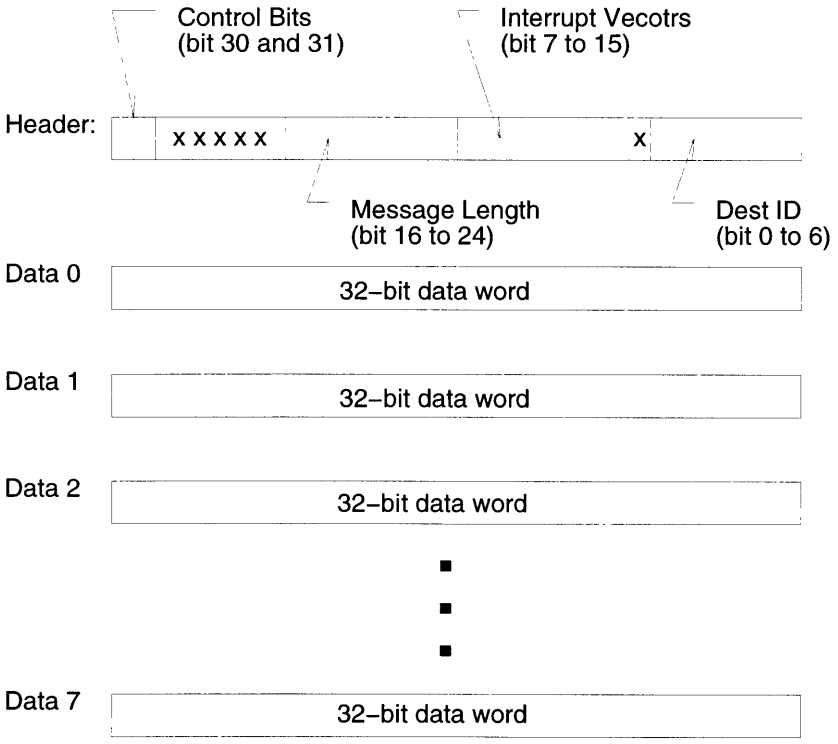


Figure 5-3: An anatomic view of a message on the secondary network

5.3.4 The Header

The header consists of four fields: the control bits, the length of the message, the interrupt vector and the destination ID. Each of these fields is described in details here:

Control Bits: The control bits field (bit 30 and bit 31 of the header) is used to distinguish the different types of messages. Classifying messages allows fast dispatching to system routines in the second network message handler. This saves the time

to walk the interrupt vector through a handler table. The current implementation supports four types of messages. Type 0 is reserved for the vectoring service. (See the use of the interrupt vector field below.)

Length: The length field (bit 16 to bit 24 of the header) identifies the length of the message in words. The current implementation uses only three bits (bit 16, 17 and 18) for the length field because the hardware controller only has room for eight 32-bit words in both the input/output message buffers. This field can be expanded in the future when more hardware buffers are available. The expansion would be particularly useful for improving the overflow paging performance. (See analysis in Chapter 4 Section 4.4.)

Interrupt Vector: Each type 0 message carries an 8-bit interrupt vector. The interrupt vector field (bit 7 to bit 15 of the handler) is used to identify which system handler to call in the interrupt routine. For example, different interrupt vectors can cause the processor to reset itself, to terminate a process or to load the message into certain memory locations, etc. This field is used for transmitting an 8-bit data in other types of messages.

Destination ID: The destination ID field (bit 0 to bit 6 of the handler) is a 7-bit ID used to identify one of the 128 nodes on the second network. Currently implementation only allows messages to be sent from one node to another. (It is possible to reserve one ID or to use bit 7 to implement a broadcasting message.)

5.3.5 Access to the Second Network Resources

Since the second network is a system network, access of the second network resources is only limited to the operating system kernel routines. For each access, the UCU checks the supervisor bit in the control lines emitted by Sparcle. It causes a protection violation trap if a user instruction tries to access the second network resources.

This section describes the network resources that are visible to the system pro-

grammer. A summary of available resources is provided in Table 5.1. Following the table, each resource is described individually below.

Name of Resource	Type	Size	ASI
Output Message Buffer	Read/Write	32-bit wide 8-word deep	0x58
Input Message Buffer	Read-Only	32-bit wide 8-word deep	0x59
Machine Info Register	Read/Write	16-bit wide	0x5A
Status/Command Register	Read/Write	16-bit wide	0x5B

Table 5.1: Summary of the visible resource to the programmer

Output Message Buffer (OMB): The Output Message Buffer (OMB) stores the message to be sent. It is a 32-bit wide 8-word deep Read/Write buffer. The first word must be a header. The second to the eighth words are data words. The number of words in the message must be recorded in the length field in the header. (See section 5.3.4 on message header format for more information.)

- The OMB is accessed through the 32-bit Sparcle databus using the special load and store instructions (*lda* and *sta*) with ASI 0x58.
- The OMB is readable and writable. By allowing the OMB to be readable, it is possible to interrupt message composition and to send an urgent message before resuming the composition phase.
- The OMB is addressed by bits 2-5 in the address field.

(See Appendix A for detailed accessing information.)

Input Message Buffer (IMB): The Input Message Buffer (IMB) is a 32-bit wide 8-word deep buffer that stores the incoming messages received from the second network. The buffer is addressed by bit 2 to bit 5 in the address field.

- The IMB is accessed using the 32-bit Sparcle databus using the special load and store instructions (*lda* and *sta*) with ASI 0x59.
- The IMB is only readable, because there is no need to write to this buffer.

- The hardware controller protects the IMB from accidentally being overwritten by another incoming message. It disables the receiver after it receives each message. The receiver is disabled (therefore, the IMB will not be overwritten) until the processor enables the receiver again. Writing an enable mask to the Status/Command register enables the receiver.

(See Appendix A for detailed accessing information.)

Machine Info Register (MIR): This register contains machine information that is necessary for the correct operation of the second network hardware controller. The MIR records two pieces of critical system information: the node ID and the machine size. The boot level software is responsible for writing this information in the MIR during the boot phase. The format of the MIR is described in figure 5-4.

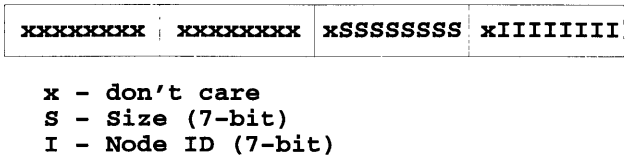


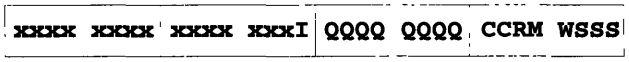
Figure 5-4: Machine Info Register

- This register is accessed using *lda* and *sta* with ASI 0x5A.
- This register must be written in the boot phase of the machine.

Status/Command Register (SCR): The Status/Command Register is the main communication interface between the processor and the second network controller. Reading the SCR provides the processor the internal status of the second network controller. Writing to the SCR sends a command to the second network controller. The Interface structure is shown in figure 5-5 below.

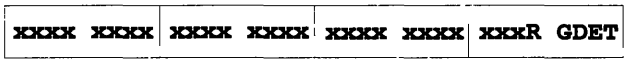
- The SCR is accessed using *lda* and *sta* with ASI 0x5B.
- The hardware controller returns the status of the controller when it is read. (Figure 5-5 shows how the status is organized when reported by the hardware controller.)

Status/Command Register Output (from Read Interface)



x - don't care
I - Interrupt Arrived
Q - Interrupt Vector (8-bit)
C - Control Bits (2-bit)
R - Ready to receive
M - Message for this node
W - Waiting to send
S - SNC state (3-bit)

Status/Command Register Input (from Write Interface)



x - don't care
R - Reset the Network
G - Generate a Token
D - Disable Receiver Flag
E - Enable Receiver Flag
T - Trigger Flag

Figure 5-5: Status/Command Register

- Commands are sent to the hardware controller by writing to the SCR with a specific mask. (Figure 5-5 shows the masks available for generating commands to the hardware controller.) Care should be taken to ensure that only one bit is written to the Status/Command register at a time. Writing multiple bits in the Status/Command register triggers multiple actions in the Second Network Controller and causes unknown effects.

5.3.6 The Second Network Hardware

This section describes the hardware of the second network. The second network hardware consists of the physical layout of the network (topology) and the design of a hardware controller. The network topology and the controller design are deeply influenced by the simulation conclusion that the second network latency and bandwidth requirements are very minimal. (See analysis in Chapter 4 for the details that lead to this conclusion.)

The Topology of the Second Network: A token ring topology is chosen for the second network. It is chosen mainly because of its ease of design and layout. For

a small machine configuration, the transmission latency of a token ring network is low enough to meet the latency requirement set forth in the analytical study. (See Section 4.3 for details.) A special layout is adopted to reduce the clock skew between neighboring nodes in the second network. Figure 5-6 illustrates the special layout of the Fugu's second network. This special topology bounds the maximum clock skew between two neighboring nodes to at most the skew for two hops.

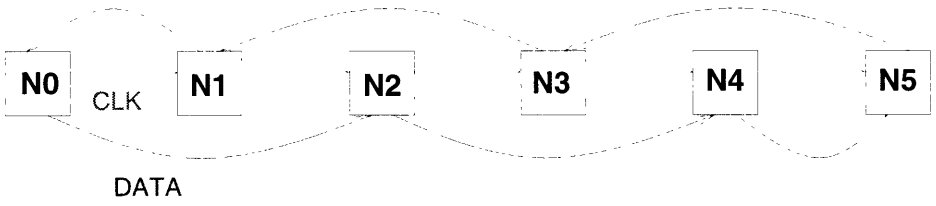


Figure 5-6: Topology for Secondary Network

The Second Network Hardware Controller: The second network hardware controller (SNC) provides the logic for the physical layer of the second network. The SNC manages the token and provides the sequential logic and datapaths to transport bits among the nodes in the second network. This section provides a brief description of the SNC hardware structure. For the detailed implementation information, see the Fugu Second Network Controller Design Document [36].

The SNC consists of five modules: the Main Control Unit, the Processor Interface Unit, the Sending Unit, the Receiving Unit and the Data Module. (Figure 5-7 shows the connectivity between these modules.)

Main Control Unit: The main control unit is responsible for providing the various control signals to the different modules in the SNC. It also manages the output to the second network. It consists of two finite state machines. One finite state machine governs the major activities within the SNC. The major activities are: (1) manage token passing, (2) send message, (3) receive message, (4) check manage ID, (5) keep track of message send and receive buffer resource and (6) manage network output. The other finite state machine controls the output of the SNC.

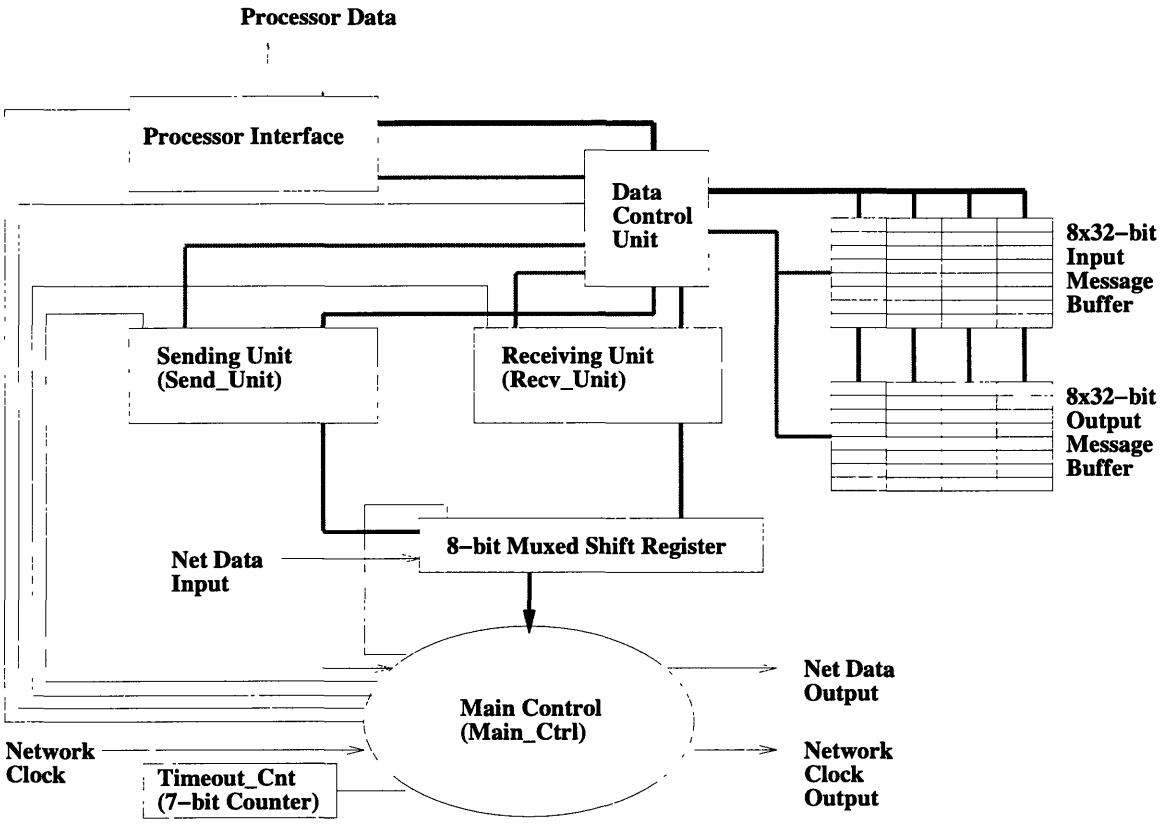


Figure 5-7: An overview of the Second Network Controller hardware

The network data input is latched at the positive edge of the clock. The data is used in the main finite state machine to determine the next state of the controller. An output is generated and latched into the output register at the negative edge. This design ensures that there is half a clock cycle tolerance against skew between two neighboring nodes.

Processor Interface Unit: The processor interface unit is integrated with the processor interface unit of the UCU. It handles all the communications between the SNC and the Sparcle processor. The processor communicates commands and data to the SNC via "colored" load and store operations. (See Appendix A for details of using these special instructions.) The SNC communicates to the Sparcle processor via the external interrupt line (EXTINT). Table 5.2 summarizes the processor interface and the corresponding signals.

Signal Name	I/O	Size	Note
Read	I	1	Sync
Write_Enable	I	1	Sync, Active Low
ASIs	I	8	Sync
Addr	I	3	Sync
Data	I/O	32	Sync
IRQ	O	1	Async

Unless specified otherwise, all signals are assumed to be active high.

Table 5.2: A Summary of the Processor Interface Signals

Sending Unit: The sending unit handles the task of sending messages in the SNC. When triggered, it reads the data from the Output Message Buffer (OMB) and sends the bits out to the network sequentially. Depending on the implementation, the width of the datapath is arbitrary. (Our current implementation selects a serial network.) In any case, the sending unit always starts from word 0 (the header) of the OMB and continues until the last word of the message is sent. The sending unit examines the header to find out the length of the message.

Receiving Unit: The receiving unit handles the task of receiving messages in the SNC. It is controlled and triggered by the main control unit. When triggered, the receiving unit latches data from the network and puts it into the input message buffer (IMB) every eight cycles (the receiving unit operates at the byte level). When a message arrives at the SNC, the main controller activates the receiving unit if the receiver's enable bit is set. The main control unit then examines the header to determine whether the message is intended for this controller. If it is, the main controller allows the receiving unit to continue receiving the rest of the message. If it is not, the main control unit stops the receiving unit. When the receiving unit completes the reception of a message, it signals the main control unit.

Data Module: The function of the data module is to redirect the read/write requests to the message buffers. There are two message buffers in the SNC: (1) the Input Message Buffer (IMB) and (2) the Output Message Buffer (OMB). Both the IMB and the OMB consist of eight 32-bit words. The IMB is written by the receiving

unit and is read by the processor interface. The OMB can both be read or written to by the processor interface, but it can only be read by the sending unit. Interlocking mechanism and software handles are provided for data security. More sophisticated locks can be implemented. Contentions between input/output buffer accesses and the SNC's solution to resolve this are described in the interlocking and protection section (Section 5.3.10).

5.3.7 The Fairness Issue for the Second Network

In the Fugu second network, the sender always releases the token after each transmission. This policy ensures fairness in the network usage. It allows other controllers down the ring to send messages.

5.3.8 The Atomicity of a Second Network Message

Messages are transmitted atomically. Once the processor commits the second network hardware controller (SNC) to send a message, the processor cannot interrupt the process before the transmission is completed. However, before the processor issues the "send" or the "launch" command, it is possible for the processor to back out from composing the current message to give way for sending a higher priority message. (See section 5.3.10)

5.3.9 The Acknowledgment of the Second Network Messages

Because of the token ring topology, there is no transmission overhead to send an acknowledgment for each message. In our design, for each message sent there is always a positive acknowledgment (ACK) or a negative acknowledgment (NACK) generated. It informs the sender the status of the message transmitted. The second network controller at the sender node will then inform the sending routine whether the transmission was successful or failed. The acknowledgments are necessary to

implement the "re-transmit on failure" messaging scheme described in Chapter 3 to ensure all second network messages will eventually be delivered.

5.3.10 The Protection Issue for the Second Network Messages

The second network is a system network. Its resources are only accessible by the operating system's kernel. This eliminates the possibility that an user application misuses or deadlocks the second network.

Although the second network is protected from the user applications, the latency in the processor interface cannot guarantee the processor can retrieve the data from the hardware buffer before a subsequent incoming message overwrites it. To circumvent this defect, the hardware controller automatically disables its receiver after it has received a message. This protects the content in the Input Message Buffer (IMB) from subsequent incoming messages. The receiver is disabled until the processor writes an enable command into the Status/Command register. See Appendix A for details of accessing the Status/Command register. Contentions between the processor's accesses and the controller's accesses to the IMB are resolved by an internal finite state machine which arbitrates simultaneous accesses.

To prevent the content in the Output Message Buffer (OMB) from being accidentally changed after one process has committed the hardware controller to send the message, we use a software semaphore to control the access to the OMB. Any routine which wishes to launch a message into the second network must first acquire the semaphore for accessing the output message buffer. Once the routine acquires the semaphore, it can commit the message to the controller. During the sending phase, the sending routine continues to hold on to the semaphore. This ensures other routines cannot alter the OMB during a transmission. The sender routine must release the semaphore when the message transmission completes. In our implementation, this semaphore is not checked out during the message composition phase. This allows higher priority tasks to interrupt the message composition and to send another

message through the second network. The routine which interrupts other process's composition phase is responsible for saving and restoring the content in the OMB before and after sending its message.

5.3.11 The Actual Design Process and the Final Product

The Second Network Controller (SNC) is designed using a top down methodology. Preliminary analyses and the specifications for the second network requirements were done in May 1995. Behavioral simulations were performed on the conceptual model of the second network controller in the summer of 1995. The behavioral Verilog code was then rewritten into the structural Verilog code. The Second Network Controller is made up of four modules in the structural code: the main control unit, the data module, the shift register module and the processor interface module. The four modules total about 1300 lines of code. (The actual code is not included here to conserve space). The structural modules were readied in September 1995. The structural code was then tested to ensure it behaves identically to the behavioral code. When that was done, the structural code was then fed into the Synopsys Verilog HDL compiler to generate gate level modules. The gate level modules were analyzed using Cadence's Design Analyzer for timing analysis. The timing analysis revealed several weaknesses in the controller's design. As a result, the structural Verilog code was rewritten to improve the timing on the critical paths. The optimized structural code was recompiled into gate level modules and the modules were fed into the Xilinx FPGA compiler to produce a netlist that can be down-loaded on to the Xilinx FPGAs. The final design requires 104 CLBs and 60 flip-flops. The Xilinx timing analysis tool, XDelay, reported a maximum frequency of 10.5MHz for the controller design running on a simulated Xilinx 4005-5 part. At the same time, a test module was developed and compiled to an FPGA netlist. When integrating the Second Network Controller and the test module together in one module, we produced a stand-alone test module that can be used for functional testing as well as performance analysis prior to the final integration with the rest of the Fugu's User Communication Unit. A 4-node network system consisted of four Xilinx 4005-5 FPGAs was built to run the functional tests.

The first functional Second Network Controller design was completed in November 1995. The actual maximum achievable clock speed in the lab is 15 MHz for the Second Network Controller which is about 50% faster than XDelay has predicted. This agrees with our experience in using Xilinx FPGA tools. The completed hardware design was then turned over to Jon Michelson to be integrated with the UCU chip. See [20] for the actual implementation of the UCU Chip.

Chapter 6

Simulations and Results

The previous chapter has described an implementation of the Fugu anti-deadlock strategy. The implementation is based on the analysis in Chapter 4. This chapter describes the simulations used to validate the analysis as well as to study the performance of the Fugu strategy.

The first section describes the architectural simulator we used for the simulations. The second section presents the study on the impact of the second network latency on the system performance. It also describes the relationship between the second network latency and some system parameters. At the end, it provides a preliminary conclusion to justify the Fugu implementation. The third section evaluates the effects of different design parameters on the network performance. The last section compares the Fugu strategy with other common anti-deadlock strategies.

6.1 Talisman2 Architectural Simulator

Using an architectural simulator, we can explore the different design dimensions without having to build the actual hardware. The goals of the simulation are to validate the analysis for the Fugu strategy and to study the performance of the Fugu anti-deadlock strategy.

Talisman2, or "T2", is the architectural simulator for the Fugu multiprocessor. The basic simulator structure was developed by Robert Bedichek using code gener-

ation and execution environment ideas from Talisman, Shade and SimOS. Its goal is to provide a software development platform for the Fugu multiprocessor prior to hardware availability.

T2 simulates the target system in an instruction-centric manner. It translates target instructions to host instructions and executes them a block at a time to allow the host to run efficiently. Moreover, these instruction blocks are cached, so that translation efforts are amortized over many executions of the same block. This simulation method is extremely efficient for multiprocessor because most parallel programs execute the same code on all processors.

Multiple processors are simulated on a single host processor by switching the simulation environment from one processor to another processor when the simulation time on the currently simulating processor exceeds a preset quantum. To facilitate the switching between processors, T2 keeps all processor specific data in a data structure that is addressed by a single pointer. Therefore, switching from one processor to another is only a matter of switching this pointer. The granularity of this multiplexing can be varied by the user to trade off simulation accuracy with simulation efficiency. That is, the user can choose fine grain execution control by simulating short instruction blocks on each processor, or the user can increase performance by allowing large translated blocks to be executed on each processor before switching.

To validate the Fugu anti-deadlock strategy, the basic T2 simulator is modified to include a new second network module and a more detailed network latency model. The second network module includes software switches that can change the topology of the second network without rebuilding the simulation. Currently, a bus base model and a token ring model are implemented. The bandwidth of the network as well as the latency can be changed by the user to study the effects of these parameters. The main network latency model allows the user to specify the size of the hardware input and output buffers. It can also specify the transmission time between two nodes. Two algorithms are implemented to calculate the transmission time. The transmission time can be calculated on a per hop basis or by using a fixed constant to model larger transmission latency. The latency model with network contention is adapted from the

network model by Agarwal in [3]. Furthermore, a network statistic gathering module is added to collect messages transmission statistics as well as the wait time statistic to analysis the network performance.

6.2 Effect of the Second Network's per Message Latency

From Chapter 4, we learn that the Fugu strategy can tolerate a relatively slow second network without degrading the main network performance. To validate this analysis, we simulate the second network latency over a large range. Two sets of experiments are performed to study the effect on the network performance and the software queue size individually. Both sets of experiments use the same benchmark program that sends 3000 messages (24 byte each) at an interval of 100 cycles to simulate a message passing application with heavy communications.

6.2.1 Impact on the Main Network

The first set of experiments studies the impact of using the anti-deadlock strategy on the main network. The impact is measured by the change in the average network utilization and the change in the average wait time for a random message to gain access to the congested network during the transmission of the 3000 test messages. The T2 simulator is programmed to collect the network statistics in a per message basis so that the channel utilization can be calculated at the end of each run. It also records the wait time a random messages would suffer if it is injected in the network at each cycle during the test. At the end of each run, T2 calculates the average wait time a random message would have suffer during the run. Data is taken from experiments with second network latency varying from 5 cycles to 50000 cycles. (The large latency can also be interpreted as sending messages to many nodes in a large system.) We also vary the handler latency from 80 cycles to 3000 cycles to see if the handler latency changes the second network latency effect. The results are shown in

figure 6-1 and 6-2.

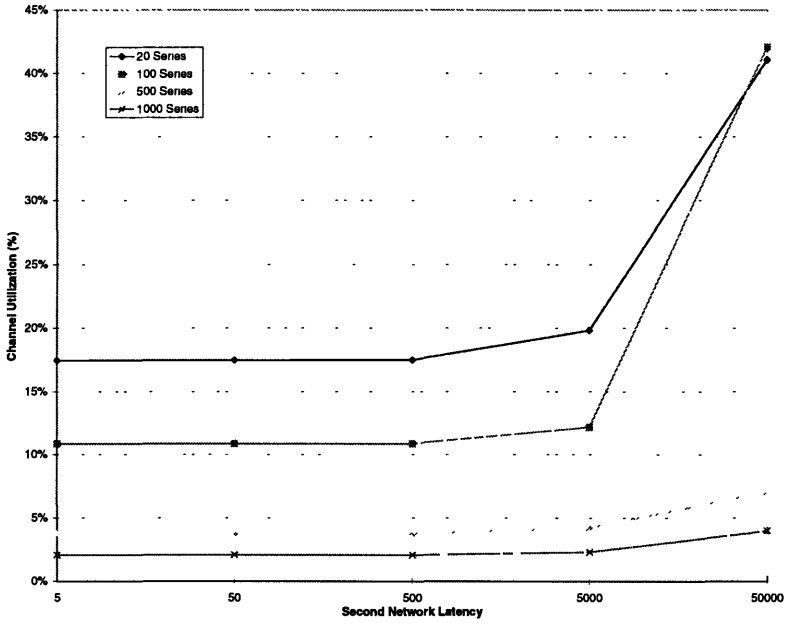


Figure 6-1: The effect of varying the second network latency on the main network utilization.

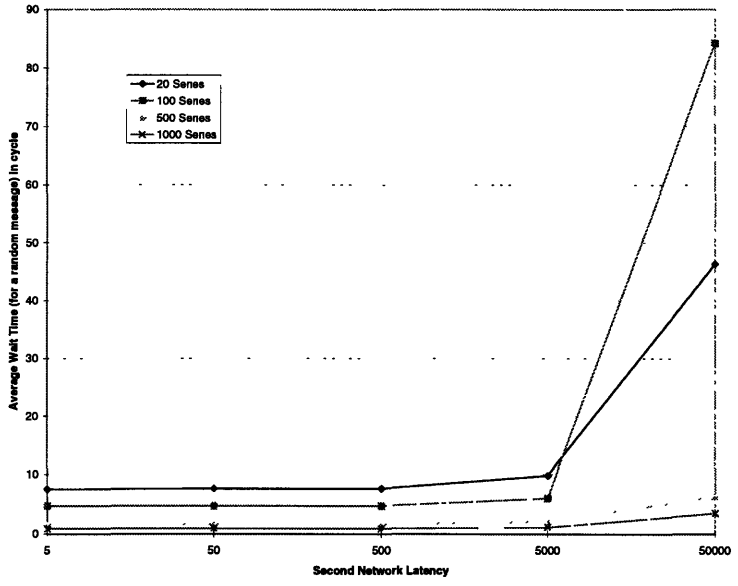


Figure 6-2: The effect of varying the second network latency on the average wait time a message experiences.

Analysis:

The simulations show that the network utilization and the average wait time remains fairly low for the second network latencies ranging from 5 to 5000 cycles. In this range of second network latencies, less than 3 % changes in the channel utilization and less than 2 cycles changes in the average wait time are observed. When the second network latency increases from 5000 to 50000 cycles, there is a moderate increase in the network utilization (+ 20 %) and in the average wait time (+ 75 cycles).

6.2.2 Relationship with the Overflow Buffer Size

The second set of experiments studies the impact of the second network latency on the overflow buffer usage. This set of experiments assumes there is no upper limit on the overflow buffer size. The second network latency is varied from 5 to 50000 cycles and the message handler latency (the inverse of the rate at which message is being handled, $\frac{1}{r_h}$) is varied from 80 to 3000 cycles. In Table 6.1, we summarize the overflow buffer usages for four different handlers over the range of second network latencies. In figure 6-3, we plot the overflow buffer size versus the second network latency when the handler latency is equal to 80 cycles.

	Second Network Latency				
Message Handler Latency	5	50	500	5000	50000
80	5	5	5	5	11
320	5	5	5	5	11
1520	5	5	5	5	11
3020	5	5	5	5	11

Table 6.1: Summary of the overflow buffer size required at different second network latencies

Analysis:

From Table 6.1, we realize that the message handler latency does not change the effect of the second network latency. This agrees with our analytical model, because once the rate at which the handler can handle messages (r_h) drops below a certain

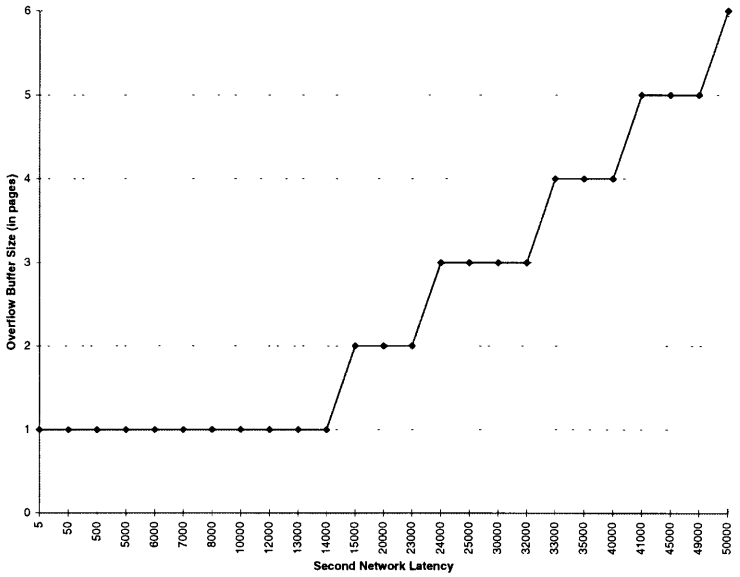


Figure 6-3: The effect of second network latency on the overflow buffer size

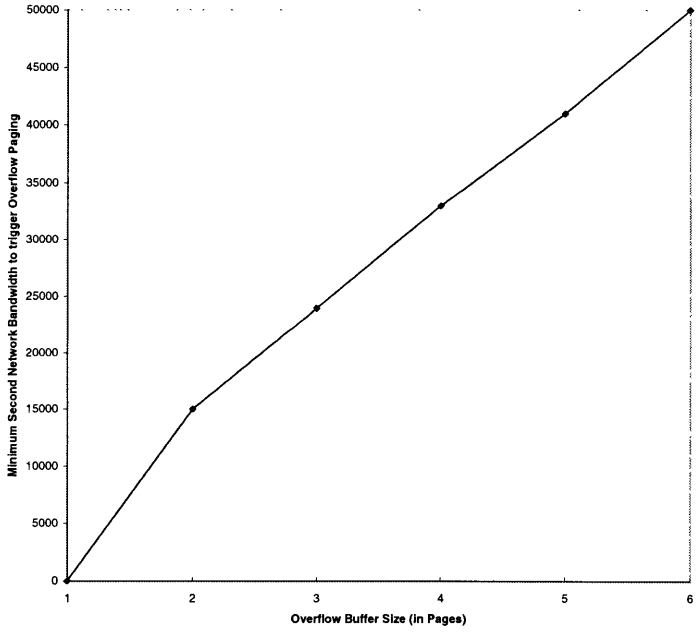


Figure 6-4: The second network latency which causes overflow at different limits of the overflow buffer size

threshold it does not affect the rate at which the software buffers are filled (r_{fill}) (See equation 4.2 and equation 4.4 in Chapter 4).

From figure 6-3, we observe that the overflow buffer size increases as the second network latency increases. By organizing the data in an inverse way, we can see clearly how the overflow buffer size sets the second network latency requirement. In figure 6-4, we plot the minimum second network latency that causes overflow paging at different sizes of the overflow buffer.

From this plot (figure 6-4), we can observe two things. First, the overflow buffer and the second network latency requirement have a linear relationship. This is not very clear from figure 6-3, but it is much more apparent in figure 6-4. This agrees with the analysis in Chapter 4. For the experimental setup (the default message buffer size is 5 pages), increasing the overflow buffer size by one page increases the second network latency tolerance by about 9000 cycles. Second, we see that an overflow buffer size of one to two pages is enough to keep the system from overflow paging in nearly all possible cases.

6.2.3 A Preliminary Conclusion

From the analytical study and the simulation experiments, we conclude that overflow paging can easily be avoided. For a moderate limit on the maximum overflow buffer size (1 to 2 pages) and on the application handler latency (< 1500 *cycles*), the second network latency requirement is minimal.

6.3 Effect of Different Design Parameters on the Overflow Paging Performance

The analysis in Chapter 4 points out that an increase in the second network bandwidth or a decrease in the hardware overhead to hardware buffer size ratio ($\frac{T_{HW}}{K}$) has a positive effect on the overflow paging performance. The analysis also identifies that the effect of increasing bandwidth is more significant than that of decreasing the $\frac{T_{HW}}{K}$

ratio. This section studies the simulations we performed to validate this analysis.

From the latency simulations in the previous section (Section 6.2), we learn that three conditions are needed to trigger the overflow paging mechanism. These conditions are 1) a very large second network transmission latency, 2) a very large message handler, and 3) a very small overflow buffer size limit. To approximate these conditions in the simulations here, 1) the second network latency is set at 50000 *cycles*, 2) the message handler latency is set at 1500 *cycles*, 3) the software buffer threshold is set at 5 *pages*, 4) the overflow buffer size limit is set at 1 *page*, and 5) the overflow reset threshold is set at 1 *page*.

Three sets of experiments are carried out. Each set of experiments varies the second network bandwidth from 1 *bit/cycle* to 100 *bits/cycle*. A different network design is used in each set of experiments. The first set of experiments uses a network design that has a large hardware overhead (190 *cycles*) per message and a small hardware buffer (an 8-word buffer). Assuming the size of the main network message is 8-word. This design has a $\frac{T_{HW}}{K}$ ratio of 190 *cycles/message*, and it approximates the Fugu second network design. The second set of experiments uses a network design that has the same hardware overhead as the first design but doubles the size of the hardware buffer (from an 8-word buffer to a 16-word buffer). The second design is used to study the effect of doubling the second network buffer size. This design has a $\frac{T_{HW}}{K}$ ratio of 95 *cycles/message* which is exactly half of the first design. The third set of experiments uses a network design that has a low hardware per message overhead (10 *cycles*) but keeps the same small 8-word hardware buffer. This design has a $\frac{T_{HW}}{K}$ ratio of 10 *cycles/message* which is the lowest of all three designs. The third design is used to study the effect of using a 2D mesh design for the second network. Figure 6-5 shows the results of the simulations.

Analysis:

The simulations in the overflow paging performance show that the bandwidth parameter has a diminishing effect on the overflow paging performance as it is increased. Increasing bandwidth from 1 *bit/cycle* to 5 *bits/cycle* realizes over 60% of the possible

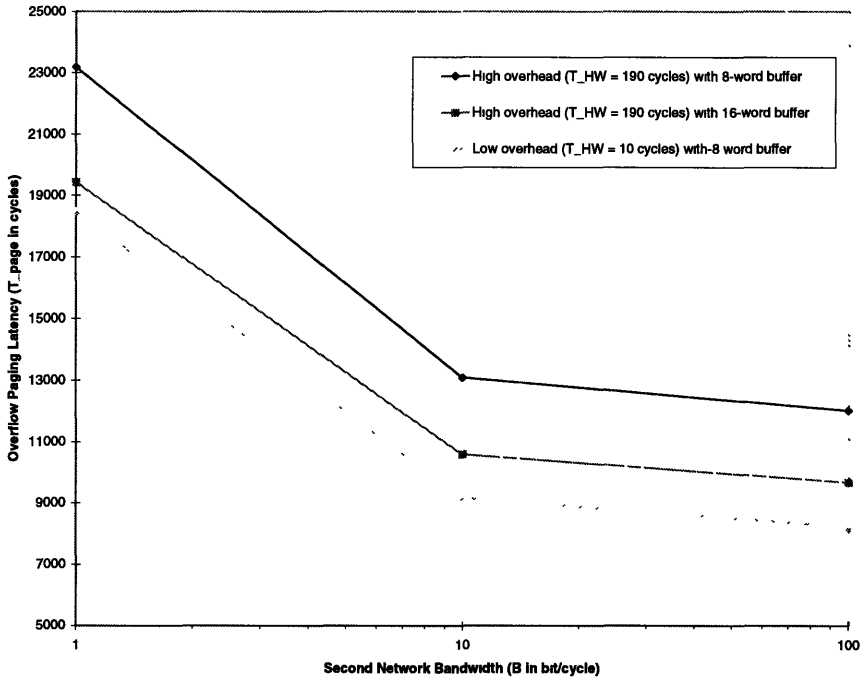


Figure 6-5: The effect of second network latency on the overflow paging performance.

improvement. This effect is observed in all three sets of experiments. In figure 6-5 we also observe that the reduction in the hardware overhead to hardware buffer size ratio improves the overflow paging performance as expected. Moreover, the figure shows that the bandwidth has a more significant impact on the performance than the overhead to buffer size ratio. All of these observations agree with the analysis in Chapter 4.

6.3.1 A Preliminary Conclusion

From the analytical analysis and the simulation experiments, we conclude that increasing the second network bandwidth or reducing the second network hardware overhead to hardware buffer size ratio improves the overflow paging performance. However, the performance effect diminishes as the second network bandwidth increases or the hardware overhead to buffer size ratio reduces. Both the analysis and the simulations agree that increasing the second network bandwidth has a more

significant effect than reducing the hardware overhead to buffer size ratio.

6.4 Comparison with other Common Anti-Deadlock Strategies

This section compares the performance of the Fugu strategy to other possible anti-deadlock strategies.

The implementations used in the comparison are: 1) a baseline model where only hardware flow control is used, 2) a software window-based flow-control model using the reservation strategy described in Section 2.1.3, and 3) a static flow-control implementation which is similar to the Fugu implementation but it differs from the Fugu strategy that in the static strategy the deschedule time is pre-determined and does not vary with the application's message handler latency.

We have implemented these strategies on the T2 simulator and compared their relative performances using the same benchmark we used in the previous simulation experiments. The benchmark sends 3000 messages (24 byte each) at an interval of 100 cycles to simulate a message passing application with heavy communications. The performance is measured by the average wait time for a random message to gain access to the congested network during the transmission of the 3000 test messages. The simulator records the wait time a random messages will suffer if it is injected in the network at each cycle during the test and computes an average at the end of each run.

The experiment is repeated for the message latency varying from 20 *cycles* to 6000 *cycles*. All models use a 512 *bytes* hardware buffer. The software window-based flow-control model provides a 1 *page* buffer to each sender. The static flow-control scheme and the Fugu scheme use an initial application message buffer of 5 *pages* and an overflow buffer of 1 *page*. The static flow-control model does not require an overflow reset threshold, but requires a predetermined deschedule time. In this experiment, we choose the deschedule time to be 1 *timeslice* which is about 500,000 *cycles*. The Fugu's strategy uses a 1 *page* overflow reset threshold. The results are plotted in

figure 6-6.

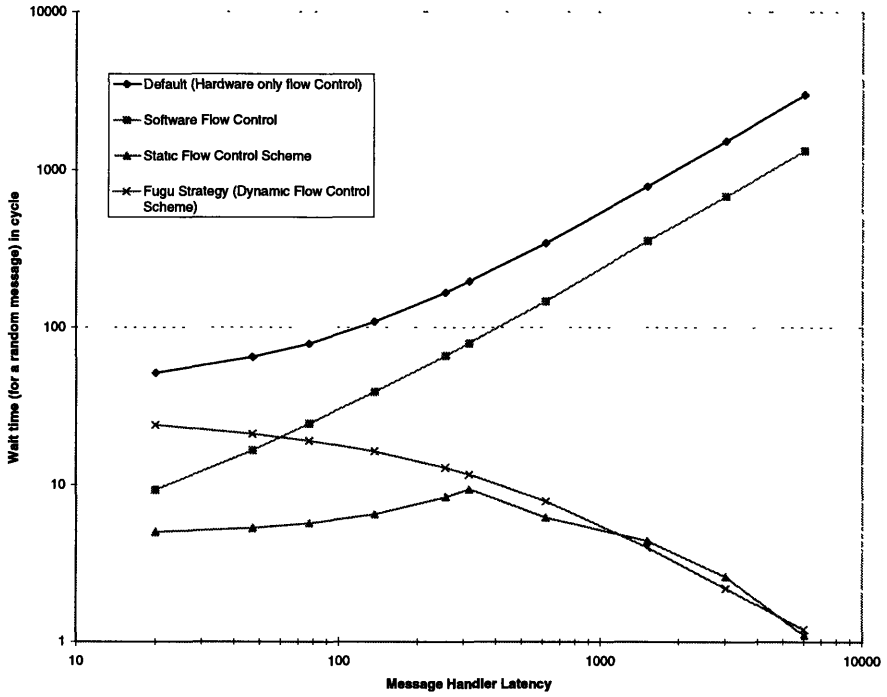


Figure 6-6: A Comparison between four anti-deadlock implementation

Analysis:

Comparison shows that all three anti-deadlock strategies perform better than the baseline case with no anti-deadlock strategy. Both the baseline model and the software window strategy display a upward trend with respect to increasing handler latency. This suggests that these two strategies do not perform well when the handler is very slow or when there is a deadlock in the network. This is not the case for the flow-control strategies. Both of the static flow-control strategy and the Fugu strategy (a dynamic flow-control strategy) display a downward trend with respect to increasing handler latency. This suggests that in the case of a very slow message handler (or even a real deadlock situation), both strategy manage to keep the network relatively free.

Comparing the static strategy with the Fugu strategy, the static strategy seems to

provide a better performance than the Fugu strategy. The average wait time under static flow-control strategy increases until it almost crosses the Fugu strategy. It then begins to drop with the Fugu strategy.

6.4.1 A Preliminary Conclusion

Analyzing the experimental data, the static strategy seems to provide the best performance of all strategies studied. However, the static flow-control strategy suffers from having a break down point at very high message handler latencies. The static scheme breaks down when the deschedule time is not long enough to let the application dequeue enough messages to reset the static scheme to the pre-triggered level. This causes the software buffer to constantly operates near the limit of the software buffers and causes the system to overflow page very often. Therefore, at high message handler latencies, the static strategy could possibly break down and the network performance will drastically decrease.

The dynamic strategy on the other hand does not have a break down point. This is because the offending senders are descheduled for a period of time that is proportional to the message handler time. When compared with the static strategy, the Fugu strategy (a dynamic flow-control one) is more desirable because its performance characteristics do not depend on applications. This provides a much better control on the overall system performance.

Chapter 7

Conclusion

In modern multiprocessor systems, interprocessor communication is extremely important. A desire to provide a communication service that guarantees forward progress and provides tolerance over misbehaving applications leads to the development of anti-deadlock strategies in multiprocessors.

This thesis presents an end-to-end anti-deadlock strategy that requires minimum architectural support and achieves good performance. Through analyzing common anti-deadlock strategies, we have identified the minimum features required to construct a deadlock-free system. These features include a deadlock-free network layer, a system timer and a reserve channel for each node to reach its backing store. This minimal set is cost effective because it reduces the system requirement to the absolute minimum. The strategy based on these features is deadlock-free because 1) no process can block the network for an extended period of time and 2) the system can off-load the data from the limited network resources to other system resources such as the main memory or the external I/O devices using the reserve channel.

We construct the Fugu strategy based on the minimal requirements, and we have added additional optimizations to improve performance. We choose to use a second network to provide the reserve channel in our implementation of the Fugu strategy. A second network is chosen also because it provides a way to perform efficient process flow-control. Under the Fugu strategy, when an application takes too long to handle its messages, the system timer expires and causes the operating system to buffer

subsequent messages. If an application exhausts its network resources in buffering messages, the operating system will allocate additional resources (an overflow buffer) for the application. At the same time, the operating system initiates a throttling process to deschedule the senders in the overflowed application to achieve a load-balancing effect on the network. If the system runs out of physical memory when allocating for overflow buffer space, the operating system uses the second network to page out part of the network buffer. Using the second network to perform the throttling process consumes a small amount of its bandwidth but increases the efficiency of the flow-control process. This drastically improves the performance of the Fugu strategy because the system scheduler can more effectively schedule processes that optimize the network usage.

Analyses and simulations in this thesis show that a simple network design is sufficient for a large multiprocessor system. We have tested an implementation of the Fugu strategy using a serial token ring network in a 32-node machine, and we find that the performance degradation is minimal. We also find that the Fugu strategy scales relatively well and performs better than some common anti-deadlock strategies. A functional prototype of the Fugu multiprocessor is under construction. It is expected to be completed in the summer of 1996.

This project benefits the architecture community by providing an accurate model of this new strategy to be used in future designs as well as insights into how this strategy achieves a good trade-off between cost and performance.

Appendix A

Accessing the Fugu Second Network Resources

Output Message Buffer (OMB)

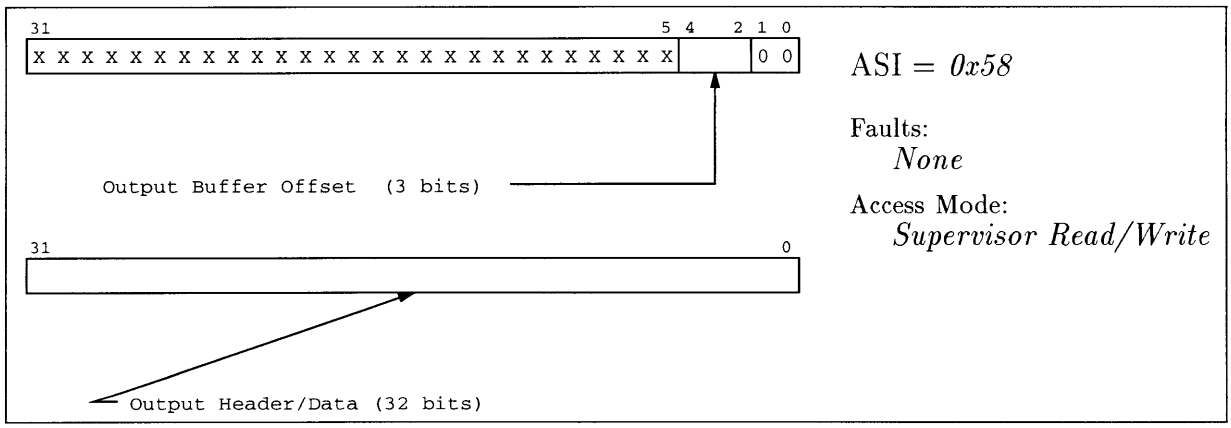


Figure A-1: Output Message Buffer

Description: This command is used to compose the outgoing message in an 8-word buffer. The first word of the message is intepreted as the header by the hardware.

Input Message Buffer (IMB)

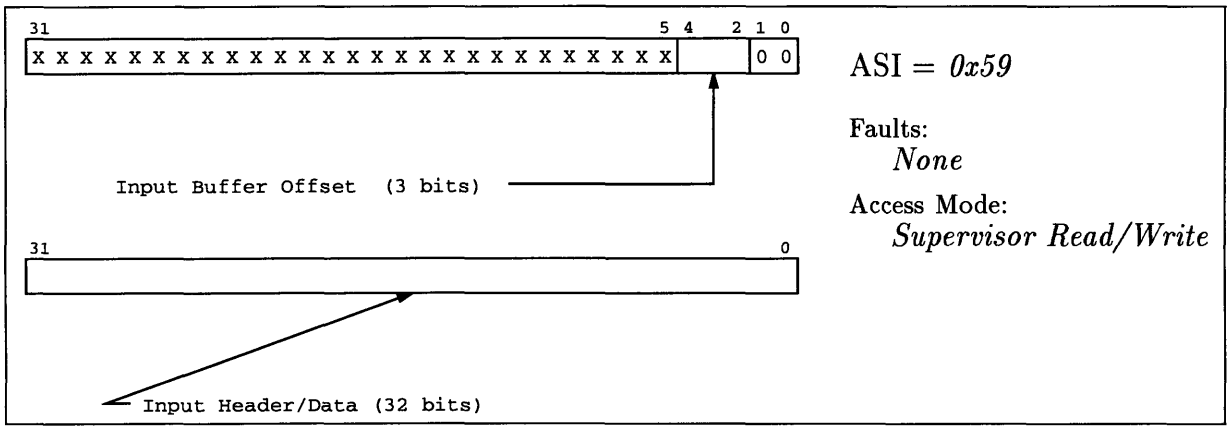


Figure A-2: Input Message Buffer

Description: This command is used to read the incoming message from its 8-word buffer. The first word of the message is the hardware header.

Machine Info Register (MIR)

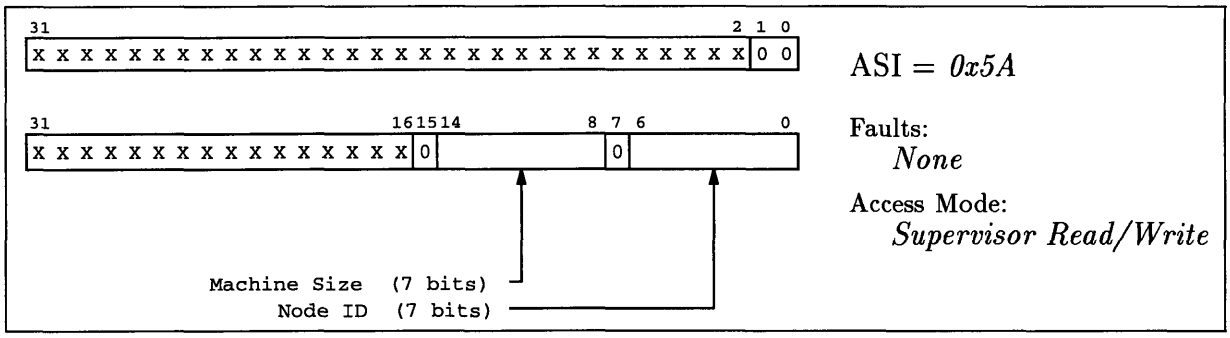


Figure A-3: Machine Info Register

Description: This register encodes the machine parameters for subsequent use by the second network. This register must be initialized before the second network may be used. The intention is that this register is written once at hard-boot time.

Machine Size The *Machine Size* field represents the total number of nodes in the second network.

Node ID The *Node ID* field represents this node's number in the network. This ID is used for identification in routing.

Machine Status Register

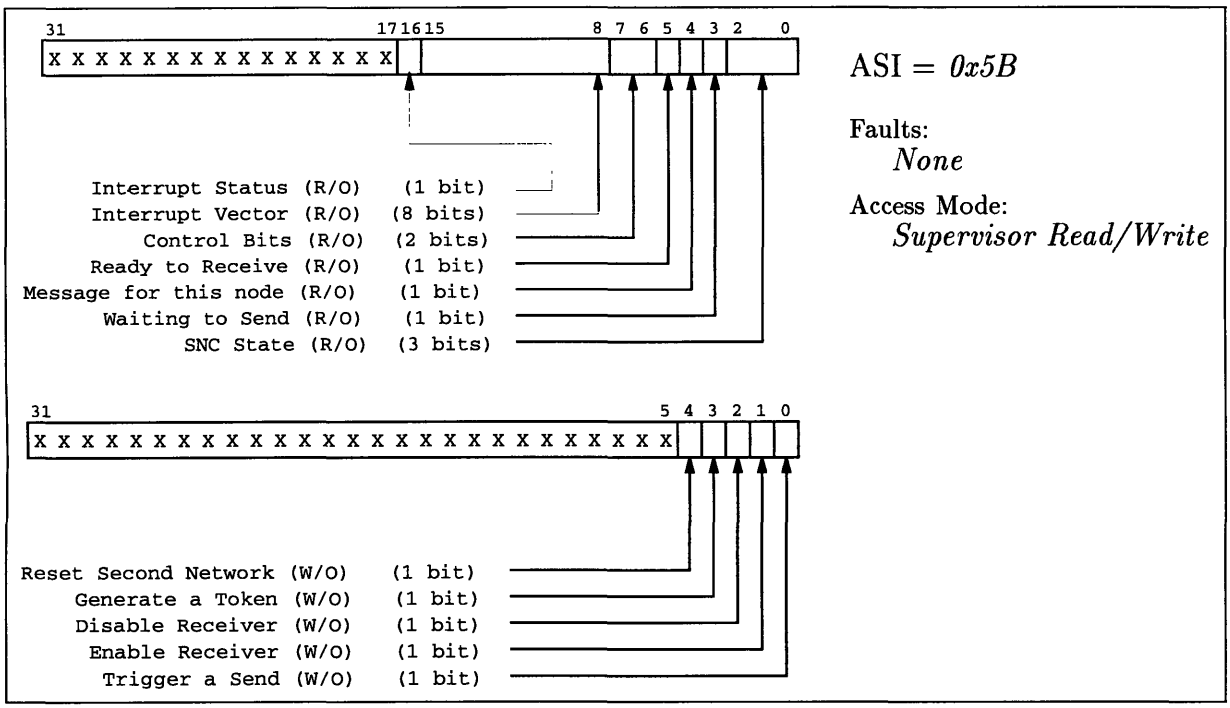


Figure A-4: Machine Status Register

Description: This register gives the current status of the network interface. Some of the fields are writable.

Bibliography

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin and D. Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. H. Lim, K. Mackenzie and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 2–13, June 1995.

[3] A. Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[4] D. Bertsekas and R. Gallager. *Data Networks.* Prentice Hall, second edition, 1992.

[5] Y. M. Boura and C. R. Das. Efficient Fully Adaptive Wormhole Routing in n-dimensional Meshes. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 589–596, 1994.

[6] D. C. Douglas et al. C. E. Leiserson, A. S. Abuhamdeh. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures.* ACM, 1992.

[7] A. A. Chien. A Cost and Speed Model for k-ary n-cube Wormhole Routers. In *Proceedings of the Hot Interconnects '93*, August 1993.

[8] D. Chiou, B. S. Ang, Arvind, M. J. Beckerle, A. Boughton, R. Greiner, J. E. Hicks and J. C. Hoe. StarT-NG: Delivering Seamless Parallel Computing. In *Proceeding of the EURO-PAR'95 Conference*, August 1992.

[9] W. J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, 1987.

[10] W. J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[11] W. J. Dally and H. Aoki. Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual-Channel. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):466–475, April 1993.

[12] W. J. Dally and C. J. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36:547–553, May 1987.

[13] J. Duato. On the Design of Deadlock-Free Adaptive Routing Algorithms for Multicomputers: Design Methologies. In *Proceedings of the Parallel Architectures and Languages*, pages 390–405, June 1991.

[14] D.E. Engler, M.F. Kaashoek, and J. O'Toole. The operating system as a secure programmable machine. In *Proceedings of the 6th ACM SIGOPS European Workshop*, Wadern, Germany, Sept. 1994.

[15] D.E. Engler, M.F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[16] C. J. Glass and L. M. Ni. The Turn Model for Adaptive Routing. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 278–287, 1992.

[17] *Intel Paragon(tm) Supercomputer Product Brochure.* Intel Corporation, 1994.

[18] V. Varavithya J. H. Upadhyay and P. Mohapatra. Efficient and balanced adaptive routing in two-dimensional meshes. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 112–121, January 1995.

[19] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[20] J. Michelson. Design and Optimization of Fugu's User Communication Unit. Master's thesis, Massachusetts Institute of Technology, June 1996.

[21] A. Agarwal K. Mackenzie, J. Kubiatowicz. UDM: User Direct Messaging for General-Purpose Multiprocessing. Submitted for publication., January 1996.

[22] K. Mackenzie, J. Kubiatowicz, A. Agarwal and F. Kaashoek. FUGU: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor. In *The MIT Laboratory for Computer Science Memo (LCS/TM/503)*, October 1994.

[23] J. H. Kim and A. A. Chien. An Evaluation of Planar-Adaptive Routing (PAR). In *Proceedings of the Fourth Symposium on Parallel and Distributed Processing*, December 1992.

[24] L. Kleinrock. Principles and lessons in packet communications. In *Proceedings of the IEEE*, pages 1320–1329, November 1978.

[25] J. Kubiatowicz. The Anatomy of a Message Send. In *MIT Student Workshop on VLSI and Parallel Systems*, page 26, July 1992. Abstract of future paper on message interface.

[26] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In

Proceedings 17th Annual International Symposium on Computer Architecture, pages 148–159, New York, June 1990.

[27] D. H. Linder and J. C. Harden. An adaptive and fault-tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Transactions on Computers*, C-40:178–186, January 1991.

[28] R. J. Littlefield. Characterizing and tuning communications performance for real applications. In *Proceedings of the First Intel DELTA Applications Workshop*, pages 179–190, February 1992. Proceedings also available as Caltech Technical Report CCSF-14-92.

[29] M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Two Virtual Memory Mapped Network Interface Designs. In *Proceedings of the Hot Interconnects II Symposium*, pages 134–142, Princeton, New Jersey, August 1994.

[30] M. D. Noakes, D. A. Wallach and W. J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceeding of the 20th International Symposium on Computer Architecture (ISCA '93)*, May 1993.

[31] P. Pierce. The NX/2 operating system. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, January 1988.

[32] R. J. Swan, S.H.Fuller and D. P. Siewiorek. Cm* - A modular, multi-microprocessor. In *Proceedings of 1977 National Computer Conference*, pages 637–644, Dallas,Texas, Jun 1977.

[33] S. Ramany and D. Eager. The Interaction between virtual channel flow control and adaptive routing in wormhole networks. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 136–145, July 1994.

[34] S. K. Reinhardt, J. R. Larus and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21nd Annual International Symposium on Computer Architecture (ISCA'94)*, April 1994.

[35] C. L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, December 1984.

[36] V. Lee. Fugu Second Network Controller Design Documentation. In *The MIT Laboratory for Computer Science Memo (Fugu Memo #3)*, October 1994.

[37] J. Duato Z. Liu and L. E. Thorelli. Grouping Virtual Channels for Deadlock-Free Adaptive Wormhole Routing. In *Proceedings of the 5th International PARLE Conference*, pages 254–265, June 1993.