A High-Speed Fault-Tolerant Interconnect Fabric for Large-Scale Multiprocessing

by

Robert Woods-Corwin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

© Robert Woods-Corwin, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by .....
John F. McKenna
Principal Member of Technical Staff, Charles Stark Draper Laboratory
Thesis Supervisor

Certified by .....
Thomas F. Knight
Senior Research Scientist, Artificial Intelligence Laboratory
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

This Page Intentionally Left Blank

A High-Speed Fault-Tolerant Interconnect Fabric for Large-Scale Multiprocessing

by

Robert Woods-Corwin

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis describes the design and synthesis of an updated routing block for a next-generation wave propagation limited fault-tolerant interconnect fabric for a large-scale shared-memory multiprocessor system. The design is based on the METRO multistage interconnection network, and is targeted at minimizing message latency. The design incorporates an efficient new tree-based allocation mechanism and an idempotent messaging protocol. A fat tree topology is the basis for the network. A Verilog implementation of the design is simulated and synthesized into physical hardware, running at speeds as high as 90MHz in an FPGA. Techniques are discussed to vastly improve performance in a potential future design using custom hardware. Further, two potential modifications to the network are considered. First, the performance effect of allocating dedicated physical wires to streamline the idempotent messaging protocol is analyzed. The modification increases the success rate of messages significantly, but the increased latency due to the space taken by the wires overwhelms the potential performance advantage. Second, a scheme for prioritizing messages is developed. This scheme improves the message success rates almost as much as the first modification, reducing the latency of idempotent messages by over 10%. However, this scheme does not increase the number of wires, and has a much smaller overhead. In addition to providing a significant performance advantage, prioritizing messages can help avoid deadlock and livelock situations.

Thesis Supervisor: John F. McKenna
Title: Principal Member of Technical Staff, Charles Stark Draper Laboratory

Thesis Supervisor: Thomas F. Knight
Title: Senior Research Scientist, Artificial Intelligence Laboratory

This Page Intentionally Left Blank

A High-Speed Fault-Tolerant Interconnect Fabric for Large-Scale Multiprocessing

by

Robert Woods-Corwin

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis describes the design and synthesis of an updated routing block for a next-generation wave propagation limited fault-tolerant interconnect fabric for a large-scale shared-memory multiprocessor system. The design is based on the METRO multistage interconnection network, and is targeted at minimizing message latency. The design incorporates an efficient new tree-based allocation mechanism and an idempotent messaging protocol. A fat tree topology is the basis for the network. A Verilog implementation of the design is simulated and synthesized into physical hardware, running at speeds as high as 90MHz in an FPGA. Techniques are discussed to vastly improve performance in a potential future design using custom hardware. Further, two potential modifications to the network are considered. First, the performance effect of allocating dedicated physical wires to streamline the idempotent messaging protocol is analyzed. The modification increases the success rate of messages significantly, but the increased latency due to the space taken by the wires overwhelms the potential performance advantage. Second, a scheme for prioritizing messages is developed. This scheme improves the message success rates almost as much as the first modification, reducing the latency of idempotent messages by over 10%. However, this scheme does not increase the number of wires, and has a much smaller overhead. In addition to providing a significant performance advantage, prioritizing messages can help avoid deadlock and livelock situations.

Thesis Supervisor: John F. McKenna
Title: Principal Member of Technical Staff, Charles Stark Draper Laboratory

Thesis Supervisor: Thomas F. Knight
Title: Senior Research Scientist, Artificial Intelligence Laboratory

This Page Intentionally Left Blank

Acknowledgments

I'd like to thank Tom Knight for providing clue, as well as creating the most enjoyable academic environment I've experienced in a long time. Thanks as well to the entire Aries group at the AI Lab. Despite the fact that most of this work was done while he was asleep, Andrew Huang provided tons of help and guidance without which I'd still be struggling aimlessly. Thanks to Brian Ginsburg for building the board on which this design will be used.

Also, I could not have done this without the aid of many people at the Charles Stark Draper Laboratory. Thanks in particular to John McKenna and Samuel Beilin for their help and their understanding.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc. under Contract NAS9-97216, sponsored by the NASA Johnson Space Center.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

.....

Robert Woods-Corwin, June 2001

This Page Intentionally Left Blank

Assignment

Draper Laboratory Report Number T-1400.

In consideration for the research opportunity and permission to prepare my thesis by and at The Charles Stark Draper Laboratory, Inc., I hereby assign my copyright of the thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.

.....

Robert Woods-Corwin, June 2001

This Page Intentionally Left Blank

This Page Intentionally Left Blank

This Page Intentionally Left Blank

Contents

List of Figures

This Page Intentionally Left Blank

List of Tables

This Page Intentionally Left Blank

Chapter 1

Introduction

The promise of performance from large-scale multiprocessing systems has been limited by network performance. For processors to work together efficiently on anything but the most parallelizable tasks, they need to communicate extensively. Both high-bandwidth and low-latency communication is desired due to the high cost of synchronization events and the cost of memory access latency in a spatially large machine. Amdahl's Law implies that as a parallel machine gets larger, the importance of non-parallelizable tasks like communication gets greater, so the network becomes the limiting factor in attaining high performance.

The interconnection fabric, along with the rest of the system, must be designed to be fault-tolerant, as the rate of hardware failure has been seen to grow linearly with the number of processors.[15] Additionally, some critical systems are designed to function in an orbital environment, where transient failures due to single-event upsets from cosmic rays are common, and where repair is difficult, if not impossible. Consequently, strong fault-tolerance is a necessity for these systems as well.

This work will discuss a class of fault-tolerant interconnect fabric designs that are robust enough to survive multiple faults while still able to deliver the highest level of performance. Also, the design and synthesis of a routing block for such a network will be described.

1.1 Goals

In order to design a high-performance fault-tolerant interconnect fabric, we must first identify the desired features of such a system.

Fault-tolerance It is necessary to integrate fault-tolerance into any design with large numbers of components. Further, a fault-tolerant network is especially useful for specific high-reliability applications.

Low latency As hard as modern processors try to play games to hide latency, network latency is still the most important limiting factor in performance. The best way to hide latency is to reduce it.

High bandwidth Secondary to latency, high bandwidth is critical in the performance of a network. Further, increasing bandwidth decreases the time that a pending message waits in a queue before being inserted into the network, thus decreasing latency.

Scalability A high-performance interconnect fabric should scale well with regard to performance and fault-tolerance.

Building on the aforementioned ideas, we have designed, simulated, and synthesized a robust high performance fat tree network. Based on Minsky's RN1, [14] our design creates an elegant and simple network. A new optimized tree structure has been developed for the critical path of allocation. The network is implemented using synthesizable Verilog, and has been simulated. Currently, a board is being built to test the design in physical hardware using an FPGA-based implementation. Clocking and synchronization issues are addressed in this implementation, and a potential high performance future implementation is analyzed.

It has been shown that a large network built from VLSI components is wire limited and that wire length and thus latency must scale at least as $O(\sqrt[3]{n})$.[12] [4] Since low latency is a primary goal of this design, we will attempt to use wires as efficiently as possible, since they are the most valuable resource available.

Toward that goal, the new design is built to use a fat tree topology and an idempotent messaging protocol. Adding wires devoted to acknowledgment in the idempotent protocol could potentially improve performance. We examine the performance impact and architectural issues involved with adding explicit hardware support for idempotent messaging.

Another consideration in network design is the potential for deadlock and livelock. We consider an extension to our design to implement a priority-based messaging scheme, which can address the problems of deadlock and livelock effectively. The ability to prioritize messages can also provide a performance improvement for a soft implementation of the idempotent protocol.

A simulation of a 4-level fat tree (1024 communication ports) has been employed to verify the functionality of the design and to evaluate the modifications.

This Page Intentionally Left Blank

Chapter 2

Background

2.1 Fault-Tolerance

Fault-tolerance is critical in any large system with many components. As the number of parts in a system scales, so does the frequency of hardware failures. It is important to use a design that is fault tolerant and degrades gracefully in order to ensure that no single hardware failure can partition the network and prevent communication between components.

Additionally, there is a large design space that requires additional fault-tolerance. Systems designed for high-reliability applications and/or for harsh environments need additional fault-tolerance. For example, systems used in an out-of-atmosphere environment experience drastically higher level of single-event upsets (SEUs) from stray cosmic rays, which often result in transient faults. Consequently, our interconnect fabric should be able to withstand many hardware failures and gracefully degrade performance while maintaining connectivity. The principles evident in this design are applicable to such an environment, although this design is not strictly targeted for these conditions.

If, for example, a network had a single point of failure, with a probability of failure of 10^{-6} per second, then the failure rate of the entire network would be at best 10^{-6} per second. Even worse, in such a network, the fault-tolerance would decrease as the number of components increased, and eventually the network would fail so often as

to become unusable.

However, consider a network in which 5 specific components with that failure rate need to fail to bring down the entire network. Now the failure rate of this network using similar components can be as high as 10^{-30} per second. It is likely to be cheaper to achieve fault tolerance by means of replication and architectures which avoid single points of failure, rather than by trying to lower the failure rates of each individual component.

Synchronization is also an important consideration for fault tolerance. In any system that uses redundancy for fault tolerance, each of the redundant subsystems must remain synchronized. To avoid a single point of failure, the redundant subsystems cannot run off of a single clock source, so a separate synchronization scheme is needed.

In the end, fault tolerance is about data integrity. For a simple fault model, encoding data using error-correcting codes (ECCs) can preserve data integrity in the event of faults, and enables easier recovery and isolation of simple faults.

Additionally and quite orthogonally, fault tolerance has a positive economic impact on system construction as it can vastly increase silicon and system yields if implemented wisely. If a single chip is designed in such a way that it still functions, albeit in a degraded fashion, in the presence of one or several manufacturing errors, then many chips which otherwise would have to be scrapped could be used, lowering the effective cost of a fabrication run.

2.1.1 Fault Model

A number of fault models can be considered when evaluating the fault-tolerance of a system. The most strict model is the Byzantine model [9], where a failed component can exhibit arbitrary, even malicious, behavior. This is a useful model when designing a system with ultra-high reliability constraints. However, the overhead of a Byzantine-resilient system often severely impacts performance.

Alternately, we can consider the types of faults that are likely to occur, and specifically design to tolerate those faults. For example, a space-borne critical system should

detect and recover quickly from single event upsets (such as those caused by cosmic rays), since such faults are common in the target environment. This methodology follows the maxim that it is best to focus on performance for the common case, and simply survive the worst case, even if recovery is relatively slow.

We expect that a critical system would require primarily full connectivity at all costs, and secondarily performance. Thus, we will target first survivability and second performance in a faulty environment.

2.1.2 Techniques

A number of fault-tolerance techniques can be applied to a network to maximize performance and survivability. We will focus on design simplicity in particular. Design errors are likely to grow at least as fast as complexity in a large system, so the strongest defense against them is a straightforward design. In particular, our design will feature a straightforward routing protocol and simple building blocks. This simplicity will also come to benefit our performance, as we will see later.

Redundancy

To ensure that connections can be made in a network in the presence of multiple faults, it is important that there be multiple paths in the network between each pair of processing elements. Dilated crossbars in the METRO routing architecture [5] achieve this effect in an elegant fashion. In the METRO system, the number of possible paths between any two processing elements increases as the network grows in size, boosting both performance and fault-tolerance. METRO also has the desirable property that no messages ever exist solely in the network. In other words, at a theoretical level the network is never relied upon to avoid loss of data. This property is quite convenient when checkpointing and recovering from failures.

For applications with high fault-tolerance demands, N-modular redundancy (NMR) is a common technique. Several copies of a piece of logic compute and then vote a result, ensuring correctness in case of failures. The level of redundancy can be ad-

justed to increase or decrease fault-tolerance, a tradeoff between performance per area and fault tolerance. Yen [17] has proposed a Specialized NMR system which can be adapted to complex network topologies. The SNMR system can effectively make use of a large scale network to efficiently provide redundant subgroups of processors that can operate with high fault-tolerance.

Data Integrity

One might also wish to ensure that data transferred through the network remains uncorrupted. A simple method is to simply append a checksum to the end of the data transferred, ensuring the detection of any single-bit error.

For a slightly larger overhead, one could send data using an error correcting code (ECC), which in addition to detecting single bit errors, can also correct them. However, we can go even further. Using the encryption method of our choice, we can encrypt our data using as complicated a method as we desire. Then, when we decrypt the message at the other end, even a malicious failure in the network could not have corrupted the data without detection. These data encoding mechanisms can be implemented at the processor's network interface or incorporated into the network explicitly.

2.2 Network Topology and Routing

For relatively small numbers (e.g. 10) of processors, a simple network topology is feasible, and can offer the best fault-tolerance and performance properties. However, in a system with a reasonably large number of processors (e.g. 1000+), more complex topologies are required to obtain adequate performance. We consider a variety of topologies for their ability to efficiently support massive scaling of the number of processors in the presence of hardware faults.

We will categorize the possible networks by how their latency grows with the size of the network and discuss the performance tradeoffs inherent to each. Also, the fault tolerance properties of each topology will be discussed.

2.2.1 O(1) Latency Network

A fully connected network has the useful property that network congestion is never a problem. This property allows for tight synchronization and fast communication. Draper Laboratory's FTPP [10] uses a fully-connected network to implement real-time Byzantine-resilient communication among 4 to 5 network elements. Taking advantage of bounded-time message-passing, the FTPP implements a real-time hardware fault-tolerant processing environment. While a fully-connected network has excellent performance, its size grows as $O(n^2)$, which rapidly becomes infeasible for a network of much larger size.

However, the fault tolerance properties are excellent for a network of this type. Any single wire failure affects only the communication between one pair of processing elements. Further, in a network containing n processing elements, any arbitrary $n-2$ failures are survivable without partitioning the network or isolating any processing elements.

2.2.2 O(n) Latency Network

At the other extreme of network architecture is a single bus. Here all processing elements share a single wire. A bus is very efficient in area, but has very poor performance and fault tolerance properties. The single communication channel must be shared by every node, and any single fault partitions the network and prevents communication between some processing elements. Further, many faults can obstruct all communications, preventing any system functionality.

2.2.3 O($\sqrt{n}$ and O($\sqrt[3]{n}$) Latency Networks

A common topology that is used as a compromise between a bus and a fully-connected network is a mesh. The two-dimensional mesh has the important property that it maps well to physical circuit board implementation. However, the complementary task of routing in a mesh is less simple.

Meshes take advantage of locality in very convenient and natural ways, and can

offer low latency if designed properly. Mesh-based networks are very good at surviving faults and maintaining connectivity. However, maintaining decent performance in the presence of failures is a complex problem with no easy solutions. In particular, routing strategies that are resistant to deadlock tend to have trouble routing effectively in the presence of faults. [11]

2.2.4 O($\log n$) Latency Networks

We would like our network to have the space efficiency of a mesh, but with the connectivity of the FTPP, so we look for a compromise in which we have higher connectivity without exorbitant area cost. There are a number of network topologies that attempt to scale latency logarithmically with size. Knight[11] argues persuasively that a multistage interconnection network can be significantly better than a simple mesh for very large networks. In particular, although total latency can be slightly worse for a very large multistage network than for a similar size mesh, the available bandwidth per processor drops very quickly in a mesh as the average message length increases with no increase in local wiring.

Regardless of the theoretical elegance of a network topology, latency must scale at least as O($\sqrt[3]{n}$) because of physical three-dimensional constraints. Since a large network is wire density limited, the degree to which those wires can be packed into three dimensions limits total latency. However, that does not mean that a three-dimensional mesh is the best that can be done. In particular, it would be desirable for a long message to traverse paths that only interfere with short messages near their endpoints, rather than all throughout as in a mesh. Put another way, we would like to scale network size while maintaining constant bandwidth per processor, which is an elegant property of many multistage networks.

Hypercube

As the number of nodes in a network grows, connections between nodes will necessarily increase in length. A popular topology for large networks is an n-dimensional

hypercube, connecting 2^n nodes with a maximum distance of $\log n$ stages. Hypercubes can also take advantage of locality of computation, as nearby nodes can be much closer than $\log n$ steps away.

However, hypercube routing networks become difficult to build efficiently as their size (and thus their dimension) increases. The physical space in which to build a network is at best three-dimensional, but a hypercube containing thousands of nodes must be flattened from tens of dimensions into just three. Physical constraints as well as routing issues prevent hypercube networks from scaling beyond hundreds of processors efficiently. This effect implies that a hypercube, like any other network, cannot scale latency better than $O(\sqrt[3]{n})$.

The fault-tolerance of a hypercube is similar to that of a two dimensional mesh. While full connectivity remains even in the case of a large number of network failures, even a single failure is very disruptive to routing. A fault tolerant routing protocol necessarily has to be able to route a message between two given nodes in multiple possible ways. Such a protocol is possible in a hypercube, but is complex and consumes overhead. In particular, Knight argues mesh-based networks are difficult to make fault tolerant because of the deterministic routing necessary to avoid locking conditions.[11]

Multibutterfly

A multistage network like the multibutterfly[1] network offers some desirable properties for our task. Figure 2-1 shows a 3 stage multibutterfly network with radix 4, dilation 2 routing blocks. Like other multistage networks, a multibutterfly allows for efficient scalable interconnect among large numbers of processors. If randomization is applied properly to wiring equivalence classes[11], certain worst cases can be avoided and thus performance guarantees are higher. Most importantly, however, a multibutterfly has excellent fault tolerance properties. In particular, a significant fraction of the total number of wires and routers can fail while maintaining complete connectivity and gracefully degraded performance.

However, any message sent through a multibutterfly network will take $\log n$ steps,

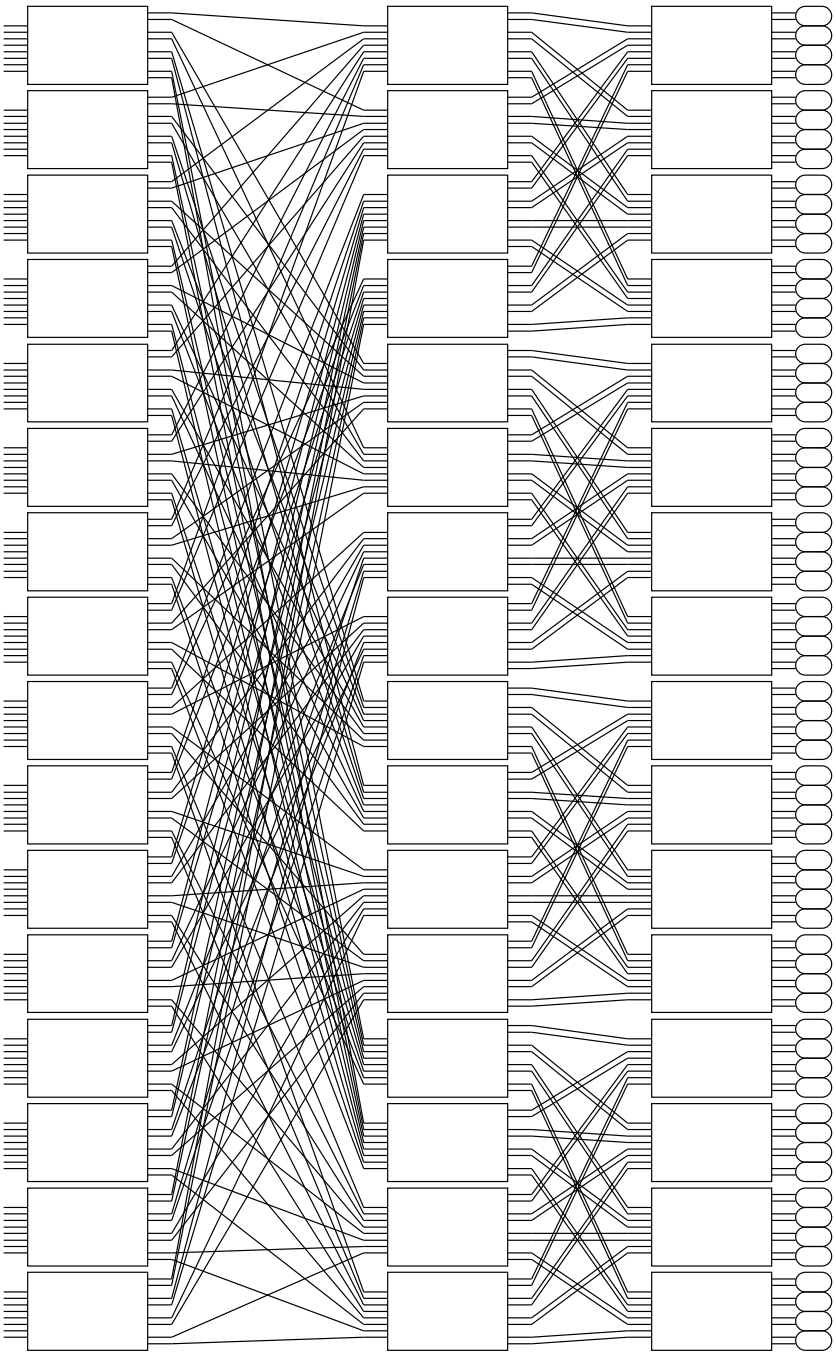

Figure 2-1: 3 Stage Multibutterfly Network

regardless of the physical relationship between the sender and the recipient. In other words, a multibutterfly cannot take advantage of computational locality.

An additional scaling problem relates to the physical realization of the multibutterfly network. In any such network, half of the first stage wires cross the midpoint of the stage. This becomes a limiting factor for implementation.

It is also hard to physically realize random wiring at large scales. Chong et al.[3] have shown a technique that can mitigate the difficulty by wiring many paths together and multiplexing the data across them, easing the physical difficulty while maintaining the performance properties of a multibutterfly. However, some fault-tolerance is lost here, as a failure in one of the fewer wide data paths has a greater impact.

Fat Tree

It is important for a network to take advantage of locality of communication. No matter how elegant the design, if the data has to traverse the entire physical network, it will be slow. The simplest way to exploit locality is with a tree-based design. A simple tree, however, has a choke point at the root. In addition to being a single point of failure, the root node is a bottleneck to communication.

How can this problem be mitigated without losing the benefits of a tree-based network? Leiserson [13] proposed the fat tree network topology, in which the tree is becomes wider near the root, much like a real tree. Thus, each leaf has connections to each of the many root nodes, and there is more bandwidth where it is needed.

Fat trees have a number of useful properties. Fat trees have $O(\log n)$ worst case latency, and also can take advantage of locality when it exists. Fat trees scale exceptionally well, and can be physically built effectively in a two-dimensional technology. Diagrams of fat trees of one to four stages are shown in Figure 2-2.

Further, fat trees exhibit desirable fault-tolerance properties. As the tree scales, so does the number of possible paths from a source to a destination, and so in the case of multiple random hardware failures, performance degrades gracefully. Figure 2-3 shows how a fat tree maintains connectivity in the presence of numerous hardware faults. In fact, the larger the network, the more failures can be absorbed for a

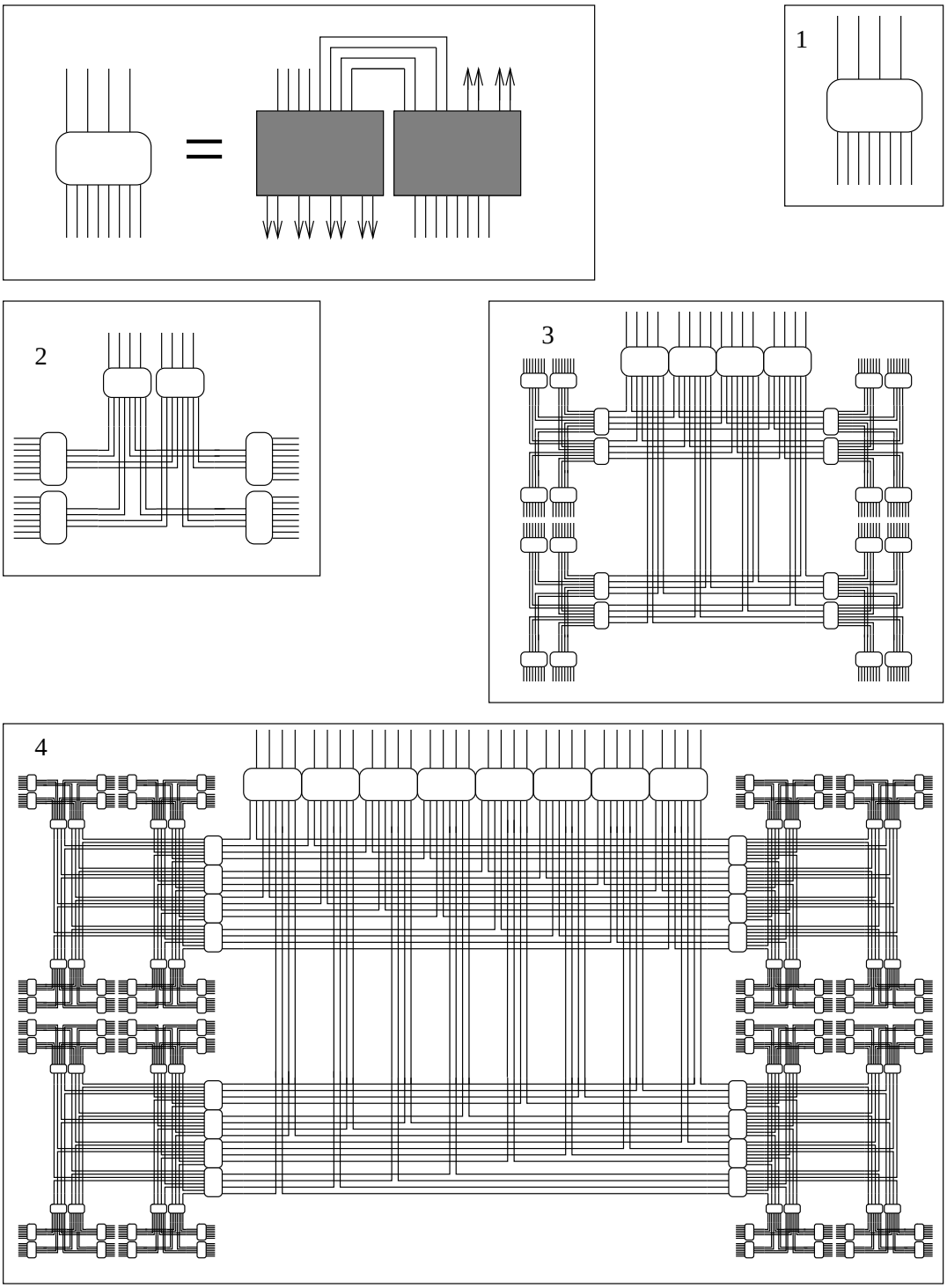

Figure 2-2: Fat Tree Networks of 1 to 4 Stages

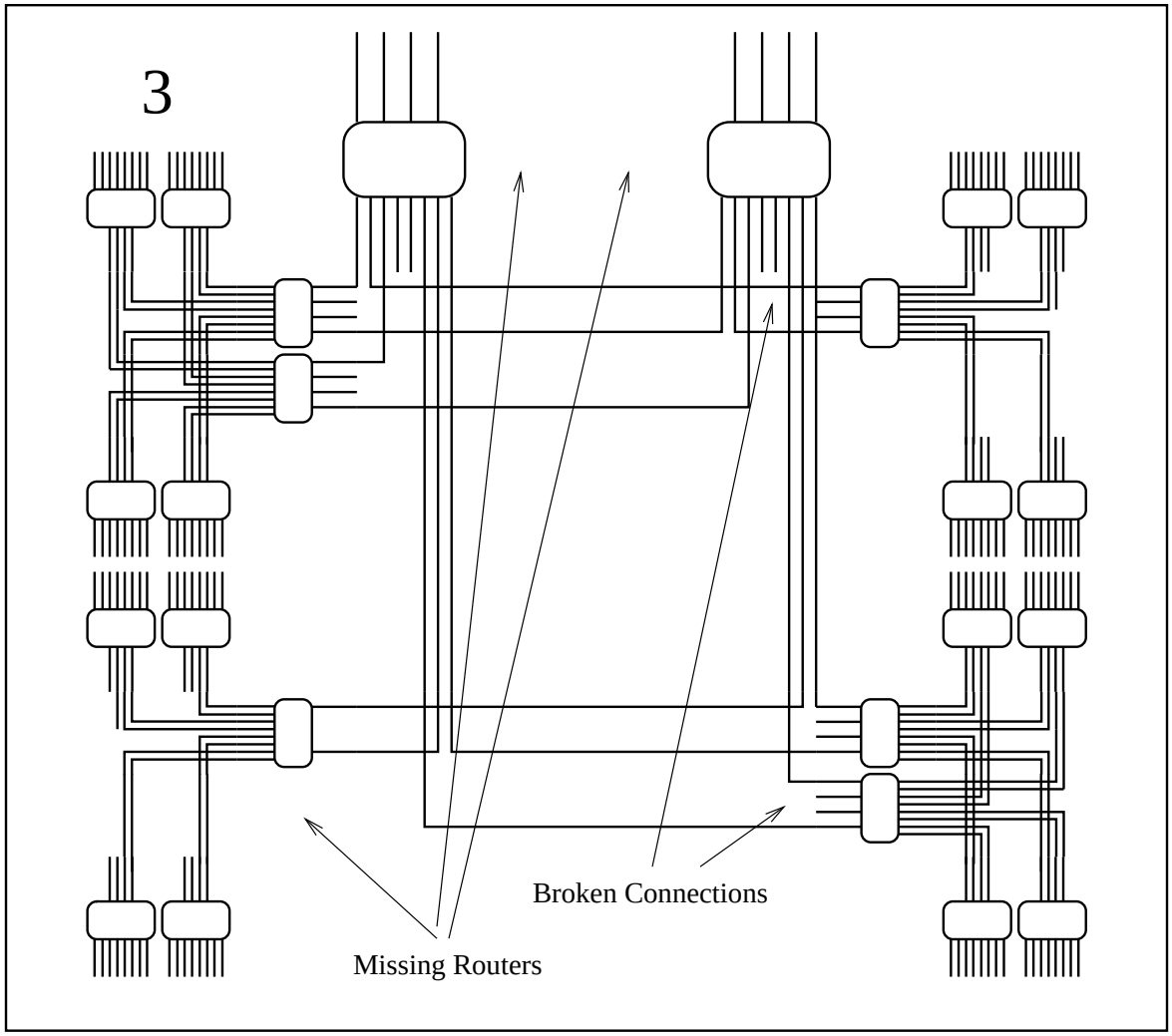

Figure 2-3: A Fat Tree with Many Faulty Components

commensurate degradation of performance. In other words, the fault tolerance scales positively with the network size, which is a very desirable property.

2.2.5 Clocking and Synchronization

The task of maintaining synchronization across a large network is a complex one. Common approaches rely on bounded-time message delivery. However, a routing protocol on a fat tree network cannot guarantee message delivery and maintain decent performance. Statistical analysis [7] shows that, even under high load, a large

percentage of messages are delivered, although it cannot be guaranteed that any specific message will be delivered successfully. The lack of bounded-time message passing hampers tight synchronization among processing elements.

Synchronization can also be implemented at a lower level. If every processor attached to the network shared a common clock source, then they could synchronize based on that. However, a single clock source creates a single point of failure, which is unacceptable.

Pratt and Nguyen [16] have developed a scheme for distributed synchronous clocking which applies well to our network. Their scheme proposes a grid of local voltage-controlled oscillators, each of which is controlled by the phase differences between itself and its neighbors. Through a variation on the basic phase detector circuit, their system robustly maintains phase-lock even in the presence of failed nodes.

2.2.6 Routing Protocol

To take advantage of a low-latency multipath network topology, we need a suitable routing scheme. The METRO Routing Protocol (MRP) [5] is an efficient protocol for a multibutterfly based topology.

MRP is a synchronous circuit-switched pipelined routing protocol where all routing decisions can be made local to the routing element. Locality of routing is important for a scalable system. In this case, the routing information consists of two bits at each stage, indicating which logical output port is desired. In a multibutterfly network, this expands to simply appending the address of the destination on the front of the data, and sending it off. Figure 2-4 shows the format of an MRP message.

MRP defines a message to consist of two parts, routing information and data. The routing information occurs at the beginning of the message, and is simply the binary address of the recipient, and the data can be of any length. For a large network, more than one byte of routing information may be needed. To handle this case, MRP "swallows" the leading byte after all of its information has been used, and the remaining stages of the network route with the second byte. This swallowing procedure can be repeated as many times as necessary.

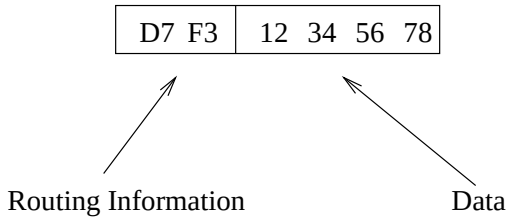

Figure 2-4: METRO Routing Protocol

Possibly the greatest strength of MRP is its simplicity. There are no buffers to overflow, and the sender of a failed message knows of the failure quickly. Routing decisions take a very short amount of time, allowing the network to run at a high clock rate. Also, since routing is straightforward, routers can be made very small and the wires between them can be proportionally shorter. This is especially good as we expect wires to be the dominant factor in the latency of a packet.

Idempotence

One feature that MRP does not support is idempotent messaging. A processing element using MRP can guarantee the delivery of at least one copy of a given message. We might like, however, for the protocol to guarantee delivery of *exactly* one copy of the message.

The basic mechanism to achieve idempotence is acknowledgment of messages. Consider the simple case in which processor A is sending a sequence of messages to processor B. First A sends a message to B, and starts a timer waiting for acknowledgment. A can continue sending messages to B as long as it has the resources to remember the timeouts. When acknowledgment of a given message returns from B, A can forget about that message.

On the other side, processor B simply sends an acknowledgment of each message it receives. However, consider the case in which B's acknowledgment gets lost in the network. Then A will timeout and resend the message. It's important that B remember enough about the message to realize that this is a duplicate message, rather than a new message.

So, we add a third message to the protocol. When B sends an acknowledgment, it sets up its own timer waiting for an acknowledgment of the acknowledgment. When said acknowledgment arrives, then B knows that A won't send any further copies of this message, so B can forget about the message. A now has the additional responsibility of replying to any acknowledgments it receives.

This resulting protocol now assures that, regardless of network failures, recipients will receive exactly one copy of each message sent. It is also argued[2] that the sender and recipient will be able to store less information using this protocol.

Idempotence is better than just a ping-acknowledge protocol because it explicitly tells receiver when it can forget about a message. In practice, a recipient will have to remember information about recent messages anyway, probably in a hardware cache, so a protocol that minimizes the amount of data that needs to be stored will also improve the performance as the cache will be less stressed.

An idempotent protocol can be layered on top of MRP, where messages are tagged sequentially, similar to the Transmission Control Protocol (TCP). However, it may be desirable to implement some support for idempotence in hardware. This thesis will consider hardware and protocol modifications to MRP to support idempotent messaging and the performance impact therein.

2.2.7 Deadlock and Livelock

It is relatively easy to show that a given network protocol is not subject to deadlock. As long as progress can always be made, one is safe from deadlock. However, it is harder to show that a network is resistant to livelock. Consider a network in which every node has a number of long messages that it needs to send, and that every node runs out of "acknowledge" resources (i.e. each node is remembering the maximum number of messages that it needs to acknowledge, so it can't accept any more messages until the acknowledgments have themselves been acknowledged). Then the nodes will all be trying to send messages which keep failing (due to a lack of acknowledgment resources at the receiver) as well as sending short acknowledgments. The acknowledgments could keep running into the long messages and dropping in the

network, and so would never get delivered. In this case, no forward progress is made, and the network is stuck in a livelock condition.

The possibility of livelock cannot be eliminated completely, but we can implement a priority system to ease the avoidance of livelock. In the example above, had the acknowledgments been higher priority than original messages, we would avoid the problem described. Additionally, priority has other features of use to a network of this type.

2.2.8 The RN1 Routing Block

RN1[14] is an implementation of a radix 4, dilation 2 self routing crossbar switch designed for the METRO network. The architecture of RN1 was created with an eye to massively scaled applications. In particular, the design expects wires and pins to be the limiting factor, and primarily aims for low latency.

This work is based heavily on Minksy's RN1 design, so here we present a brief overview of the original RN1 architecture. Subsequently, we will examine some modifications to improve the performance and features of the design.

RN1 consists of a grid of crosspoint switches connecting horizontal input wires to vertical output wires. The grid is surrounded by blocks of logic that support routing and port allocation. A block diagram of the architecture is shown in Figure 2-5. RN1 is designed to be used with the METRO Routing Protocol described above, and in particular it is designed to be small and fast.

Further, however, the design of RN1 is flexible. With minor modifications, RN1 is suitable for many self-routing multistage network topologies. The design does not require the interconnecting paths to all have the same length. Using wave-pipelining, there can be several cycles of data in a wire at a time. This flexibility greatly aids the physical design task of the network.

As mentioned, RN1 at the highest level consists of several subcomponents, each of which handle a separate task associated with the routing process. Here a short overview of their functionality is presented.

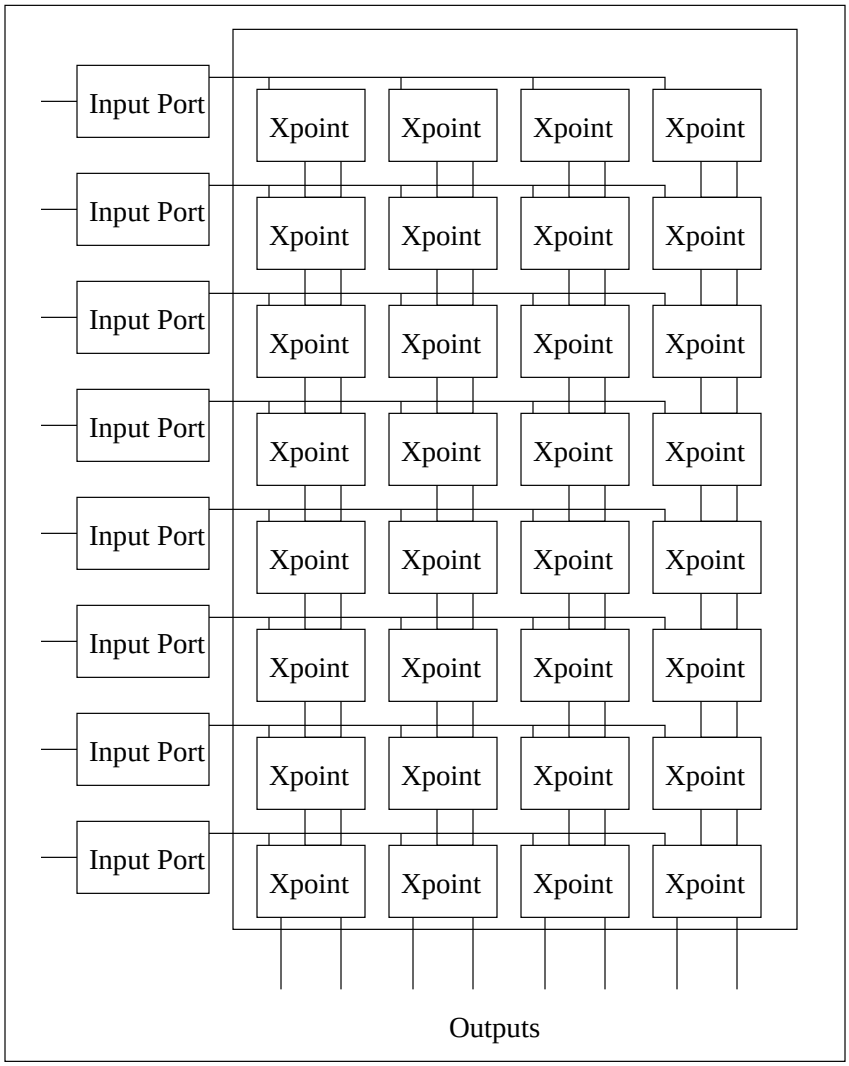

Figure 2-5: RN1 Block Diagram

Forward Ports

The forward ports are primarily responsible for decoding routing information from incoming messages. When a new message arrives, the forward port needs to figure out which back port is desired and then send the appropriate allocation request to the crosspoint fabric. The forward port also needs to swallow the leading byte when appropriate.

A small finite state machine (FSM) is used to decode the routing information and send allocation requests. As mentioned before, each stage uses two bits from the leading byte as routing information. Every fourth router needs to swallow a byte, so the next stage sees the next byte with the appropriate routing information.

In summation, the forward ports parse incoming data and request appropriate routing.

Back Ports

The back ports are responsible primarily for the turning feature, which allows for quick message acknowledgment. When a message is sent through the network, if it reaches its destination, the sender would like an acknowledgment of that fact. On the other hand, if a message is interrupted, the sender would like to know so that it can retry as soon as possible. The ability to turn a connection around enables both of these.

However, this feature requires wires to be bidirectional, and also requires that connections are held open when no data is being sent on them. As the network scales, messages get smaller compared to the expected latency of the network, and this wastage is exacerbated.

Crosspoint Fabric

The crosspoint array provides the connections through which any forward port may connect to any back port. It consists of an 8 by 4 grid of crosspoint modules. This array maintains the state of which connections are active, and transmits allocation

information to and from the forward ports.

Crosspoints and Allocation Logic

A crosspoint can connect or isolate a forward port to one of two logically equivalent back ports. Each cycle, the crosspoint knows which of its back ports are available, and whether the front port is requesting this connection. When the forward port requests a connection which is available, the connection is granted and the state is updated.

The crosspoint is a very simple circuit, consisting essentially just of two connections which can be on or off. However, the allocation logic which determines which connections are made and broken is slightly more complex.

Each cycle, the potential allocation of a free back port ripples down the chain of forward ports, stopping when a request is made. So, for example, forward port 3 has a higher priority than forward port 5 when requesting a given back port. Once a request has been granted, that connection remains open until it is dropped, regardless of which other requests occur. To ensure fairness, half of the back ports give priority to lower numbered forward ports, and the other half give priority to higher numbered forward ports.

This is the critical path of the RN1 design, and Minsky developed custom circuitry to speed it up. In 1991 technology, he was able to run the routing block as high as 50MHz. In the ten years since then, there has been quite a bit of process technology advancement. We will have to show that in a custom process, we can do quite a bit better.

This Page Intentionally Left Blank

Chapter 3

Architecture

The architecture of our design targets several simple goals:

- Low Latency
- Scalability
- Short Wires
- Fault Tolerance

Most networks advertise their performance in terms of bandwidth. However, that is only one side of the story. Latency becomes more important than bandwidth as messages become small relative to the size of the network. For example, when trading music files over the internet, it's not so important whether the first bytes of the files begin to arrive after 50 milliseconds or 2000 milliseconds, as the complete file transfer will take a minute or two anyway. It is more important to be able to send more bits per second, that is, to have a higher bandwidth. In other words, since the data size is large compared to the latency of the network, latency is a relatively unimportant component of overall network performance.

Consider, however, the network connecting nodes of a shared memory system. The common traffic in this network will consist of memory accesses, which are very small messages. Memory requests will be a few bytes of addressing information, and

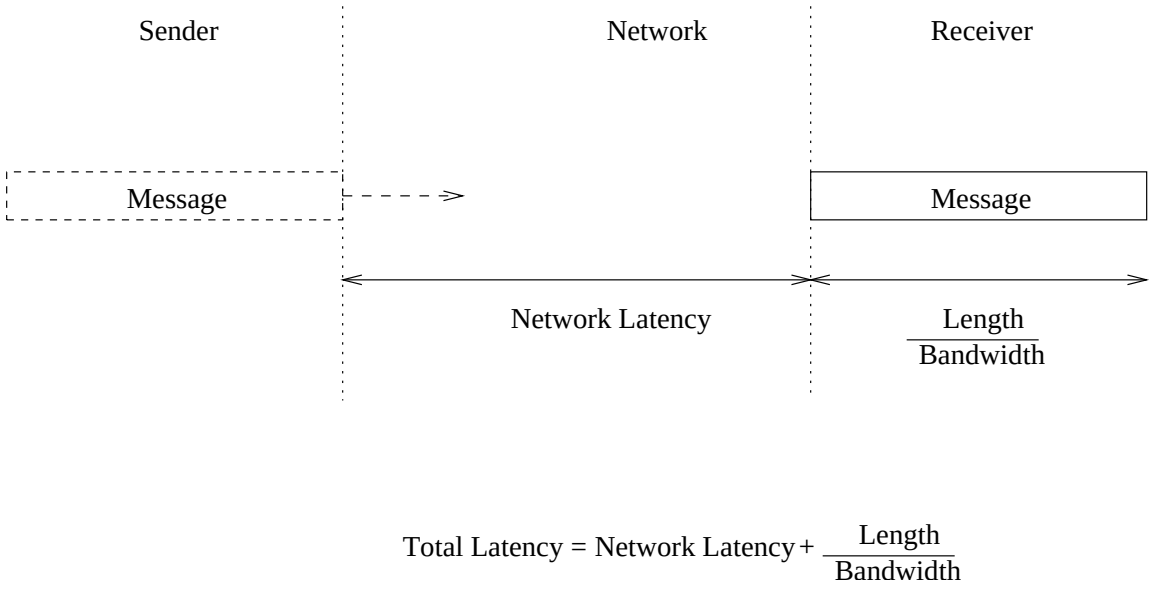

Figure 3-1: Latency Components

the data not much larger. In this case, the sender of the request may be stalling on the result, and the latency is the most important factor. Techniques like out-of-order execution and multithreading can hide some of the latency of remote memory accesses, but latency is still a major limiting factor in system performance. Total message latency will be the primary metric for network performance in this thesis. Figure 3-1 shows how total message latency depends on both network latency and bandwidth.

And the story only gets worse in a large system. In any attempt to scale a network to a size larger than a few hundred nodes, several points become clear quickly. If one desires low latency, the speed of light quickly becomes the most important limiting factor. The network becomes physically large, and the size of the wires becomes the important factor in determining how close together routing stages may be placed. Consider that the number of stages in a reasonably scalable network topology can grow as $O(\log n)$, but the length of the wires must grow at $O(\sqrt[3]{n})$ in a three dimensional world. Thus, the wires themselves become the limiting factor in determining total message latency, regardless of how elegant a theoretical network topology may be.

So, as n becomes large, the number of routing stages data must traverse diminishes in importance, and instead, the length of wire that the data travels becomes important in determining latency and thus performance. Thus we design with the goal of using wires as efficiently as possible, since they are the most valuable resource available to the network designer.

As the size of a network grows, the number and frequency of failures of network components or processor components grows as well. Whether we are considering SEUs or permanent hardware failures, the frequency of failures likely will grow at or near O(n) for a network of size n. This is not a statistical certainty, but it has been observed[15] in a number of large systems. In any reasonably large system, the best way to deal with the growing failure rate is to design in redundancy, and plan to survive multiple failures.

At a minimum the network should be able to degrade gracefully in the presence of a significant number of failures. It is additionally desirable to transmit data correctly in the event of hardware failure, but it is often more efficient to simply be able to detect when there is a data error and resend.

We attempt to design an architecture that balances these design goals effectively, creating a high-performance robust fault-tolerant network.

3.1 Architectural Overview

In a large enough network, the length of wires will be the most important factor in determining the latency. The routing block is pin-limited by packaging, so there should be a lot of extra silicon to spare. In addition to latency, bandwidth is also an important factor in the network's performance, and it is limited by the clock rate at which the routing block can be run. Consequently, in designing the architecture for an individual routing component, we try to maximize the clock rate at which our design can be run by spending some of our extra area with as small a latency impact as possible.

In particular, the allocation logic is optimized to use a new tree-based mechanism.

By replicating logic and using extra area, the critical path computation is streamlined.

Figure 3-2 shows the high level-organization of the design. The task of routing in a crossbar switch can be broken into three major steps, and each step can be implemented in a pipeline stage if desired.

1. For each input, decide whether we want to request an output, and if so, which one.

2. Allocate the outputs to the requesting inputs.

3. Move the data from input to output.

The decision of whether to pipeline or not is a complex one. At the extreme, we can take the point of view that latency is everything, and there should be no pipelining. But if we can increase the bandwidth significantly with only a minor increase in latency, it's probably worthwhile. Consider that the real latency of a message is network latency + (message length / bandwidth) (see Figure 3-1).

So, if the goal is to deliver complete messages as quickly as possible, in many cases pipelining makes sense. Consider the effect of adding a pipeline stage to the router. There are two major cases. In the first, the amount of time that a message spends in a pipeline stage of a router is much smaller than the amount of time spent in the wires. In this case, total performance can be increased with good pipelining, since the total latency is largely wires anyway. However, if the amount of time spent in the wires is comparable to or less than the time spent in the routers, then it may not be worthwhile to further add to the latency for a small bandwidth gain. The line separating these two cases will be different for different applications, and pipelining can be added and removed relatively late in the design cycle, as it is mostly orthogonal to the rest of the design.

We expect that we will be able to run our routing components at sufficiently high clock rates such that interchip communication will be a complicated design task in itself. These interchip wires will likely be several clock cycles long, and thus we will wave pipeline the data along them. For this case, bidirectional wires are a bad

idea. The ability to turn a connection around incurs a several cycle penalty, but more importantly, it complicates the critical task of building a high-performance signaling system. Instead, for the highest performance, we will use unidirectional series terminated lines for interchip communication.

However, with the removal of bidirectional wires, we have also lost the ability to directly turn connections around. Messages should nonetheless be acknowledged as quickly as possible, and so we will consider devoting narrower banks of wires to the task of acknowledgment in explicit hardware support of an idempotent messaging protocol.

Our target topology is a fat-tree, which requires a slightly different routing protocol. We modify the implementation to support the swallowing issues associated with the new protocol.

Finally, we examine clocking and synchronization issues. RN1 used a two-phase non-overlapping clocking system to facilitate some circuit optimizations. Our design is simple and fully synchronous, as it is targeting an FPGA. More complex clocking schemes may be warranted for a higher performance technology in the future, however.

RN1 requires that interchip wires be a fixed number of clock cycles long, but does not specify how many. A mesochronous clocking scheme with wave pipelining allows for greater flexibility and ease of physical design. With the mesochronous clocking scheme, interchip wires need only be consistent in length within a single connector; there is no need for global consistency of wire lengths, and each set of wires may be of arbitrary length.

3.1.1 Critical Path

As mentioned above, we break up the design into three stages:

Request connection The input port is responsible primarily for decoding routing information from incoming data, and making allocation requests. This is implemented with a simple FSM and some combinational logic.

Part of the decoded routing information includes knowledge of whether to pro-

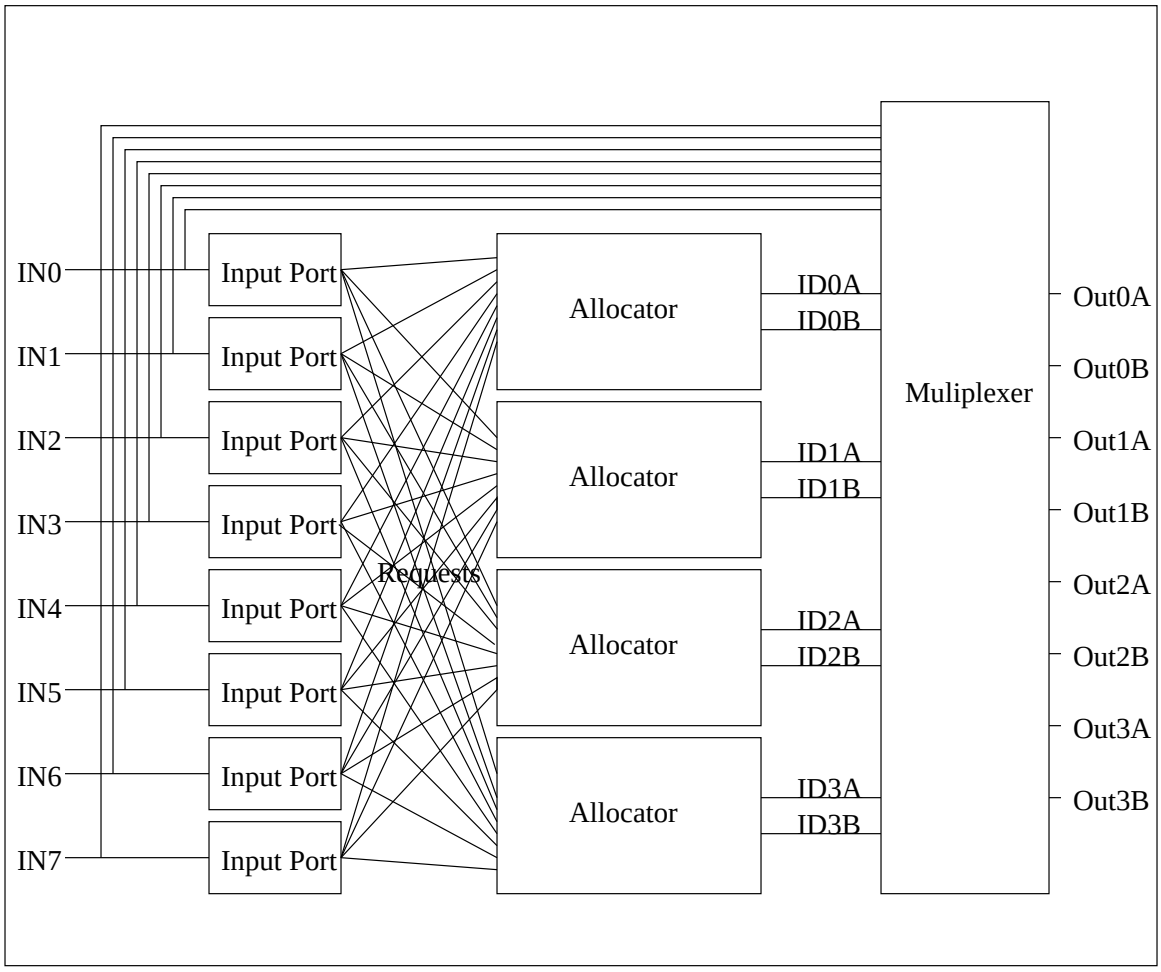

Figure 3-2: High Level Architecture

ceed with the allocation or to swallow this byte and route using the next one. The routing protocol requires that routers swallow every n stages, where $2n$ is the number of bits of routing information in a single word. So, when $n = 4$ and the lowest level is zero, we swallow on the way up at the levels which are 3 mod 4, and on the way down, at levels which are 0 mod 4. Further, we swallow anytime a message is routed across from the up tree to the down tree.

Each pair of bits in the leading word contain the routing information for their respective stages of the network. From this information, the input port decodes which output port is being requested, and sends the appropriate signal to the allocation stage.

Allocate The allocation stage of the pipeline is the critical path in the design. RN1 implements this section with a cascading chain of requests. This chain requires that the critical path be at least 8 stages long. We introduce a modification using a tree structure which can shorten that portion of the critical path to much fewer stages.

We design for the case of varying message priority, and normal messages are routed as a special case. Priority information is contained within the routing blocks of a message. Figure 3-3 shows the design that is used here. Each cycle, the allocator needs to compute the two highest priority messages requesting the logical output port. It is relatively straightforward to build a binary tree to compute which input port has the highest priority, but to find the second highest is more complex.

To quickly find the second highest priority, eight additional parallel trees are used, each of which omits the request of a different input. Each cycle, the highest priority is computed, and that information is used to select from the eight potential second highest priorities. The critical path of this logic is only one stage longer than the basic tree.

This technique uses replication to take advantage of the plentiful area available. Since all possible second best paths can be computed at once and without

waiting for the result of the primary tree, the computation is very short and fast.

In a full custom implementation this may not be much better than a chain due to the circuit techniques that can streamline a chain, but in an FPGA, even short wires are slow, so we optimize to have as few combinational stages as possible. In the extreme case, we can use the 8 allocation bits to index a single 256 entry block ram to determine the routing in one "stage" of logic.

Route The routing block has a simple task, but one which is wire-intensive. Given a list of input-output connections, it simply connects the wires accordingly. FPGA hardware supports reasonably large multiplexers efficiently, so the design uses those rather than a crossbar switch with tristate buffers. The design is shown in Figure 3-4.

3.1.2 Fat-Tree Topology

The original MRP design can be easily adapted to the fat tree topology. We primarily need to make changes in the swallow logic, and we can additionally optimize to have different up and down routers.

Unlike in the multibutterfly design, where all messages conceptually travel in the same direction, in a fat tree, messages start at a leaf, travel up the tree for a while, then turn around and go down a different branch of the tree to the destination leaf. Figure 3-5 shows an example of message being routed in a fat tree.

The fat tree is implemented by having two routing blocks at each logical node of the tree, one routing upward data, and one routing downward data. These blocks have fundamentally different tasks, and need to be distinguished in hardware.

In the original MRP, a word was swallowed every 4 stages, when the routing info in that word had been used up. In a fat tree, a given down router could be the 2nd stage in a message traveling locally, but the 8th stage in a message which has longer to travel.

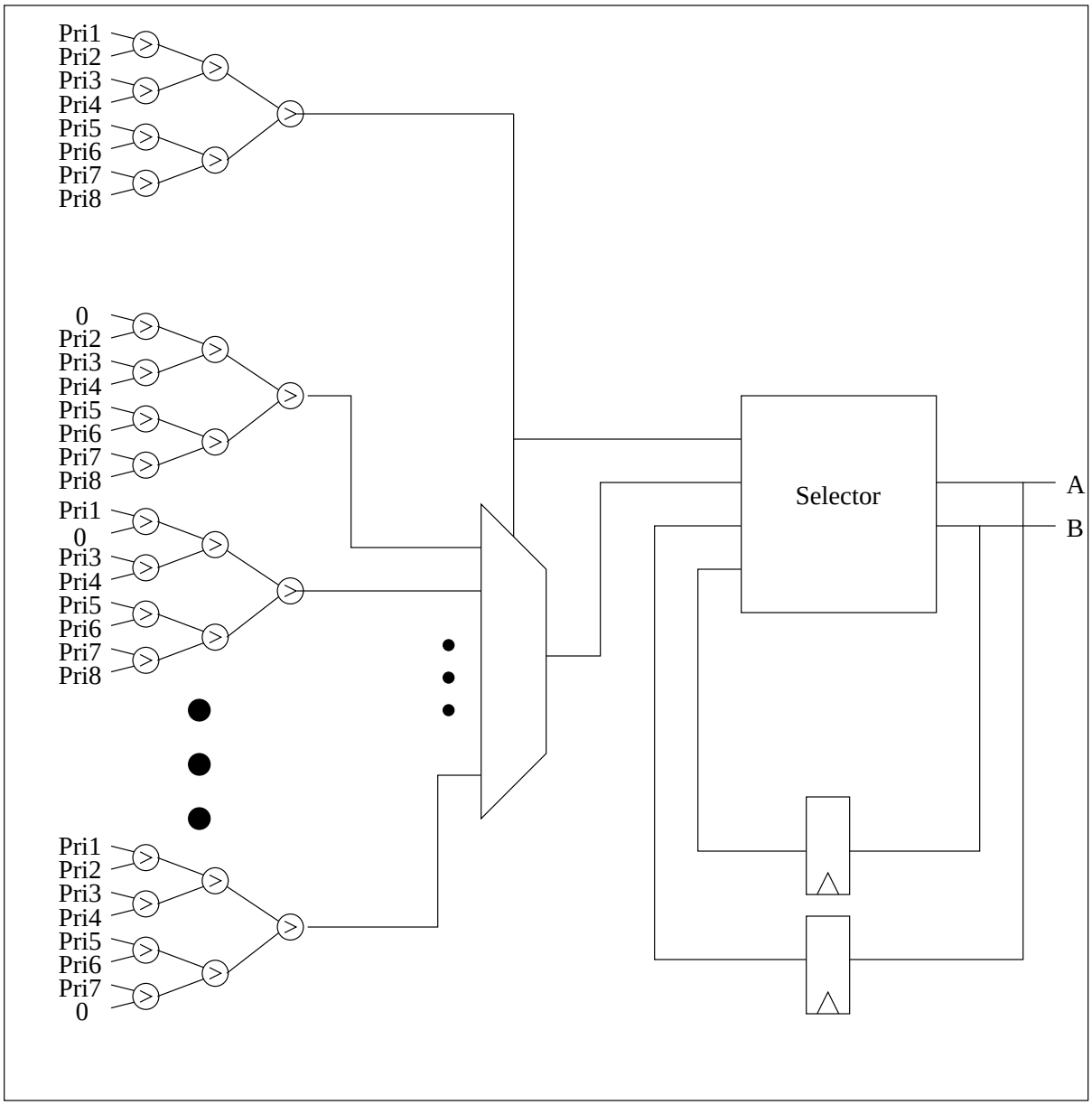

Figure 3-3: Allocation Logic

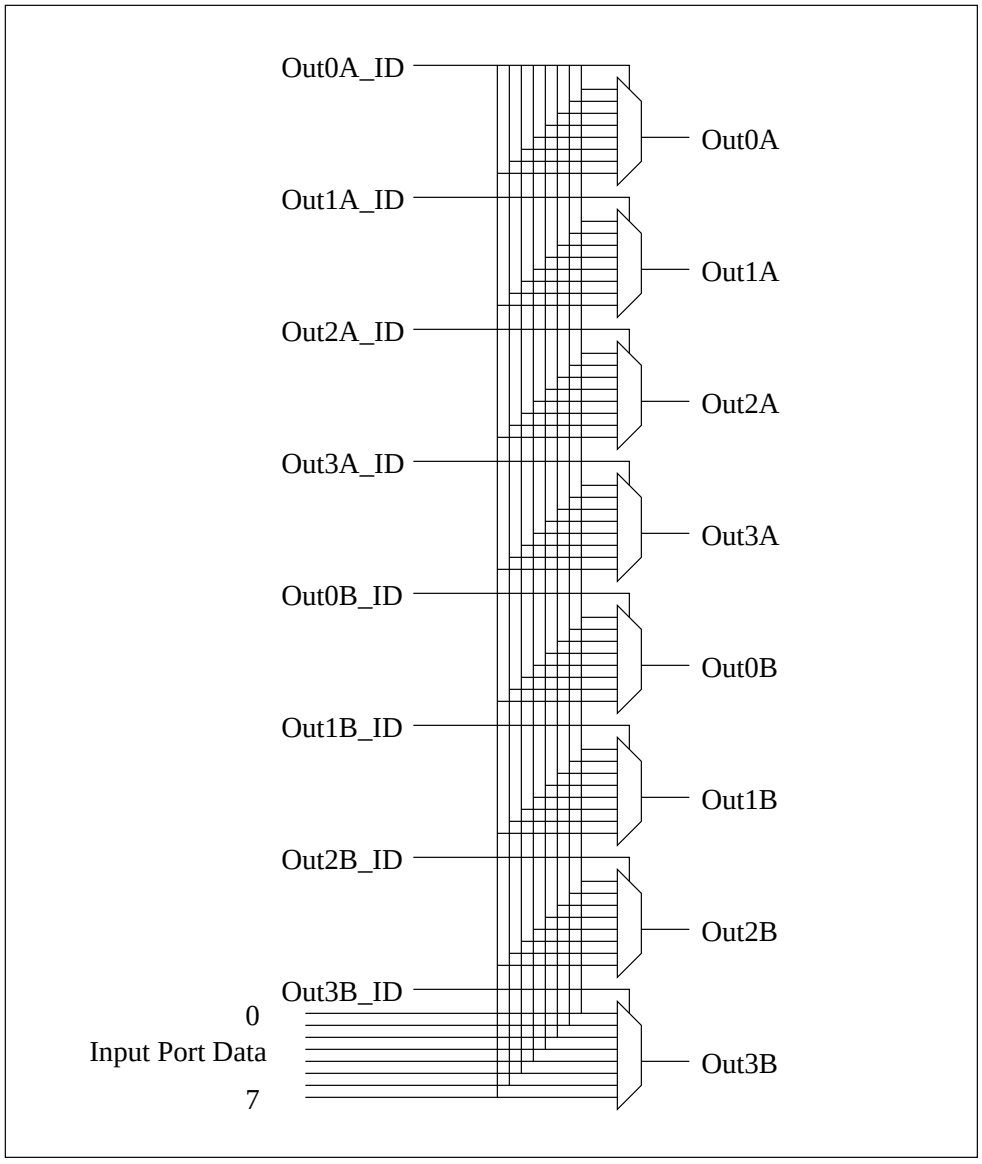

Figure 3-4: Multiplexed Routing Logic

To cope with this condition, the protocol is modified to always swallow on the transition between an up router and a down router, in addition to the normal swallowing every n stages. Now a message will always have at least two routing words, one to go up, and one to go down. See Figure 3-5 for an example.

There are two further optimizations that we can make, given the fat tree topology. First, the topology allows routers to have a different number of ports dedicated to up/down traffic than to across traffic. In particular, DeHon[6] suggests that bandwidth should increase by a factor of $\sqrt[3]{16}$ at each stage, rather than a factor of 2 to preserve the volume universality property of the network. This has no effect on downward routers, but an upward router could be made more flexible. Perhaps 5 of the 8 ports should correspond to "up" and 3 should correspond to "across." In general, when going up one would like a router with variable dilation on each logical output port.

The second optimization flows from the realization that a message can move up the tree in fewer stages than it can move down, since less information is needed to describe the destination. We can rewire the tree so that at each up router, instead of simply deciding between "up" and "across," the remaining height of the tree is partitioned into 4 sections. Then the routing is done using using existing radix 4 switches, and the total number of stages can be reduced from $(2*\log n)$ to $(\log n+\log(\log n))$. In addition to reducing the number of stages a message traverses and thus the message latency, this simplifies the routing protocol and uses uniform routing hardware, but greatly complicates the wiring of the network. Consider that in the simpler design described above, every upwards path between two routers has a complementary downward path, and every path connects adjacent levels. Both of these features can be heavily exploited by physical designers. However, the optimization for logarithmic upward routing is less regular and requires that many paths bypass many levels at a time. This adds to the difficulty of an already challenging signaling task, because drivers would need to drive a signal through multiple connectors and cables, as well as for a significantly longer distance.

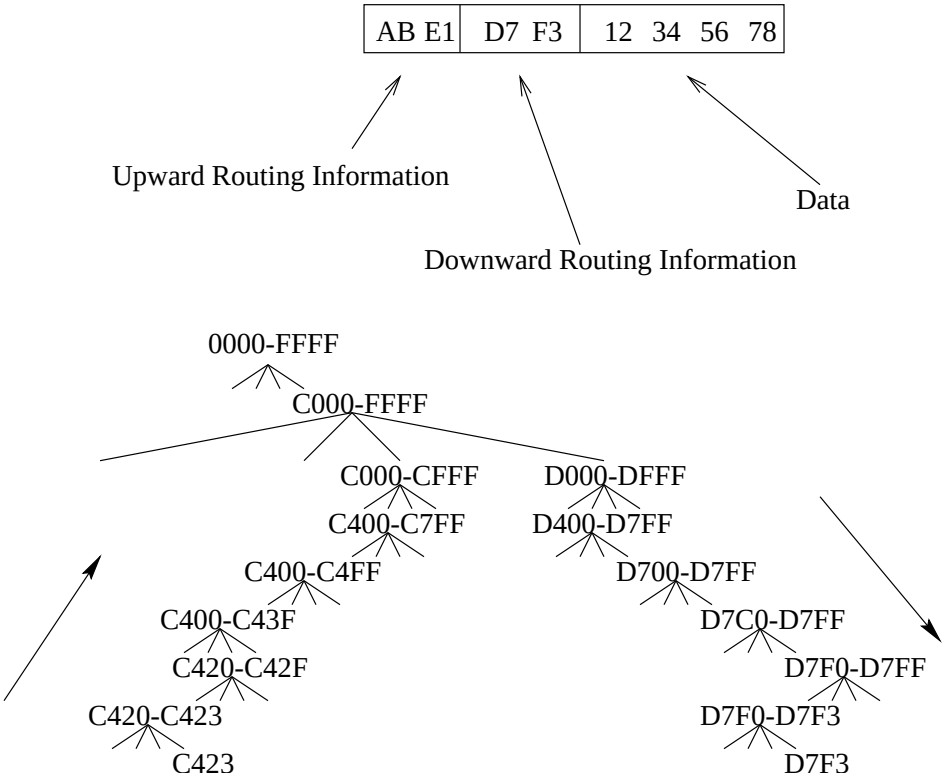

Figure 3-5: Routing Example

3.1.3 Routing Protocol

We assume here that the network is a fat tree consisting entirely of radix 4 routing components. In the up direction, output ports 0 and 1 connect across, and ports 2 and 3 connect up to higher levels of the tree. To route in a fat tree, the sender needs to calculate more than just the receiver ID that MRP uses as routing data.

First, the sender must calculate how high in the tree the message must go to reach the destination. This can be done by comparing the most significant bits of the sender's ID and the receiver's ID until they differ. The index of the highest bit that differs divided by two is the number of levels of the network that the message must ascend.

The first words of the message (exactly how many words is unspecified; routing just keeps going until the message arrives somewhere) correspond to the "up" half of the path the message travels, with bit pairs 1X indicating "go up" and 0X indicating "go across."[1] When the message crosses to the down half of the tree, the last up byte is swallowed. Then the down routers route normally as in MRP. We must ensure that the down routing word's bit pairs must be right-justified so further swallowing happens appropriately.

So, for example, to send a message from processor 0xC423 to processor 0xD7F3 in a 2^{16} node system containing the data 0x12345678, we might send the following sequence of bytes: 0xABE1D7F312345678 (see Figure 3-5). This data encodes a path which goes up six times, then across and down to processor 0xD7F3. Note that not all of the 0xD7 byte is used, since we crossed at level 6, but the down routers know where in a given byte to look for data appropriate to their level.

3.1.4 Clocking

A globally synchronous clocking system is not feasible for a large network and the speeds at which we plan to run. In particular, we need to address the issue of clock skew between chips. A solution is to use a mesochronous clocking system,

[1] X can be chosen randomly to minimize collisions

which tolerates skew but still requires no slippage. The mesochronous system involves sending a clock along with any off-chip data as a qualifying signal so that the receiver knows when the data is valid.

The specific protocol used requires that the data be valid from before the incoming clock goes high to after the incoming clock goes low, and also that said period spans at least one edge of the local clock (the first condition alone is insufficient due to jitter). Thus, if we sample on both edges of the local clock, we are guaranteed that at least one of the two has correct data, and we can select by choosing the one during which the incoming clock was low.

In a clocking system which is not globally synchronous, metastability is a concern. While no system can be provably invulnerable to metastability, we can design to minimize the likelihood of it. In particular, we ensure that at least one of the mesochronous sampling points is guaranteed to fall in a zone that is not transitioning. This can be achieved by having more than two sampling points, or by cleverly tweaking the duty cycles of the sampling clock. Then, a state machine can determine which of the sampled results is not metastable and will select valid data.

This is good enough to build a small system, but for fault tolerance in a large system, the single crystal clock source is a single point of failure. What we really want is a way to have each switch independently clocked, while maintaining synchronization for inter-board communication.

The distributed synchronous clocking scheme described previously addresses this problem effectively. This scheme simplifies some aspects of synchronization as well, as local timers are guaranteed to remain identical to remote timers since there is no slippage.

Chapter 4

Synthesis

The next step after simulation is to actually build a system in hardware. Taking the step from simulation to synthesis requires the consideration of a new set of design challenges. Given the constraints of cost, packaging, and time, we need to decide how large of a network we can build, and what types of hardware to use.

A real commercial implementation of a large scale network of this type would have vastly different cost and design time constraints than does a small research group with a few graduate students. First, we'll discuss how one might build this system in a resource-laden environment, and then what we are actually doing now.

4.1 Resource-Intensive Synthesis

With many experienced designers and a large supply of money and time, our network could be built in such a way as to fully take advantage of the fault tolerant and high performance architectural features of the design and of fat tree networks in general. In particular, the following areas of the design can be optimized for maximal performance

Clocking: It is imperative that our network not have a single point of failure for fault tolerance reasons, so we need to have multiple independent clock sources. However, we need to handle the slippage between different clock domains. A

nice solution to this is to use Ward and Nguyen's NuMesh[16], which uses PLLs to keep a grid of independent oscillators in sync.

This solution still allows for skew between clock domains, but the use of a mesochronous clocking scheme for all interchip communications addresses that issue. With extremely high performance signaling requirements over RC transmission lines, we would like to be able to have as small as possible a portion of the clock period in which the data is required to be valid. This can be achieved by sampling the external data and clock at tighter intervals. Consider for example a system in which we require the data to be valid for the 1/128 of the cycle after the rising edge of the external clock. This can be done by making 128 copies of the internal clock, each phase-shifted by 1/128 of the clock period, and sampling on each rising edge. The first sample in which the external clock is high contains valid data. This technique can loosen the constraints on the signaling system and allow for higher performance wires.

ASIC Technology: Given the pin-limited nature of existing packaging for our application, the on-chip design can spend area to increase bandwidth with small cost in latency. A simple method would be to duplicate the critical path and run the two copies in parallel on alternating inputs.

Signaling: Since wires are expected to be our limiting resource, it is important to use the highest performance signaling methods possible. At least for long off-chip and off-board wires, and probably for shorter wires as well, we would use a differential signaling method.

A major limit on the performance of many signaling methods are process technology unknowns. For example, it is difficult to precisely match impedances for terminated transmission lines before fabrication. However, feedback systems have been developed which deploy adaptive impedance control [8] to optimize performance by adapting to static manufacturing parameters automatically. Using these methods and others, designers have achieved signaling rates as high as several Gb/s per wire. Our design will aim to approach that level of perfor-

mance.

Packaging and Pin Count: Considering the vast performance advantage of on-chip wires over wires that cross chip boundaries, it is desirable to fit as much on a single chip as possible. Our design is overwhelmingly pin-limited, so we would choose a package with a maximal pin count. A level 2 fat tree with 8 bit wide data paths would require about 880 I/O signals, or 1760 pins differentially signaled. This is within a reasonable factor of the bleeding edge of current packaging technology today in 2001.[1] In the event that an entire level 2 fat tree could not be fit on a single chip, we could instead fit a level 1 fat tree and widen the data path to 16, 32, or 64 bits. This would have the beneficial effect of effectively shortening messages, which would lead to less network congestion in addition to higher bandwidth per node.

Another possibility is to design the network using a fat tree topology, but at the leaf nodes, instead of a single processing element, placing a small multibutterfly network containing several processors.[6] This smaller multibutterfly network could potentially fit on a single chip. A 2-stage multibutterfly of radix 4, dilation 2 routing blocks with 8 bit wide data paths would require 704 I/O signals (1408 pins differentially signaled).

This deep integration has an impact on the fault-tolerance of the network. Suppose a single chip fails. Then a large portion of the network becomes inoperable. This effect is a necessary evil of a highly integrated network. The only way to avoid it would be to use more distinct components, which directly impacts performance. The same problem exists at the board level and anywhere else that is susceptible to common-mode failures.

Board Design: Analogous to the performance difference between on-chip wires and wires which cross chip boundaries wires is the difference between on-board wires and wires that must travel off-board through connectors and cables. Conse-

[1] IBM ASIC technology offers 2577 pin packages commercially as of May 2001

quently, we would like to fit as many chips on a board as possible. If a two level fat tree fits on a single chip, a four level tree could reasonably fit on a single board, requiring 22 chips and connecting down to 256 processors (2 in ports and 2 out ports per processor). The board would need connectors with 1280 pins worth of upward signals to connect to higher stages of the network. This is a large but feasible amount of wiring to fit on a board.

Cables: In a large network, even with all of the effort described above to keep wires short, there will be long off-board wires, since as the number of stages in the network grows as $O(\log n)$, the three dimensional space into which the network fits must grow at least as $O(\sqrt[3]{n})$. Consequently, total message latency will be significantly affected by the properties of relatively long cables. We would choose a cable well adapted to wave-pipelining a large number of differential signals in a space-efficient manner. Further desirable attributes of a good cabling solution are a low dielectric constant and low signal loss at high frequencies.

4.2 Proof of Concept Synthesis

As wonderfully high performance as all of the aforementioned techniques would be, many are beyond the capabilities of a small research group. This section is a description of what is actually being built by the Aries group with our small budget and staff. Nonetheless, we do our best to still have a wire-aware design methodology for our synthesis, and we hope to demonstrate solid performance even with these cheaper and faster techniques:

Clocking: Although using only a single clock source creates a potential single point of failure, it is a lot simpler to design and build. We chose to focus in this implementation on showing the performance qualities of this architecture. A scheme like nuMesh can address the single point of failure, but it is orthogonal to our focus, so we will omit it in this implementation.

Nonetheless, we need to deal with the skew among different routing components,

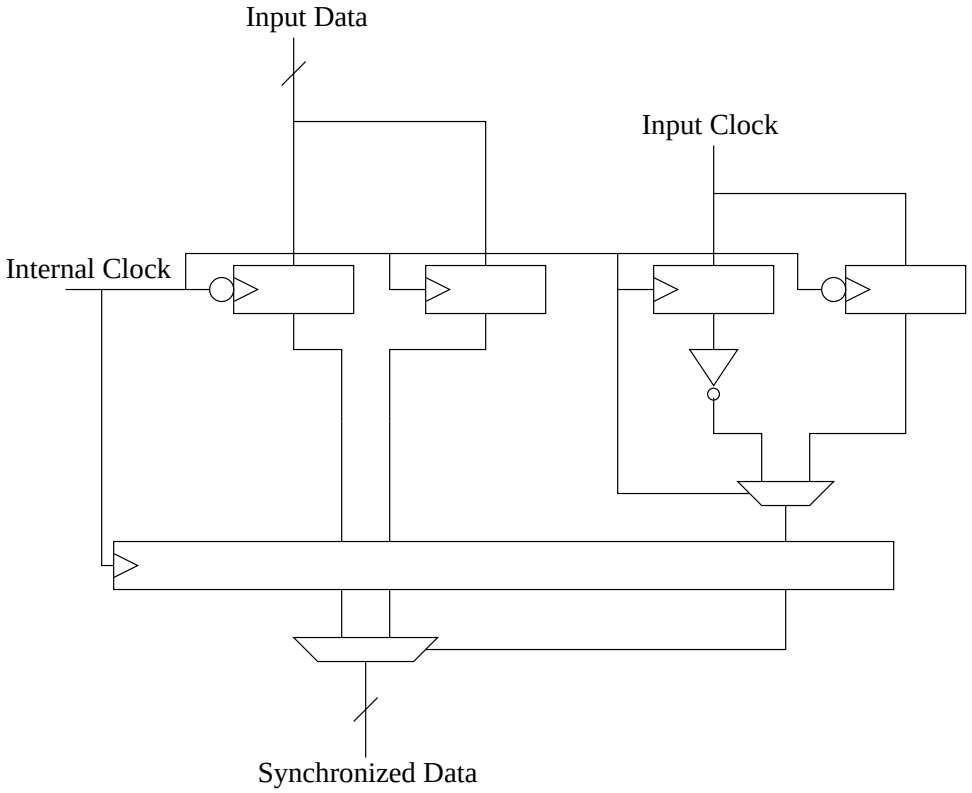

Figure 4-1: Mesochronous Clocking Logic

so we have implemented a simple mesochronous system as described above. The design of this block is detailed in Figure 4-1. A waveform demonstrating the functionality of the block is shown in Figure 4-2.

FPGA Technology: For ease and speed of design, the technology of choice is an FPGA. As FPGAs have matured as a technology, their performance has increased enormously, and is slowly closing the performance gap to a full custom ASIC solution. However, a large performance difference still remains. Thanks to the generosity of Xilinx, Inc. we have found ourselves in possession of a number of Virtex-E FPGAs. In particular, the routing block is implemented targeting XCV1000EFG1156CES devices. This chip has 660 I/O pins, and more than enough internal resources for this type of design.

The major performance loss due to the use of an FPGA in this design is that

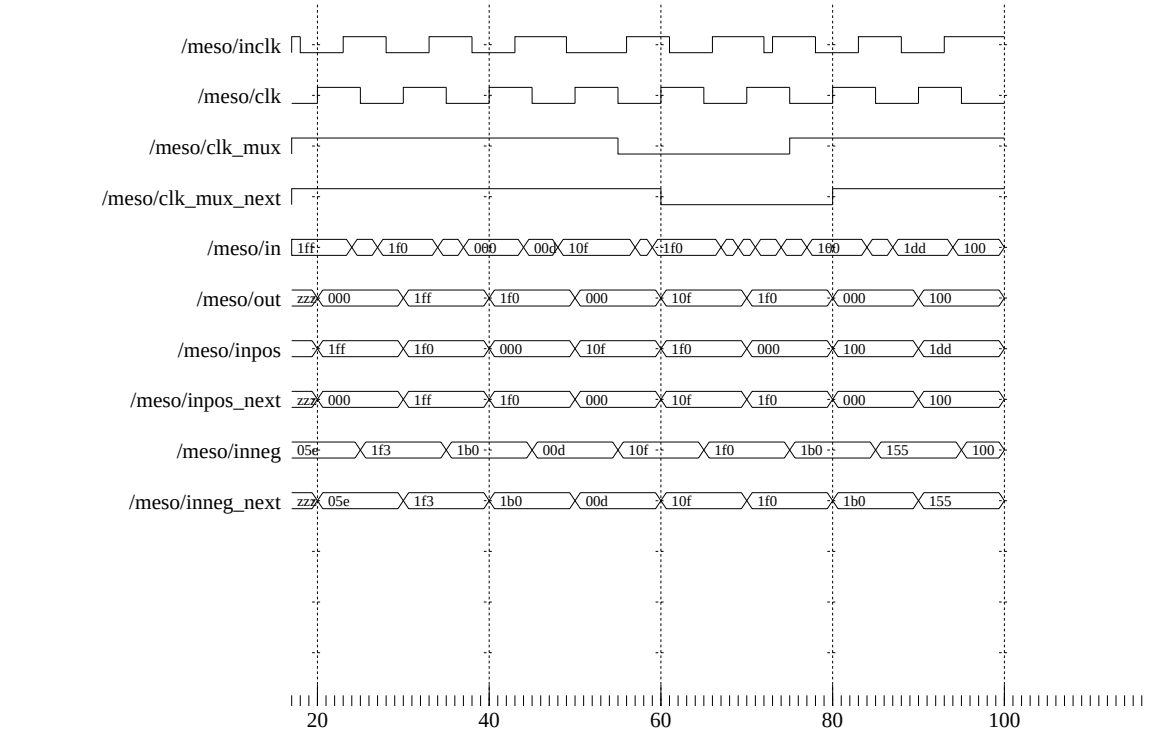

Figure 4-2: Mesochronous Clocking Simulation

the internal wiring resources are very inefficient compared to wiring in a custom ASIC. Since most of the delay on the FPGA is in the wiring[2], we expect that we are losing a significant amount of bandwidth and latency to this effect. In particular, a pipelined version of our design synthesizes at just over 90 MHz, a respectable if not earth-shattering speed. The critical path in this implementation remains the allocation tree logic.

Signaling: On the bright side, however, our relatively low clock rate allows us to expend less effort on the signaling front. At 90MHz, we can use the FPGA's simple I/O drivers to directly drive signals between routing blocks, even across cables. At these speeds, higher performance signaling methods are not mandatory.

Nonetheless, for demonstration purposes, the network will use CTT (center tap terminated)[3] lines as a signaling standard. CTT is a destination-only terminated signaling standard which terminates to a voltage of 1.5V. It uses a V_{cco} of 3.3V, but has sized push-pull driver FETs that look effectively like 8 mA current sources, so the total voltage swing is limited to the incident current wave times the wire impedance (about 800 mV in the case of our cables). A differential comparator in the receiver compares the voltage on the line to a 1.5V reference, with noise margins of about 200 mV.

Packaging and Pin Count: As mentioned before, our design is pin-limited. The XCV1000EFG1156CES has 660 dedicated I/O pins, which can be effectively utilized by a 1 level fat tree with a 16 bit wide data path. This subtree can be elegantly used as a building block for an entire fat tree network. In particular, the wires that cross from the up tree to the down tree are now all on-chip and thus are very fast. Further, they avoid the extra cycle of latency induced by chip to chip mesochronous communication (See Figure 4-3).

Q's Ansible Board: For simplicity of design, Brian Ginsburg is building a board intended to contain a one level fat tree in an FPGA. This board will serve as a

[2]In typical runs, as much as 75% of each path consists of interconnect
[3]JEDEC Standard JESD8-4

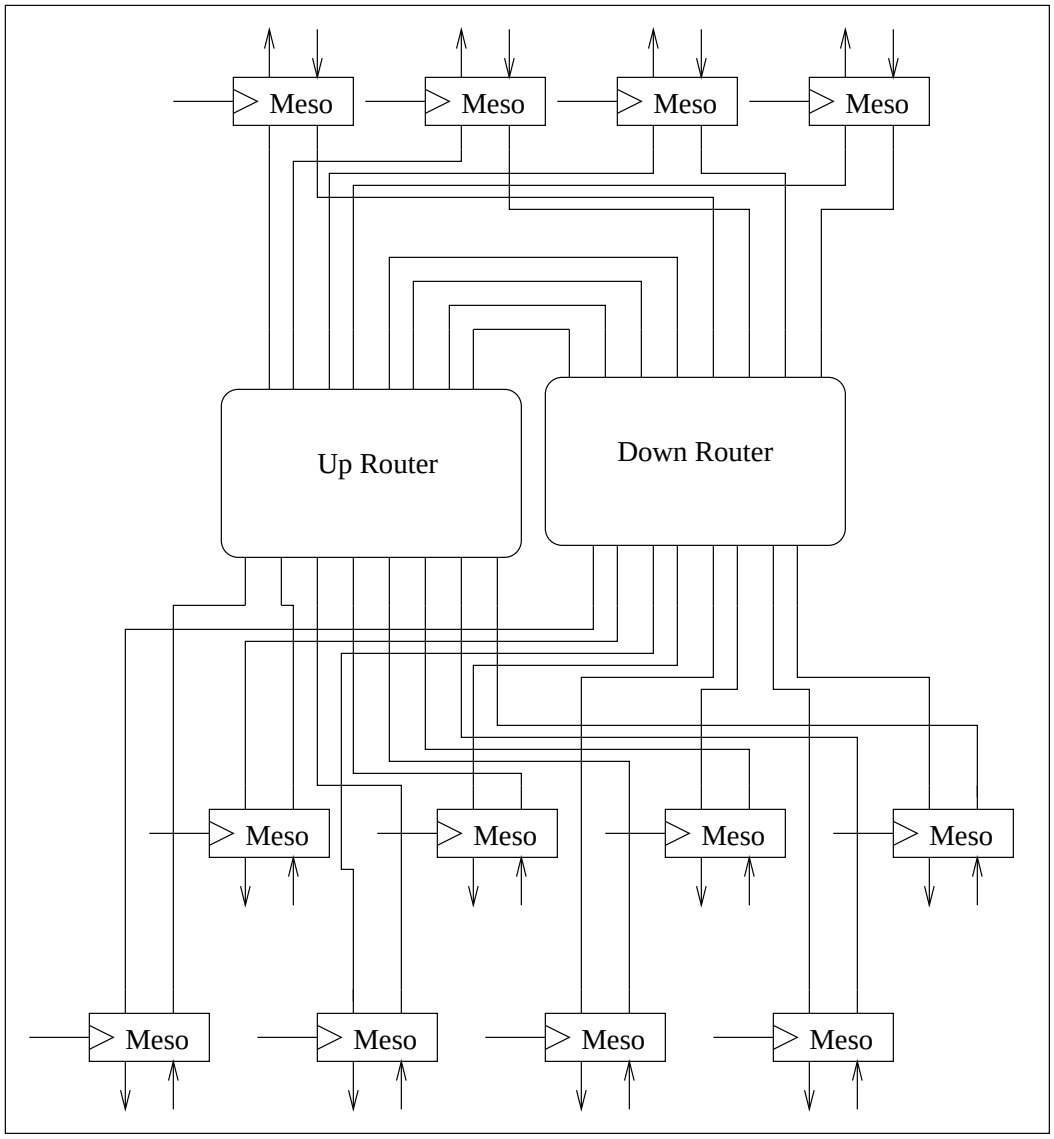

Figure 4-3: One Level Fat Tree with Mesochronous Clocking

proof of concept of this network design, and is designed to offer visibility and control to an out-of band debugging system.

The board's centerpiece is a single Xilinx Virtex-E FPGA device which contains the one-level fat tree. Wires are routed out from the FPGA through series termination resistors to a set of connectors. These connectors will connect via ribbon cables to other similar boards, creating a higher level fat tree network.

The board also contains a StrongARM processor running at 206MHz and 64MB of RAM nearby. These are used for debugging, and are connected to the outside world via a ring topology network running at around 50-100 MHz. The FPGA hangs off of the StrongARM's main bus, and has a 10-bit address space and a 16-bit wide data path to the processor.

Using memory-mapped I/O over this connection, the debugging system can control the network using single-step clocks, and can read internal network state and other debugging information.

This board is not suited for ultra-high performance networking, but rather it is a proof of concept. The flexible debug hardware will allow for detailed analysis of the design and verification of our simulation results.

Cables: The relatively low clock rate of 90MHz allows us to use slower edge rates and thus simple cheap ribbon cables to transmit data between boards. However, we do need to place alternating ground wires between signal wires to mitigate the effects of crosstalk. This is especially important since our wires are likely to all switch logical values simultaneously as new data is sent out onto the bus, and because we are wave-pipelining data and cannot wait for lines to settle.

The number of signals traveling between two routers on a given port is the data width, plus one for the control bit, and one for the backwards drop wire. There is one such path in each direction between any two given routers, and we add a mesochronous clock in each direction, for a total of 38 signals. Adding alternating grounds makes 76 wires. We use one 40-pin ribbon cable with alternating signals and grounds in each direction. By sharing a cable with the signals that

they qualify, the mesochronous clock arrives at the destination at the proper time.

Chapter 5

Hardware Support for Idempotence

Having a basic architecture for our fat tree network, we consider adding features to improve performance. As mentioned before, an idempotent messaging protocol is a desirable thing to have on top of the network. We explore how to design explicit hardware support for such a protocol on top of our fat tree network.

5.1 Motivation

Idempotence is certainly a desirable trait for a messaging scheme. The interesting question is whether it is useful to provide explicit hardware resources to support idempotence, or if it can be done just as well without them.

The idempotent protocol to be used is the three-message protocol described above. The sending processor awaits acknowledgment of each message sent. Upon receipt of the acknowledgment, the original sender sends a third message telling the recipient that the acknowledgment has been received. If an acknowledgment is not received within a certain amount of time, the message is resent.

The receiving processor is responsible for acknowledging messages. If the acknowledgments are not themselves acknowledged within a certain amount of time, the acknowledgments are resent. In the case of a failure which completely prevents

the delivery of the message or acknowledgment, messages will only be attempted a finite but large number of times (a few tens of thousands should be a good indication that communication is unlikely to succeed).

On one hand, explicit hardware support can reduce latency, since all three messages are routed at once. Assuming adequate back wires, this should minimize collisions, and thus total latency. However, these wires have a twofold cost. First, they take up valuable space, enlarging the total network and increasing latency. Second, the routing of these back wires incurs an overhead in the routing block.

5.2 Architecture

The primary architectural modification to support idempotence in hardware is of course the extra wires. Along each existing data path, we add a number of narrower second paths devoted to acknowledgments, and a larger number of narrow third paths devoted to acknowledging the acknowledgments. The exact numbers of paths to add are not obvious and will be discussed further.

The basic procedure for routing a normal message will proceed as usual, except that a second and 3rd path are also allocated through the same set of routers. If the original message is blocked at any point, then we must drop the connection and the associated second and third paths. This information can propagate back through the connection via the second path.

If, however, the message is delivered successfully, the first path is freed, and the second path goes to work and delivers a reply. When the reply is received by the original sender, the third path delivers its reply. Note that the second path is reserved for at least twice as long as the first path, and the third path is reserved for at least three times as long as the first path. It gets even worse when the network becomes large compared to the size of a message, since the first path could have sent out several messages before the reply to the first of the messages is returned.

Which back paths are chosen for a given message can vary at each stage, so the connection state must be maintained locally at each router. This adds complexity to

allocate	0x10Y
idle	0x100
data	0x1XX
drop	0x1XX

Table 5.1: Protocol for 3rd Path (Y = 2nd path ID, X = don't care)

the allocation process and requires physical space near the critical path.

5.3 Implementation

The details of implementing this scheme are complex. First, the allocation mechanism needs to be modified to allocate back paths. Second, the protocol for dropping/interrupting a message needs to be similarly modified.

The allocation process is mostly the same here as in the original design, except that here two additional paths need to be selected and associated with this connection. We could have the message contain routing information about the back paths, but at several bits each, that would create significant overhead. Instead, the incoming third path will have data on it indicating that it is connected to the allocation, as well as data about which second path is associated with the message.

Once the main message routing is determined, we allocate new back paths and maintain connection information locally until the message completes. The selection of back paths within an output port is arbitrary.

A good deal more information must now be stored about each potential connection. Where previous there was the binary state of the connection, now there must be identifiers for both the input and output back paths for each active connection, of which there can be many.

Next there must be support for dropping connections, whether due to completion of a message segment or interruption by a higher priority message. Dropping of the first messages occurs mostly as before. Now however, we need to cancel the back paths as well. The second path carries the drop information backwards, and at each

idle	0x100
data	0x1XX
done	0x000
drop	0x00Y

Table 5.2: Protocol for 2nd Path (Y = 3rd path ID, X = don't care)

stage the third path is thus dropped.

The complexity to implement these changes in the routing block is considerable. Instead of a single bit of information per possible connection, now there must be several bytes. This will inevitably translate into a physically larger routing block, which will require longer wires in the critical path.

This design provides dedicated hardware for the idempotent protocol. The protocol can also be implemented in software/firmware at the network interface to the processors. In this soft implementation, each processor keeps queues of pending first, second, and third messages. Later stage messages are sent first, to avoid overflowing the queues and to reduce total latency. If the network supports priority, later stage messages should be given higher priority than earlier ones.

Also, each processor keeps a table of previously sent messages with timeouts. When replies are received, the messages are removed from the tables, and when timeouts expire, messages are placed back in the appropriate queue to be resent. In the end, the same messages are sent using this soft implementation, except now they are all sent along the same set of wires, rather than along dedicated wires.

5.4 Expected Performance Impact

The size of the network is an important factor in deciding how many back paths to wire. Let n be the ratio of the average latency of the network to the length (in time) of an average first message. To a first approximation, there should be $2n$ second paths and $3n$ third paths per first path to avoid limiting the network with allocation of back paths. If n is close to 1, and we assume that replies are shorter than normal messages

and thus can use much narrower paths, we can build the idempotent network with only a factor of two more wires. But the number of additional wires per path scales as the size of the network, which adversely affects the total latency of the network.

The major performance tradeoff stems from the fact that hardware support for idempotence uses many more wires. If we were to double the number of wires in a normal network, we should expect a $\sqrt[3]{2}$ increase in latency, because the physical length of a wire across the network scales at best as the cube root of the volume, and we expect that wires are volume-limited.

In return, as long as we have adequate reply wires, only one message needs to be routed and sent across the main wires, which is faster and lowers congestion.

There is an additional cost in the overhead of making the now more complicated routing decisions, primarily in that each crosspoint needs to store much more state. This forces the crosspoint grid to be larger, increasing the length of on-chip wires in the critical path of the design, and hurting both latency and bandwidth.

This Page Intentionally Left Blank

Chapter 6

Message Priority

We consider adding the feature of message priority to our network. In the original design, any message that is blocked fails. With priority, if a higher priority message is blocked by a lower priority message, it can override the lower priority message and take its place.

6.1 Motivation

Deadlock and livelock are common problems of large network architectures. For example, if the highest priority is restricted to a devoted kernel thread, simple cases of deadlock and livelock can be avoided in software. Further, it's a convenient hook to the operating system to provide priority. Many applications have their own sense of priority, and would also benefit from this option. For example, a tightly synchronized fault tolerant computer might want to assign synchronization messages a higher priority than normal data traffic.

As mentioned in Section 2.2.7, deadlock and livelock issues surround the use of acknowledgment in an idempotent protocol. Without priority the acknowledgments could keep running into long original messages and dropping, and original messages couldn't be stored at the destination because the queue was full, and no overall progress would be made. With priority, however, acknowledgments are delivered with higher priority, and livelock is much less likely.

Priority does not prevent the theoretical possibility of livelock, but it is a powerful tool that can be used to combat it. In particular, if the highest priority is used all the time, then the original livelock problem recurs. However, if the highest priority is reserved for a kernel task that explicitly is careful to avoid livelock situations, a designer can be less cautious about the use of lower levels of priority while still retaining a way out in case of trouble.

There is also a performance motivation for priority alluded to earlier. Consider a network whose traffic consists of messages and their acknowledgments. With the potential for subsequent messages to block acknowledgments from getting through, a heavily loaded network can accumulate large numbers of messages in transit, which need to be remembered by the senders until they complete. This stored data will eventually have to be a cache, and so minimizing the use of this data can improve latency. We can define acknowledgments to have higher priority than original messages, thus ensuring that in case of high traffic on the network, acknowledgments get through first, lowering the latency of messages.

6.2 Architecture

The main idea behind prioritizing messages is that a higher priority message should be routed over a lower priority message, even if the latter is already in progress. From an architecture standpoint, this means we need to modify the allocation mechanism. Additionally we need to modify the protocol to support dropping a message part way through.

See Figure 6-1 for an example of a new message overriding an existing message. The first two incoming messages request and are granted logical output port 0. In the subsequent cycle, a third message arrives with a higher priority. The lower priority current message is overridden and replaced by the new message.

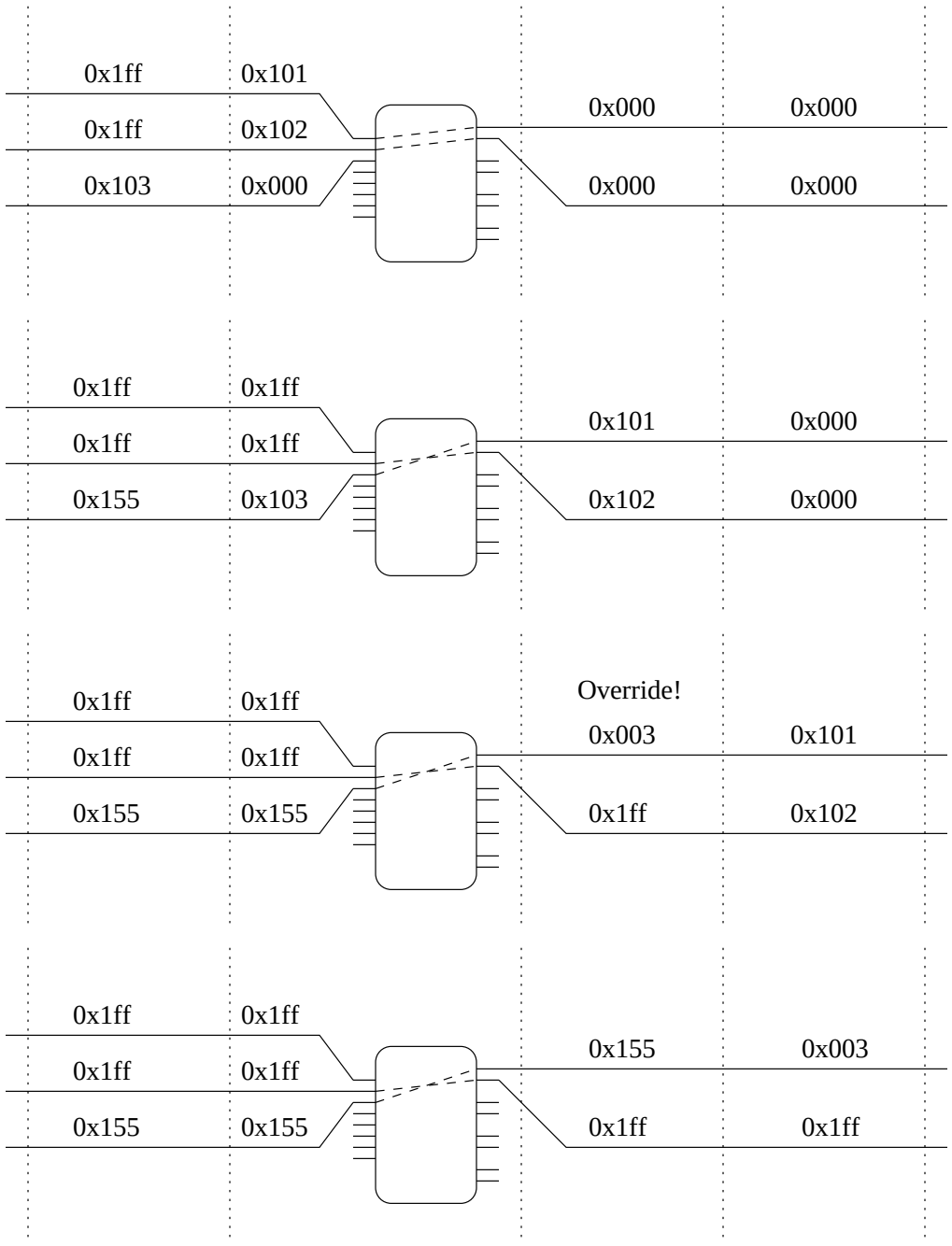

Figure 6-1: Message Priority Example

6.3 Implementation

Several changes need to be made to the basic architecture developed in Chapter 3 to support priority. The first obvious change is that we need to check the relative priority of messages when performing allocation, and also consider the possibility of having to override an existing message. Additionally, we need to modify the routing protocol and the hardware that implements it.

6.3.1 Priority-Based Allocation

The tree-based design described above is well-suited for adaptation to priority. The tree is simply modified to compare 4-bit priority values instead of 1-bit requests. Each logical output maintains the state of its current priority, and routing proceeds as before. Note that this method behaves identically to the original method if all priorities are equivalent.

In the event that a new message overrides the existing message, two things need to happen. First, the new message needs to be routed in place of the old message. Second, the input port associated with the old message is notified of the override. Then the port can forward drop information back to clear the now dead message out of the way.

6.3.2 Routing Protocol

The routing protocol needs to be modified in several ways to support priority. First, the priority information needs to be available in every routing byte for allocation. So, we reserve the least significant four bits of every word for priority information. This necessarily means that each packet can encode fewer stages of routing information.

Consider what happens when a message is overridden. The new message will be sent over the same wires as the previous message in the very next cycle. There needs to be a way to encode the fact that this is a new message, or the next router will not recognize it and route it separately.

We refine the protocol to define (for an 8-bit data word):

drop 0x00 with a low control bit

drop and reroute any other word with a low control bit

Now we can begin a new message and signify the end of the previous message simultaneously. But what happens at the next stage? Consider the following stream of words (including control bit): 0x155,0x0ff,0x134,0x154,0x000. A message routed toward 55 has been interrupted by a message routed toward ff. At the next routing stage the stub 0x155 will be routed toward output port 1, but the remainder of the stream should be rerouted toward output port 3. We need to detect this condition and modify the result on output port 3 to begin with 0x1ff, so that future routing stages interpret the information correctly.

6.4 Expected Performance Impact

It is expected that the use of priority will lengthen the allocation path, because the tree compares 4-bit values instead of single bits. It is further the case that there will be a small but nonzero number of truncated message stubs which continue to worm their way through the network despite no longer being useful. These will contribute to the congestion of the network slightly.

On the other hand, priority should enhance the performance of the idempotent protocol. If third and second messages have higher priority, queues should be smaller, and overall message success rates should be higher.

This Page Intentionally Left Blank

Chapter 7

Results

7.1 Functional Verification

Having created this design, it would be nice to have some assurance that it works as specified before going out to build a thousand-node machine based on it. A series of tests have been run to verify the functionality of this design. A Verilog testbench is used to verify the functionality of the various components of the routing block.

Each input port is tested:

1. Sending a message to each output port
2. Attempting to send a message which is blocked
3. Sending a message which had been overridden in a previous stage
4. Sending messages in quick succession
5. Verifying whether swallowing occurs correctly

The allocation unit is tested:

1. Allocating each input port to each output port
2. Performing multiple simultaneous allocations
3. Overriding with higher priority messages

The routing multiplexers are fully tested by sending a message from each input port through each output port.

A typical waveform showing several of these tests in a priority routing system is shown in Figure 7-1. Note the routing of inputs 5 and 6, which were overridden in the middle in a previous cycle, and also note the swallowing where appropriate.

7.1.1 Timing Verification

Having satisfied ourselves that the initial Verilog description of the design is correct, it still remains to show that the timing of the design works in real hardware. It is all too easy to generate code that simulates correctly, but does not actually work. Thus, the timing-annotated output of the Xilinx FPGA synthesis was also tested for functionality.

Running the timing-annotated output through a similar series of simulations showed that the synthesized design functions identically to the simulated design. The final step of verification will be to test the design in actual hardware, when the hardware becomes available.

7.2 Performance

7.2.1 Basic Network

The performance of a network can be measured using a number of possible metrics. We primarily are concerned with latency, but are also interested in bandwidth. The peak latency and bandwidth are fairly easy to determine. For example, in the FPGA implementation described here, a pipelined design can run as high as 90MHz. The latency can be determined by multiplying the number of stages by the period of a cycle. However, the performance is expected to be significantly less than peak in most usage.

Overall latency of an idempotent message is the sum of the latencies of the three messages that constitute the idempotent protocol. The message latency of each of

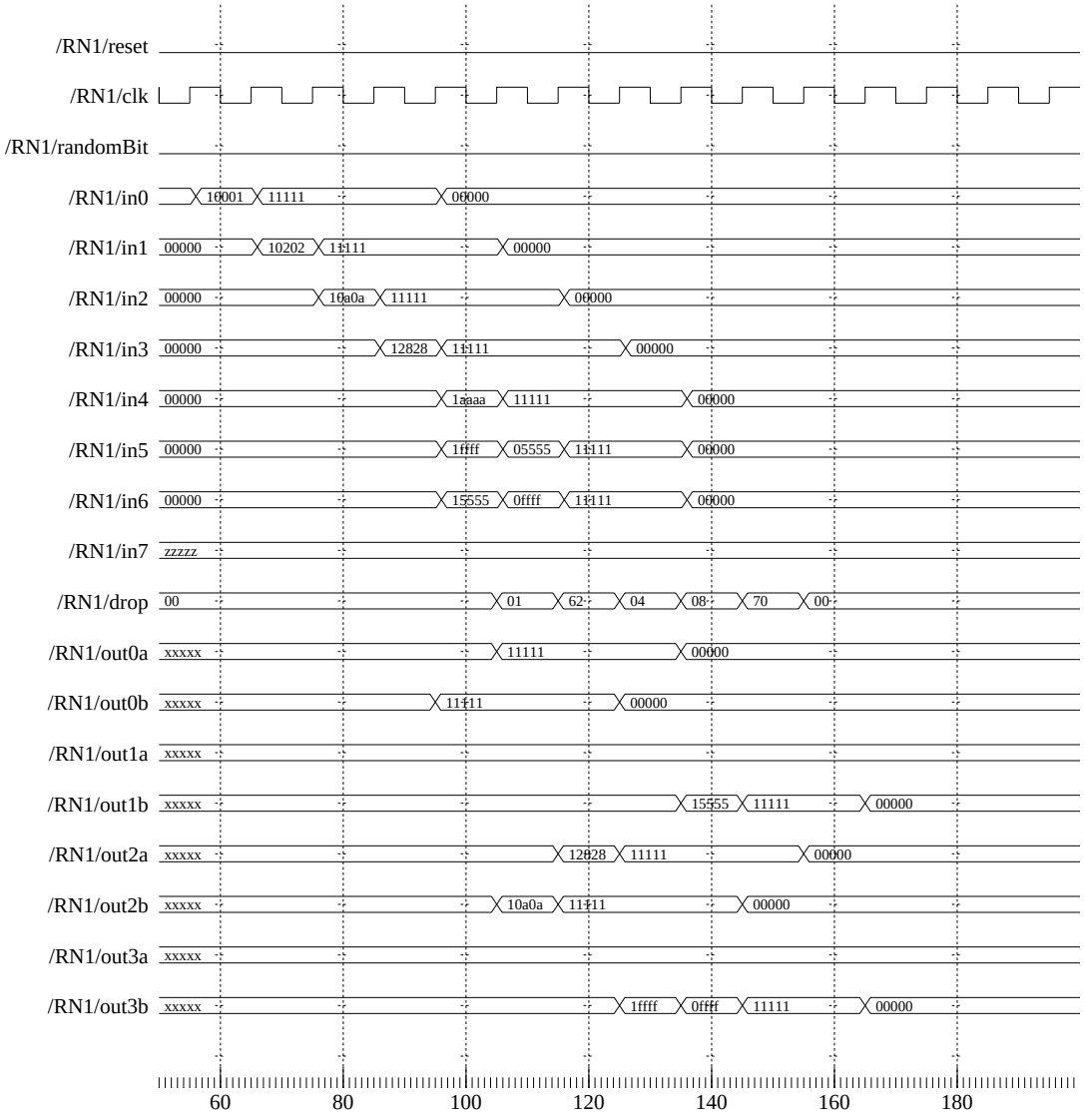

Figure 7-1: Basic Test Waveform

those messages is the average length of a path through the network times the expected number of attempts necessary to deliver the message. Said expected value is the inverse of the probability of a successful message. So the total expected latency can be expressed as the sum of the inverses of the probabilities of success of each of the three types of messages. This is the basic benchmark by which we will evaluate network performance.

7.2.2 Idempotence

In order to quantitatively evaluate the modifications for hardware support for idempotent messaging, a network simulation was built in Verilog. The measured performance of the network is then adjusted by the expected hardware overhead. A comparison of the resulting data indicates whether the modification is worthwhile, and under what conditions.

We simulated a four level fat tree, with each of 256 processors having 2 in and 2 out ports. Measuring the performance of the idempotent network, a first approximation is that the back paths are always successfully allocated. Thus, the resulting latency and bandwidth depend entirely on the routing of first messages in a normal network, adjusted for the overhead of routing complexity.

Our modification for hardware idempotence should be compared to a basic network which implements the idempotent protocol offline in the network interface of each processor. This is simulated in a behavioral Verilog model for each processor.

Each processor has three queues of messages, one for each phase of the protocol. Later messages are sent with higher priority. The processor adds acknowledgments to the appropriate queues upon receipt of messages. The processor additionally has a timeout mechanism to resend messages when replies are not received soon enough.

In a four level fat tree, an average message will traverse 4 routers, 8 cycles in total with no pipelining. To first order, in this network the message length is comparable to the size of the network. Suppose that replies contain 20% of the data that an original message does. Then we can make the reply paths five times narrower and have similar bandwidth. Considering that second message paths sit idle during the

first message, and third paths sit idle during both of the first two messages, we need twice as many second paths and three times as many third paths to avoid constraining the network with reply path availability. So even ignoring concerns of message failures and overhead, hardware support for idempotence would require twice the total number of wires of a standard network. This corresponds to a $\sqrt[3]{2}$ increase in total latency, and probably closer to $\sqrt{2}$ at the lower stages of the network which we expect to be constrained to two-dimensions.

Additionally, the overhead to implement the routing hardware decreases bandwidth for nearly a factor of 2, according to the timing report from the Xilinx synthesis tools. Careful optimization could lessen the overhead somewhat, but the bandwidth will be lower with hardware support for idempotence.

On the other side of the coin, there fewer messages in the primary network, since replies are confined to their dedicated wires. Our analysis shows a 56% increase in the overall message bandwidth without the presence of the short reply messages (more messages can now be sent per unit time)[1]. The overall message latency is improved by around 10% (due to the fact that 2nd and 3rd messages never fail). See Figure 7-2 for the effect of idempotence on message success rates. However, the latency needs to be adjusted 25% slower due to the $\sqrt[3]{2}$ effect from the additional wires. The improvement in latency due to the hardware support for idempotence is insufficient to justify the extra wires.

7.2.3 Priority

Priority by itself is difficult to analyze from a performance perspective. It is unclear what metric to use and what type of traffic to assume. However, there is a natural application of priority to the idempotent messaging protocol. We analyze the performance effect of prioritizing acknowledgments higher than original messages, and compare to the basic network in which all three types of messages are of the same priority. Figure 7-3 shows the effect of priority on the success rates of messages.

[1] 2598 messages received with the extra wires vs. 1659 first messages received without the extra wires

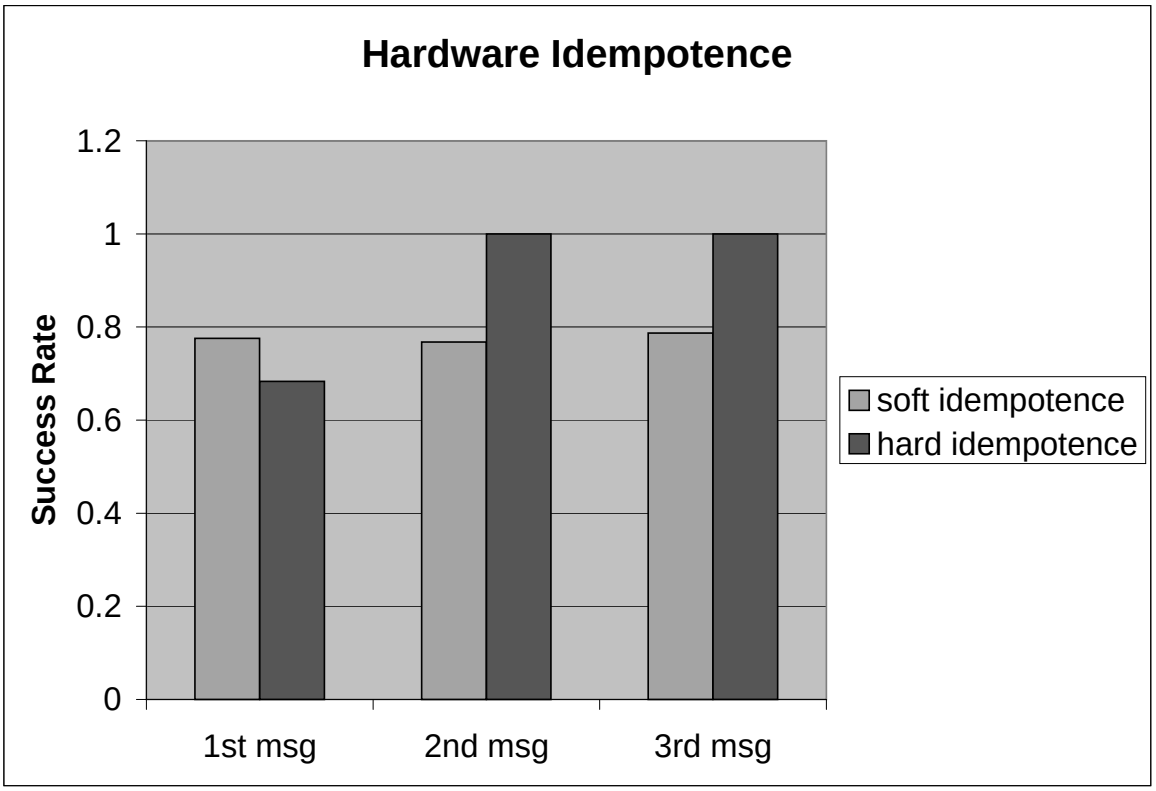

Figure 7-2: Hard-Wired Idempotence Performance

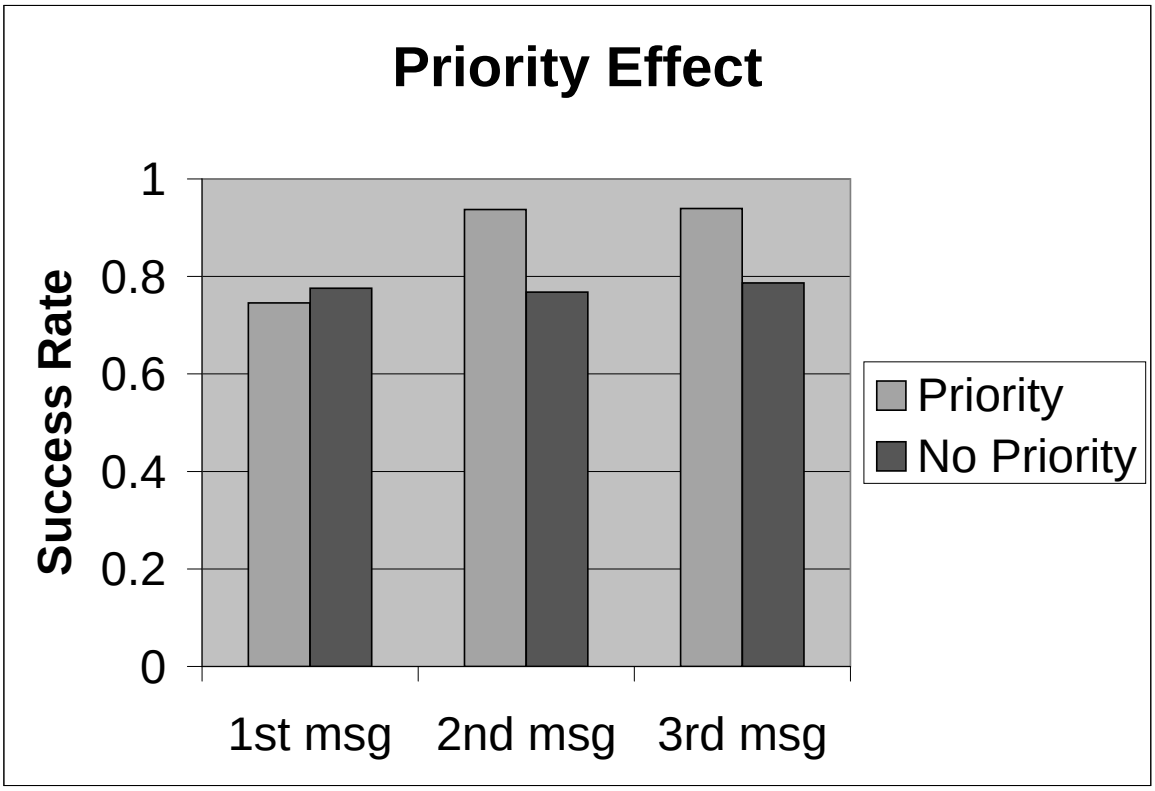

Figure 7-3: Priority Performance

In the normal network, all three types of messages have similar success rates, about 77% for a reasonable size and frequency of messages. With the same network traffic in a prioritized network, first message success rates decline to 74% due to the cases where a later message will override a first message, but second and third message success rates climb to 94% each because of the overriding effect. As a result, the total expected latency decreases 10% due to the priority change. This is a fairly significant improvement, and is worth the bandwidth overhead for most applications.

7.2.4 Other Factors

There are a number of other variables which can affect the performance of the network. We examine the effects of varying message size, message density, and the expected distribution of recipients.

Message Distribution

Each application will have its own distinct network traffic patterns. However, we can break down the possible network traffic types into two major types: those which exploit locality and those which don't exploit locality.

We examine the difference in performance of the network between fully randomized messages and localized random messages. The network performs rather poorly on fully random message traffic, because although the root of the tree is wide, it is not wide enough to handle the expected 50% of all messages that would be routed through it. Thus the root is a major bottleneck, and the network is inefficient.

However, we expect that realistic usage of this type of network will fall closer to a second type of message traffic, which assumes some locality of computation. In this type, half of a processor's messages are local to the first stage router. Half of the remaining are local to the second stage router, and so forth recursively until the root of the tree is reached.

Figure 7-4 shows the difference in message success rates for the two types of message distribution. The fully random messages succeed only 23% of the time in total, due to the very low success rates for first and second messages, compared with 77% of the time for the localized messages. The expected latency for the fully random messages is nearly three times as high as the localized messages.

Message Length

One would expect that shorter messages should have a greater success rate than longer messages for several reasons. First, a failed long message leaves a larger stub in the network, which can block other messages before petering out. Second, longer messages imply greater total usage of the network and more potential for collisions.

Figure 7-5 shows the effect of message length on message success rate. A network filled with messages 21 words long had a success rate of 74%, and a network filled with 7 word messages had a success rate of 86%. At those two extremes, the difference accounted for an 17% difference in total expected latency.

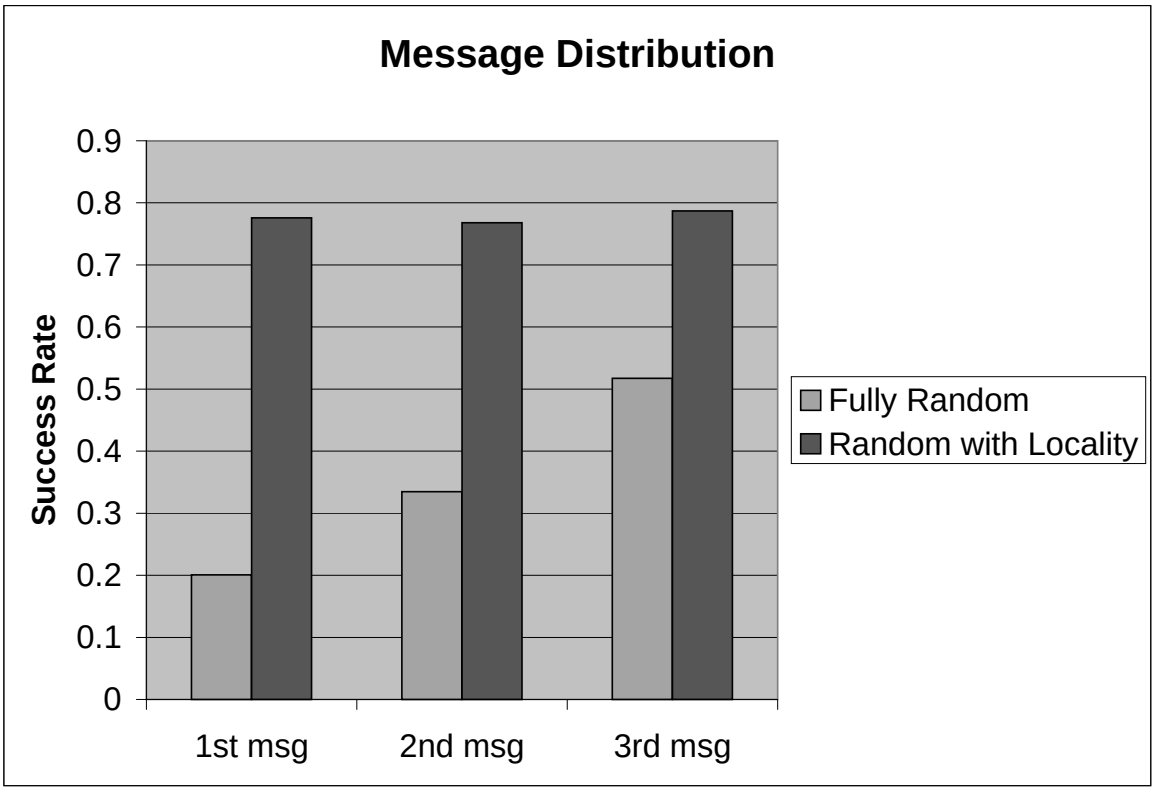

Figure 7-4: Message Distribution

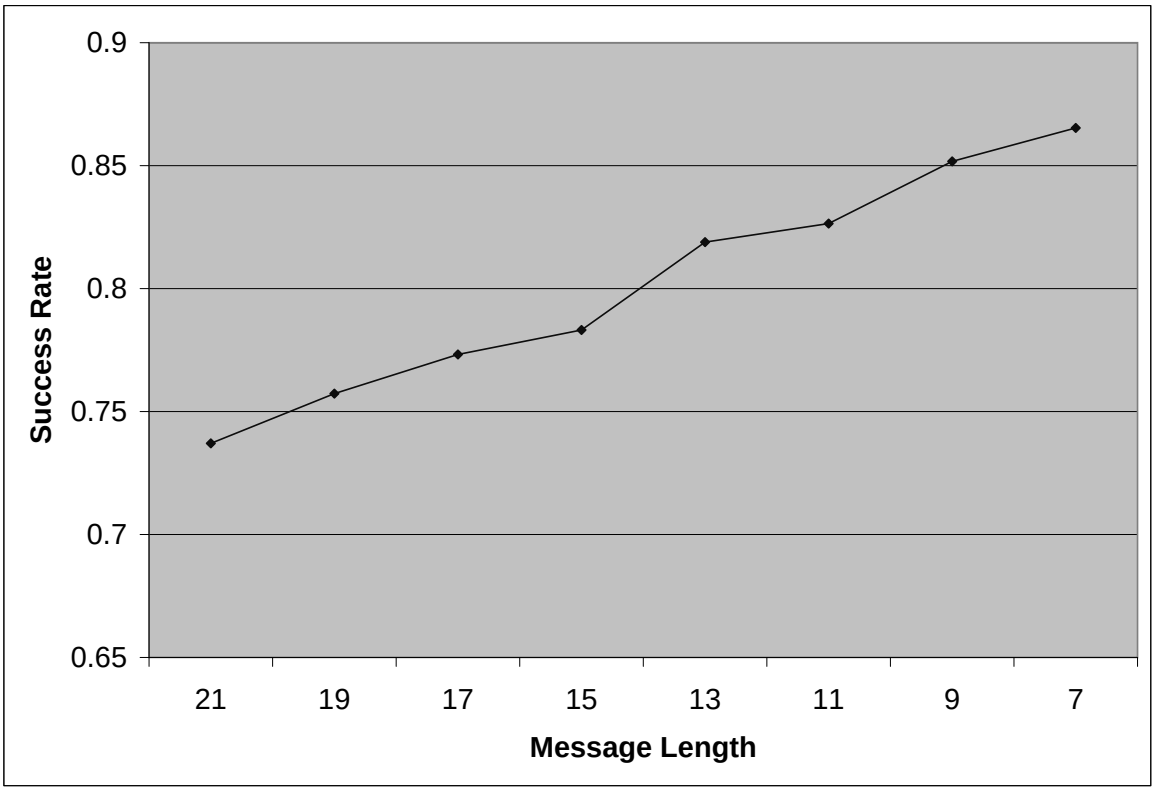

Figure 7-5: Message Length

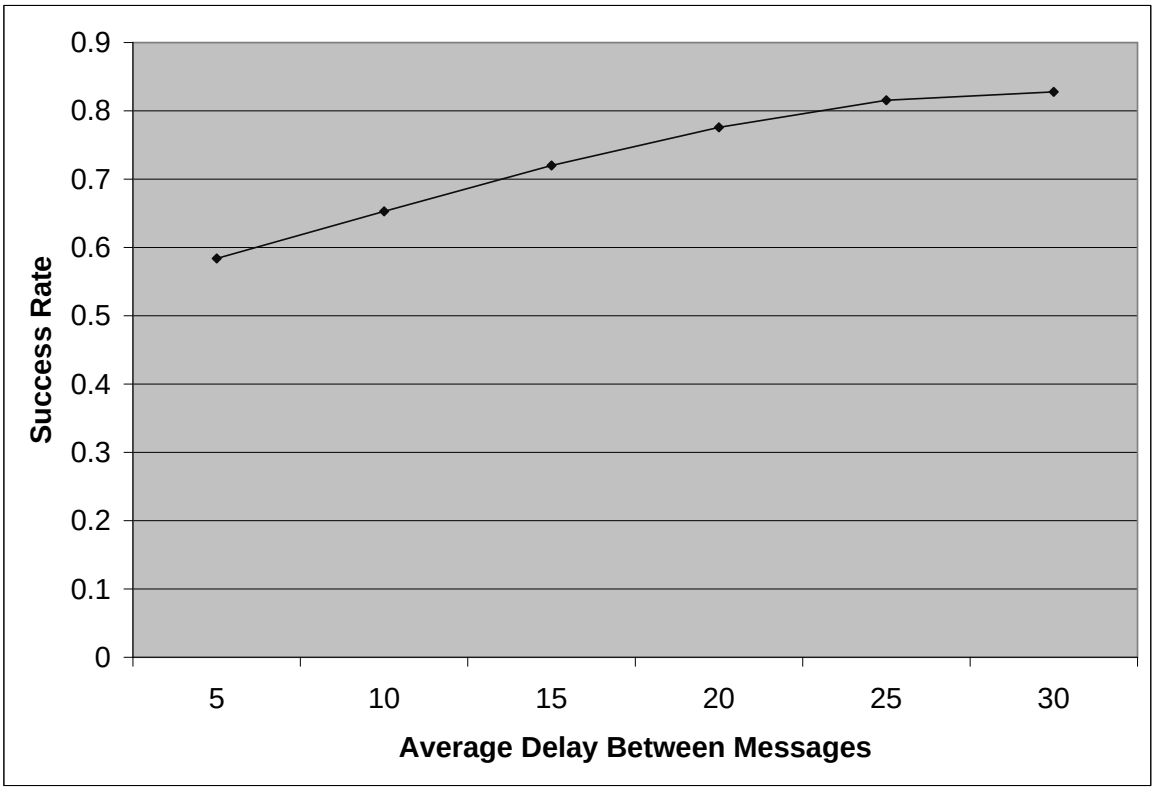

Figure 7-6: Message Density

Message Density

Message density has similar properties to message length. Increasing the average delay between messages has a positive effect on the success rate of the messages. Figure 7-6 shows the effect of varying the delay between messages on success rate. Messages in this simulation are 10 cycles long. Ranging the average delay between half the message length and three times the message length induced a corresponding change in success rate from 58% to 82%. As the delay increases beyond three times the message length, the additional benefit becomes small.

This Page Intentionally Left Blank

Chapter 8

Conclusions

When building large multiprocessing systems, the network becomes increasingly important as the system grows in size. In particular, network latency becomes a significant limiting factor. But many networks today are built with an eye only to bandwidth, often at the expense of latency.

A latency-aware network architecture should use a topology which is physically realizable and takes advantage of locality of communication. It should be composed of small and fast routing components, decreasing latency both by shrinking the total network size and by moving the data through the routing components more quickly.

Our design, based on Minsky's RN1 and the METRO network, modified for a fat tree topology, addresses these needs. A new tree-based allocation mechanism shortens the critical path of allocation.

Two further modifications were considered. First, the possibility of hardware support for idempotence by means of dedicated reply wires was analyzed. The results suggest that in its current form, such specialized wiring is not worth the overhead, and as the network gets larger, the penalty increases. A latency increase of 10% can be achieved, but at a cost of around double the total number of wires, which itself is expected to increase latency by 25%.

Second, a scheme for prioritizing messages was developed and simulated. This scheme has a slight bandwidth penalty, but drastically increases the success rate of messages when using the idempotent protocol. The use of prioritized messages causes

a 10% decrease in total latency. The ability to prioritize messages also provides valuable features to the operating system.

A prototype of the design was simulated and synthesized, and it will be tested in physical hardware. The physical implementation will be targeting Xilinx Virtex-E FPGAs, and synthesized supporting clock rates as high as 90MHz for pipelined versions.

If a resource-laden design team were to build this network, a very high performance would be achievable, as the core architecture is intended not for the proof of concept being built now, but for a large scale high performance massively scaled computing system. Techniques were discussed to use advanced packaging, signaling, and integration to reach high bandwidth while maintaining the very low latency inherent in the architecture. With 16 8-bit wide ports per router, each at 90MHz, the design has a peak data rate of around 3Gb/s at each router. Taking this architecture to a resource intensive full custom process has the potential to achieve well over an order of magnitude higher bandwidth. The promise of such performance justifies further research and development of fat tree based networks for multiprocessing.

Bibliography

[1] L. A. Bassalygo and M. S. Pinsker. "Complexity of an Optimum Nonblocking Switching Network Without Reconnections". *Problems of Information Transmission*, 9(3):64–66, 1974.

[2] Jeremy H. Brown. "An Idempotent Message Protocol". Aries Memo no. 014, MIT AI Lab, May 2001.

[3] Frederic T. Chong, Eric A. Brewer, F. Thomson Leighton, and Thomas F. Knight. "Building a Better Butterfly: The Multiplexed Metabutterfly". In *International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 65–72, 1994.

[4] W. J. Dally. "Performance Analysis of k-ary n-cube Interconnection Networks". *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[5] A. DeHon, F. Chong, M. Becker, E. Egozy, H. Minsky, S. Peretz, and T. F. Knight. "METRO: A Router Architecture for High-Performance Short-Haul Routing Networks". *Proceedings of the 21st International Symposium on Computer Architecture*, pages 266–277, 1994.

[6] André DeHon. "Fat-Tree Routing for Transit". Technical Report 1224, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, February 1990.

[7] André DeHon. *Robust, High-Speed Network Design for Large Scale Multipro-*

cessing. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1993.

[8] André DeHon, Thomas F. Knight, and T. Simon. "Automatic Impedance Control". *Proceedings of the International Solid-State Circuits Conference*, pages 164–5, 1993.

[9] D. Dolev. "The Byzantine Generals Strike Again". *Journal of Algorithms*, 3:14–30, 1982.

[10] R.E. Harper, J.H. Lala, and J.J. Deyst. "Fault Tolerant Parallel Processor Architecture Overview". *18th Intl. Symp. on Fault-Tolerant Computing*, pages 252–257, June 1988.

[11] Thomas F. Knight. "Transit: Reliable High Speed Interconnection Technology". In *International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 350–357, 1994.

[12] B. Landman and R. L. Russo. "On a Pin vs. Block Relationship for Partitioning of Logic Graphs". *IEEE Transactions on Computers*, C-20(12):1469–1479, December 1971.

[13] Charles E. Leiserson. "Fat-Trees: Universal Networks for Hardware Efficient Supercomputing". *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[14] Henry Minsky. A parallel crossbar routing chip for a shared memory multiprocessor. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1991.

[15] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*, pages 126–144. Prentice Hall, 1996.

[16] Gill A. Pratt and John Nguyen. "Distributed Synchronous Clocking". *IEEE Transactions on Parallel and Distributed Systems*, 6(3):314–328, March 1995.

[17] I-Ling Yen. "Specialized N-modular Redundant Processors in Large-Scale Distributed Systems". *Proceedings of 15th Symposium on Reliable Distributed Systems*, pages 12–21, 1996.