ABSTRACT

Title of dissertation: MESH-OF-TREES INTERCONNECTION NETWORK FOR AN EXPLICITLY MULTI-THREADED PARALLEL COMPUTER ARCHITECTURE

Aydin Osman Balkan
Doctor of Philosophy, 2008

Dissertation directed by: Professor Uzi Vishkin
Department of Electrical and Computer Engineering

As the multiple-decade long increase in clock rates starts to slow down, mainstream general-purpose processors evolve towards single-chip parallel processing. On-chip interconnection networks are essential components of such machines, supporting the communication between processors and the memory system. This task is especially challenging for some easy-to-program parallel computers, which are designed with performance-demanding memory systems.

This study proposes an interconnection network, with a novel implementation of the Mesh-of-Trees (MoT) topology. The MoT network is evaluated relative to metrics such as wire area complexity, total register count, bandwidth, network diameter, single switch delay, maximum throughput per area, trade-offs between throughput and latency, and post-layout performance. It is also compared with some other traditional network topologies, such as mesh, ring, hypercube, butterfly, fat trees, butterfly fat trees, and replicated butterfly networks. Concrete results

show that MoT provides higher throughput and lower latency especially when the input traffic (or the on-chip parallelism) is high, at comparable area cost. The layout of MoT network is evaluated using standard cell design methodology. A prototype chip with 8-terminal MoT network was taped out at 90nm technology and tested. In the context of an easy-to-program single-chip parallel processor, MoT network is embedded in the eXplicit Multi-Threading (XMT) architecture, and evaluated by running parallel applications. In addition to the basic MoT architecture, a novel hybrid extension of MoT is proposed, which allows significant area savings with a small reduction in throughput.

MESH-OF-TREES INTERCONNECTION NETWORK FOR AN EXPLICITLY MULTI-THREADED PARALLEL COMPUTER ARCHITECTURE

by

Aydin Osman Balkan

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor Uzi Vishkin, Chair/Advisor
Associate Professor Clyde P. Kruskal
Professor Steven Nowick
Professor Martin Peckerar
Associate Professor Donald Yeung

© Copyright by
Aydin Osman Balkan
2008

Dedication

This dissertation is dedicated to my parents:

Y. Afet Balkan

Örsçelik Balkan

I am eternally grateful.

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Uzi Vishkin, for providing me this opportunity for contributing to the XMT project. I also thank him for providing a creative and challenging research environment that allowed me to improve myself.

I am also very grateful to my co-advisor Prof. Gang Qu, for his guidance and directions that led me to high quality results and publications. I am a better researcher today, thanks to our conversations and technical discussions.

I would like to thank my Ph.D. dissertation committee members, Prof. Clyde P. Kruskal, Prof. Steven Nowick, Prof. Martin Peckerar, and Prof. Donald Yeung, for their time, effort, and comments that improved this dissertation.

I would like to specially thank Prof. Steven Nowick, for traveling from New York to participate my defense. I also appreciate his hospitality at Columbia University, and our fruitful discussions.

I also specially thank Prof. Martin Peckerar, for his help in our layout efforts.

I would like to thank all XMT research team members for their friendship and collaboration during my studies. Especially, I thank Michael Horak, for his tireless efforts, and positive attitude during our long hours with of chip layout tools. I thank Xingzhi Wen, for his collaboration in integrating my work into the XMT processor. I also thank Mary Kiemb for her contributions in our testing efforts.

I would like to thank ECE Chair Prof. Patrick O'Shea, and all ECE staff for providing a peaceful and productive research environment. I also thank all

my friends at University of Maryland, for their friendship and support during my studies.

I would like to thank my parents, Afet and Örsçelik, and my brother Gökçe, for providing a sympathetic and patient ear, whenever I needed one, and for their unconditional love.

Last, but not least, I would like to thank Sanem Koçak, for her friendship, her companionship; and for being my beacon, when I felt lost.

Table of Contents

List of Tables ix

List of Figures x

List of Abbreviations xiii

1 Introduction 1

2 Background and Related Work 5
- 2.1 PRAM-On-Chip Vision 5
- 2.2 Underlying Memory Model 6
- 2.3 Review of Existing Interconnection Network Models 9
  - 2.3.1 Definitions 9
  - 2.3.2 Bus 11
  - 2.3.3 Crossbar 12
  - 2.3.4 Fat Tree Networks 13
  - 2.3.5 Mesh Networks 14
  - 2.3.6 Ring Networks 14
  - 2.3.7 Hypercube Networks 17
  - 2.3.8 Butterfly Networks 17
- 2.4 Performance Improvement with Additional Resources 19
  - 2.4.1 Virtual-Channel Routers 19
  - 2.4.2 Virtual Output Queuing and Buffered Crossbars 20
  - 2.4.3 Tuned Butterfly Networks 22
- 2.5 Deficiency of the Existing Interconnection Networks 23
  - 2.5.1 Interference 23
  - 2.5.2 Global Synchronization 23
- 2.6 Advantages of MoT Network 23
- 2.7 Earlier Implementations of Mesh-of-Trees Network 24

3 General Methodology of Evaluation 26
- 3.1 Introduction 26
- 3.2 Topology Evaluation 27
  - 3.2.1 Wire Area Complexity 27
  - 3.2.2 Register Count 28
  - 3.2.3 Bisection Bandwidth 30
  - 3.2.4 Network Diameter 30
  - 3.2.5 Deadlock 31
- 3.3 Switch Evaluation 31
  - 3.3.1 Modeling Interconnection Network Components as Queues 31
  - 3.3.2 Hardware Models 33
  - 3.3.3 Switch Delay 34
- 3.4 Network Performance Evaluation by Simulation 35

3.4.1 The Network Simulator 35
3.4.2 Artificially Generated Traffic 36
3.5 Layout Evaluation 39
3.5.1 Layout Design and Verification 39
3.5.2 Cycle-Accurate Validation 39
3.5.3 Physical Testing of Network Chip 40
3.6 Mesh-of-Trees Network in XMT Context 43

4 Mesh-of-Trees Interconnection Network 44
4.1 Introduction 44
4.2 Topology 44
4.3 Routing 47
4.4 Flow Control 47
4.5 Floorplan 49
4.6 Differences with Existing MoT Implementations 50
4.7 Evaluation 51
4.7.1 Wire Area Complexity 51
4.7.2 Register Count 53
4.7.3 Bisection Bandwidth 57
4.7.4 Network Diameter 57
4.7.5 Deadlock 58
4.7.6 Interference 59
4.8 Summary 62

5 Switches of MoT Network 63
5.1 Introduction 63
5.2 Queue Model of MoT Network 64
5.3 Earlier Arbitrate-and-Move Primitive Implementations 66
5.3.1 Asynchronous Implementation 68
5.3.2 Reduced Synchrony Implementation 68
5.3.2.1 Static Gate Implementation (RS-Static) 70
5.3.2.2 Dynamic Gate Implementation (RS-Dynamic) 71
5.3.3 Simulation Results 72
5.3.4 Discussion 74
5.4 Synchronous Switch Primitives 77
5.4.1 Pipeline Primitive 78
5.4.2 Routing Primitive 80
5.4.3 Arbitration Primitive 82
5.4.3.1 Arbitration Method 83
5.4.3.2 N-input to 1-output Arbitration 85
5.4.3.3 Winner-Take-All Arbitration for “Store” Operations 86
5.4.4 Butterfly Primitive 88
5.5 Evaluation 89
5.5.1 Logic Delay of Switch Primitives 89
5.5.2 Maximum Network Throughput 92

5.5.3 Throughput and Latency Under Varying Traffic 95
5.6 Summary 98

6 Layout 99
6.1 Introduction 99
6.2 Network Layout 100
6.2.1 Terminal Circuits 102
6.2.2 Pipeline Insertion 102
6.3 Results and Discussion 104
6.3.1 Simulation Results 104
6.3.2 Layout Results 108
6.4 Physical Testing 110
6.4.1 Lessons Learned 113

7 Area Improvement Through Hybridization 115
7.1 Introduction 115
7.2 Hybrid MoT Network 116
7.2.1 Network Architecture 116
7.3 Evaluation 119
7.3.1 Register Count 119
7.3.2 Minimum Latency 121
7.3.3 Throughput-Area Trade-off 122
7.3.4 Latency and Throughput vs. Traffic 123
7.3.5 Post-Layout Throughput 125
7.4 Summary 126

8 MoT Network as Part of XMT Parallel Processor 128
8.1 Deadlock 128
8.1.1 Conditions for Deadlock 129
8.1.2 Deadlock Prevention Methods for XMT 131
8.1.3 Cost of Deadlock Prevention 132
8.1.4 Summary 134
8.2 Application Simulation on XMT 135
8.2.1 Application Traffic and Execution Time 137
8.3 Layout of XMT ASIC chip 139

9 Discussion 142
9.1 Limiting Factors for Clock Rate 142
9.1.1 Clock Rate Decrease Between Development Stages 142
9.1.2 Limitations of Standard-Cell Design Method 145
9.2 Potential Impact of Multi-GHz Operation 146
9.2.1 Case Study 147
9.3 Applicability to Other Systems 151

10 Future Directions and Conclusion 154
10.1 Future Directions 154
10.2 Conclusion 161

A MoT Network in XMT Architecture 167

Bibliography 171

List of Tables

4.1 Asymptotic area comparison of networks. 56

4.2 Asymptotic diameter comparison of networks. 58

5.1 Comparison of asynchronous and reduced synchrony arbitrate-and-move circuits. 75

5.2 Comparison of N-to-1 arbitration methods. 86

5.3 Single switch delay of various networks (in FO4). Replicated butterfly and MoT do not have virtual channels. 91

5.4 Maximum throughput 94

6.1 Wire and cell area (in mm^2). 100

6.2 Simulation results for different network configurations. 106

6.3 Layout Results 108

7.1 Register count of some hybrid MoT-BF networks normalized to MoT with same number of terminals. 121

8.1 Comparison of deadlock avoidance methods. 135

8.2 Simulation Results for Execution Time and Traffic Rate. 139

9.1 Performance improvement with square floorplan. 143

9.2 Benefits of high-frequency network operation. 150

10.1 Summary of important results in absulute terms (quantitative measurements). 165

10.2 Summary of important results in relative terms (comparison to other networks). 166

List of Figures

2.1 Memory System 7

2.2 Bus with $N = 8$ terminals. 11

2.3 Crossbar with 3 source and 4 destination terminals. 13

2.4 Two types of fat trees with constant switch size. (a) k-ary n-tree with $k = 2, n = 4, N = k^n = 16$; (b) Butterfly Fat Tree with $N = 16$. . . . 15

2.5 (a) Ring, and (b)4×4 2-dimensional mesh topologies with $N = 16$. . 16

2.6 A physical implementation of 4-dimensional hypercube. 18

2.7 (a) Butterfly network and (b) its layout with $N = 8$ PCs as shown in [107]. 18

2.8 4×4 crossbar with virtual output queues (VOQ). 21

2.9 4×4 Combined Input and Crosspoint Queued (CICQ) crossbar. . . . 21

3.1 A system with queue buffers, and server with arrival and service parameters λ and μ. 32

3.2 Markov chain representation of a queue with arrival and service parameters λ and μ. 32

3.3 Setup for testing network chip. 41

4.1 Mesh of Trees with 4 Clusters and 4 Memory Modules 45

4.2 Possible implementation of network terminal node. 46

4.3 Switch primitives of MoT network. 48

4.4 Mesh-of-Trees network floorplan 50

5.1 Queue model of fan-out tree. 65

5.2 Asynchronous arbitrate-and-move primitive. 69

5.3 Reduced-synchrony arbitrate-and-move primitive. 70

5.4 Schematic of RS-Static arbitrate-and-move circuit 71

5.5 Schematic of RS-Dynamic arbitrate-and-move circuit 72

5.6 Operation of reduced synchrony arbitrate-and-move circuits 73

5.7 Pipeline primitive. 79

5.8 Block diagram of pipeline primitive. 80

5.9 Block diagram of routing primitive. 81

5.10 Block diagram of arbitration primitive. 83

5.11 Block diagram of butterfly primitive. 88

5.12 Register-to-output delay of routing primitive with different number of outputs. 92

5.13 Cost-performance comparison of networks. 96

5.14 Throughput and latency of various networks for $N = 64$ terminals. . . 97

6.1 Wire and cell areas for $90nm$ and $32nm$ technology nodes. 101

6.2 High level chip floorplan for 8-terminal network. 102

6.3 Throughput, and latency of 64-terminal MoT. 105

6.4 64-terminal MoT simulation results for different methods of handling store operations. 107

6.5 Final layout of 8-terminal chip. 109

6.6 Die photo of 8-terminal chip. 111

6.7 Simulation output. 111

6.8 Test output 1. 112

6.9 Test output 2. 113

7.1 Butterfly, Mesh-of-Trees, and Hybrid Networks. 117

7.2 Cost-performance comparison of different network configurations. . . 122

7.3 Latency and throughput of 64-terminal hybrid networks. 124

7.4 Post-layout throughput of MoT, replicated BF and MoT-h-BF networks. 125

8.1 Single MoT network in XMT. 129

8.2 Two-network configuration of XMT with 4 PCs and 4 MMs. 132

8.3 $10mm \times 10mm$ layout of XMT ASIC chip with 64 processors. 140

8.4 Modules of XMT ASIC chip. 141

A.1 Block diagram of synchronous network in XMT. 168

A.2 Block diagram of asynchronous network in XMT. 170

List of Abbreviations

ASIC	Application Specific Integrated Circuit
BF	Butterfly
FO4	Technology-independent delay of inverter driving 4 identical inverters
fpc	Flits per cycle
LSTRM	Length of Sequence of Round-Trips to Memory
MoT	Mesh-of-Trees
MM	Memory Module
PC	Processing Cluster
PRAM	Parallel Random Access Machine/Model
QD	Queuing Delay
RTM	Round-Trip to Memory
XMT	eXplicit Multi-Threading
N	Number of terminals of interconnection network
λ	Traffic rate in terms of flits per cycle per port

Chapter 1

Introduction

The advent of the Billion-transistor chip era coupled with a slow down in clock rate improvement brought about a growing interest in parallel computing. Ongoing expansion in the demands of scientific and commercial computing workloads also contributes to this growth in interests. To date, the outreach of parallel computing has fallen short of historical expectations. This has primarily been attributed to programmability shortcomings of parallel computers.

The Parallel Random Access Model (PRAM) is an easy model for parallel algorithmic thinking and for programming. Given a task, PRAM allows users to express its parallelism independently of the underlying architecture. As a consequence, PRAM has been considered as the desired model of parallel computing during 1980s and 1990s. On the other hand, PRAM assumes that a high-performance shared memory system with Uniform Memory Access (UMA) feature is available to support high level of parallelism. Historically, several challenges, including the ones in implementing such a system, led to decline of attention to PRAM. Nevertheless, experts noted that building a computer "that can look to the programmer like a PRAM" is an achievement of revolutionary magnitude [24].

The PRAM-on-Chip vision at University of Maryland aims to build a single-chip multi-core parallel processor, called *eXplicit Multi-Threading (XMT)* architec-

ture, that looks to the programmer like a PRAM [12, 71, 101, 102, 104, 105]. To handle the high level of parallelism and memory access needed for a PRAM-like architecture, XMT uses a memory architecture where partitioning of data memory starts from the first level of the on-chip cache [72]. It is a challenging task to build a high-throughput low-latency interconnection network on such architecture. A not well designed interconnection network may create many on-chip queuing bottlenecks when concurrent read and/or write requests are issued to the memory, consume large portions of die area, or suffer from long wire delays. Such problems can significantly hurt the overall system performance.

We study the interconnection network design problem for a memory architecture designed to achieve high single-chip parallelism. We propose using the Mesh-of-Trees (MoT) topology with a novel approach of implementation. As a result, we achieve high bandwidth, high throughput, high operating frequencies and low latency.

We follow a top-down design, and bottom-up implementation strategy. In other words, we start our evaluations at high-level features, such as topology, maximum bandwidth and network diameter; and then advance to design and evaluation the low level details of network switches [9, 10]. This concludes our top down design and analysis stage. After observing the advantages of the proposed network architecture, we discuss hardware implementation models using standard-cell libraries, and building larger components with them, as they have led to a prototype layout of Mesh-of-Trees network [8]. Then, we discuss fabrication and testing of the prototype chip.

Next, we advance the MoT network concept by proposing a novel hybridization approach, where some parts of it is replaced by Butterfly (BF) networks of small scale. As a result, we achieve significant reduction in area cost with acceptable reduction in throughput [11]. Hybridization with increased intensity shows a trade-off between area cost and throughput, which is more cost-effective than popular networks such as meshes and rings.

Finally, we incorporate the MoT network to a prototype design (as shown in [105]) of XMT architecture. We evaluate deadlock conditions that may arise due to the MoT-XMT interaction, and evaluate methods of deadlock prevention. Following that, we evaluate the effectiveness of the MoT network with real applications, which are compiled and executed on a complete parallel computing platform that supports PRAM-like programming, and implements the memory system envisioned in [72].

The main results in this thesis are based on synchronous CMOS circuit design principles. With asynchronous design principles [94], one can achieve low power consumption, since such components do not require a periodic "clock" signal, which is a significant power consumer in synchronous circuits. In fact, the study reported in [42] focuses on designing a low-power network using asynchronous design principles, based on the MoT architecture discussed in the current thesis.

This thesis is organized as follows: Following the introduction, we discuss background and related work in Chapter 2. Next, we present our methodology of evaluation in Chapter 3. We introduce the Mesh-of-Trees topology and evaluate its inherent features based on its topology in Chapter 4. Next, we discuss the details of network switches, their performance, and consequently, the overall cost

and performance of the MoT network in Chapter 5. Chapter 6 follows with the layout design and evaluation; and physical testing of the fabricated prototype chip. In Chapter 7, we present and evaluate our hybridization method. Next, in Chapter 8 we discuss the integration of MoT and XMT architecture. We discuss the limitations of our approach in Chapter 9; and conclude in Chapter 10.

Chapter 2

Background and Related Work

2.1 PRAM-On-Chip Vision

The Parallel Random Access Model (PRAM) is an easy model for parallel algorithmic thinking and programming [101]. It has been developed mostly during 1980s and early 1990s, and it provides the second largest algorithmic knowledge base, following the serial algorithms.

PRAM is a natural extension of serial RAM (Random Access Model), which is the basis of current programming model in serial computers. In PRAM, architecture details are abstracted. The algorithm designer (or software programmer) focuses on the actual problem, and explicitly declares all available parallelism, instead of dealing with architecture details and marginal optimizations. The hardware orchestrates efficient execution of expressed parallelism. This is one of the features that provide the highly sought-after ease-of-programming.

Earlier multi-chip multiprocessor designs that aim to support the PRAM (such as Tera/Cray MTA [2] and SB-PRAM [7]), although interesting, are constrained by inter-chip interconnections. Latency and bandwidth problems have limited their success in supporting PRAM. With the continuing advances in integrated circuit technology, it becomes possible to build a single-chip parallel processor, as is being demonstrated in the Explicit Multi-Threading (XMT) project [71, 102–105] that

seeks to prototype the *PRAM-On-Chip* vision.

2.2 Underlying Memory Model

Parallel computing generally requires a larger number of memory accesses than serial computation per clock. A standard technique for hiding access latencies is by feeding functional units with instructions coming from multiple hardware threads. This allows, for example, overlapping several arithmetic instructions as well as read instructions each requiring waiting for data. Such overlap implies a steady and high demand for memory accesses. To facilitate concurrent accesses by many processing elements, memory is normally partitioned on parallel machines [50]. For example, Tera/Cray MTA [2] uses 512 memory modules of 128MB each, and SB-PRAM [7] uses as many memory modules as processing elements.

In addition to employing multiple processing elements and memory modules, the XMT architecture uses *Independence of Order Semantics* (IOS), which allows concurrent threads to advance independently from each other, without busy-waiting [102]. As a result, threads issue more memory requests, creating a very high demand for memory bandwidth.

We consider a memory system that is designed to support high levels of parallelism, as expected from a PRAM-like operation. Ideally, such a system has same memory latency for a single access and multiple concurrent accesses. In reality, such a system can be approximated by assuming *Uniform Memory Access* (UMA) models, as opposed to *Non-Uniform Memory Access* (NUMA) models. The main

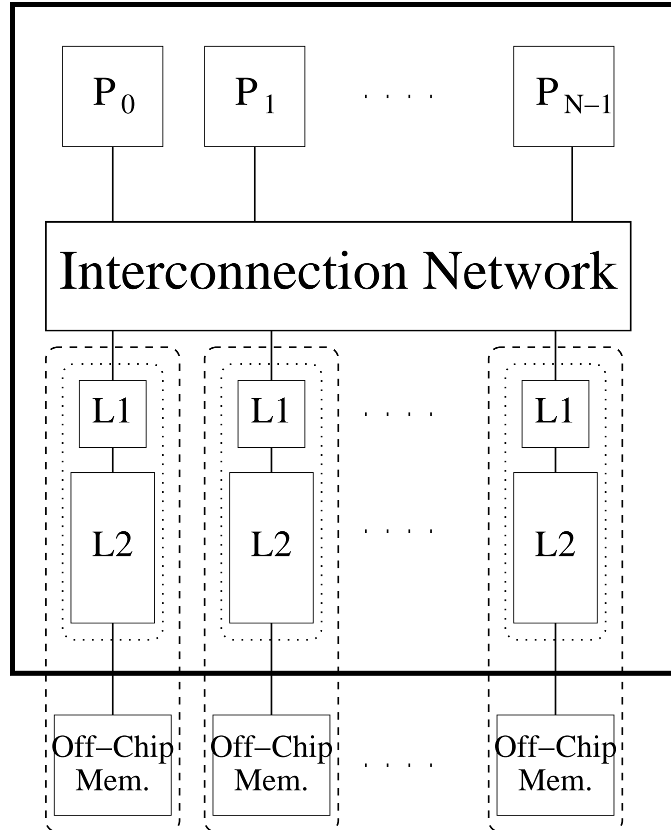


Figure 2.1: Global memory is partitioned into modules (separated by dashed lines). Each module has its own possibly multi-level on-chip caches (within dotted lines).

difference is that in UMA model, all memory modules are at same logical distance to all processors; whereas in NUMA model this is not true.

A memory access consists of two parts:

1. The processors send a *request* to memory modules.

2. The memory modules send a corresponding *response*.

For example, for a *load* instruction, the processor sends an address as request, and memory modules respond with the data. In some cases, the response may be omitted. For example, depending on the communication protocol, a *store* operation

may or may not require any response from memory.

*Request*s and *response*s are physically realized by sending data packets between processors and memory modules. From the network's point of view, both are attempts to communicate between a source and a destination. Therefore, unless we explicitly state otherwise, we do not distinguish them, and call them as *request*.

In the general case of concurrent memory accesses, the access latency of a message between processors and memory modules breaks into two parts:

1. *Flight time*, where memory requests advance towards their destinations, and,

2. *Waiting time*, where they wait in queues while some other requests are being serviced.

The advantage of an UMA model is that the former is the same for all memory requests, regardless of different source-destination pairs. The latter part depends on the traffic amount in the network, and distribution of the traffic on all destination modules. High traffic implies higher waiting times, and balanced traffic implies that the waiting time seen by the sources is similar to each other. The flight time and waiting time components can be used in performance modeling of PRAM-like programs, as shown in [101]. The performance model of [101] considers the sum of these components for *memory request* and *memory response*, and calls them *Round-trip Time to Memory* (RTM) and *Queuing Delay* (QD). For high-performance execution, both RTM and QD are desired to be low.

The following memory architecture is used in the XMT single-chip parallel processor, [72], which is designed to optimize single-task completion time (Figure 2.1).

A globally shared memory space is partitioned into multiple memory modules. Each memory module consists of on-chip cache and off-chip memory portions. A hashing function is used to avoid pathological access patterns (similar to [2, 7, 35]). This structure completely avoids cache coherence issues because the processors do not have writable private caches. In a recent implementation [105], *read-only* buffers were used for groups of processors. This architecture, along with PRAM performance model of [100], imposes significant challenges for the interconnection network design.

1. The network needs to provide high throughput between processors and first level of memory caches, especially when the traffic is high.

2. The network needs to support an UMA model. Namely, the distance between each source-destination pair needs to be the same.

3. For low execution time, low RTM and QD are desired.

2.3 Review of Existing Interconnection Network Models

2.3.1 Definitions

The topology of an interconnection network is represented by a set of source terminal nodes S, a set of destination terminal nodes D, a set of internal nodes I. All nodes, $N^* = S \cup D \cup I$, are connected by a set of channels C.

In general, each channel $c = (x, y) \in C$ uni-directionally connects two nodes x and y, where $x, y \in N^*$; and characterized by its width w_c (in terms of bits), and

frequency of transmission f_c. The bandwidth of a channel is $b_c = w_c f_c$, and it is measured in *bits per second*. In the notation $c = (x, y)$, the node x is the source, and the node y is the destination of the channel. In specific cases (see Section 2.3.2) a channel may be bi-directional, and have multiple sources and destinations.

A *cut* of a network, $C(N_1^*, N_2^*)$ is defined as the set of channels, so that removing them from the network partitions the nodes N^* into disjoint sets of N_1^* and N_2^*. A *bisection* is a cut that partitions the network, including the source and destination terminals, nearly in half[1]. The *bandwidth* between two disjoint sets is equal to the sum of bandwidths of all channels in the cut. The *bisection bandwidth* B_B of a network is the defined as the minimum bandwidth over all possible bisections.

We consider *packet-switched interconnection networks*, where messages are transmitted from sources to destinations in form of data packets. A message consists of one or more data packets. Packets can be further divided into *flow-control digits* or *flits* [28], which are the smallest units recognized by the control circuits in the network.

In the single-chip multiprocessor context, terminals in S and D are realized by processing elements and memory modules; and internal nodes in I are realized by simple switch primitives, or more complex router circuits. In general, a channel c is realized by w_c parallel wires. Specific VLSI implementations can insert repeater buffers or pipeline stages on these wires, in order to improve signal delay on wires [39]. In synchronous implementations with a periodic clock signal, each channel

[1]If there are even number of nodes, each partition has equal number of nodes. If there are odd number of nodes, one partition has one more node than the other.

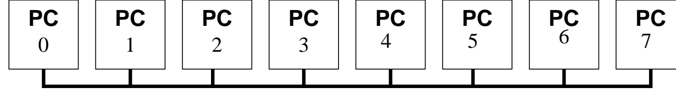


Figure 2.2: Bus with $N = 8$ terminals.

transmits one flit per clock cycle. Asynchronous implementations do not have a traditional periodic *clock*. However, transmission occurs in cycles, which may not be periodic; and each channel transmits one flit per cycle.

2.3.2 Bus

Bus is one of the simplest interconnection networks. It consists of a single channel, with no intermediate nodes ($I = \emptyset$). A bus with $N = 8$ terminals is shown in Figure 2.2.

In practice, bus networks are used for systems with small number of interconnected components [32]. A commonly used open-source bus standard is AMBA (Advanced Micro-Controller Bus Architecture) [5].

On the other hand, the use of multi-stage interconnection networks is shown to be more effective for large number of components, including Multi-Processor Systems-on-Chip (MPSoC) [3, 4, 40, 41]. Therefore, we briefly discuss busses for completeness, but focus on multi-stage interconnection networks for our main studies and evaluations.

Buses operate in cycles. In each cycle, if a source node transmits a message on the bus, it is broadcast to all destinations. The intended destination node accepts and processes the message, while other terminal nodes ignore it. Since multiple

independent source nodes may want to transmit on the bus in a given cycle, an arbitration mechanism is necessary. One source, called *master*, has control over the bus in any given cycle, and it transmits its message during that cycle [5, 28]. An arbitration protocol decides which source will control the bus in the next cycle [5].

2.3.3 Crossbar

Traditionally, an $N \times M$ *crossbar* (also called *bus matrix* [77]) connects N inputs to M outputs without any internal stages. Such a crossbar has $N \cdot M$ *crosspoints*, where N input lines and M output lines intersect. Each crosspoint can be implemented as a switch. If a switch is closed, source and destination modules corresponding to the switch's input and output lines are connected. Figure 2.3 shows a 3×4 crossbar.

Several studies considered crossbars for connecting processors, memory modules and application-specific components on Multi-Processor Systems-on-Chip (MPSoC) [70, 77]. Such networks are built by connecting multiple pipelined buses, based on communication bandwidth requirements between heterogeneous components. While diversity in required bandwidth allows efficient hardware optimization for different applications, such optimizations are not expected to provide the same level of benefit when the connectivity of components is symmetric and high bandwidth is required between general-purpose processors and globally shared memory. The pipelined crossbar network of Cyclops-64 processor (IBM) [111] connects such processors and globally-shared memory modules. However due to its centralized

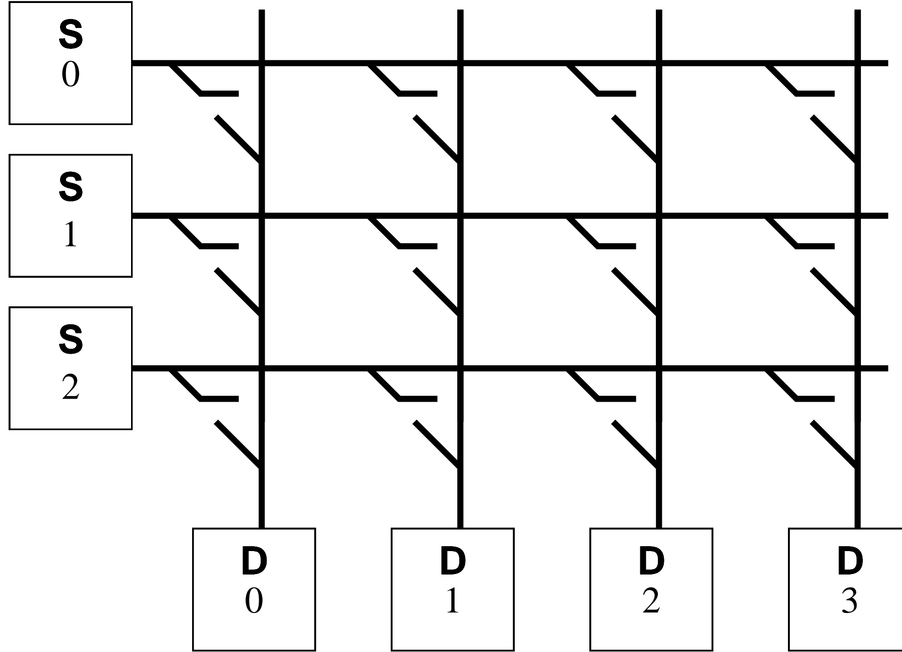


Figure 2.3: Crossbar with 3 source and 4 destination terminals. Horizontal and vertical lines represent input and output lines respectively.

architecture, it is unable to provide the desired performance.

2.3.4 Fat Tree Networks

Fat tree network [58] provides multiple paths between each pair of nodes. A disadvantage of fat tree is its large switch size. The following two structures were proposed to overcome this disadvantage.

1. The *k-ary n-tree* [80] connects $N = k^n$ PCs with a fat tree of n levels as illustrated in Figure 2.4(a). Root nodes (small circles in the center) have k children, switch nodes (oval shape internal nodes) have k children and k parents, and there are two unidirectional links between a child and parent. Thus there are $2k$ input ports and $2k$ output ports for each switch node.

2. Figure 2.4(b) depicts a *butterfly fat tree* (BFT) with $N = 16$ PCs [74]. Each internal switch node (the square with no label surrounded by 4 PCs) is connected to 4 PCs and the 2 root switch nodes. Thus it has 6 input ports and 6 output ports.

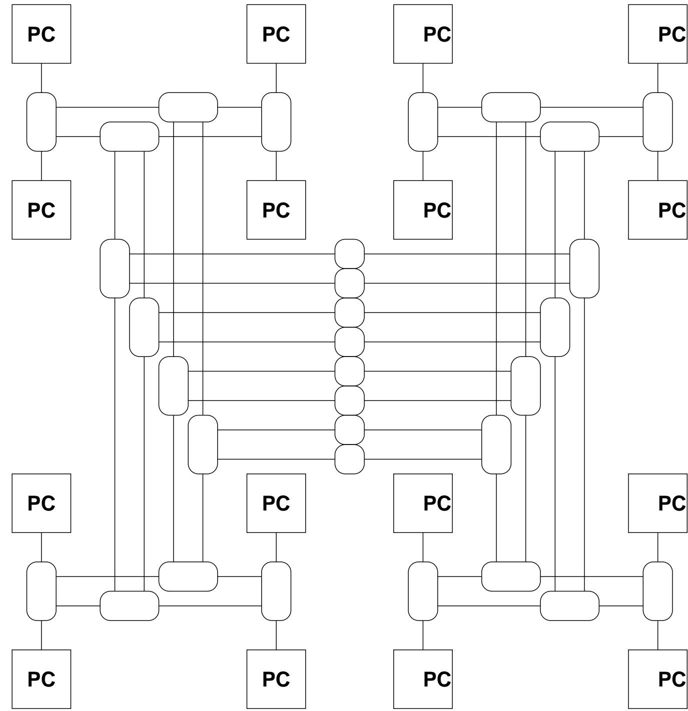
2.3.5 Mesh Networks

In general, *2-dimensional (2D) mesh* networks can be connected by an $m \times n$ grid, where m and n represent number of rows and columns. Such a network interconnects $N = mn$ components. Figure 2.5(b) shows a case where $m = n = 4$.

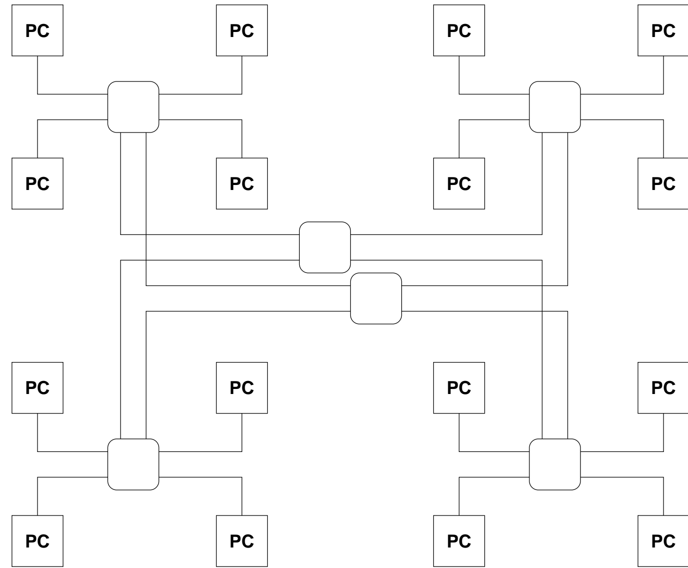
Meshes can have higher dimensions [28], however most common practice is based on 2D meshes. The interconnection network of Teraflop processor (Intel) [99] connects 80 processing cores in an 8×10 2D-mesh. This topology has also been used in academic projects such as RAW (MIT) [95], and TRIPS (UT-Austin) [36]. Furthermore, several network-on-chip (NoC) studies chose 2D-mesh as underlying topology, due to its regularity and low hardware complexity [4, 69, 83].

2.3.6 Ring Networks

Ring network consists of a 1-dimensional (1D) array of modules, where the last component is also connected to the first to form a cycle. A ring network interconnecting $N = 16$ components is shown in Figure 2.5(a). This network is a specific member of a family known as *cube*, or *torus* networks [28]. Ring represents 1-dimensional interconnection of N components. On the other hand, *hypercube*

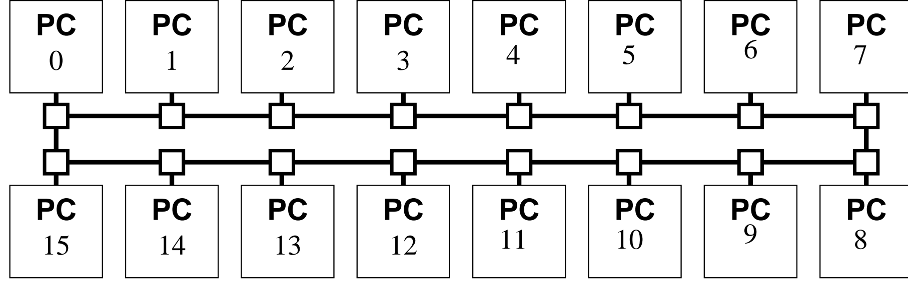


(a) 2-ary 4-tree

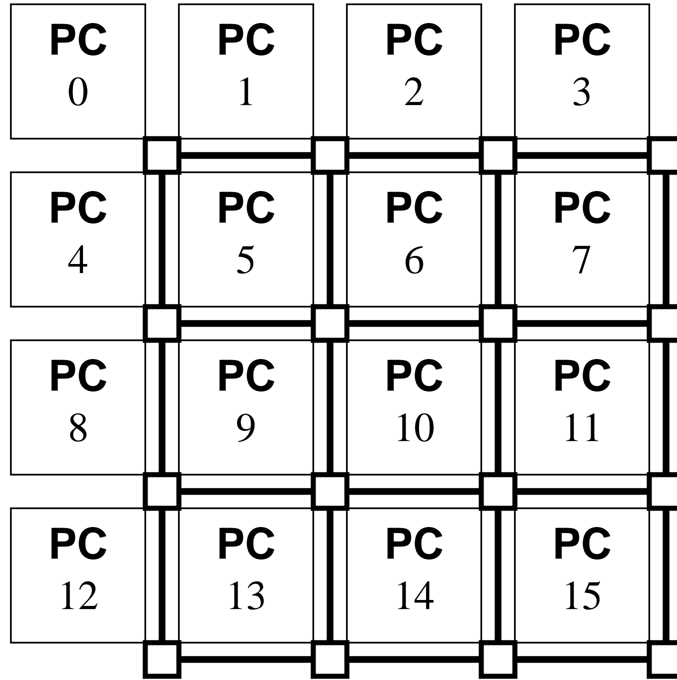


(b) Butterfly Fat Tree

Figure 2.4: Two types of fat trees with constant switch size. (a) k-ary n-tree with $k = 2, n = 4, N = k^n = 16$; (b) Butterfly Fat Tree with $N = 16$.



(a) Ring



(b) 2D-Mesh

Figure 2.5: (a) Ring, and (b)4×4 2-dimensional mesh topologies with $N = 16$.

network of the same family represents $\log_2 N$ dimensional interconnection. For a given N, these networks correspond to lowest and highest dimensions in this family respectively.

Recently, new topologies, such as *spidergon*, are proposed and evaluated as a compromise among the ring and 2D-mesh topologies [16, 23].

Ring networks also received attention from recent industrial projects. The

Element Interconnect Bus of Cell processor (SONY, Toshiba, IBM) [46] is designed with 4 parallel rings that connect 12 cores consisting of processing elements and peripherals.

2.3.7 Hypercube Networks

Hypercube is a member of *cube* or *torus* network family [28]. An n-dimensional hypercube, Q_n, connects $N = 2^n$ nodes by connecting a node to n other nodes. If we label the nodes from 0 to $N - 1$ in binary, a pair of nodes are connected directly if and only if their labels differ by one bit [28, 38, 57]. This connection consists of two uni-directional physical communication channels (wires). Figure 2.3.7 depicts the best known implementation of Q_4 in terms of area efficiency [38]. $N = 16$ nodes (PC stands for *Processing Cluster* in the figure) are connected by wires in tracks shown in the shaded areas between the PCs.

2.3.8 Butterfly Networks

Butterfly network is one of the most extensively studied interconnection networks (e.g. [31, 47, 66]). Figure 2.7(a) shows a binary butterfly that connects $N = 2^3 = 8$ nodes. The 8 PCs are connected to each other through switch nodes labeled (by their vertical layers) A, B, C, and D. For example, the connection between source 0 and destination 5 and the connection between source 6 and destination 6 are highlighted. Figure 2.7(b) shows the best known physical layout to implement the same butterfly network for MPSoC context, based on [107].

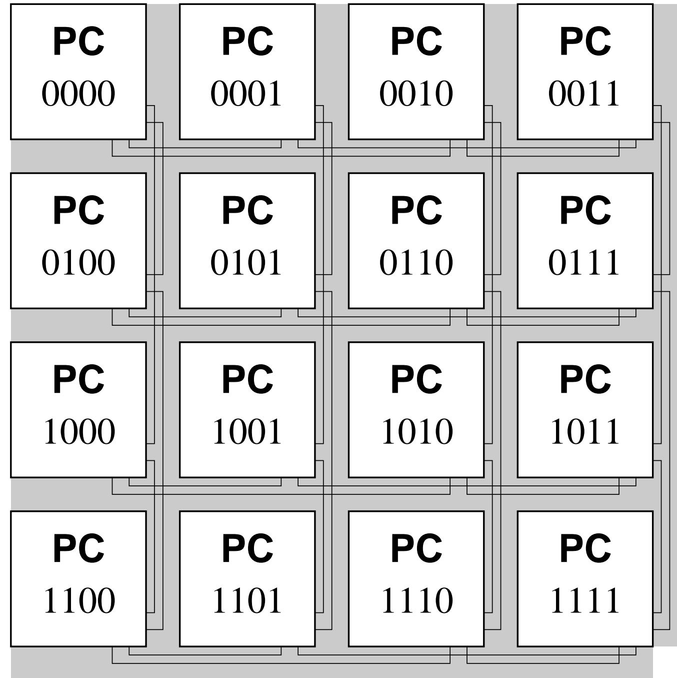
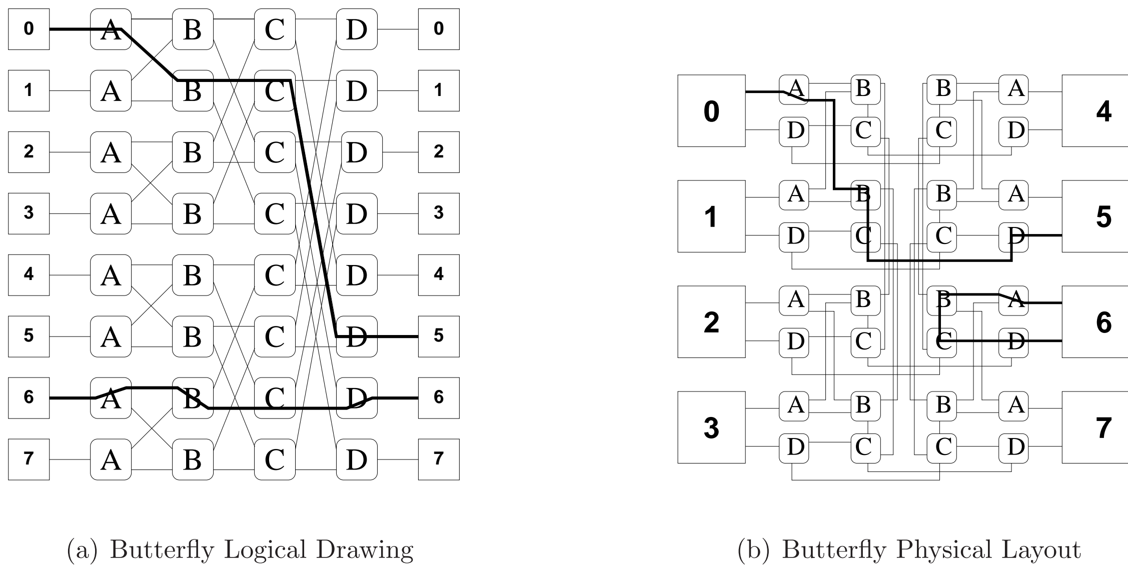


Figure 2.6: A physical implementation of 4-dimensional hypercube.



(a) Butterfly Logical Drawing

(b) Butterfly Physical Layout

Figure 2.7: (a) Butterfly network and (b) its layout with $N = 8$ PCs as shown in [107].

2.4 Performance Improvement with Additional Resources

In many cases *per-cycle throughput* of interconnection networks can be improved by increasing the amount of resources. In this section, we summarize three common methods.

2.4.1 Virtual-Channel Routers

Virtual channels [27] can be used as buffers for incoming data packets that are stalled due to contention in later stages. A packet is stored in a virtual channel in the switch until an output port and physical channel toward its destination becomes available [27, 28, 37, 80]. These studies typically use 2 or 4 virtual channels per physical channel. Using more virtual channels improves *per-cycle* throughput, by increasing utilization of physical channels (wires) in cases of contention. It is possible to build a virtually non-interfering network by using N virtual channels per switch, where N is the number of terminals. However, in addition to the area cost of virtual channel buffers, this approach also increases the complexity of the network switch. With more virtual channels, there are more candidate packets that require to use a single physical channel or output port. This results in a more sophisticated routing and arbitration logic, and longer switch delay.

Recently, *Express Virtual Channels* (EVCs) [51, 52] were proposed to reduce energy and performance overheads of regular virtual channels. Results on a 7×7 2D-Mesh network (49 terminals) show 84% reduction in packet latency and up to 23% improvement in throughput while reducing the average router energy consumption

by up to 38% compared to state-of-the-art virtual channel implementations [52]. The EVCs have also been evaluated on larger networks with 100 terminals (10×10), and have shown improvements over the baseline VC configuration [52]. The improvements are achieved by allowing some packets, which need to travel several hops without changing directions, to bypass some stages in each hop such as routing and arbitration.

While EVCs seem to be feasible and promising on 2D-Mesh networks, it is not clear if it would work similarly on high-bandwidth networks such as butterfly, where packets may need to change directions more frequently. Furthermore, the network switches with EVCs require additional logic that could increase the switch delay; and additional flit buffers that could increase the area cost of the network.

2.4.2 Virtual Output Queuing and Buffered Crossbars

Virtual Output Queues (VOQ) are the most commonly used methods to achieve maximal throughput with crossbars. In its classical implementation, $O(N)$ buffers per input port ($O(N^2)$ buffers total) precede the inputs of a monolithic crossbar [65]. Figure 2.8 shows a 4×4 crossbar with VOQ buffers between sources and the crossbar. In such crossbars, the complexity of the arbitration and scheduling may affect the length of clock cycles, similar to the effect of virtual channels discussed above.

Alternatively, *buffered crossbars*, or *crosspoint-buffered crossbars* use buffers at each crosspoint instead of inputs. More advanced crossbar architectures are built by combining input buffers and crosspoint buffers (CICQ crossbar, Figure 2.9), and

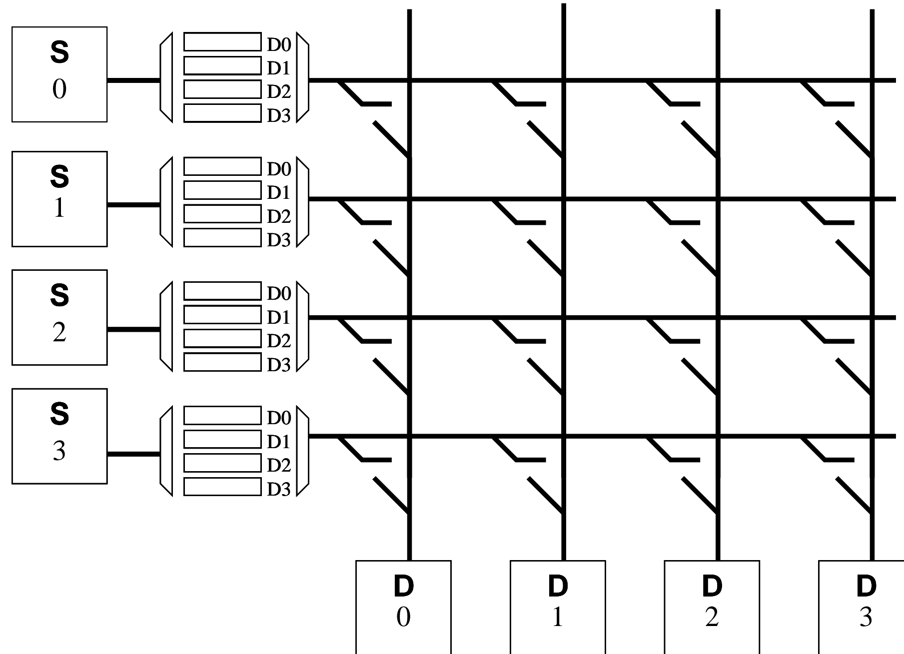


Figure 2.8: 4×4 crossbar with virtual output queues (VOQ).

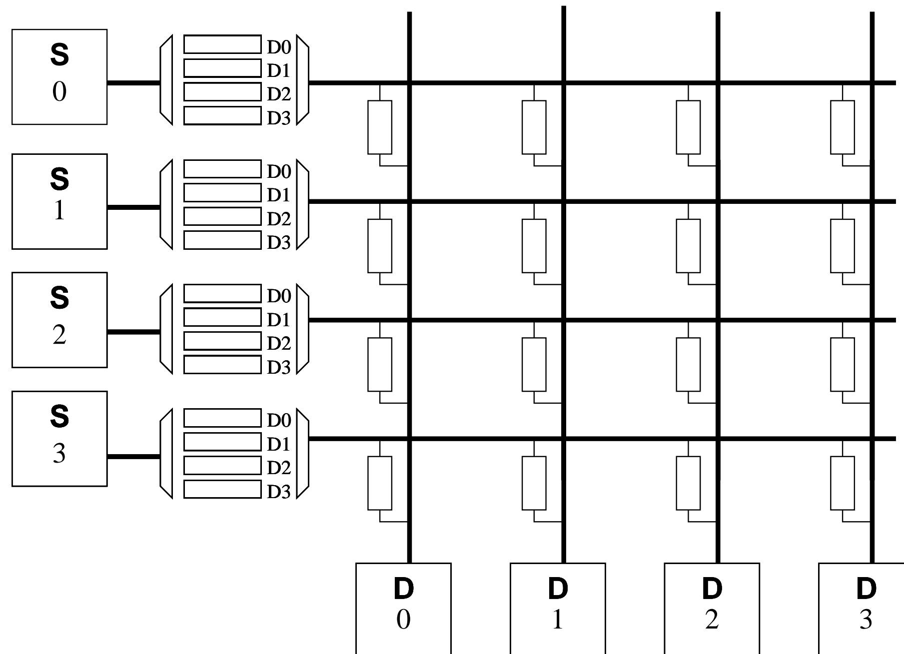


Figure 2.9: 4×4 Combined Input and Crosspoint Queued (CICQ) crossbar.

generally used in large-scale network routers [1]. Using some amount of crosspoint buffers effectively decouples input scheduling from output scheduling; however, it does not completely eliminate them as centralized operations [48]. Therefore, such architectures may still require long clock cycles, or multiple iterations of arbitration.

2.4.3 Tuned Butterfly Networks

The butterfly network is one of the most extensively studied interconnection networks. Many variants of butterfly network have been developed to improve its performance. The improvement is achieved by increasing the resources.

One group of networks extend the regular butterfly vertically, by adding parallel resources. Extra hardware provides additional bandwidth, reduces congestion and improves throughput. Examples of this approach include *multi-butterfly* [98], *dilated butterfly* [49, 87], and *replicated butterfly* [33, 49].

Another group of networks extend the regular butterfly horizontally, by adding extra stages. This approach adds alternative paths between sources and destinations, improves traffic distribution in the network, and reduces congestion. However, without additional bandwidth, throughput improvement is limited. Some examples of this approach include *extended butterfly networks* [66], *selective extra stage butterfly networks* [47], and *augmented butterfly networks* [53]. The historical *Beneš networks* [14] can also be considered in this group, since it is equivalent to two back-to-back butterfly networks.

2.5 Deficiency of the Existing Interconnection Networks

2.5.1 Interference

Interference occurs, when traffic destined to D_a interferes with or "steals" bandwidth from traffic destined to D_b, where $0 \leq a, b \leq N-1$ and $a \neq b$ [28]. In other words, two packets P_i, from $s_i \in S$ to $d_i \in D$ and P_j from $s_j \in S$ to $d_j \in D$ interfere with each other, if one prevents the other from advancing towards their destination.

2.5.2 Global Synchronization

Crossbar networks provide one standard type of high-throughput interconnection networks, where packets do not interfere. They achieve this by scheduling the switches based on the global state of the network. The overhead for global scheduling may be acceptable with large payloads in messages. However, in the XMT single-chip parallelism context, the messages between processors and cache are very small (one word for load instructions and at most two words for the store instructions). Therefore, the networks that need to globally schedule the switches will incur significant overheads.

2.6 Advantages of MoT Network

We propose a new interconnection network implementation based on mesh of trees (MoT) topology. We will demonstrate, both analytically and experimentally,

that the proposed MoT network can provide high throughput with low latency within a reasonable area cost.

The MoT topology and routing method guarantee that unless the memory access traffic is extremely unbalanced, packets between different sources and destinations will not interfere. Therefore, MoT network provides high *per-cycle throughput*, very close to its peak throughput.

MoT network consists of less complex switches compared to other networks. Furthermore, packets in MoT network are routed without global scheduling. This allows higher operating frequencies.

Notable differences between Mesh-of-Trees network and CICQ crossbars [1] include the distribution of buffers over trees with logarithmic depth; the use of completely decentralized and decoupled constant-complexity routing and arbitration operations; and combining arbitration with data traversal over the network, similar to [9]. Among those differences, logarithmic depth is especially important, since a MoT with comparable area distributes wire load over $O(\log N)$ pipeline stages as opposed to $O(1)$ stages of crossbar. This feature allows MoT to operate at higher clock rate.

2.7 Earlier Implementations of Mesh-of-Trees Network

The *mesh of trees* (MoT) concept has been discussed earlier in books such as [57], and papers such as [15, 21, 30, 56].

In an interconnection network based on their approach, functional units (pro-

cessing units and memory modules) will be placed at the leaves of the trees, and a communication path involves climbing up and down some part of the tree. This approach is not interference free and would create performance bottlenecks, as children of internal nodes would compete for resources (connections in the communication path) even when their destinations are different.

Chapter 3

General Methodology of Evaluation

3.1 Introduction

The aim of this thesis is to show that the proposed interconnection network is a competitive alternative for interconnecting memory systems as described in Section 2.2. For that purpose, we characterize and evaluate our network in several aspects, and compare to the networks described earlier.

This chapter lists definitions, and our assumptions and general methods of evaluation.

We start with the network topology, and related metrics that affect cost and performance (Chapter 4). We use wire area and register count as cost metrics; and bisection bandwidth and diameter of the network as a performance metric. We derive analytic expressions for these metrics, and compare with other networks discussed in Section 2.3. Additionally, we analyze deadlock and interference in the proposed network, and qualitatively compare it with above mentioned networks.

Following the general properties of the network, we discuss switches, which are the building blocks of the network (Chapter 5). Using queuing models for switches, we analyze the expected throughput of the proposed network. We perform simulations to accurately evaluate network performance under certain expected traffic conditions. We compare our results with other networks of Section 2.3. We also use

hardware models of switches to analyze the circuit delay of each switch, which is an important component of operating frequency of the network. We integrate the hardware model of our network into XMT processor architecture, which embodies the memory architecture of Section 2.2, and evaluate our network under application-generated traffic.

We build layout of our network using commercial tools and standard cell libraries. We evaluate layout-accurate area, performance and power consumption (Chapter 6) while taking physical constraints, such as wire lengths, into consideration.

Finally, we integrate our network in XMT architecture context, and evaluate its performance with real applications (Chapter 8).

3.2 Topology Evaluation

3.2.1 Wire Area Complexity

We follow the following grid assumptions of Thompson's classical VLSI complexity theory [96]: (i) Width of wires and square switches are assumed to be one unit; (ii) there are two levels of metal wires; (iii) two wires can intersect in one unit square, if one is horizontal, the other is vertical, and they belong to different levels.

We define the area of a network that is laid-out according to above assumptions as its *wire area*.

These assumptions are sufficient to evaluate and compare asymptotic wire area complexity of networks in this study. However, in modern VLSI processes, there are

more than 2 levels of metal available for wiring. As a result, the real wire area of a circuit is less than the area indicated by the asymptotic complexity by a technology-dependent constant factor [13,29,108,109]. The switch nodes are usually wider (and taller) than wires by another constant factor. As a result, the cumulative switch area may dominate the real area cost of the network.

3.2.2 Register Count

We discussed *wire area complexity* as a measure of cost in previous section. In addition to wires, a network consists of switches that route the flits from their sources to their destinations. We consider the cumulative switch area as a metric to evaluate the area cost of networks.

Our goal is to come up with a simple and accurate model for computing and comparing the switch area A_{sw} of various networks. We divide the switch area into two components (3.1): (i) control logic to orchestrate their flow from inputs to outputs A_L, and (ii) storage units for flits A_S.

$$A_{sw} = A_L + A_S \tag{3.1}$$

In general, storage units can be implemented as registers, using edge-triggered flip-flops, or level-sensitive latches. We use the term *register* to cover all implementations of such storage units.

The control logic usually consists of a small amount of gates per register. On the other hand, a register consists of multiple bits, for example, b-bits for a b-bit-wide flit. The value of b depends on the data that is carried through the network.

The signals generated by the control logic is broadcast to all relevant bits of the registers. Furthermore, techniques such as *virtual channels* (Section 2.4.1 allow multiple registers to use the same control logic at different times [28]. As a result, the registers require more hardware than the control logic. Therefore, we assume that A_L is negligible compared to A_S. Then, A_S can be used to approximate the switch area, and compare with other networks.

Finally, in order to generate a technology-independent model, we use the number of b-bit registers R instead of the actual hardware area. If one bit of register requires A_{bit} silicon area at a given technology, then the storage area is computed as $A_S = R \cdot b \cdot A_{bit}$.

To summarize, we consider the cumulative switch area as a cost metric of networks. For a single switch, we use the area model given in (3.2). We assume that the logic area A_L is negligible compared to register area. A register consists of b bits depending on the intended purpose of the network, and we assume that it is same for all evaluated networks. We use a technology-dependent constant A_{bit} to represent the silicon area of one bit of register, and we assume that it remains constant for the evaluated networks. R is different for each network, and it can be varied for the same network to obtain higher performance. As a result, we use register count R to evaluate the area cost of the networks in this study.

$$A_{sw} = A_L + R \cdot b \cdot A_{bit} \tag{3.2}$$

3.2.3 Bisection Bandwidth

We defined bisection bandwidth earlier in Section 2.3.1. Here we discuss its relevance and importance as a performance characteristic.

The bisection bandwidth is the bandwidth of the smallest cut that partitions the network nearly in half[1] [28]. The upper bound of per-terminal throughput of a network under uniform traffic is proportional to its bisection bandwidth, and inversely proportional to the number of terminals [28]. As a result, under uniform traffic, the upper bound of overall throughput of a network is proportional to bisection bandwidth. Therefore it is an important performance measure to consider for the memory system of Section 2.2. In such a memory system, a network with higher bisection width is expected to have higher performance.

3.2.4 Network Diameter

Packets in a network advance in *hop*s from one node to the next, as they travel from the source to the destination. The diameter of a network is the largest number of hops among all shortest paths [28]. Since the network we intend to build is placed between processors and the first level of globally shared cache memory, low and scalable diameter is desirable.

[1]Exactly in half, if number of nodes and terminals are even.

3.2.5 Deadlock

Deadlock occurs in a network, when a set of flits cannot move, because each flit is waiting for some resource, that is held by another flit. If these flits form a cycle, the network is deadlocked [28].

Deadlock prevents the operation of a network. Therefore, if deadlock can happen in a network, additional measures are needed to either prevent the occurrence of deadlock, or resolve when it occurs.

We consider deadlock in two cases. For the MoT network in isolation, we discuss deadlock issues in Section 4.7.5, together with topology and routing. We revisit the issue later in Section 8.1, when we embed the MoT in the XMT processor, since the environment of the network may impose new conditions for deadlock.

3.3 Switch Evaluation

Packet-switched interconnection networks consist of a set of switches (or routers, or switch primitives), which contain a set of buffers and some control logic circuit. Incoming packets are stored in the buffers until the time they are processed and/or forwarded to the next stage. This process can be modeled as a queue, with an arrival process and a service process [28].

3.3.1 Modeling Interconnection Network Components as Queues

Figure 3.1 shows a simple queue system, where packets enter on the left. They are stored in buffers until they are served and leave the system. λ and μ are called

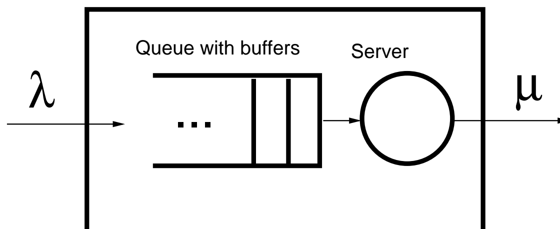


Figure 3.1: A system with queue buffers, and server with arrival and service parameters λ and μ.

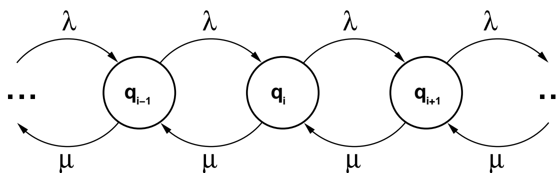


Figure 3.2: Markov chain representation of a queue with arrival and service parameters λ and μ.

arrival and *service* rates respectively.

We consider the so-called "Markovian" queues, which are used in modeling interconnection network switches [28, 85]. Such queues have stochastic arrival and service processes with exponential distribution of inter-arrival and service times. This condition is satisfied by assuming Poisson process for continuous-time systems, and Bernoulli process for discrete-time systems such as computer interconnection networks. In other words, if the arrival and service processes of a queue system are modeled by Bernoulli processes, the queue system is called "Markovian", and it can be modeled and analyzed using *Markov Chain* representation [25, 85]. Such systems can be analyzed to evaluate the system at steady state, if such a steady state behavior exists [25, 85].

Switches in interconnection networks are commonly modeled with Markovian queue systems, where packets arrive with a rate of λ packets per second, stored in an infinite-length queue until they are ready to be served, and processed at a rate of μ packets per second [28]. The *Markov chain* representation of such a system is shown in Figure 3.2. The state of the system represents the number of waiting packets in the queue. A new arrival at a rate of λ transitions the state from q_i to q_{i+1}; a service at a rate of μ transitions it from q_i to q_{i-1}.

Although this model is easy to build, solve, and analyze the network [28], the *infinite queue* assumption introduces inaccuracies. Therefore, for more accurate evaluation, we build simulation models, and evaluate our network by simulations.

3.3.2 Hardware Models

We generate hardware models of the network switches for the following purposes:

1. To evaluate the logic complexity and delay of the switches.
2. To generate and evaluate network layout using industry-standard design tools.
3. To embed the proposed network into a hardware model of a parallel processor that embodies the memory system described in Section 2.2, and evaluate the network in real application traffic.

We use Verilog hardware description language to model our switches using Register Transfer Level (RTL) abstraction. In RTL modeling, the hardware is described as sequential and combinatorial logic. We specify how and when the value of

a register changes. This gives us control over the operations that are performed in one clock cycle. As a result we are able to perform manual optimizations to improve the operating frequency of the network.

In order to build the hardware model of a specific network instance, we need multiple instantiations of each switch, connected in a specific way. We automatize this process by developing a high-level synthesizer, which generates all necessary Verilog models of a network, based on a configuration file. This will be further discussed in Section 3.4.1.

3.3.3 Switch Delay

Peh and Dally showed in [79] that the critical delay of a routing switch increases, when (i) number of input and output ports, or (ii) number of virtual channels increases. Longer critical delay requires a slower clock rate, and this reduces peak and average throughput of the network.

In some cases, it might be sufficient to tune the network for highest *per-cycle throughput*, although this imposes a slow global clock for the entire chip. This may be an acceptable price to pay, because currently, there is an apparent cap on the clock rate for microprocessors. Clock speed does not increase as predicted earlier [88, 89], in part due to challenges of distributing the clock signal on the entire chip. However, it is still reasonable to run a small centralized module with a fast clock, and multiple other modules with a slower clock (derived from the fast clock), e.g. 2× or 4× slower than the fast clock. Therefore, it is important to seek short critical delays in the

interconnection network.

3.4 Network Performance Evaluation by Simulation

Simulations are widely used to characterize interconnection network performance [28]. We apply two different approaches for this purpose.

First, we build a network simulator to test our network in isolation. We inject traffic that is artificially generated based on several parameters. This analysis helps us to understand the limits and characteristics of our network in a controlled setting.

Second, we use appropriate hardware models of our network and integrate it to the hardware model of the XMT processor [105]. We compile and run parallel applications and measure network performance under application-generated traffic.

3.4.1 The Network Simulator

SystemC [45] is a standard hardware modeling and design approach based on C++ language. It is freely available for use in stand-alone mode; and recently it has been integrated in some commercial Electronic Design Automation (EDA) tools. It combines the flexibility of a standard and general-purpose computer language (C++) with the accuracy of RTL modeling. Therefore, we chose SystemC for our simulation environment.

Our SystemC simulator mainly consists of three parts. A high-level synthesizer generates the network (Device Under Test or DUT), based on configuration files. The network is hierarchically generated by instantiating and connecting switches,

and some macro components built of multiple switches. This is a high-level synthesis because the generation stops at switch level; as opposed to a regular synthesis, where all components are modeled with low-level logic gates or standard cells.

The second part of our simulator is the actual implementation of the switches. Each switch is modeled based on its operation, and implemented as a C++ class with class-specific C++ functions. SystemC libraries provide the ability of trigger such functions based on value changes in signals. We used the global clock signal to trigger the operation of our switches, similar to a real hardware operation.

The third part is the environment around the network, which consists of traffic generating terminal modules, packet tracking and performance measurement functions. We also have a text-based user interface to report progress and results.

More detailed and up-to-date information about the operation of specific functions is available as part of the simulator's on-line documentation.

3.4.2 Artificially Generated Traffic

In order to evaluate the network, we apply different traffic patterns at its input, and observe average throughput and latency at the outputs. This section describes different traffic patterns that we used in this evaluation.

We classify our traffic patterns with respect to length of packets, and temporal and spatial distribution.

In general, processors issue *load* and *store* instructions. In the underlying memory architecture, all *load* instructions that are issued by processors, returning

data and store-acknowledgment packets can fit in single-flit packets. Two-flit packets are used when sending *store* instructions. Architectures such as [105] may have some specific and less frequently issued instructions, which can fit into two flits. As a result, packets are most frequently one-flit, and less frequently two-flit long, and longer packets do not occur. Our main results based on artificially generated traffic assumes single-flit packets, and covers most common packets described above. Our simulations with real-life traffic evaluates our network with all of the above instructions.

The temporal distribution of a traffic pattern describes its characteristics as time advances. This is also called the *injection process* [28]. Three most common approaches are *periodic*, *Bernoulli* and *Markov modulated* (MMP) processes [28].

The spatial distribution of a traffic pattern describes its characteristics with respect to destination modules. Two most common approaches are *random* (or *uniform*) and *permutation* traffic.

Next, we briefly describe each of the temporal and spatial distributions.

Periodic Process In a periodic process, a flit is injected into the network every T_i cycles. This process is characterized by the injection rate is $r = 1/T_i$.

Bernoulli Process Bernoulli process is a random process, where the injection of a flit depends on the outcome of a weighted coin toss. In other words, the probability of flit injection at any given cycle is r. As a result, flit injection events are geometrically distributed over time. The average injection rate of this process is r.

Markov Modulated Process In a Markov modulated process (MMP) the Bernoulli injection process is modulated by a Markov process. It is used to simulate bursty traffic, where the injection rate changes in time. A two state MMP is characterized by three parameters, α, β, and r_{on}. The process has two states, *on* and *off*, and at any given cycle the process is at one of them. At the *off* state, no flits are injected. With probability α the process can transition from *off* state to *on* state. At the *on* state flits are injected as a Bernoulli process with injection rate r_{on}.

Random Traffic In a random traffic distribution, each packet is assigned a destination with equal probability. In other words, if there are N possible destinations, the probability of a packet to have destination i, $0 \leq i < N$ is equal to $1/N$. Since the distribution is also called *uniform distribution*, this pattern is sometimes called *uniform* traffic. This traffic pattern is the expected pattern for the memory architecture described in Section 2.2. It is a reasonable assumption due to the use of hashing mechanism, which has an effect similar to randomization that distributes the memory accesses evenly among modules [2, 7, 35, 64].

Permutation Traffic In permutation traffic, the destination address of a packet is determined by its source address and a permutation function. As a result all traffic from one source targets one destination [28]. Various functions have been used in earlier network studies to simulate typical communication patterns of specific applications, or worst-case or best-case patterns for networks. This kind of traffic pattern is not suitable for our evaluations, because of the underlying memory model,

and general-purpose use of the surrounding parallel processor system.

3.5 Layout Evaluation

We build the layout of the proposed network using standard cell design flow. We using commercial CAD tools, with commercially available technology IPs, such as standard cells. Standard cell design is flexible, whereas a custom layout is project specific, and more time consuming. It also may yield better performance due to lowest level optimizations.

3.5.1 Layout Design and Verification

In this section we first explain the importance of validating the previous results on MoT network with cycle-accurate Verilog simulator. We then modify the arbitration primitive to support the *store* operation. We describe the physical design of the MoT network as a further step towards evaluating its layout-accurate performance. Finally, pipelines are inserted to deal with the long wire delays.

3.5.2 Cycle-Accurate Validation

In [10], the performance model of the MoT network has been evaluated using a custom-made simulator, written in C++ using SystemC libraries. There was no earlier study of a cycle-accurate simulator for verifying the MoT network model in [10]. To demonstrate accuracy, some butterfly network simulations has been compared with the "booksim" simulator of [28]. However, the simulator in [10] is

optimized for MoT network, and the simulator in [28] is optimized for traditional networks such as hypercube and butterfly. Therefore, the accuracy of the comparison was limited.

Prior to the current paper, switch primitives have been individually synthesized into generic technology, but the whole MoT network has not been synthesized and verified. Therefore, a realistic hardware model was not available for validation. In this paper we derive a synthesizable verilog model of the full MoT network using our own high level synthesis tool. We perform RTL and gate-level netlist simulations, and validate earlier results.

We assume uniform traffic pattern, which is expected for the memory architecture described in [71], due to the use of a hashing mechanism [2, 7, 35, 64].

3.5.3 Physical Testing of Network Chip

During the layout design of the MoT chip, we verified functional correctness by using a testbench module that simulate a typical operation of the network. Specifically, our testbench serially uploads a short sequence of parameters into the terminal circuits; runs all terminals for a determined period of time; and serially downloads the accumulated statistics from the chip. We call this the *write-execute-read* sequence.

For physical testing of the chip, we aim to create similar conditions in order to compare the outcome with verilog simulations. For that purpose, we designed the setup shown in Figure 3.3. We use the following components in this setup:

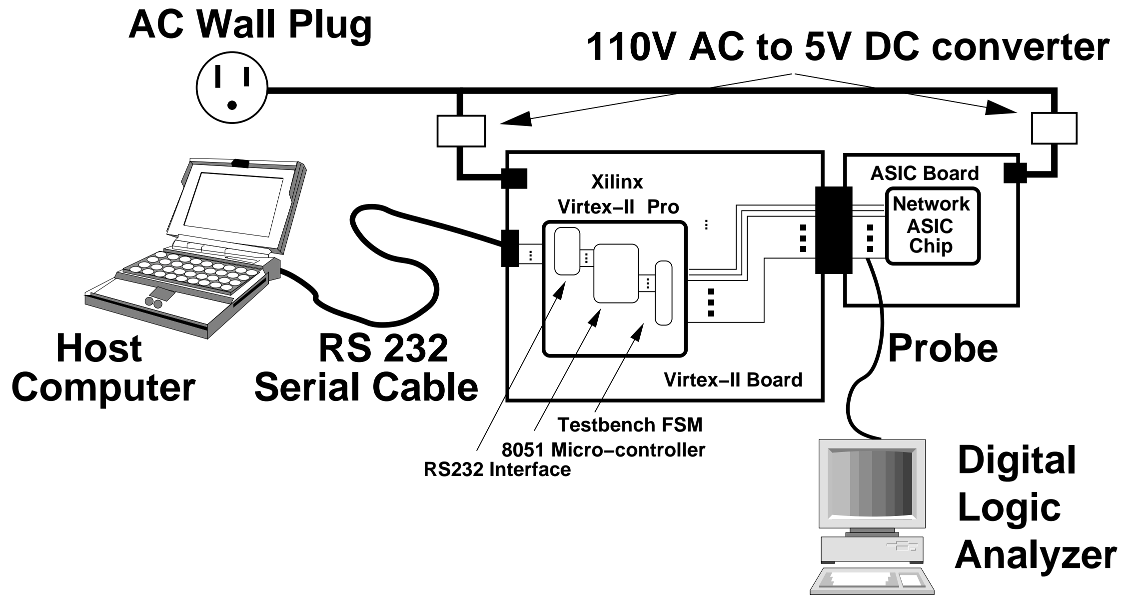


Figure 3.3: Setup for testing network chip.

- FPGA board with Xilinx Virtex-II Pro chip. FPGA contains the following components:
  - **RS232 serial communication interface**: Establishes communication between host computer and 8051 micro-controller.
  - **8051 embedded micro-controller**: Programmable controller for orchestrating test sequence; setting PLL inputs on the chip; and directing reset and "go" signals based on user input.
  - **Testbench Finite State Machine**: Circuit synthesized from verilog testbench. It generates the input signals for the chip as seen in the verilog simulations.
- Custom chip board, developed in-house, to connect ASIC chip to the FPGA board.

- Digital logic analyzer (DLA), to observe the signals at the input/output of the ASIC chip.

The chips are tested as follows:

1. For initial testings, we disabled PLL by setting proper inputs. We send a clock signal with $4MHz$ to the ASIC chip, and observed $\approx 4kHz$ output from *clk1k* output. This output indicates that the clock signal entered the chip, and it is divided by 1024 correctly.

2. We reset the ASIC chip from host computer.

3. We run the Testbench FSM by entering the “go” command from host computer. With this command, the FSM goes through the *write-execute-read* sequence, similar to the verilog simulations.

4. Downloaded throughput and latency data are written in RAM blocks in the FPGA, and later sent to the host computer through the serial interface.

5. We observe chip output from the host computer screen, as well as using DLA.

6. We compare the throughput/latency output with verilog simulation output; compare chip signals with verilog simulation waveforms.

Ideally, we expect that the throughput and latency numbers would match with the output we observe at verilog simulations. This would verify that the chip is operating as shown in the verilog simulations.

3.6 Mesh-of-Trees Network in XMT Context

We conduct preliminary study on the execution of real life programs to demonstrate the effectiveness of the proposed MoT network. We modify our hardware model to fit the XMT architecture [104]. This modification is presented in Appendix A. Next, we develop XMT programs, and measure execution time and the amount of network traffic during execution. In order to show the effectiveness of MoT, we repeat same experiments using a butterfly network instead of MoT.

Chapter 4

Mesh-of-Trees Interconnection Network

4.1 Introduction

This chapter presents the MoT network for single-chip parallelism. We focus on a single network with N source and N destination terminals for characterization. We discuss its topology, routing methods, and flow control mechanisms, and propose a floorplan, considering Thompson's assumptions for VLSI complexity [96]. We contrast our proposed implementation with earlier implementations of MoT topology and other networks that we discussed in Section 2.3.

4.2 Topology

Figure 4.1 shows our implementation of MoT network with four processing clusters (PCs) and four memory modules (MMs). Without loss of generality, we consider the PCs as sources for packets and the MMs as destinations.

The terminal nodes can be built in various ways. In this study, we do not focus on these nodes, or impose any specific design. A recent and practical implementation for such terminals in XMT architecture context is discussed in [104]. We use the following model in our discussion. A PC and an MM can share a terminal node of a single network as shown in Figure 4.2. The PC with and instruction cache is

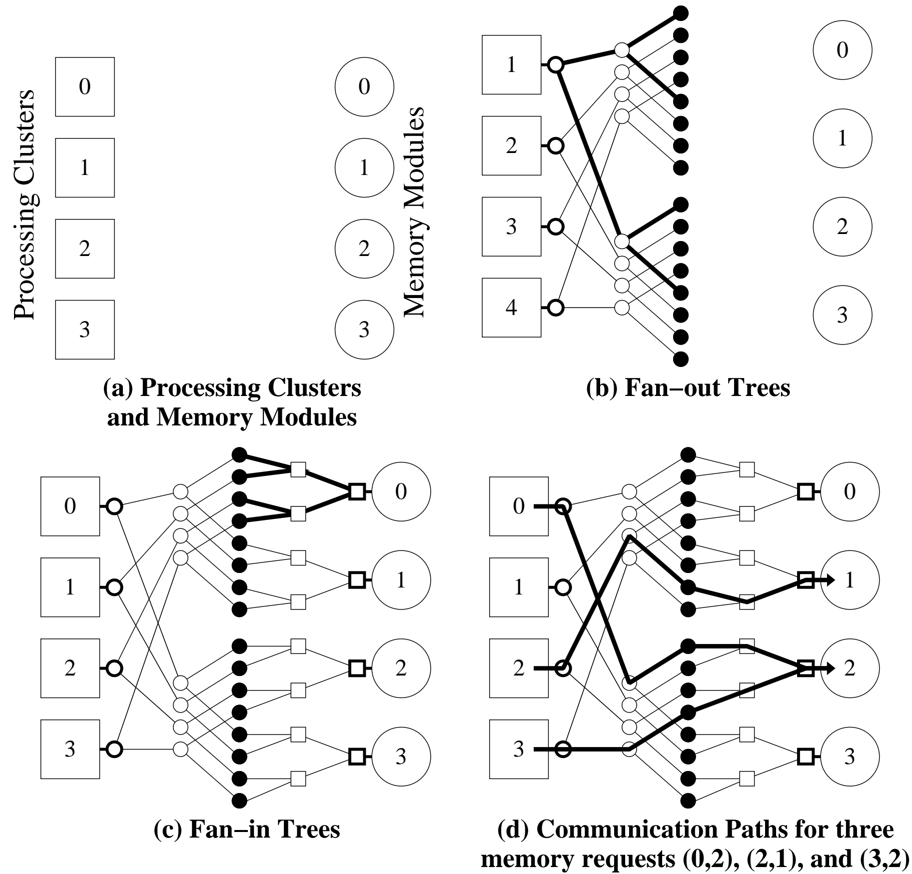


Figure 4.1: Mesh of Trees with 4 Clusters and 4 Memory Modules

marked with letters P and I, and two level data caches are marked as L1-D and L2-D. The network interface is marked as NIF, and it directs network traffic to and from processor and memory. In Chapter 8, we discuss the use of a second network ("response" network) in the context of XMT architecture, for data packets returning from MMs.

The network consists of two main structures, a set of fan-out trees and a set of fan-in trees[1]. Figure 4.1(b) shows the binary fan-out trees, where each PC is a

[1]They are called row and column trees in [57]. We use the names 'fan-out' and 'fan-in' to convey the data flow direction.

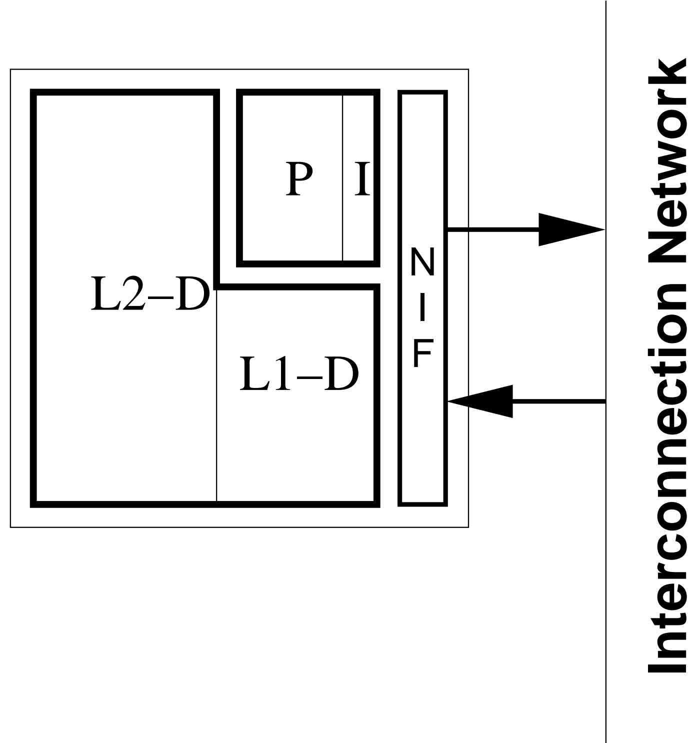


Figure 4.2: Possible implementation of network terminal node. P: Processor, I: Instruction Cache, L1-D, L2-D: Two levels of data cache, NIF: Network interface.

root and connects to two children (we call them *up child* and *down child*), each child will have two children of their own. The 16 leaf nodes also represent the leaf nodes in the binary fan-in trees that have MMs as their roots (Figure 4.1(c)).

Each source (PC) and destination (MM) terminal is labeled as S_i and D_i respectively, where $0 \leq i \leq N-1$ is the address of the terminal node. We label each internal fan-out node as $R_{s,i}$ and each fan-in node as $A_{d,i}$. Here, s and d represent the terminal, where the root of each respected tree is connected; and $1 \leq i \leq N$ is the label of each node in the tree. In the case of binary trees the children of a node $R_{s,i}$ are denoted as $R_{s,2i}$ and $R_{s,2i+1}$.

4.3 Routing

Routing is the process of finding a path for each packet from its source to its destination. Figure 4.1(d) gives the communication paths from PCs to MMs for three memory requests. Each memory request will travel from the PC (source) through a fan-out tree and then a fan-in tree before it reaches the MM (destination). There is no routing decision to be made in the fan-in trees as all packets move toward the root. In fan-out trees, routing decision is trivial from the binary representation of the destination. For example, when PC 0 sends a packet to MM 2 (10 in binary) as shown in Figure 4.1(d), the packet goes from the root to its down child (because of the first bit 1 in 10) and then it selects the up child (because of the 0 in the next bit position in 10) and reaches the leaf. This simple routing scheme also ensures that the fan-out tree part of the network is non-interfering. Similarly, packets with different destinations will not interfere in the fan-in trees.

4.4 Flow Control

Flow control mechanisms manage the allocation of channels and buffers to flits. Figure 4.3 illustrates the switch primitives in our MoT network. Each node in the fan-out and fan-in trees of the network will be implemented as the fan-out (routing) or fan-in (arbitration) primitives as shown in Figures 4.3 (a) and (b) respectively. The pipeline primitive in Figure 4.3 (c) is used to divide long wires into multiple short segments. We discuss the primitives in detail in Chapter 5.

In a given cycle, switch primitives send two signals in addition to data signals,

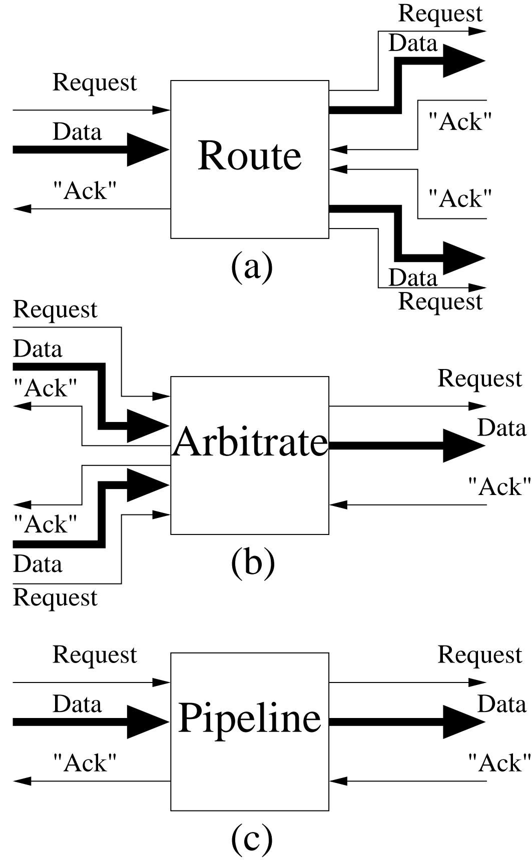


Figure 4.3: Switch primitives of MoT network.

one to their predecessor, and one to their successor. These signals are marked as "Ack" and "Req" respectively in Figure 4.3. Upon receiving these signals, each switch primitive computes their next state; and state transition occurs in synchrony with the clock signal.

In absence of contention, a flit advances one level per cycle. In cases of contention, a flit may stall, and wait in the buffer of a switch primitive, until the contention is resolved.

Contention occurs when flits compete for common resources. In general, competition does not occur in the fan-out trees. Flits of fan-out trees stall only when

the fan-in trees stall and the stall condition propagates back to the fan-out trees. On the other hand, competition may occur frequently in the fan-in trees, since all flits in a fan-in tree have the same destination module. From the flit's point of view, it may experience increased latency; however, from the destination module's point of view, flits arrive at a high rate of throughput.

4.5 Floorplan

Figure 4.4 depicts our proposed floorplan for the *MoT networks*. We first explain the layout of the fan-out and fan-in trees. Both the fan-out and fan-in trees are placed in pairs for better area utilization. Figure 4.4 (a) shows such a pair of 8-leaf fan-out trees for an MoT network with $N = 8$ clusters. The two root nodes of the two fan-out trees are connected to the source clusters by the thick lines. Empty circles are internal nodes and crosses are leaf connections. Figure 4.4 (c) shows the same layout for a pair of 32-leaf fan-out trees. Figure 4.4 (d) shows a pair of 8-leaf fan-in tree. Leaves are connected to internal nodes represented by the empty squares. Roots of the two fan-in trees are connected to the destination clusters through the connections with arrowhead. Figure 4.4 (e) gives the layout of a pair of 32-leaf fan-in trees.

Figure 4.4 (b) shows how the fan-out and fan-in trees are placed between the eight sources and destinations marked as S_i and D_i respectively, where $0 \leq i \leq 7$. Each pair of the fan-out trees is placed vertically and each pair of fan-in trees is laid out horizontally. The leaves of fan-out tree are connected to the leaves of fan-in

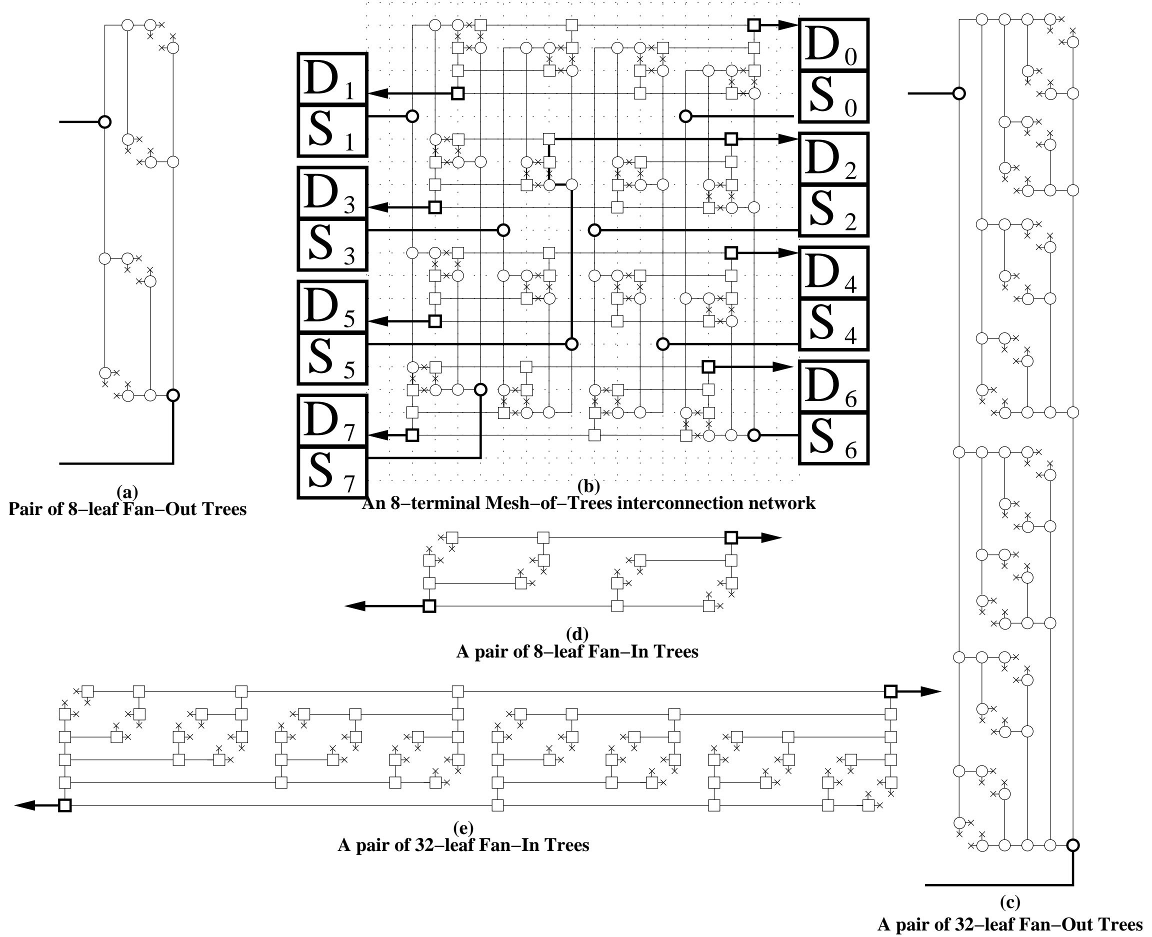


Figure 4.4: Detailed floorplan of the MoT interconnection network. (b): the 8-cluster MoT network floorplan; (a) and (d): details of a fan-out tree pair and a fan-in tree pair in (b); (c) and (e): layout of 32-leaf trees.

trees. The path of a packet from source 5 to destination 2 is highlighted.

4.6 Differences with Existing MoT Implementations

Unlike the conventional MoT approach (Section 2.7), we put the PCs and MMs at the roots of the trees instead of the leaves. As a result, there are two major differences of our MoT network.

1. **Unique Routing Path** There is a unique path between each source and each destination. This simplifies the operation of the switching circuits and allows faster implementation which translates into improvement in throughput when pipelining a path (registers need to be added to separate pipeline stages).

2. **Low Interference** Conventional MoT is prune to interference of packets between different sources and destinations as described in case (i) of Section 4.7.6. Our implementation eliminates this type of interference and related performance loss.

4.7 Evaluation

4.7.1 Wire Area Complexity

This section evaluates the wire area complexity of MoT network and compares it with other networks such as hypercube, butterfly, ring, 2D-mesh, fat trees and buffered crossbar.

Earlier work [10] assumed a fixed chip size, and compared wire areas of different networks. We generalize that discussion by removing the chip size constraint. Our results show that the wire complexity of MoT network is asymptotically larger than other networks by a factor of $\log^2 N$ or N (Table 4.1).

In following analysis we use w_c for number of bits per channel, and d_w for wire pitch, a technology-dependent parameter specifying the minimum distance between two adjacent wires.

For MoT network, we consider Figure 4.4 (b). The width of wire area is $w_c \cdot d_w \cdot \frac{N}{2} \cdot (\log N + 2)$ and height is $w_c \cdot d_w \cdot \left(\frac{N}{2} \cdot (2\log N + 1) + 1\right)$. Their product gives the wire area of MoT network (4.1).

$$A_w = \tfrac{1}{4}N^2 \cdot \left(\log^2 N + 3\log N + 2\right) \cdot \left(w_c \cdot d_w\right)^2 = O(N^2 \log^2 N) \tag{4.1}$$

Hypercube area is computed as shown in Figure 2.3.7. The chip area is $N \cdot (s + w_w)^2$, where s is the size of a PC, $w_w = 2 \cdot t(Q_{n/2}) \cdot w_c \cdot d_w$ is the width of the wire area between two PCs. The constant 2 is due to the use of unidirectional channels, $t(Q_{n/2})$ is the number of tracks in such area, and $n = \log N$. We use the formula for $t(Q_{n/2})$ from [38], assume unit width for PCs, and obtain hypercube network's wire area (4.2).

$$A_w = \left(\frac{16}{9}N^2 + \frac{8}{3}N\sqrt{N} + N\right) \cdot \left(w_c \cdot d_w\right)^2 = O(N^2) \tag{4.2}$$

For butterfly network, the number of wire tracks required in both dimensions can be obtained from the layout of [107]. Similar to the MoT approach, their product gives the wire area in equation (4.3).

$$A_w = (N + logN - 3) \cdot \left(4N - \frac{3\sqrt{2}}{2}\sqrt{N}\right) \cdot \left(w_c \cdot d_w\right)^2 = O(N^2) \tag{4.3}$$

The wire area of a replicated butterfly network is r times the area of single butterfly times $O(\log r)$ for connecting multiple copies to sources and destinations.

We consider fat trees as the two practical implementations, namely *k-ary n-trees* and *butterfly fat trees*, as discussed earlier in Section 2.3.4. The wire area does not change for k-ary n-trees with different values of k and n as long as $N = k^n$ is kept constant. We calculate the total wire area by iteratively adding wires starting from

the root. Assuming unit size for PCs, the wire area of k-ary n-trees and butterfly fat trees are given in equations (4.4) and (4.5) respectively.

$$A_w = \left(2N^2 - N\sqrt{N}\right) \cdot \left(w_c \cdot d_w\right)^2 = O(N^2) \tag{4.4}$$

$$A_w = \left(\frac{1}{4}N\log^2 N + N\log N + N\right) \cdot \left(w_c \cdot d_w\right)^2 = O(N\log^2 N) \tag{4.5}$$

For the wire area of 2D-mesh, we consider only switches and links, and assume unit width for PCs (4.6).

$$A_w = \sqrt{N} \cdot \sqrt{N} \cdot \left(w_c \cdot d_w\right)^2 = O(N) \tag{4.6}$$

The wire area of ring networks is computed similarly to the 2D-mesh, and shown in equation (4.7).

$$A_w = 2 \cdot \frac{N}{2} \cdot \left(w_c \cdot d_w\right)^2 = O(N) \tag{4.7}$$

The wire complexity of a traditional crossbar is $O(N^2)$. The buffered crossbar adds constant overhead per crosspoint buffer. Therefore, its wire complexity remains $O(N^2)$. We assume that VOQ-based crossbars also have similar wire complexities.

$$A_w = O(N^2) \tag{4.8}$$

4.7.2 Register Count

Routing switches consist of several data registers of w_c-bits each, and some control circuit that handles resource allocation, and forward and backward signaling. In typical virtual-channel routing switches [28], there are v *virtual channels* per input and output port to improve performance. Each virtual channel uses at least

one w_c-bit register for one data packet. In our proposed MoT network, each switch primitive has several input and output ports, and B first-in-first-out (FIFO) buffers at each input port. In both types of switches, the control circuit consumes negligible area compared to data registers.

Next, we discuss register count for different networks. If v is constant, hypercube, butterfly and fat trees have asymptotically fewer registers than MoT network. However, if $v = O(N)$ so that these networks become non-interfering, register count of these networks reach or exceed MoT (Table 4.1).

As shown in Figures 4.1 and 4.4, the MoT network consists of N fan-out and N fan-in trees, each with $(N-1)$ nodes. The leaves do not contain switching circuits, since they are only wire connections. Each switch primitive contains $B \geq 1$ registers per input. We compute the register count for $B = 2$, which maximizes local throughput [20]. Each register is w_c-bit wide. The total number of w_c-bit registers is computed in equation (4.9).

$$R = 3 \cdot BN\,(N-1) = 6N\,(N-1) = O(N^2) \tag{4.9}$$

Hypercube has N switching nodes, each with $\log N + 1$ input and output ports. Each of the input and output ports contains v data registers [28], one per virtual channel. As a result, the total number of w_c-bit registers is computed in equation (4.10).

$$R = 2 \cdot v \cdot N\,(\log N + 1) = O(vN \log N) \tag{4.10}$$

Similarly, the switches of butterfly network have a total of $2 \cdot N \log N$ input

and output ports with v virtual channels each (4.11).

$$R = 2 \cdot v \cdot N \log N = O(vN \log N) \tag{4.11}$$

We consider replicated butterfly switches B registers per input, similar to MoT network, and no virtual channels. Similar to MoT, we assume the minimum value for B, and discuss larger values in Chapter 5. The network consists of r copies of a regular butterfly, and binary trees between the network and source/destination modules. The total number of registers in replicated butterfly with r copies is shown in equation (4.12).

$$R = 3 \cdot B \cdot N(r-1) + B \cdot r \cdot N \log N \quad = 6 \cdot N(r-1) + 2 \cdot r \cdot N \log N = O(rN \log N) \tag{4.12}$$

The k-ary n-tree with $N = k^n$ terminals has N root nodes with $2 \cdot k \cdot N$ total ports, and $N \log_k N$ internal nodes with $4 \cdot k \cdot N \log_k N$ total ports. In total they require $2 \cdot v \cdot k \cdot (2 \cdot N + \log_k N)$ b-bit registers (4.13).

$$R = 2 \cdot v \cdot k \cdot (2 \cdot N + \log_k N) = O(vkN) \tag{4.13}$$

In the butterfly-fat-tree, the total number of switches approaches to $N/2$ as N grows [29, 37, 74]. Each switch node has 6 input and 6 output ports. Therefore the total number of w_c-bit registers will be approximately $6 \cdot v \cdot N$ (4.14).

$$R = 6 \cdot v \cdot N = O(vN) \tag{4.14}$$

2D-mesh network has N switches. Each switch has 5 input and output ports for connections towards North, South, East, West and corresponding PC. In total

there will be $10 \cdot v \cdot N$ b-bit registers (4.15).

$$R = 10 \cdot v \cdot N = O(vN) \tag{4.15}$$

In a ring, each switch has 3 ports, for connections towards its predecessor, successor and the corresponding PC. As a result, it has $6 \cdot v \cdot N$ registers (4.16).

$$R = 6 \cdot v \cdot N = O(vN) \tag{4.16}$$

In a buffered crossbar or VOQ-based crossbar, there are N^2 buffers (or queues) with depth q, which is usually a small constant (4.17).

$$R = q \cdot N^2 = O(N^2) \tag{4.17}$$

Table 4.1: Asymptotic area comparison of networks.

Network	**Wire complexity**	**Register Count**
MoT	$O(N^2 \log^2 N)$	$O(N^2)$
Hypercube	$O(N^2)$	$O(vN \log N)$
Butterfly	$O(N^2)$	$O(vN \log N)$
Replicated Butterfly	$O(N^2 r \log r)$	$O(rN \log N)$
Fat Tree (k-ary n-tree)	$O(N^2)$	$O(vkN)$
Butterfly Fat Tree	$O(N \log^2 N)$	$O(vN)$
2D-Mesh	$O(N)$	$O(vN)$
Ring	$O(N)$	$O(vN)$
Buffered Crossbar	$O(N^2)$	$O(N^2)$

4.7.3 Bisection Bandwidth

We consider that N is the number of terminals of a network, and links between switches have identical bandwidths.

The bisecting cut of a MoT network contains N links. As a result, its bisection bandwidth is $O(N)$.

A ring network has $O(1)$, a buterfly fat tree has $O(\sqrt{N})$, and a 2D-mesh network has at most $O(\sqrt{N})$ bisection bandwidth. If the number of rows and columns (m, n) are not equal, the bisection bandwidth of 2D-mesh is $O(\min(m, n))$. While these networks may be better suited for localized traffic patterns, for parallel processors with uniform memory access patterns they face scalability challenges. Networks can be replicated to improve the bisection bandwidth, such as the 4-ring network of the Cell processor [46]. However, there may be challenges in further scaling (e.g. up to $O(N)$ rings), and we are unaware of any comprehensive performance study of such replicated networks.

Other networks that we consider in this study have $O(N)$ bisection bandwidth.

4.7.4 Network Diameter

Table 4.2 summarizes the diameter of the networks discussed in Section 2.3. Most networks have logarithmic ($O(\log N)$) diameter, similar to MoT network. For 2D-mesh, the longest minimum distance (network diameter) is between the terminals at opposite corners, for example between nodes 3 and 12. As a result, the diameter of 2D-mesh is $O(\sqrt{N})$. Similarly, the diameter of the ring networks is

$O(N)$. Buffered crossbar, VOQ-crossbar and CICQ-crossbar have a constant number of stages between inputs and outputs, therefore their diameter is $O(1)$.

Table 4.2: Asymptotic diameter comparison of networks.

Network	**Diameter**
MoT	$O(\log N)$
Hypercube	$O(\log N)$
Butterfly	$O(\log N)$
Replicated Butterfly	$O(\log N)$
Fat Tree (k-ary n-tree)	$O(\log N)$
Butterfly Fat Tree	$O(\log N)$
2D-Mesh	$O(\sqrt{N})$
Ring	$O(N)$
Buffered Crossbar	$O(1)$

4.7.5 Deadlock

A necessary condition for deadlock in a network is existence of cyclic dependencies on resources. The MoT network is deadlock-free by design, because it does not contain any cycles as shown in Figure 4.1.

Similar to other deadlock-free networks such as butterfly, certain external dependencies can cause deadlock. For example, the memory module receives a memory request from the network, processes it, and injects it back into the network as a response. Here, the memory module creates a cycle (externally) that includes

network resources, and can cause deadlock. This is called *high-level deadlock*, or *protocol deadlock*. Using separate networks is a known approach to avoid such conditions [28]. We discuss the deadlock conditions caused by external dependencies in a parallel processor context in Chapter 8.

4.7.6 Interference

We defined interference in Section 2.5.1. In this section we analyze and evaluate interference conditions in MoT network.

Consider two packets P_i and P_j, with source terminals $s_i, s_j \in S$ and destination terminals $d_i, d_j \in D$. In order to evaluate interference in MoT network, we consider two cases, (i) $s_i \neq s_j$ and (ii) $s_i = s_j$.

In case (i) we show that the paths of P_i and P_j do not share common resources (Figure 4.1). Consider the set of internal nodes[2] $\Phi_i = \{R_{s_i,k} | k \in [1, N]\} \cup \{A_{d_i,k} | k \in [0, N]\}$. The first part of Φ_i represents the nodes on the fan-out tree that originates from source terminal s_i. Its second part represents the nodes on the fan-in tree that ends at destination terminal d_i. The path of P_i consists of a subset of Φ_i. Similarly, the path of P_j consists of a subset of internal nodes $\Phi_j = \{R_{s_j,k} | k \in [1, N]\} \cup \{A_{d_j,k} | k \in [1, N]\}$. It is easy to see that if $s_i \neq s_j$ and $d_i \neq d_j$, then $\Phi_i \cap \Phi_j = \oslash$. As a result, P_i and P_j do not share any nodes. Any link that P_i passes through has its both ends in Φ_i; likewise any link that P_j passes through has its both ends in Φ_j. If P_i and P_j do not share any nodes, they also cannot share any links. As a result, P_i and P_j do not share any resources. Therefore, they do not

[2] Node notation was defined earlier in Section 4.2

interfere with each other.

In case (ii) packets originate from same source terminal. Then, $\Phi_i \cap \Phi_j = \{R_{s_i,k} | k \in [1, N]\}$, in other words, they may use a common resource in the fan-out tree, such as a buffer in a node or a link between nodes. We denote this common resource as c. Assuming that each terminal injects at most one packet per cycle, packets P_i and P_j must be injected at different times. Without loss of generality we assume that P_i is injected before P_j. We split case (ii) in two parts. In (ii.a), P_i packet advances one hop every cycle on its path in the fan-out tree of S_i, until it enters the fan-in tree of terminal D_i. In (ii.b), P_i does not advance one hop per cycle.

In case (ii.a) P_i uses and releases the common resource c in consecutive cycles. Since P_j is at least one cycle behind P_i, it will be able to use common resource c after P_i releases it. As a result, P_i and P_j will not require to use the same resource at the same time. Therefore, they will not interfere with each other.

In case (ii.b) P_i may hold on to resource c for longer than one cycle. This may interfere with P_j, because it cannot use c. This case occurs, when P_i cannot advance (stalls), while it is still in the fan-out tree. In the fan-out tree, where packets advance one stage per cycle as long as the next stage is available, a packet stalls only if one or more of packets stall in later stages. So, if P_i stalls, There must be another stalled packet P_k ahead of P_i in the network.

If P_k is in the fan-out tree, we can repeat the same argument and conclude that there must be another packet, P_l, stalled ahead of P_k, and continue analyzing P_l. If P_k is in the fan-in tree, it can stall for two reasons: (1) similar to previous

cases, there is a stall ahead; or (2) because it lost the arbitration.

Arbitration is expected to create such stalls in cases of high traffic towards a single destination. Our above argument shows that their effects can spill over to fan-out trees under two conditions: (1) stall happens at the leaves of fan-in tree, and it effects the following packets, which did not leave fan-out trees yet; or (2) stall happens closer to the root of fan-in tree, and packets are backed up due to limited storage capacity.

In summary, (ii.b) is the only case that interference can happen in MoT network. For this case to occur, the demand for the a particular destination needs to be high. Due to address hashing mechanisms used in the memory system, as described earlier in Section 2.2, this condition does not happen frequently.

Case (ii.b) is similar to *head-of-line* blocking in switches of other networks, where data destined for one switch output is blocked behind data waiting for another port [28]. This can be prevented by use of a separate set of virtual channels for each output port.

Networks such as bus, ring, mesh, butterfly, hypercube and fat trees suffer performance losses due to interference described in case (i). Our implementation of MoT topology eliminates the kind of interference in case (i), and improves network performance.

4.8 Summary

We introduced a new implementation of Mesh-of-Trees (MoT) network between processing clusters (PCs) and memory modules (MMs) of the memory architecture described in Section 2.2. We described the topology, routing and flow control methods. We presented a floorplan for the MoT networks, discussed the differences from earlier implementations of MoT topology, and the advantages of our implementation.

Chapter 5

Switches of MoT Network

5.1 Introduction

The Mesh-of-Trees (MoT) network is built by connecting multiple instances of different types of switch primitives, following the network topology described in Section 4.2. Two main primitives are *routing* and *arbitration* primitives, which are used in *fan-out* and *fan-in* trees respectively. A basic MoT network can be built with these two primitives. A third *pipeline* primitive can be used to reduce delays on long wires by splitting such wires in smaller segments. The *butterfly* primitive can be used as part of a hybrid network for reducing area cost (Chapter 7). It is not part of the basic MoT network, but discussed in this chapter for sake of completeness.

In this chapter, we first present our earlier studies on reduced-synchrony arbitrate-and-move circuits [9]. These circuits contain elements of both synchronous and asynchronous design principles: the periodic clock signal is combined with asynchronous handshake signals between consecutive switch primitives of the network. Functionally, they correspond to an *arbitration* primitive, where the circuit arbitrates among flits at the inputs, and passes one to the next stage.

Reduced synchrony circuits promise high frequency operation, especially when implemented using dynamic logic circuits [9]. For reasons that we will discuss after presenting these circuits, we chose to focus on a fully synchronous implementation,

and defer reduced-synchrony or fully asynchronous designs for future studies. In fact, an alternative fully asynchronous implementation, presented recently in [42], is based on the current study.

Following reduced synchrony circuits, we continue with synchronous switch primitives. We explain their operation, and analyze and compare their performance with existing switch models. Finally, we build a MoT network using the synchronous primitives, and analyze and discuss the performance of the network.

5.2 Queue Model of MoT Network

The topology of MoT network shows that it consists of identical fan-out trees and identical fan-in trees. Considering the queue model presented in Section 3.3.1 [28], we discuss the queue model of MoT by considering those trees individually.

Figure 5.1 shows the queue model of the fan-out tree. When we split a packet stream with exponential interarrival times, the sum of the traffic rates of each output stream is equal to the traffic rate of the input stream [28]. Assuming uniform random distribution, and exponential interarrival times between packets (such as in Bernoulli process), the traffic generated at the root (with generation rate λ) will be halved at each level of the tree. Assuming the tree has $l = \log N$ levels, the traffic will be diluted to $\frac{\lambda}{N}$ at the leaves.

Similar to splitting, combination of multiple input streams with exponential interarrival time results in one output stream with the rate equal to the sum of input streams [28]. Then the operation of the fan-in tree is similar to the fan-out

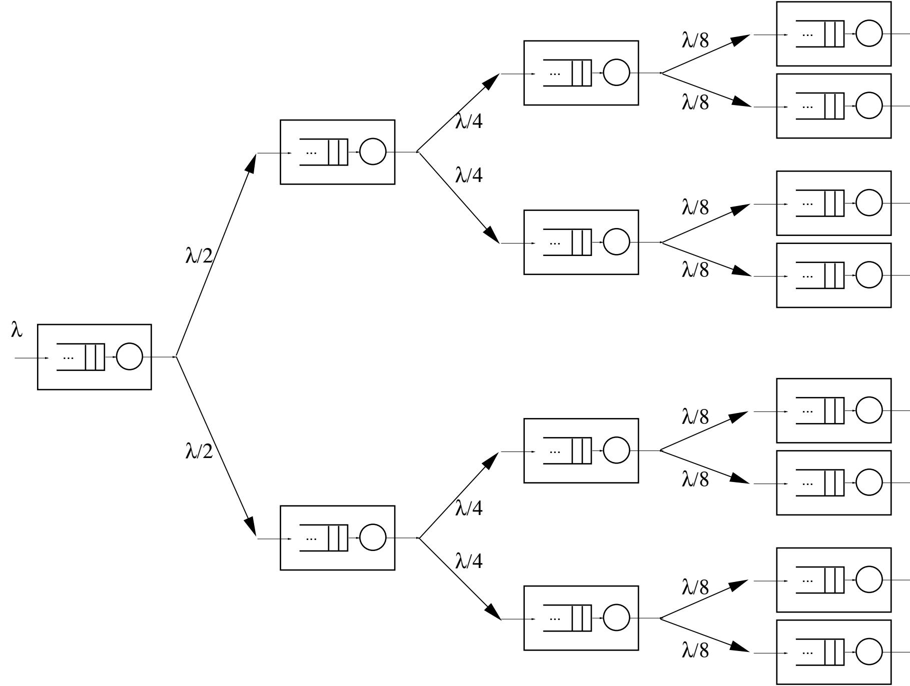


Figure 5.1: Queue model of fan-out tree.

tree, but in reverse direction.

A leaf of fan-out tree is directly connected to a leaf of a fan-in tree. Bases on our analysis of fan-out trees, each leaf of fan-in tree will receive $\frac{\lambda}{N}$ traffic. At next level, the traffic coming from two branches will combine to $2\frac{\lambda}{N}$. As a result, the traffic rate leaving the root node of a fan-in tree will be equal to λ.

As we discussed earlier, this model assumes that the queues are arbitrarily deep. This condition is difficult to realize, because of hardware cost of buffers. Therefore, we build more accurate simulation models to evaluate our network. The queue model described in this section represents the ideal case that we want to achieve.

5.3 Earlier Arbitrate-and-Move Primitive Implementations

Flits in the arbitration (fan-in) tree of a MoT network advance to the root by step-wise arbitration. In each arbitration cycle, one among incoming requests is arbitrated at each fan-in tree node, and moved to its parent node. We call them *arbitrate-and-move* circuits [9].

Research on state-of-the-art arbiter circuits were motivated by applications, where the performance of crossbar switches and bus-based networks is directly related to the latency of arbitration (e.g. [43, 81, 91, 112]). These works focused on N-to-1 arbitration, where a request vector of N-bits is given, and an N-bit grant vector is generated with only one granting signal among N bits.

In cases of crossbar network applications ([43,91,112]), the grant vector is used to configure the switches to connect input ports to output ports. Communication (moving data from input to output port) follows switch configuration. In case of bus network applications [81], the grant signal decides which source owns the bus in the next cycle to transmit data. In all of the above approaches, arbitration among N elements precedes the data movement.

In a crossbar context, *pipelined multiplexers* are used in [106], where standard multiplexers and flip-flop embedded multiplexers are connected in an alternating way to move data in a pipelined path. However, this design lacks arbitration function, and the steering signals for multiplexers are obtained from a global crossbar scheduler. On the other hand, [55] describes a distributed crossbar scheduling in binary-tree form, however the decision and data flits do not advance in a pipelined

path.

The latency of traditional arbiter circuits is measured from the instant the request vector is modified to the instant when the grant vector is updated. N-bit arbiter circuits in each of [81, 91, 112], were built in a tree structure, using 2-input or 4-input primitives. Critical delay path, which influences latency directly, extends from leaves to the root [81] and in some cases, from leaves to the root and then back to leaves [91, 112].

In contrast, we combine arbitration and data movement. We move the data one level up in the binary tree, towards the root (output port), as soon as we arbitrate between two neighboring requests. Arbitration and movement repeat at each stage until the data reaches the root. Each stage of the pipeline has the latency of a single (2-to-1) arbitrate-and-move primitive, which promises great increase in throughput.

Arbitrate-and-move circuits can be implemented following asynchronous, synchronous (see Section 5.4.3), or reduced-synchrony methods. We first summarize a fully asynchronous design based on the concept of Micropipelines [94]. We then discuss the reduced synchrony (RS) circuits and present two implementations with static CMOS gates (RS-Static) and dynamic TSPC [110] gates (RS-Dynamic) respectively. Then, we summarize our results in [9].

The main idea of reduced-synchrony arbitrate-and-move primitives is the meaning and implementation of the clock signal, which propagates in the tree in reverse direction of the data, i.e. from root to leaves. It is also used as an acknowledgment signal, similar to asynchronous handshake protocols. If a child node has a request

towards its parent, a clock pulse from the parent indicates the acknowledgment of the request.

5.3.1 Asynchronous Implementation

Single-input-single-output pipelines built by this approach operate at the speed of a single C-gate [90, 92]. Arbiter and Call blocks [94] are required along with the pipeline segment, to build the node circuit. Figure 5.2 shows the block diagram of this circuit. Empty squares represent control logic components of micropipeline segments, and empty ellipses represent the data latches or registers.

We implemented Arbiter and Call blocks as described in [67] and [59] respectively. The former generates two mutually exclusive grant signals, which, in turn, are converted to a single request by the latter. According to [67], a second request will be served at least 11-gate delays after the first one.

We ran a simulation of a ring oscillator [84] circuit to figure out the minimum gate delay for $0.18\mu m$ design technology. 50ps propagation delay of an inverter suggests that our node circuit cannot operate at a rate faster than 550ps between consecutive requests. Detailed simulations show that the delay is much higher than 550ps, due to high fan-out of the gates.

5.3.2 Reduced Synchrony Implementation

The reduced synchrony circuits proposed below share some properties with the synchronous and asynchronous approaches, yet they do not fit exactly into any of

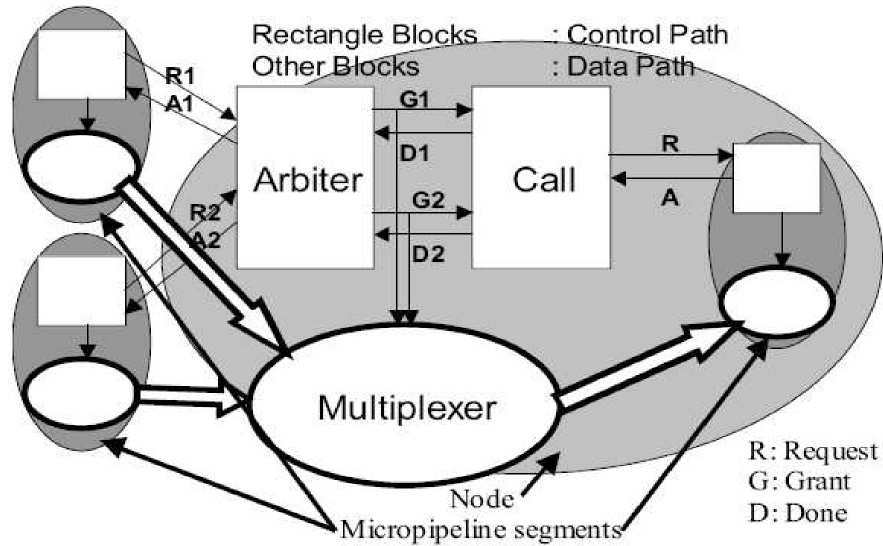


Figure 5.2: Asynchronous arbitrate-and-move primitive.

them. We unified acknowledgement signal and clock pulse as a new design approach, to simplify the implementation and conserve power.

These circuits are not connected to a global clock directly. An external fast-clock signal is connected to the root circuit only. The children receive clock pulses from their parents, only when needed. There is synchrony between a parent and its immediate children, but there is no global synchrony as in a fully synchronous circuit. Hence, we use the term, reduced synchrony.

We initially designed such a circuit using static CMOS gates. Further observations suggested that a design with dynamic gates could yield better performance in terms of speed and power.

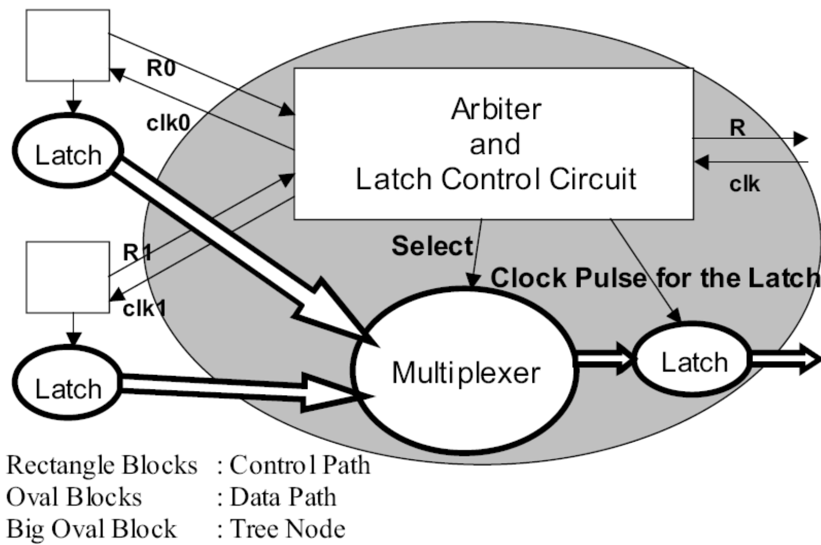


Figure 5.3: Reduced-synchrony arbitrate-and-move primitive.

5.3.2.1 Static Gate Implementation (RS-Static)

The node is connected to its children as shown in Figure 2. It operates as follows:

1. If no requests come from the children, no request signal is sent to the parent, and both children receive a clock pulse.

2. If a single request comes from a child, it passes, and the clock pulse is passed to both of the children.

3. If two requests come, one passes first and then the other. Only the child with passing request receives a clock pulse.

The implementation can be seen in Figure 3. The critical delay, which determines the clock period of this circuit, is the amount of time between the generation of the clock pulse at the parent node and the update of the request signal at the

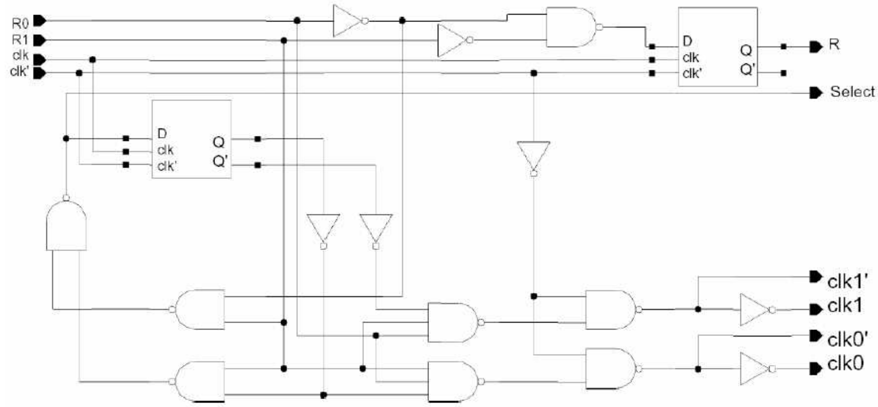


Figure 5.4: Schematic of RS-Static arbitrate-and-move circuit

child node. The former consists of a 2-input nand gate and two inverters, and the latter consists of a D-latch, implemented as a transmission gate and 2 inverters. Since some of these gates have a fan-out of 3 or 4, we cannot achieve the delay of a gate of the reference ring oscillator as described for the asynchronous circuit in Section 3.1.

5.3.2.2 Dynamic Gate Implementation (RS-Dynamic)

The high-level node structure is the same as in Figure 2 except that the clock signals to the children are unified to clk-out signal. Dynamic logic gates from True Single Phase Clocking (TSPC) family [110] are used for this implementation (Figure 4). Each gate executes a simple logic function and latches the result for one clock period. This allows the parent node to modify signals of its children. The high-level algorithm is as follows:

1. If no requests come from the children, no request signal is sent to the parent, and both children receive a clock pulse.

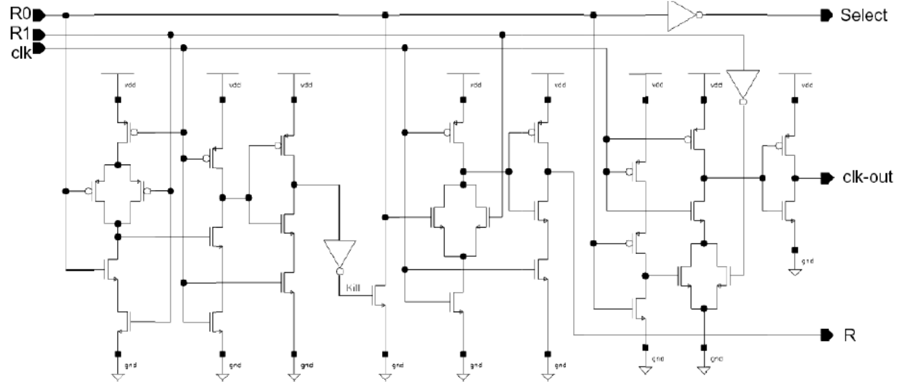


Figure 5.5: Schematic of RS-Dynamic arbitrate-and-move circuit

2. If a single request comes from a child, it passes, and the clock pulse is passed to both of the children.

3. If two requests come, the one from Child 0 passes, then the parent kills that request. The clock signal is not passed to the children. (At the next cycle, the request at Child 1 remains but Child 0 does not generate a new request)

If both children send a request at a given cycle, the request of child 1 passes as the only remaining request at the next cycle. Therefore, arbitration is fair, despite of the built-in priority of Child 0. (Figure 5)

The critical delay path is similar to that of RS-Static, however fewer gates are used: Clock pulse is generated through two half gates (dynamic gates) and a buffer. Request is generated through one dynamic gate.

5.3.3 Simulation Results

In this section, we describe our simulation setup for the study of the proposed circuits and report our preliminary results. We used Cadence tools (SPECTRE sim-

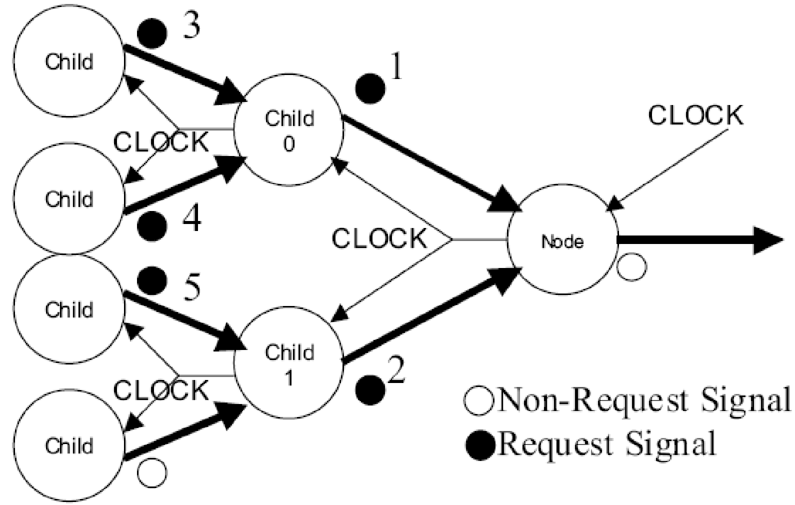


Figure 5.6: Operation of reduced synchrony arbitrate and move circuits: As the Node receives continuous clock pulses, requests pass in order 1, 2, 3, 5, and 4 through the Node

ulator) for all simulations and delay and power measurements. SPICE parameters for $0.18\mu m$ technology are obtained from [68]. A 1.8V source is used.

We simulated a single node of the asynchronous arbitrate-and-move circuit with minimum sized transistors, and an 8-leaf-7-node binary tree for both RS-Static and RS-Dynamic circuits. RS-Dynamic (Figure 5.5) is optimized using Cadence Analog Circuit Optimizer. RS-Static (Figure 5.4) is optimized manually.

As reference, we built a 7-stage ring oscillator using inverters with a $W_p/W_n =$ 13/6 ratio at the same technology. Here, W_p and W_n represent the width of pmos and nmos transistors respectively. We used minimum length for transistors that is available by this technology. Each inverter showed a 49ps propagation delay (50% of the input to 50% of the output) of full voltage swing) and a 73ps rise time (10%

to 90% of full voltage swing). We thus targeted a rise time of 80ps for the clock pulse when we simulate the arbitrate-and-move circuits.

Table 5.1 reports the performance, measured by speed, area, and power, of different circuit implementations for a test design explained above. The root node of RS circuits is connected to a clock generator with various clock periods for our simulation. Cycle is measured as the fastest clock (or equivalently, the shortest clock period) that the circuit can keep pace with for the RS circuits, and the time interval between the request and done signals for the asynchronous circuit. Area is measured as the total transistor area ($width \times length$) and the number of transistors required per node circuit. Power consumption of each circuit is measured as the average power consumed by the global power source on the same request pattern.

The shortest clock period for the RS-static circuit to work correctly is $800ps$, while the RS-dynamic circuit operates correctly at a clock period as fast as $500ps$. The critical delay path of RS-dynamic circuit is even shorter, but we observed that for shorter cycle times, the clock signal does not complete full voltage swing. The asynchronous circuit completes one arbitration cycle in $2.5ns$. No other requests can be processed during this time.

5.3.4 Discussion

All circuits described above contain state holding elements. In the asynchronous one, these elements may go into a metastable state as the node receives two request signals concurrently. Extra circuitry, which reduces the speed of the

Implementation	Cycle	T. Area	T. Count	Power
Asynchronous	2.5 ns	$\approx 35um^2$	188	N/A
RS-Static	800 ps	$38um^2$	72	8.8 mW
RS-Dynamic	500 ps	$28um^2$	34	16.4 mW

Table 5.1: Comparison of asynchronous and reduced synchrony arbitrate-and-move circuits. "T" stands for transistor.

circuit, is required to prevent this from happening. As a result, cycle time increases and overall throughput decreases.

Power consumption is proportional to switching frequency, and load capacitance. A major power consumer in synchronous circuits is the clock tree, which delivers the periodic clock signal from one input to all synchronous elements in the chip. It is usually designed and implemented separately from the functional operation, during chip layout preparation. Asynchronous and reduced-synchrony circuits do not require an external clock tree. In RS circuits, a node sends clock pulse to its children only when it is ready for a new request. Therefore the inner nodes will not always receive the fastest clock signal. In the case when the interconnect tree is fully loaded with requests, there will be only one single branch driven by the clock, from the root towards the leaves, and the remaining part of the tree will not consume any power. Table 5.1 shows that RS-Static is slower but consumes less power compared to the RS-Dynamic circuit. We note that the results were obtained through simulations with synthetic inputs, and not with real data traffic. A more realistic result could be observed with such traffic.

Although the results indicate that reduced synchrony circuits may be a good candidate for the MoT network, we focus on synchronous circuits for our further design and implementation. The following reasons summarize the rationale behind our decision.

1. A drawback of the reduced synchrony design is that selective propagation of the clock signal may reduce overall throughput in terms of flits per cycle, when there are large gaps between consecutive flits. In a synchronous or fully asynchronous design, flits behind the gap may advance even when flits in the front are stalled during arbitration. In a reduced synchrony implementation, the lack of clock signal will slow down such advance. This feature needs improvement in order to provide high throughput in all cases.

2. From the system-wide point-of-view, Mesh-of-Trees network as presented in Chapter 4 did not have a reference implementation in practice at that time. A study of synchronous network would establish foundations for its functional operation and building blocks, and evaluate its cost and performance metrics with respect to other well known synchronous network architectures in practice.

3. Although there are benefits in using asynchronous or reduced-synchrony circuits in the network, the environment of the network, namely processors and memory modules are expected to be synchronous, at least for the foreseeable future. Considering the critical location of the network, robust and efficient interfaces are needed for handling communication between processors, network

and memory. Promising examples of such interfaces [20] were at pre-layout level of maturity around the same time frame as [9], and have not been tested as part of a large-scale system. On the other hand, fully synchronous approach has mature design methodologies from concept to layout, with reusable component libraries that are used for such systems.

4. Finally, based on the design of [9], such interfaces would be required at leaves of each arbitration tree. This would amount in $O(N^2)$ additional cost, and potential performance overheads.

5.4 Synchronous Switch Primitives

In this section, we model the switch primitives as synchronous finite state machines. In a given cycle, the primitives evaluate their own state, and input signals from their predecessors and successors, to determine their next state. The MoT network does not have any central component to schedule or keep track of the states of all primitives. As a result, each switch primitive operates independent of others, except for its immediate neighbors.

In general, primitive has one or more *input channel*s, and one or more *output channel*s. A *channel* consists of a w_c-bit data signal d^i_{XY}, a *request* signal r^i_{XY}, and a negative acknowledgment signal that we call *kill-and-switch* ks^i_{YX}. In this notation, the superscript index i represents the order of the channels, starting from 0. We omit this index if the primitive has only one input or output channel. For subscript indices we use letters A, B and C to represent the predecessor, the primitive under

consideration, and the successor respectively. The order represents the direction of the signal, which originates from the first letter, and terminates at the second. Note that the direction of ks signals are opposite to the data and request signals of same channel. With this notation, we uniquely identify each signal associated to a primitive. For example, Figure 5.7(a) shows a pipeline primitive with its connections to its predecessor and successor.

An assertion of signal r (logic value 1), means that the source of this signal has a packet to pass to its successor on that channel. A de-asserted r (logic value 0) means that the source does not have a packet for its successor on that channel. An asserted ks signal means that the source primitive is able to accept a packet through that channel in the next cycle. A de-asserted ks signal means that the source will not accept a new packet through that channel in the next cycle.

A switch primitive evaluates its state and the signals from its predecessor and successor; and decides on its new state and output for the next clock cycle. State transition occurs in synchrony with the clock pulse.

Next, we discuss each of the switch primitives in the order of increasing complexity, namely, pipeline, routing, arbitration and butterfly.

5.4.1 Pipeline Primitive

This simplest primitive has one input and one output channel (Figure 5.7(a)), and consists of $B \geq 1$ storage registers and control logic. Following the latency-insensitive design methodology of [18], $B = 2$ registers are necessary and sufficient to

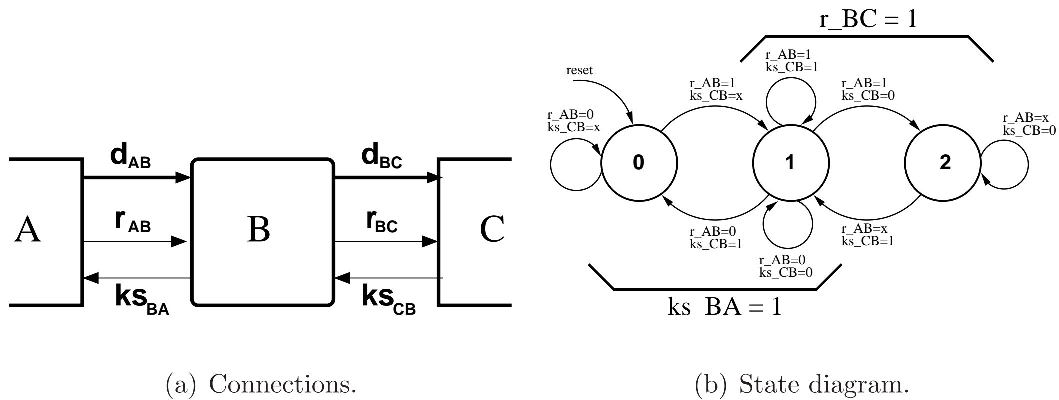
satisfy maximum local throughput, i.e. one flit per cycle, under following conditions:

1. Stall condition is propagated from destination towards source, opposite to the data flow, as opposed to being broadcasted instantly to all primitives

2. Data must be preserved, i.e. not overwritten or discarded.

Therefore, we assume $B = 2$ through this study.

A high-level state diagram of a pipeline primitive with $B = 2$ is shown in Figure 5.7(b). There are $B + 1 = 3$ states, denoted as 0, 1, and 2; they represent cases with 0, 1 and 2 full registers respectively. The arcs represent the necessary input signals to switch between states. On the top and the bottom of the state diagram we show the output signals. In states 1 and 2 r_{BC} is equal to 1, and 0 otherwise. Similarly, ks_{BA} is equal to 1 in states 0 and 1, and 0 otherwise.



(a) Connections. (b) State diagram.

Figure 5.7: Pipeline primitive.

The block diagram of the pipeline primitive is shown in Figure 5.8. Rectangles marked as B_0 and B_1 represent the data registers. Vertical trapezoidal blocks represent multiplexers. Their output is connected to one of the numbered inputs (on the left), depending on the value of the third input (at the bottom). Thick lines

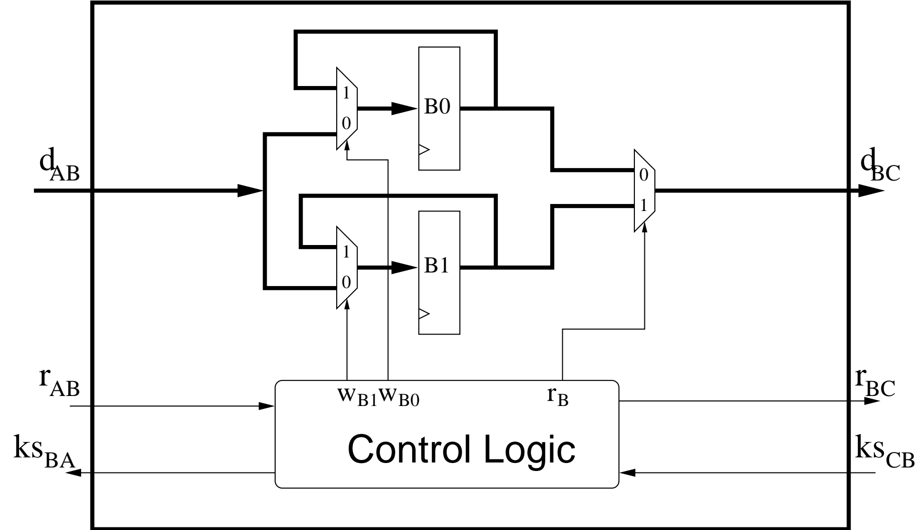


Figure 5.8: Block diagram of pipeline primitive.

represent data paths with w_c-bit signals, thin lines represent control paths with 1-bit signals. Clock and reset signals are omitted for clarity.

5.4.2 Routing Primitive

The purpose of the routing primitive (or fan-out primitive) is to direct a packet from one source to one of the multiple destinations. We classify the routing primitives based on the number of their output channels. A *k-ary* routing primitive has $k \geq 2$ output channels.

A binary routing primitive ($k = 2$) has one input and two output channels. It can be used to build binary routing trees (*fan-out trees*). An l-level tree has an input at the root, and 2^l outputs at the leaves. Using *k-ary* routing primitives with k output channels, *k-ary* trees with k^l leaves can be built similarly.

The routing primitive contains one pipeline primitive attached to the input

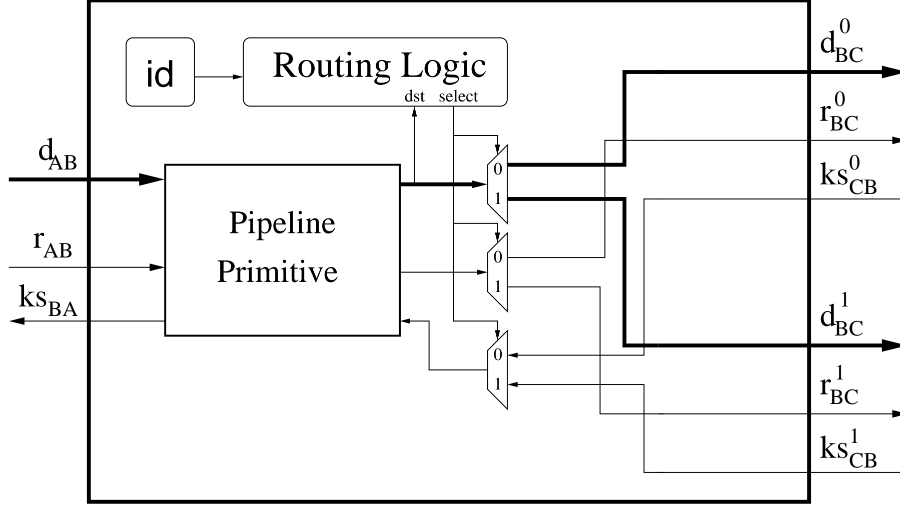


Figure 5.9: Block diagram of routing primitive.

port, and additional combinatorial routing logic. Each routing primitive has a constant *id* signal, that is used in determining the packet direction. Figure 5.9 shows the block diagram of a binary routing primitive.

The high level operation is same for all k. Incoming flit is stored in the pipeline primitive. The destination (signal *dst*) of each packet is encoded as part of the data signal. The signal *select* is derived through a comparison of the encoded destination bits with the *id* signal, which depends on the depth of each routing primitive in the fan-out tree. Based on the outcome the flit is directed to one of the outputs, and other outputs are disabled by setting their corresponding request signal to 0. In other words, if a flit is destined for output port i, then we set $r^i_{BC} = 1$, and $r^j_{BC} = 0$, where $0 \leq j \leq k-1$, $j \neq i$. Similarly, the ks_{CB} signal of the pipeline primitive is connected to the ks^i_{CB} input. This operation is implemented using demultiplexers for data and request signals, and a multiplexer for the negative acknowledgment (ks) signal.

Depending on k, the circuit implementation will have the following differences: A $k-ary$ routing primitive needs to compare $\log k$ bits of dst signal to determine the destination of the flit. In addition, the logic depth of demultiplexer and multiplexer components also increase with k. As a result, the overall complexity of routing increases as k increases. Therefore, we focus on $k = 2$ in this study, without ruling-out the possibility of $k > 2$ cases in future studies and realizations.

5.4.3 Arbitration Primitive

The purpose of the arbitration primitive (or fan-in primitive) is to select one of multiple competing packets at its inputs, and forward it to its output. We use the number of input channels to classify such primitives. A k-*ary* arbitration primitive has k input channels and one output channel.

A binary arbitration primitive has one output and two input channels; and it is used to build a binary arbitration tree (*fan-in tree*). An l-level arbitration tree has 2^l inputs at its leaves, and a single output at its root. Similar to routing trees, k-*ary* primitives can be used to build k-*ary* trees.

A k-ary arbitration primitive contains k pipeline primitives, and combinatorial arbitration logic, with an additional state machine to provide fairness and prevent starving. Incoming flits are stored in pipeline primitives. Based on the dynamically changing priority and arbitration method, one of the stored flits is directed to the output. Other stored flits wait for another cycle.

Figure 5.10 shows the block diagram of a binary arbitration primitive. The

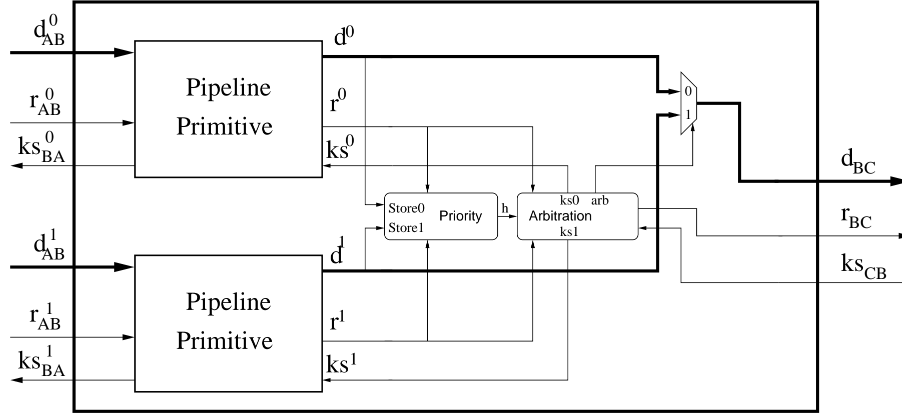


Figure 5.10: Block diagram of arbitration primitive.

Priority block keeps and updates the state of the current priority based on request signals r^0 and r^1. It consists of a priority register and the combinatorial logic for the update. Optionally, the flits can have a single-bit signal, if the current flit and its successor must remain together, for example, in case of a *store word* instruction from processors to memory modules. We discuss this option in Section 5.4.3.3. The *Arbitration* block performs the arbitration based on dynamic priority (denoted h for *history*), r^0, r^1, and ks_{CB} signals. It sets ks^0 and ks^1 signals for the pipeline primitives, the r_{BC} output signal, and the *arb* signal for multiplexing the output data. This block consists of combinatorial logic only.

5.4.3.1 Arbitration Method

We use an arbitration method that prevents starvation of local (switch primitive) and global (network) input sources. Among local inputs of each arbitration primitive, each request is served equally often, when there are continuous requests on each input. This is called *strong fairness*. On the other hand, a single arbitration

primitive is not aware of the global state of network inputs. As a result, network inputs may not be served equally often. However, every request will be eventually served. In other words, there is an upper bound of time, between the appearance of a packet at an input port and its exit at the output port. This is called *weak fairness* [28].

For k-ary arbitration primitives, where $k > 2$, we use round-robin arbiters with dynamic priority assignment [28, 91]. A $k \times k$ arbiter evaluates k inputs, and generates one "winner" among at most k requests (k-to-1 arbitration). The "winner" receives a *grant* signal, whereas other requests wait for the next arbitration cycle. The winner is determined by a dynamic priority setting. At any given cycle, one of k inputs, say input i, $0 < i < k - 1$ has priority. If i has a request at that cycle, it is declared "winner", and it receives the grant signal. If i does not have a request, then $i + 1$ is considered. If $i + 1$ has a request, it becomes the "winner" and receives the grant; if not then $i + 2$ is considered, and so on. After considering input $k - 1$, the order rotates back to input 0, and it continues until input $i - 1$. Here, i has the highest priority, and $i - 1$ has the lowest priority.

In order to guarantee *strong fairness* in the primitive, the dynamic priority is updated every cycle as follows. Regardless of priority holder i, if input j receives *grant* signal at any cycle, input $j + 1$ will have the highest priority at the next cycle. In other words, the input receiving the grant signal has the lowest priority in the next arbitration cycle.

Studies show that the area cost of the arbitration logic in this model is $O(k)$, and the logic delay is $O(\log k)$ with logarithmic-depth tree implementations [55].

5.4.3.2 N-input to 1-output Arbitration

For N-to-1 arbitration, one can use a single-stage arbitration primitive with $k = N$, as shown in many studies of crossbar networks. For this implementation, area cost is $O(N)$, and shortest logic delay is $O(\log N)$. If there is no competition, a request reaches the output in one cycle. In other words, minimum arbitration latency is 1. If there are multiple requests, the worst-case latency is $O(N)$.

In order to build a high-throughput network, arbitration trees with short cycle times are desirable. This can be achieved by using multiple levels of simpler and faster primitives with short cycle times.

Assuming all input signals are available at the rising edge of the clock pulse, the arbitration result *arb* of the binary arbitration primitive in Figure 5.10 can be computed with two levels of NAND gates (or a single AND-OR-INVERT gate in current technology libraries [6]), implementing the following logic function: $arb = (h \cdot r^0 + r^{1\,\prime})'$, where $\cdot$ and $+$ represent logical *AND* and *OR* operations respectively, and x' represents inverted value of x. Here, h represents the *history bit* which is used to keep track of the dynamic priority.

A $\log_k N$ level arbitration tree built with $k = 2$ primitives will consist of $N - 1$ primitives, each with $O(1)$ area cost and logic delay for one clock cycle.

Table 5.2 compares "single k-ary primitive" with "tree of binary primitives". The area cost of both methods are $O(N)$. The clock rate of latter method is $O(1)$, compared to $O(\log N)$ [55] of the former. The minimum and maximum arbitration latency is 1 and $O(N)$ cycles in the former, and $\log N$ and $O(N)$ cycles, in the latter.

Method	Area	Clock Cycle	Min Latency	Max Latency
Single N-ary primitive	$O(N)$	$O(\log N)$	$O(1)$	$O(N)$
Tree of binary primitives	$O(N)$	$O(1)$	$O(\log N)$	$O(N)$

Table 5.2: Comparison of N-to-1 arbitration methods.

As a result, an N-to-1 arbitration tree with $k = 2$ configuration will provide a faster cycle time and higher throughput, compared to $k > 2$ configurations, at the expense of increased minimum arbitration latency.

In the underlying memory system, a high traffic rate is expected through the network. As multiple packets target the same destination module, achieving minimum arbitration latency will be less likely. In that case, the disadvantage of an arbitration tree with $k = 2$ configuration reduces relative to $k > 2$ configurations, without losing the fast clock rate advantage. Therefore, our study focuses on arbitration trees with $k = 2$ configuration.

5.4.3.3 Winner-Take-All Arbitration for "Store" Operations

A part of our performance model is based on exchanging single-flit packets between terminals. In case of a *load* operation, the processor sends the address to the memory module, and the memory module responds with the requested data. In this most common mode of operation, each packet consists of a single flit with sufficiently many bits, that contains either the address or the data.

In case of a *store* operation the processor sends the address and the data to the memory module. A flit could be sufficiently wide to hold both the address and the

data, however this would waste bandwidth when *load* instructions are sent through the network. Alternatively, a *store* packet could consist of two flits that are injected consecutively to the network. In this case additional effort is required to relate address and data pairs that belong together, and perform the correct operation. We consider the following two options for handling *store* operations.

- Both flits of address–data pair can be marked with an identifier tag, and sent as individual single-flit packets. The memory commits the operation when the second flit with the matching tag arrives. This method requires computation on the processor and the memory module. The network remains unchanged. This is called *fair bandwidth* arbitration [28], since the arbitration primitives perform fair arbitration regardless of the type of the packet.

- Second flit is chained to the first one, and they follow each other in the network. The memory receives the pair consecutively.This method requires computation in the network. Specifically, the arbitration primitive must ensure that second flit immediately follows the first one. This introduces a temporary bias to the arbitration operation. The processor and the memory modules remain unchanged. This method is called *winner-take-all* arbitration [28]. Extra logic in the arbitration primitive may increase clock period and, therefore, reduce throughput. On the other hand, this method reduces average packet latency for multi-flit packets in terms of clock cycles. The optional signals called $Store_0$ and $Store_1$ in Figure 5.10 are used to perform *winner-take-all* arbitration.

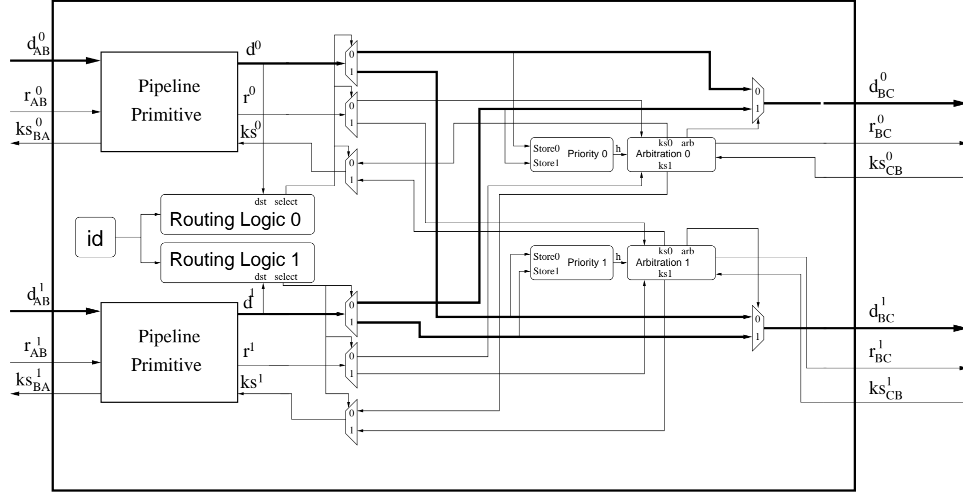


Figure 5.11: Block diagram of butterfly primitive.

We implement these both types arbitration primitives and evaluate their performance with a cycle-accurate verilog simulator. The results show that they both provide similar throughput improvement over the single-flit arbitration. The improvement is significant especially when load operation dominates. See Section 6.3.1 for details.

5.4.4 Butterfly Primitive

The original MoT tree [10] can be built with three switch primitives (Figures 5.8- 5.10). An additional butterfly primitive (Figure 5.11) is used for building the hybrid MoT-BF network. A butterfly primitive consists of one pipeline primitive per input channel, and additional control logic that handles routing and arbitration together. Similar to other switch primitives, a flit spends one clock cycle in the primitive if there is no contention. In case of contention and stalls, proper backward signaling and using the second w_c-bit buffer prevents overwriting stalled data.

5.5 Evaluation

5.5.1 Logic Delay of Switch Primitives

As we stated earlier in Section 2.5, long logic delay of switches in existing networks limits the performance. In this section we evaluate switch delay of the networks.

For switches with virtual channels, we use the analytical results of [79]. For MoT and replicated butterfly network, we build verilog models of switches and synthesize them using Cadence tools, ARM regular-V_t standard cell library [6] and IBM 90nm (9SF) CMOS technology [44]. We normalize all results using technology-independent $FO4$ delay unit that represents the delay of an inverter driving four identical inverters. For the technology and standard cell library that we used, the $FO4$ delay at slow operating corner with 1.08V voltage and 125 C temperature is equivalent to 66.5ps [6][1]

The results are summarized in Table 5.3. For butterfly, ring, 2D-mesh, hypercube and fat trees, we increase the number of virtual channels. MoT and replicated butterfly don't have virtual channels. For MoT, we used the longest critical delay of three switch primitives discussed in Section 5.4. For replicated butterfly, we used the delay of the butterfly primitive (Section 5.4.4), which is the longest delay among all of its switch primitives. We assumed 32-bit data path in our computations. Our results show that attempts to improve throughput by increasing virtual channels

[1]We also note that in a typical operating corner, with 1.2V source and 25 C temperature, the $FO4$ delay is approximately 35ps.

will increase switch delay at the same time, and reduce clock rate.

Switches of hypercube network have $\log N + 1$ input and output ports, where N is the number of network terminals. Therefore, peak throughput of hypercube would reduce with increasing number of terminals. Similarly, a butterfly network built with larger switches with e.g. 4-ports, would have longer clock period, and therefore lower peak throughput, compared to a butterfly with 2-port switches.

Leiserson [58] states that the root capacity of the fat tree for N terminals is between $N^{2/3}$ and N, where the capacity between the network and each processor at the leaves is defined as 1. The capacity increases exponentially at each level between leaves and the root. The number of the input and output ports to the switching nodes is proportional to the capacity at that level of the tree. Therefore, in Leiserson's fat tree, number of switch ports increases between leaves and root, reaching $N^{2/3}$ at the root switch. Alternative fat tree architectures as described earlier in Section 2.3.4 are built with small switches with constant number of ports. Smallest of these switches has 4 input and 4 output ports. As a result of this, they will provide higher peak throughput compared to Leiserson's fat tree, however they will fall short compared to the peak throughput of butterfly network with 2-port switches.

In [79], the routing stage of switches are not analyzed, and they are assumed to be less than 20 $FO4$ delay, which used to be the typical clock rate for earlier serial processors. In most networks that we consider, routing can be performed by checking a single bit at each stage. Therefore we think that this will not affect the critical path for switches of butterfly, hypercube and fat trees. However, if routing

Number of VCs (v)	2	4	8	16	32	64
2-port (butterfly)	10.9	14.2	17.5	20.8	24.1	27.4
3-port (ring)	11.9	15.2	18.5	21.8	25.1	28.4
4-port (fat tree)	12.5	15.9	19.2	22.5	25.8	29.1
5-port (2D-mesh)	13.1	16.4	19.7	23.0	26.3	29.6
7-port (hypercube-64)	13.9	17.2	20.5	23.8	27.1	30.4
Replicated butterfly	9.40					
MoT	8.98					

Table 5.3: Single switch delay of various networks (in FO4). Replicated butterfly and MoT do not have virtual channels.

takes as long as 20 $FO4$, the minimum switch delay for these networks will be 20 $FO4$.

Next, we discuss the delay for routing primitives with $k > 2$ outputs. Similar studies for arbitration primitives with $k > 2$ inputs [91] show that the logic delay increases as $O(\log k)$. In order to estimate the additional switch complexity for $k > 2$, we synthesize routing primitives for $k = 4$, 8, and 16 outputs; and measure the worst delay between clocked registers and output signals. Note that this delay is less than 8.98 $FO4$ as shown in Table 5.3, because (i) it is measured between registers and output, where additional circuit for $k > 2$ is inserted; and (ii) the routing primitive is relatively simpler than the arbitration primitive.

This delay depends on the load at the outputs, and we do not know the actual load before the layout. Therefore, we report results, where the outputs are

loaded with the smallest (BUFX2) and largest (BUFX20) buffer cells available in the standard cell library [6]. According to the data book, these cells correspond to a capacitive load of $5.0fF$ and $8.8fF$ respectively. If buffers are inserted to reduce long wire delays, it is reasonable to expect loads within this range. Figure 5.12 shows that the measured delay increases logarithmically with increasing k.

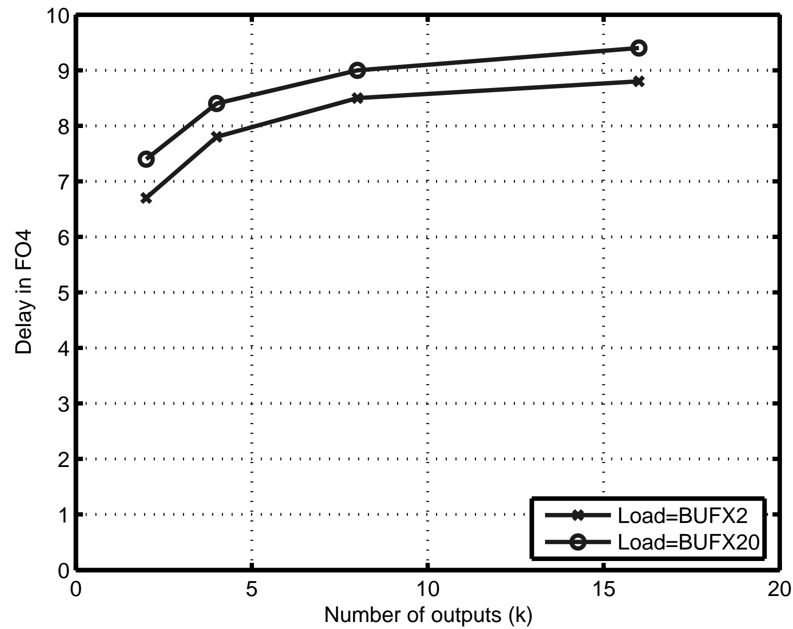


Figure 5.12: Register-to-output delay of routing primitive with different number of outputs.

5.5.2 Maximum Network Throughput

In order to evaluate the maximum throughput provided by each network model, we assume the maximum traffic generation rate of one flit per cycle (1.0 fpc)

at each input port of the network[2]. At this generation rate, the network will saturate with packets, and the injection and delivery rates will come to balance at the maximum throughput. We assume random traffic pattern.

We obtain the results for hypercube, butterfly, 2D-mesh, and ring networks from simulation using the simulator of [28]. The results for fat tree networks are from [80] and [75]. For MoT and replicated butterfly we used our own simulator as discussed in Section 3.4.1. As one can see from Table 5.4, the proposed MoT network can provide the highest maximum throughput, which is 76% and 28% higher than butterfly and hypercube with $v = 4$ virtual channels, and 3% and 16% higher than butterfly and hypercube with 64 virtual channels, respectively. The maximum throughput of 2D-Mesh and Ring networks decrease as N increases.

Configuration	**Max Throughput**
Hypercube $N = 16$ $v = 4$	0.777
Hypercube $N = 64$ $v = 4$	0.763
Hypercube $N = 16$ $v = 16$	0.787
Hypercube $N = 64$ $v = 64$	0.843
Butterfly $N = 16$ $v = 4$	0.602
Butterfly $N = 64$ $v = 4$	0.553
Butterfly $N = 16$ $v = 16$	0.861
Butterfly $N = 64$ $v = 64$	0.946
2D-Mesh $N = 16$ $v = 4$	0.677

[2]Note that our single-flit per packet assumption represents the majority of processor-memory communication, as explained in Section 3.4.2

Configuration (cont.)	**Max Throughput**
2D-Mesh $N = 16$ $v = 16$	0.800
2D-Mesh $N = 64$ $v = 4$	0.352
2D-Mesh $N = 64$ $v = 64$	0.500
Ring $N = 16$ $v = 4$	0.239
Ring $N = 16$ $v = 16$	0.472
Ring $N = 64$ $v = 4$	0.061
Ring $N = 64$ $v = 64$	0.179
Fat Tree $N = 256$ $k = 4$ $n = 4$ $v = 2$	0.55
Fat Tree $N = 256$ $k = 4$ $n = 4$ $v = 4$	0.72
BFT $N = 64$ $v = 4$	0.28
BFT $N = 64$ $v = 8$	0.30
Mesh of Trees $N = 16$	0.951
Mesh of Trees $N = 32$	0.963
Mesh of Trees $N = 64$	0.977

Table 5.4: Maximum throughput (in *flits per cycle per port*) provided by different networks (N: numbner of terminals, v: number of virtual channels, BFT: *Butterfly Fat Tree.*).

We plot area cost vs performance for these networks and MoT for various number of terminals (Figure 5.13). Area cost is computed as the number of data registers in Section 4.7.2. Performance is obtained by maximum throughput simulations as

described above.

Cost and performance of replicated butterfly increases as we increase the number of copies (r); and cost and performance of virtual-channel networks increase as we increase number of virtual channels (v). There is single MoT configuration on each chart for a given number of terminals (N). Replicated butterfly achieves higher performance at comparable cost, with respect to virtual-channel networks. On the other hand, MoT network achieves higher throughput for comparable cost, or comparable throughput at lower cost for up to $N = 64$ terminals.

5.5.3 Throughput and Latency Under Varying Traffic

As traffic in the network increases, packets will experience longer latencies. We follow the guidelines in [28] to design simulations in order to evaluate the throughput and latency of various network models under different input traffic. We use Bernoulli process and random traffic pattern. (Section 3.4.2).

The network is warmed-up until the throughput stabilizes, then marked packets are injected for latency measurement. We are particularly interested in the case when the input traffic, or the on-chip parallelism, is high. 2D-Mesh, Ring, Hypercube and butterfly networks are simulated on the simulator provided by [28] with $N = 64$ terminals, and different number of virtual channels, namely a typical $v = 4$ setting and an aggressive $v = 64$ setting. Router switches have three cycle switch latency per *speculative virtual channel router* design of [79]. MoT network is simulated using an RTL SystemC simulator that we implemented and validated [8,10].

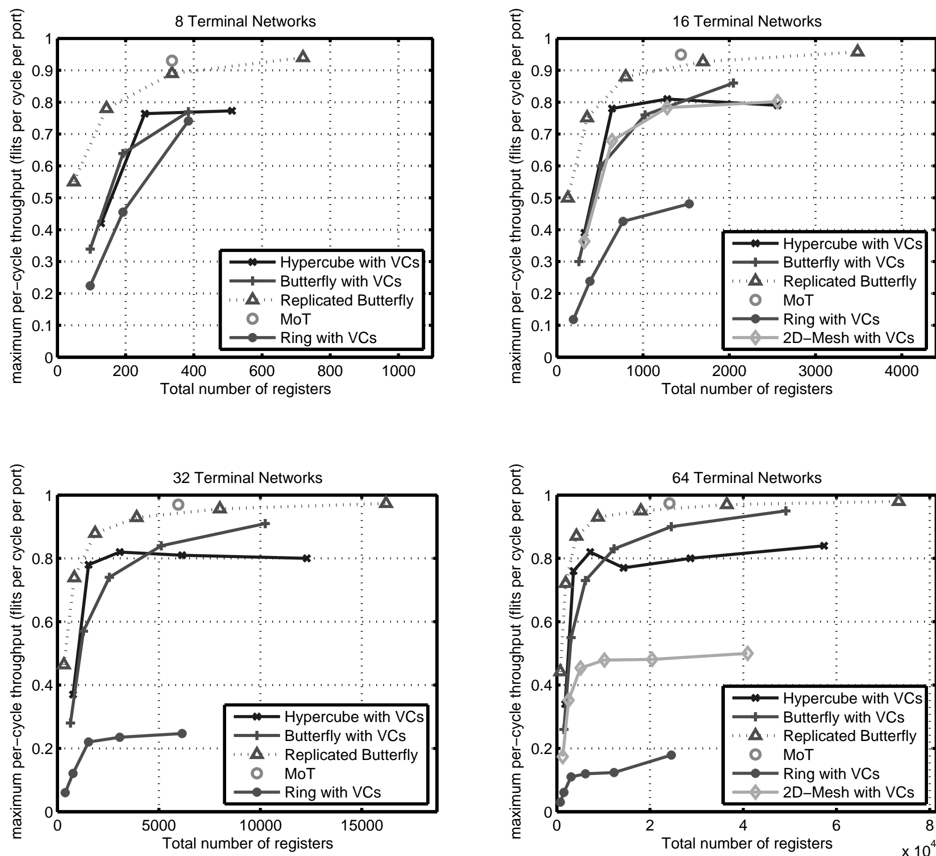


Figure 5.13: Cost-performance comparison of networks. On the curves, number of virtual channels for hypercube, butterfly, 2D-mesh and ring are doubled from left ($v = 2$) to right ($v = N$). With Replicated butterfly, the number of copies is doubled from left ($r = 1$) to right ($r = N$).

We vary the input traffic from the low 0.1 fpc per port to the maximum 1.0 fpc. The latency of a flit is measured as the time from it is generated to the time it is received at the destination, which includes the waiting time at the source queue. For each input traffic rate, we use different seeds to generate a set of traffic with the same traffic rate. These input traffic sets are injected to simulators for each

network model and the average throughput and latency are reported in Figure 5.14.

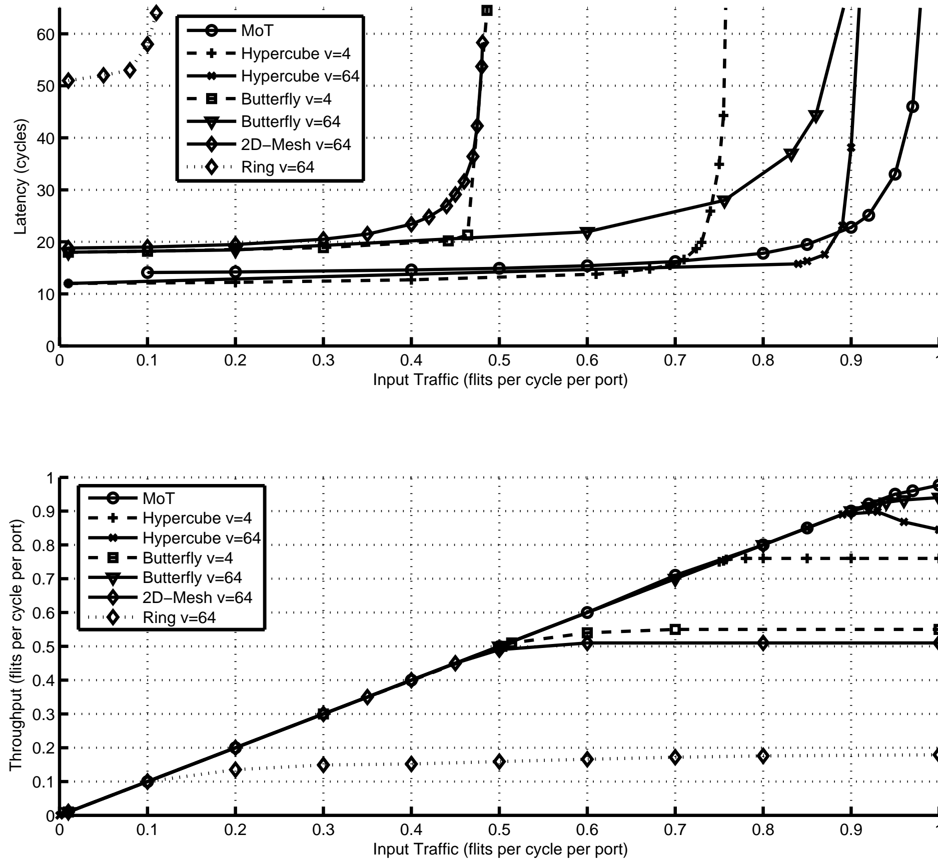


Figure 5.14: Throughput and latency of various networks for $N = 64$ terminals.

MoT network provides competitive throughput and latency and has a clear advantage over others when the input traffic is high. More importantly, MoT network has a more predicable latency when the input traffic varies. For example, when we increase the input traffic from 0.1 fpc per port to 0.9 fpc, the hypercube latency increases by a factor of 3.2, butterfly network latency increases by a factor of 3.9, while MoT latency increases only by a factor of 1.6. This could allow more accurate design and analysis of algorithms as described in [101].

5.6 Summary

In this chapter, we described the switch primitives of MoT network. First, we discussed reduced synchrony arbitrate-and-move circuits, and evaluated them. Next we detailed the operation of synchronous switch primitives. We synthesized each primitive, and compared experimental results with existing switches for similar purposes. We built a MoT network with these primitives, and simulated it with a cycle-accurate simulator written in C++ and SystemC. We compared the network performance with existing network architectures.

Our results show that MoT switch primitives operate faster than virtual-channel switches, that are used with popular network topologies such as 2D-mesh, ring, butterfly, hypercube, and fat trees. When wire delays are not considered, this advantage immediately translates into higher throughput in terms of Gbps. Wire delays will cause some degradation in performance, which we evaluate in Chapter 6. We note that wire delays can be reduced by inserting pipeline stages following latency-insensitive design principles [18,19]. Therefore, the simplicity and high speed of MoT switches is a significant advantage over more complicated virtual-channel switches.

Chapter 6

Layout

6.1 Introduction

Previous chapters presented the Mesh of Trees network, and evaluated its cost and performance with comparisons to existing networks. This chapter brings the concept of MoT network closer to reality by generating a chip layout, and evaluating its layout-accurate cost and performance. Specifically, we measure clock rate, and power consumption of MoT networks of different sizes, and show the effects of wire pipelining. The layout of an 8-terminal network is fabricated using IBM 90nm technology [44], and tested.

Our design flow starts with RTL-level verilog description of switch primitives. Our own high-level synthesizer generates verilog files higher level modules, such as balanced binary trees. All verilog files are then synthesized and mapped to ARM standard cells [6] using *Cadence RTL Compiler* tool, and placed and routed using *Cadence Encounter* tool. Measurements are taken using the Cadence Encounter tool.

We connect the fabricated chip to an FPGA board for testing. The FPGA is programmed to generate the input signals we used during verilog simulations. We expect to obtain the same output signals as we observed during verilog simulations. Although we lowered operating frequency significantly, we were unable to observe

Terminals	4	8	16	32	64
Bits per flit	26	28	30	32	34
Cell Area	0.064	0.314	1.419	6.166	26.289
Wire Area	0.003	0.020	0.135	0.863	5.197

Table 6.1: Wire and cell area (in mm^2).

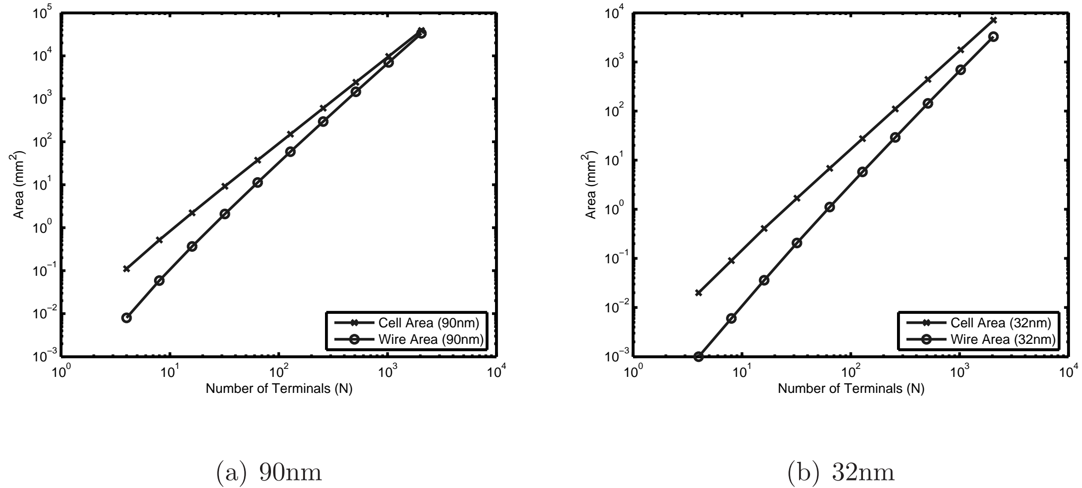
any meaningful output. From output pins, we observe that none of the tested chips transition to the output state. This is a clear mismatch with gate-level simulations before we release to manufacturing, and could indicate manufacturing defects.

6.2 Network Layout

The wire area of the MoT network grows as $O(N^2 \log^2 N)$, and the number of tree nodes grow as $O(N^2)$, where N is the number of terminals [10]. This would imply that the wire area will dominate the cell area, and the floorplanning must consider wire area constraints. Synthesis results with this particular technology and standard cell library show that cell area is larger than the wire area for practical number of terminals. Table 6.1 shows these results for different network configurations that are considered in this paper.

Wire area grows faster and it can exceed cell area for higher number of terminals and bits per flit. We estimate wire and cell area for different number of terminals for $90nm$ and $32nm$ technologies. Our estimation (Figure 6.1) shows that this does not happen for networks with practical sizes at least until $32nm$ technology node. This justifies our assumption that register area and count is a reasonable

measure for area cost.



(a) 90nm

(b) 32nm

Figure 6.1: Wire and cell areas for $90nm$ and $32nm$ technology nodes. For $90nm$, we assumed 52-bit wide flits, and 8 layers of metal. For $32nm$, we assumed 80-bit flits, and 11 layers of metal. In both technology nodes we reserved bottom 3 and top 2 layers of metal for standard cell and power routing respectively, and did not include in our estimation. For the rest of the routing, we assumed 2 vertical and 1 horizontal layers with $90nm$, and 3 vertical and 2 horizontal layers with $32nm$.

Our floorplan and placement strategy in this study is based on the cell area of the network. In a network with N terminals, we create $N/2$ partitions in order to improve layout quality during placement and routing. Figure 6.2 shows a network with 8 terminals that has 4 partitions marked P_0 to P_3. An initially square floorplan is separated into partitions, and each partition is individually placed, routed, and optimized. Depending on other geometrical factors, such as height and width of terminal modules, two partitions could be separated by a gap.

6.2.1 Terminal Circuits

Ideally, our network would interconnect parallel processors and memory modules. We use a terminal node to replace a pair of cluster and memory module. In order to focus on the interconnection network, these nodes are dummy terminals that generate random requests based on programmable parameters, and record statistics upon receiving a packet.

The terminal modules do not affect critical delay path of the network modules. However, since they are generating packets and recording arrivals at each cycle, their critical delay path affects the operation frequency of our taped-out chip. Therefore, we report critical delays for the network module separately.

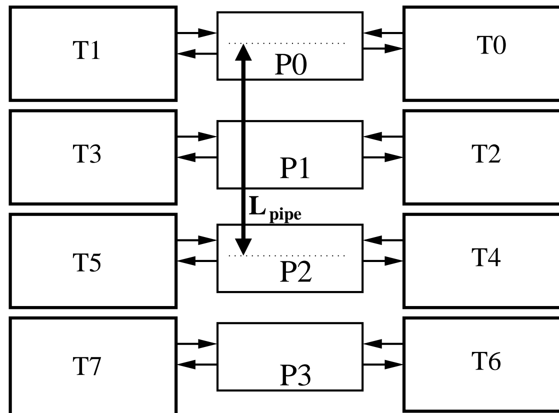


Figure 6.2: High level chip floorplan for 8-terminal network. Terminal modules: T_0 to T_7. Network partitions: P_0 to P_3.

6.2.2 Pipeline Insertion

Long wires of MoT network could increase the clock period and reduce the throughput. Inserting pipeline registers to long wires would improve performance

[4,22,39,62]. Earlier work [10] proposed to use a pipeline primitive to cut long wires in shorter segments. However, the benefits could not be demonstrated without a physical layout.

Pipeline insertion can be automatized by several ways. State of the art synthesis tools are capable of inserting repeaters. However, they are usually unaware of final wire lengths. Place and route tools can insert any standard cell or module to an existing netlist and connect them to rest of the circuit. However, this requires use of low level commands of the specific tools, and may not be portable. Furthermore, state changes in the circuit cannot be traced back to RTL-level. This could complicate verification and performance evaluation. Our high level synthesis tool inserts pipeline registers at RTL level. Then, the network would have a portable and coherent state machine view through the entire physical design flow.

It is challenging to estimate the optimal wire length to fit in a single pipeline stage. It involves multiple physical design iterations. Furthermore, CAD tools perform several proprietary and heuristic optimizations. Therefore, it is virtually impossible to estimate the exact wire length between two consecutive registers before the layout is finalized.

In this prototyping study, we follow a high level heuristic approach to determine the amount of pipelining, guided by the wire length between the centers of partition P_i and the second partition P_{i+2}, denoted as L_{pipe} in Figure 6.2. Thus, we allow the signals to pass over one full partition P_{i+1} without being stored in a pipeline register. For lack of space, we only note that following this model, an 8-terminal network would not require pipelining. Furthermore, 16 and 32-terminal

networks will ideally operate at the same frequency as the 8-terminal network.

6.3 Results and Discussion

In this section, we first present simulation results that validate the claims of [10] and provides average throughput per cycle. Then, we lay out networks with 4, 8, 16 and 32 terminals, and obtain their clock rate. The combination of both results will give layout-accurate average throughput for MoT. Finally, we taped-out the 8-terminal design for fabrication.

We used IBM CMOS9SF 90nm technology [44] and regular ARM/Artisan SAGE-X standard cells [6]. Typical operating conditions ($V_{DD}; T$) for this library is given as $1.2V; 25°C$. In this paper we report delay estimations for a slow corner (worst case) operating conditions, such as $1.08V; 125°C$. We use NC-Verilog for simulations, Cadence RTL Compiler for synthesis, and Cadence SOC Encounter for layout generation. For tape-out, we use Synopsys Hercules for DRC, and Cadence Virtuoso for final details in layout.

6.3.1 Simulation Results

Latency and throughput results with verilog simulations for a 64-terminal network is compared with results in Section 5.5.3 in Figure 6.3. Table 6.2 compares the average throughput at highest traffic rate, and latency at three traffic levels. *Low*, *High*, and *Max* represent flit generation rates of 10%, 90%, 100% of network capacity. Throughput is averaged over all terminal ports, and latency is averaged

over all recorded packets.

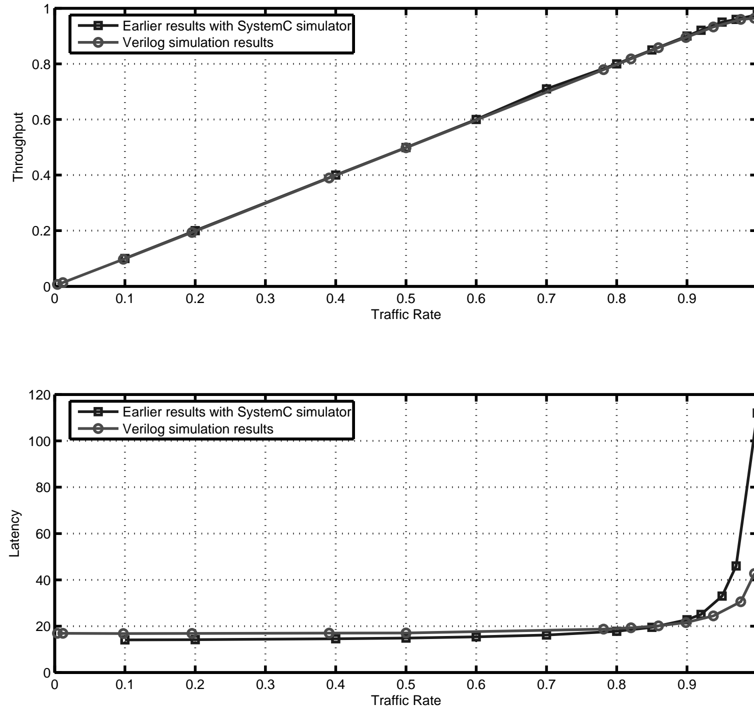


Figure 6.3: Throughput, and latency of 64-terminal MoT at various traffic rates. Verilog simulations compared to earlier results. Throughput is measured in terms of flits per cycle and averaged over all ports.

Compared to results of [10], throughput differs between 1% to 2%. Latency results for 64 terminal MoT network are 17% higher for low-traffic case and 6.5% lower for high-traffic case. Such deviations are expected due to different implementation of *source queue* component as described in Section 3.5.2.

Next, we simulate the network with different ratios of 1-flit and 2-flit packets, to model a mixture of *load* and *store* operations. Traffic rate is adjusted for each run so that average flit injection rate remains constant at the maximum capacity of

Terminals	4	8	16	32	64
Tput from [10]	N/A	N/A	0.95	0.96	0.98
Average Tput	0.88	0.91	0.93	0.95	0.96
Latency (low)	8.64	10.8	12.8	14.8	16.9
Latency (high)	18.0	16.9	17.9	19.3	21.6
Latency (max)	26.6	29.8	33.6	38.0	42.7

Table 6.2: Simulation results for different network configurations. Throughput is measured in flits per cycle per port, at the maximum traffic generation rate of 1 flit per cycle per port. Latency is measured in cycles.

the network, namely 1 flit per cycle per port. Higher traffic rates would saturate the source queue in the terminal. In that case several packets would be dropped, and the mixture rate could change. For example, a mixture ratio of 30% means that each cycle there is a 77% probability of generating a packet. Additionally, the generated packet has two flits with a probability of 30%, and one flit with a probability of 70%. As a result, the average rate of flit generation is 1.0 per cycle.

We simulated *fair arbitration* and *winner-take-all* arbitration methods as described in Section 5.4.3.3. The variation in latency and throughput for 64-terminal network is shown in Figure 6.4. The *wide flit* case assumes that the flit width is doubled so that any one of *load* or *store* operations fits in a single flit.

Simulations show that using multiple flits for *store* instructions improves throughput for almost all mixture ratios. There is no significant difference between two methods of arbitration. Layout of both arbitration primitives shows that the in-

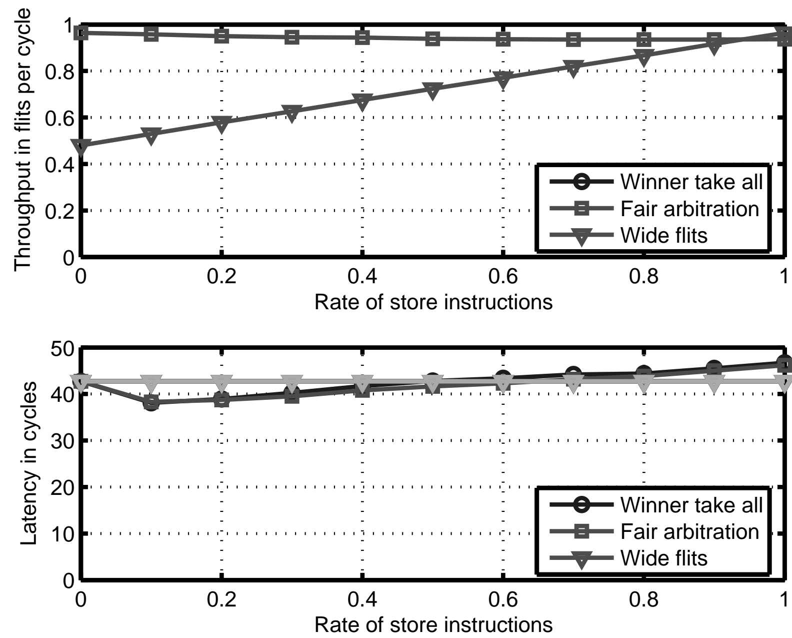


Figure 6.4: 64-terminal MoT simulation results for different methods of handling store operations.

crease in clock period due to additional logic is negligible. Latency is improved for low amounts of *store* instructions, but this could also be caused by the source queue implementation.

For 64-terminal network, *fair arbitration* has slightly lower latency. Additional simulations show that for a 4-terminal network, *winner-take-all* has lower latency. We conclude that the number of flits in a *store* instruction is not sufficiently high to make a difference in latency. Further studies with more flits per packet would be beneficial to evaluate MoT performance for cases where multiple data words are moved through the network, such as loading or storing long vectors or streams.

Config.	**4**	**8**	**16**	**32**	**16 p**	**32 p**
Clock Rate	970	890	680	578	748	764
Bits per flit	26	28	30	32	30	32
Peak Tput	101	199	326	592	359	782
Avg Tput	88.6	180	302	563	334	747
Low trf lat	8.64	10.8	12.8	14.8	13.5	17.8
High trf lat	18.0	16.9	17.9	19.3	18.7	22.6
Cell area	0.08	0.41	1.89	6.5	1.88	7.3
BBbox area	0.16	0.74	3.21	13.4	3.21	13.4
Power	72	268	794	N/A*	967	N/A*

Table 6.3: Comparison of MoT configurations after layout. The letter 'p' indicates pipelined configuration. "BBox" stands for bounding box. Clock rates are in MHz; throughput values are in $Gbps$; latency values are in cycles; area values are in mm^2; power is in mW. *Due to constraints on computing resources, these results are not available.

6.3.2 Layout Results

Following the standard flow of the Cadence tools, we synthesized, placed and routed networks with different configurations. Table 6.3 shows the area and performance results.

We extended the 8-terminal configuration with power routing and I/O pads for fabrication. The final layout is shown in Figure 6.5.

Table 6.3 shows that the clock frequency decreases as the number of terminals

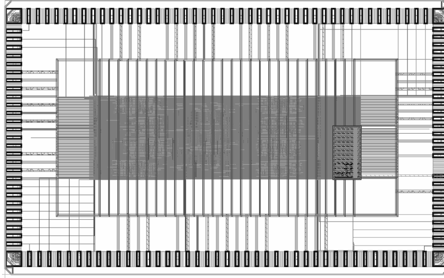


Figure 6.5: Final layout of 8-terminal chip.

increases. This is mainly caused by longer wires on the critical path. Results of pipelined configurations 16p and 32p show the benefit of pipelining on frequency and throughput. Average latency increases in pipelined configurations due to increased number of stages between some sources and destinations.

Partitioning constraints prevented optimal pipeline placement on long wires. Therefore, the improved frequency did not reach the expected level of an 8-terminal network. Reducing the critical length for pipelining could improve performance. Pipeline circuits would be placed within the partitions, instead of between them. Such improvements could incur additional area and latency cost. Evaluation of these trade-offs requires further studies.

Table 6.3 shows that the cell area of laid-out networks exceeds estimations (Table 6.1), since the layout tool optimizes for performance by inserting repeaters and using larger cells.

Cell area of 32p is larger than 32, as expected, due to additional pipeline stages. In 16p, the area of added pipeline stages turn out to be comparable to large repeaters on long wires of 16. Therefore, the area of 16p is approximately equal to the area of 16.

The area of the bounding box is approximately twice as much as the cell area, because of the gaps between partitions, and overestimated design margins. We introduced gaps between partitions in order to level the partitions with the terminals (Figure 6.2). The amount of gaps depend on the area and aspect ratio of terminal circuits. In an ongoing study, we are investigating the relationship between processor geometry, and MoT area and performance. In this prototyping study we did not optimize for the area. However, based on Table 6.1, we expect the actual area to be close to the cell area.

Power consumption has been estimated based on the layout, and simulated switching activity with highest traffic rate. As expected, the power consumption grows quadratically with the number of terminals, that is, at the same rate as the number of cells. Pipelining increases power consumption by both adding more cells, and increasing operating frequency. In this study, we did not optimize for power consumption. However, typical approaches such as clock-gating could reduce power consumption.

6.4 Physical Testing

After fabrication, bare dies of the MoT chip (Figure 6.6) have been packaged (by third-party suppliers), and tested, as previously described in Section 3.5.3.

A functionally correct chip is expected to operate similar to the simulations, and generate same statistical data for throughput and latency. It turns out that none of the tested chips generate any data, and they show signs of being defective.

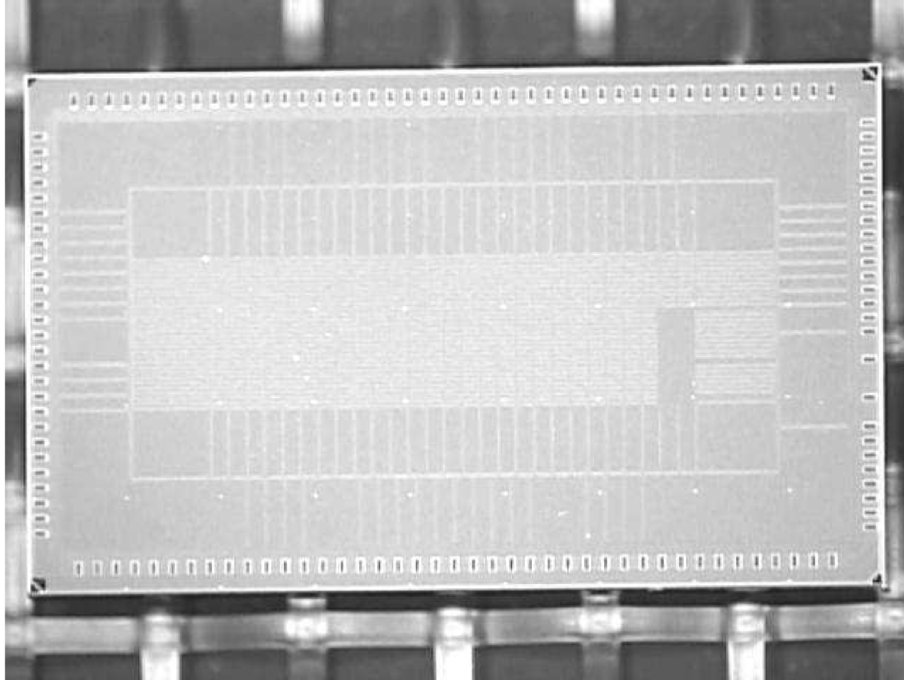


Figure 6.6: Die photo of 8-terminal chip.

Our tests and results are listed below.

First, we verified that the inputs to the network chip are similar to the simulation signals. In that case, a regular *write-execute-read* sequence is expected to produce the simulated waveform shown in Figure 6.7. Instead, we observed the waveform in Figure 6.8, where no output is produced at *SEROUT* signal.

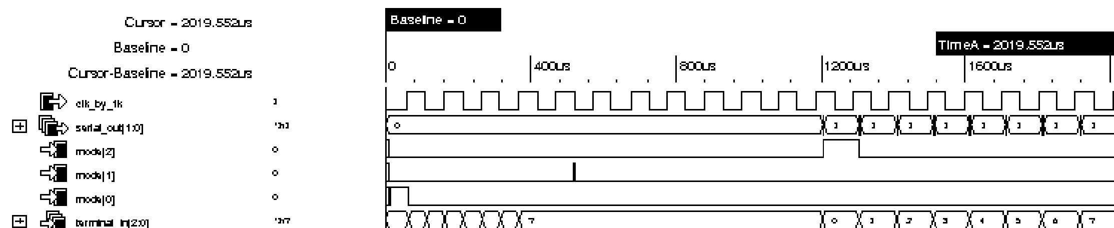


Figure 6.7: Simulation output.

Next, we tried to observe the *read* phase of our regular sequence in isolation.

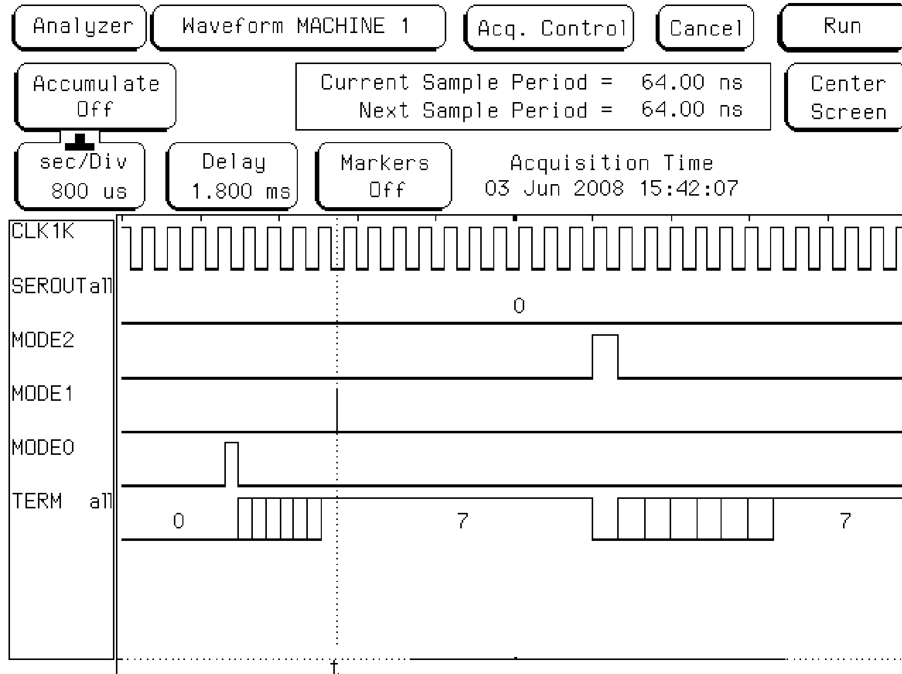


Figure 6.8: Test output 1.

Simulation suggests that we expect to observe a change in *SEROUT* signal, which is a 2-bit wide serial output signal. However, as shown in Figure 6.9, no output is observed, again.

This behavior, namely no output at *SEROUT*, is unexpected; because according to the state machine design, and the simulations of the terminal module, a state change occurs when *MODE2* input receives a pulse. In the new state, *SEROUT* cannot have the value 0, but it must have a value of 1 or 3. Two experiments that we described above suggest that this state change may not be occurring as expected. Another alternative is that the state changes properly, but *SEROUT* is defective.

Our tests are not conclusive, because we cannot observe any repeatable pattern as response to out inputs. Furthermore, we cannot observe or control any signal in the chip. Next, we discuss how this can be achieved in the future.

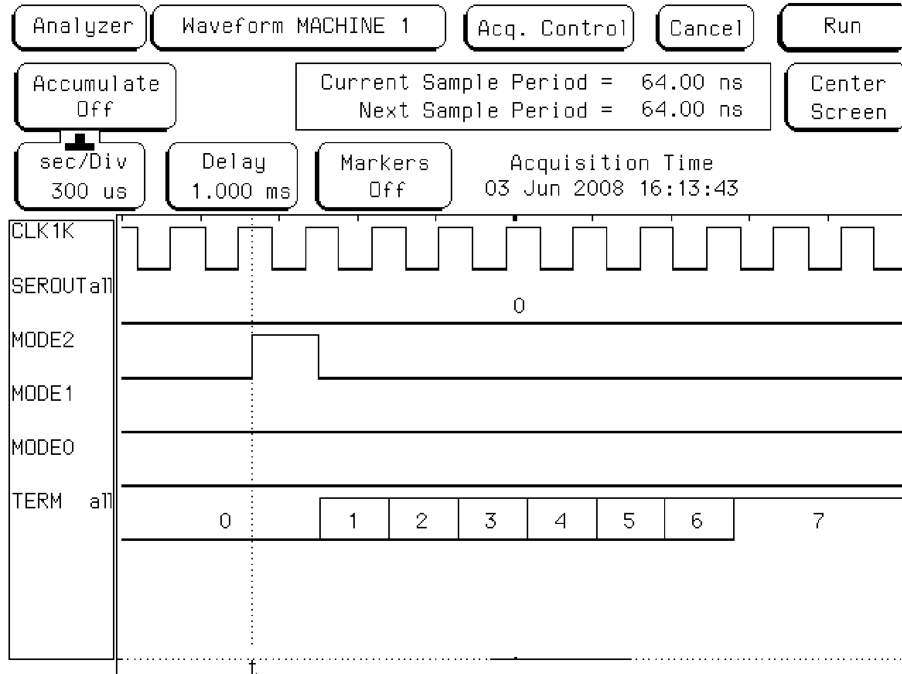


Figure 6.9: Test output 2.

6.4.1 Lessons Learned

As a result of this experiment, we realize that on-chip diagnosis and testing mechanisms are crucial for such prototyping studies. These mechanisms are available as part of industry-standard design tools, and they are called "Design For Testing" (DFT) components [17].

Our network chip did not include these components because of time constraints that we had during layout design. The layout that has been embedded in XMT processor design has been finished later, and as a result it contains these components.

According to [17], DFT components can be inserted in a design with either partial, or full coverage over all flip-flops in the design. Full coverage provides the ability of observing all flip-flops at a desired instant during operation. On the other

hand, with partial coverage some flip-flops cannot be observed.

Based on our experience with this prototyping study, we strongly believe that there are great benefits to apply full DFT coverage for any chip that contains complicated digital systems such as the MoT network and multiple terminal circuits, at least for the first prototype.

Chapter 7

Area Improvement Through Hybridization

7.1 Introduction

In earlier chapters we discussed the Mesh-of-Trees (MoT) network, and evaluated its cost and performance. Our results show that the register area of MoT grows quadratically with number of network terminals, making it impractical for large systems with many terminals.

In this Chapter, we propose hybrid MoT networks called MoT-BF, where we replace part of MoT network by butterfly (BF) networks of small scale. A BF network is area efficient, but it performs poorly under heavy traffic in terms of throughput and latency, particularly when the number of network terminals is large. In a hybrid network, traffic is diluted through MoT network; hence each mini-BF is subject to low traffic, mitigating the high traffic performance loss of pure BF network. We conduct a comprehensive evaluation of the proposed hybrid MoT-BF network in terms of area, latency and throughput. Mathematical analysis, cycle-accurate simulation and post-layout results all show that the proposed hybrid MoT-BF network can significantly reduce the area cost of MoT network with negligible performance degradation.

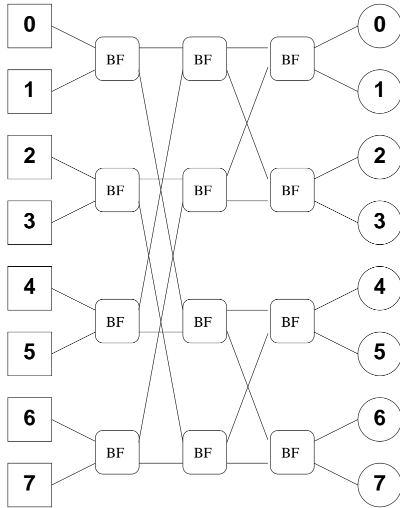
7.2 Hybrid MoT Network

Earlier studies considered hybrid networks to optimize network cost and performance. A notable example is the Cube Connected Cycles (CCC) network [82], proposed to optimize high switch degree of hypercube networks. CCC network is built by replacing corners of a 3-dimensional cube with a group of terminals that are interconnected by a smaller ring network. This reduces the degree of each switch node of a CCC network from $O(\log N)$ to $O(1)$, where N is the number of terminals.

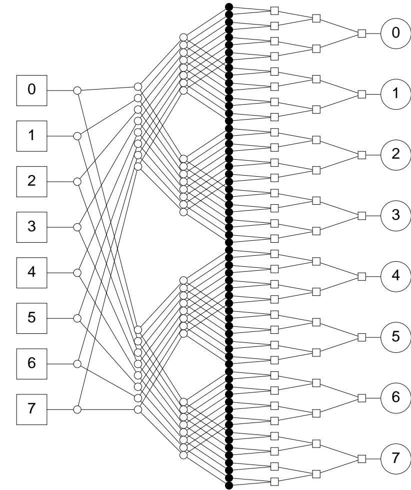
We propose a hybrid MoT-BF network, where inner levels of trees are replaced by mini-butterfly networks. We chose BF network due to its proven area efficiency [28].

7.2.1 Network Architecture

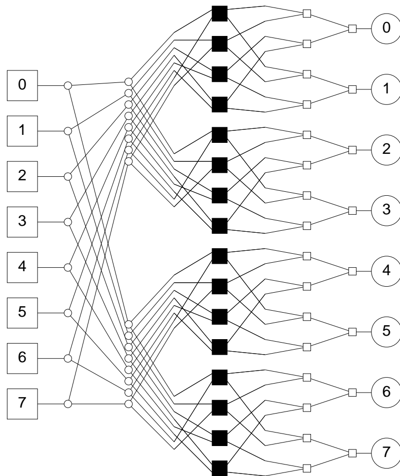
In a regular MoT network with N PCs and N MMs, we enumerate the levels of fan-out and fan-in trees by $\{0, 1, ..., \log N - 1\}$, where the root node is at level 0, its children are at level 1, and so on. In a hybrid MoT-h-BF network, we replace the h inner levels (levels numbered $\{\log N - h, \log N - h + 1, ... \log N - 1\}$) of both fan-out and fin-in trees by BF networks. We refer the number $h \in \{0, 1, 2, ... \log N - 1, \log N\}$ to as the *hybridization level*. The remaining fan-out and fan-in trees both have $\log N - h$ levels. They are connected by $(N/2^h)^2$ mini-BF networks with h stages or 2^h terminals. (see Figures 7.1(b) and 7.1(c) for an MoT-1-BF network with N=8 terminals). Also note that pure MoT and pure BF networks can be represented as MoT-0-BF and MoT-log N-BF, respectively.



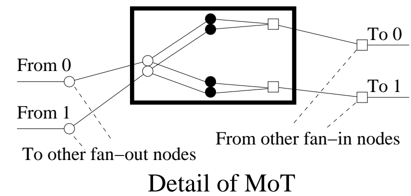
(a) 8-terminal butterfly network.



(b) 8-Terminal MoT.



(c) 8-Terminal MoT-1-BF.



From 0
From 1
BF
To 0
To 1
To other mini−BF
From other mini−BF
Detail of MoT−1−BF

(d)

Figure 7.1: Butterfly, Mesh-of-Trees, and Hybrid Networks. (a) Butterfly network with 2x2 BF switch primitives, connecting 8 sources (numbered squares) and 8 destinations (numbered circles). (b) MoT network. (c) MoT-1-BF network. 3 Innermost columns of MoT network are replaced by mini-BF networks (black squares). (d) Details of the conversion. BF box in (a) and (d) represents the butterfly primitive in Figure 5.11.

The main drawback of BF network is its poor performance (low throughput and high latency) at high traffic rate [10]. The proposed hybrid MoT-h-BF network reduces the traffic through mini-BF networks by the fan-out trees. Each root of the fan-out tree in the MoT-h-BF network will have $2^{\log N-h} = N/2^h$ leaves. If λ is the amount of uniform traffic in terms of flits per cycle that enters the root of fan-out tree, each input to the mini-BF networks, which is the leaf of the fan-out tree, will have a reduced traffic rate of $2^h\lambda/N$ in average, which is 0.25λ for the MoT-1-BF network with N=8 in Figure 7.1(c). This will significantly reduce the congestion and performance loss in BF networks at high traffic rates.

The original MoT tree [10] is built with three primitives (Figures 5.8, 5.9, and 5.10). The fan-out tree primitive performs a routing operation by directing an incoming flit to one of the two outputs. A fan-in tree primitive performs an arbitration between two incoming flits and sends the winner to the next stage. Finally, a pipeline primitive is used to cut long wires in shorter segments if necessary. An additional butterfly primitive (Figure 5.11) is used for building the hybrid MoT-BF network. Each primitive consists of 2 clocked w_c-bit[1] registers per input channel, several mux and demux and control logic that handles routing and arbitration. In an empty network, a packet spends one clock cycle in each primitive. In case of contention and stalls, proper backward signaling and using the second w_c-bit buffer prevents overwriting stalled data.

[1] w_c is the number of bits in a channel, typically 32.

7.3 Evaluation

We evaluate the proposed hybrid MoT-BF network in five categories, register count, minimum latency, throughput-area trade-off, network latency at different traffic rates, and post-layout throughput. We compare the proposed network to MoT, replicated BF, and virtual-channel BF networks.

7.3.1 Register Count

Modern VLSI processes can provide almost up to 10 metal layers, and this number increases every few generations. As a result, wire complexity becomes a secondary concern, at least for reasonably small scale networks, such as 64 terminals. The network area is dominated by the data registers. Therefore, we measure register count of networks, which is directly related to the area cost.

In typical virtual-channel routing switches [28], there are v *virtual channels* per input and output port to improve performance. Each virtual channel uses at least one w_c-bit register for one data packet. In MoT, RBF and MoT-h-BF networks, each switch primitive has either one or two input and output ports and no virtual channels. In all types of switches, the control circuit consumes negligible area compared to data registers.

Mesh of Trees A MoT network consists of N fan-out and N fan-in trees, each with $(N-1)$ nodes. The leaves do not contain switching circuits, since they are only wire connections. Using the primitive circuits of [10], the total number of w_c-bit registers is $R = 6N\,(N-1) = O(N^2)$.

Virtual-Channel Butterfly Switches of butterfly network have a total of $2 \cdot N \log N$ input and output ports with v virtual channels each. Then, the number of registers becomes $R = 2 \cdot v \cdot N \log N = O(vN \log N)$.

Replicated Butterfly Replicated butterfly switches have two registers per input, and no virtual channels. The network consists of r copies of a regular butterfly, and binary trees between the network and source/destination modules In total, they have $R = 6 \cdot N(r-1) + 2 \cdot r \cdot N \log N = O(rN \log N)$ registers.

Hybrid MoT-BF A MoT-h-BF network with N terminals has N fan-out and fan-in trees, with $\log N - h$ levels. Additionally, there are $(N/2^h)^2$ mini-BF networks with h stages. BF primitives have two registers per input, and no virtual channels. As a result, a MoT-h-BF network has $R = 6N(N/2^h - 1) + (N/2^h)^2 \cdot 2h \cdot 2^h = O(hN^2/2^h)$ registers.

Table 7.1 compares register counts of MoT-BF and pure MoT networks for small number of terminals, up to $N = 64$. A 64-terminal MoT-1-BF network has approximately 34% less registers compared to pure MoT.

It is also important to observe the asymptotical behavior of register count. Since h varies between 0 and $\log N$, the number of registers for MoT-h-BF network is asymptotically upper bounded by pure MoT network; and asymptotically lower bounded by pure BF network. For example, if $h = \log N/2$, then $R = O(N\sqrt{N} \log N)$.

N	8	16	32	64
MoT	336	1440	5952	24192
MoT-1-BF	0.62	0.64	0.66	0.66
MoT-2-BF	0.33	0.38	0.40	0.41
MoT-3-BF	0.14	0.20	0.23	0.24

Table 7.1: Register count of some hybrid MoT-BF networks normalized to MoT with same number of terminals.

7.3.2 Minimum Latency

Minimum latency is the time in clock cycles, for a packet to travel from source to destination through an empty network. Usually it is averaged over all source-destination pairs, however it does not vary between such pairs in any of the considered networks.

Mesh of Trees A packet travels $\log N$ stages in the fan-out tree, and $\log N$ stages in the fan-in tree. Each stage takes one clock cycle [10]. The overall latency is $L = 2 \log N$.

Virtual-Channel Butterfly The butterfly network has $\log N$ stages of switch nodes. A packet takes three cycles to pass through a regular virtual-channel switch, or at least two cycles to pass through a speculative virtual-channel switch [79]. Assuming regular switches, the minimum latency of a virtual-channel butterfly is $L = 3 \log N$.

Replicated Butterfly In a replicated butterfly network with r copies, the packets travel through a $\log r$ stage trees before and after the butterfly. Assuming

single-cycle switches, the minimum latency is $L = 2 \log r + \log N$.

Hybrid MoT-BF In a MoT-h-BF network, packets pass through $\log N - h$ levels of fan-out and fan-in trees before and after h level butterfly. With single-cycle switches, the minimum latency is $L = 2 \log N - h$.

As h is limited between 0 and $\log N$, the minimum latency of MoT-h-BF network varies between $\log N$ and $2 \log N$ for different hybridization levels.

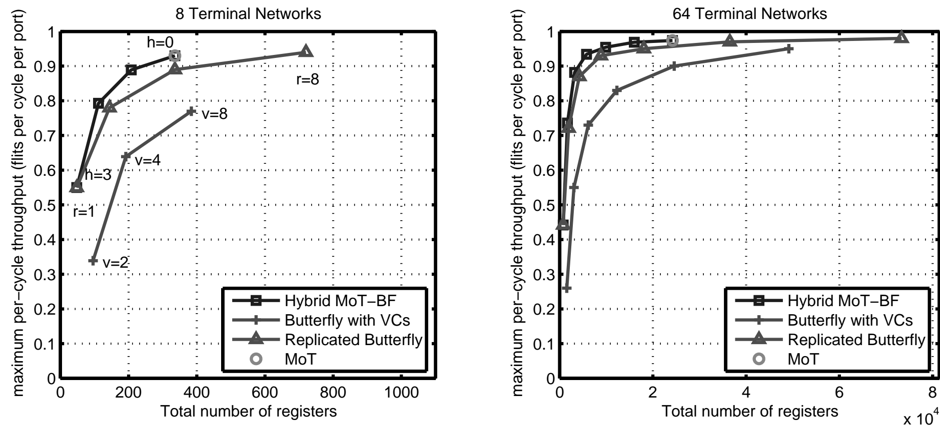


Figure 7.2: Cost-performance comparison of different network configurations. In each plot, upper left region represents high performance and low area. On the curves, number of virtual channels for VCBF doubles from left ($v = 2$) to right ($v = N$). For RBF, the number of copies doubles from left ($r = 1$) to right ($r = N$). For MoT-h-BF, the hybridization level decreases from left ($h = \log N$) to right ($h = 0$).

7.3.3 Throughput-Area Trade-off

We evaluated maximum throughput of each network by cycle-accurate simulations. For virtual-channel butterfly network, we used the simulator of [28]. For

other networks, we use the simulator of [10].

In order to evaluate the maximum throughput provided by each network model, we assume the maximum packet generation rate of one flit per cycle at each input port of the network[2]. At this generation rate, the network will saturate with packets, and the injection and delivery rates will come to balance at the maximum throughput. We assume uniform traffic pattern, which is expected for the memory architecture described in Section 2.2. Uniform traffic pattern is a reasonable assumption due to the use of hashing mechanism, which has an effect similar to randomization that distributes the memory accesses evenly among modules [2,7,35].

We simulated networks for $N = 8, 16, 32$ and 64 (see Figure 7.2 for $N = 8$ and $N = 64$). For each network size, we tuned the throughput by modifying the amount of registers, which are directly related to area cost. Specifically, we modified number of virtual channels v in virtual-channel butterfly, and number of copies r in replicated butterfly networks. As expected, we see that the maximum throughput increases for each network as the number of registers increases, Hybrid MoT-BF network outperforms both BF networks.

7.3.4 Latency and Throughput vs. Traffic

As network traffic increases, packets will experience longer latencies, and network throughput will saturate. We use a Bernoulli model to generate packets [28], with generation rates varying from 0.1 to 1.0 flits per cycle per port. The network is

[2]Note that in several other studies of interconnection networks, long data packets may be divided into shorter units, called *flits*. In this network, each packet consists of a single flit.

warmed up until throughput saturates, then marked packets are injected for latency measurement. We are particularly interested in the case when traffic rate, or the on-chip parallelism is high.

Results are shown in Figure 7.3. At lower traffic rates networks with high hybridization levels have lower latency, because they have fewer stages. At higher traffic rates, packets start to interfere with each other. Networks with lower hybridization levels perform better, and their throughput saturates at higher traffic rates.

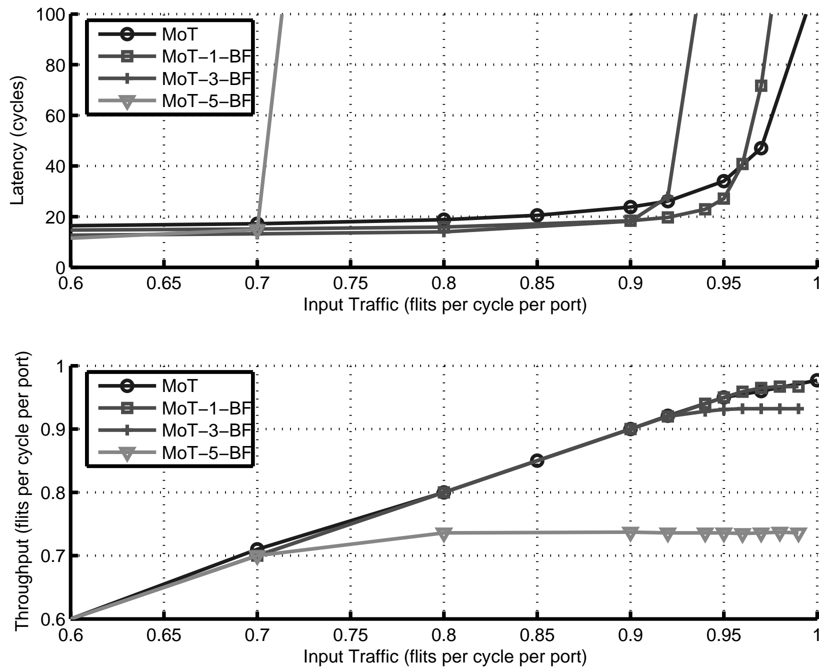


Figure 7.3: Latency and throughput of 64-terminal hybrid networks as input traffic changes. There is no notable difference among networks when traffic rate is lower than 0.6 flits per cycle. Pure MoT results are also plotted for comparison.

7.3.5 Post-Layout Throughput

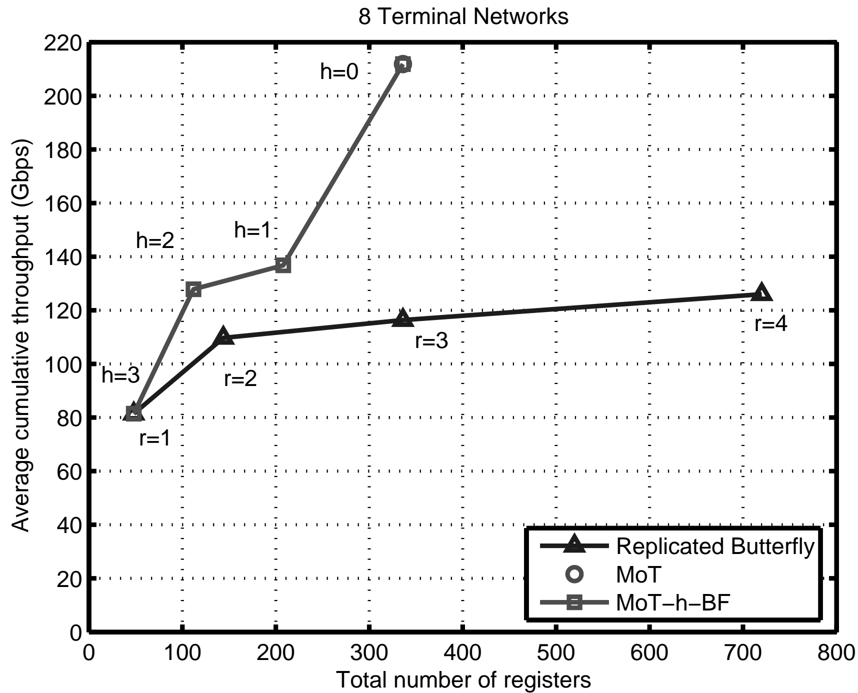


Figure 7.4: Post-layout throughput of MoT, replicated BF and MoT-h-BF networks. Each flit is assumed to be 32-bits wide.

In general, throughput is measured in terms of *Gigabits per second* (Gbps). This value is determined by number of bits in a flit, number of flits delivered per cycle (*per-cycle throughput*), and clock rate. The number of bits in a flit usually depends on the width of the data path, and it depends on the environment, i.e. the parts of the system that remain outside the network. We assume that this parameter will remain constant with different networks and configurations. Per-cycle throughput depends on network type and architecture. It is usually measured through network simulations (Sections 7.3.3 and 7.3.4). The clock rate depends on technology-specific parameters, network and router architecture, and physical layout of the network.

Many earlier studies of interconnection networks omit this component, or make safe assumptions because VLSI issues such as wire delays could be neglected at older technologies. On the other hand, some recent studies recognize the importance of the issues and discuss clock rate as well [4].

We create layouts for 8-terminal MoT, RBF and MoT-h-BF networks in order to compute their highest clock frequency and layout-accurate throughput in terms of *Gbps*. We use ARM regular-V_t standard cell library and IBM 90nm (9SF) CMOS technology. Results are shown in Figure 7.4.

Clock rate of a pure MoT network is the highest among all measured networks, because MoT primitives have shortest delays. Therefore, it has highest throughput in Figure 7.4. BF primitives perform more complex operation, and this increases their delay. Hybrid networks contain both faster MoT primitives and slower BF primitives. Place and route tools are capable of balancing the wire delays among these primitives to optimize clock rate. As a result, the operation frequency of hybrid MoT-BF networks lies between pure MoT and pure BF networks.

7.4 Summary

A hybrid network architecture incorporating mesh-of-trees (MoT) and butterfly (BF) networks was presented. MoT provides superior performance with $O(N^2)$ area cost, where N is the number of network terminals. BF network provides poor performance with $O(N \log N)$ area cost. We replaced inner levels of MoT network with mini-BF networks to build the MoT-BF hybrid network. Based on our analy-

sis, area cost of MoT-BF network lies between pure MoT and BF networks. Under uniform traffic assumption, traffic through mini-BF networks is diluted by preceding fan-out trees. This reduces congestion and related performance loss in mini-BF networks. Simulation results validated that MoT-BF performance is between MoT and BF networks up to 64 network terminals.

Operating at same clock rate, a 64-terminal MoT-1-BF network gains 34% area by sacrificing only 0.5% throughput with respect to pure MoT network. At the same time it has approximately 2.5% higher throughput with respect to a RBF network with similar area.

Combining simple MoT primitives with more complex BF primitives allows place-and-route tools to balance wire delays accordingly. Resulting layouts of hybrid networks have maximum clock frequencies between pure MoT and pure BF networks. Post-layout throughput of 8-terminal MoT-1-BF is 22% higher than a RBF network with comparable area cost. Pure MoT network has much higher throughput, mostly because of its higher clock rate.

Chapter 8

MoT Network as Part of XMT Parallel Processor

In earlier chapters we presented and evaluated the MoT network. In this chapter we embed the MoT network in eXplicit Mult-Threading (XMT) parallel processor architecture, and evaluate in that context.

First, we discuss deadlock conditions, and evaluate methods to eliminate deadlock. Next, we run several applications on the XMT processor, and evaluate the MoT network by comparing it with a butterfly network.

8.1 Deadlock

Earlier, we focused on a single MoT network, that transmits messages from sources to destinations. In a single-chip parallel processing context, the communication between PCs and MMs is two-directional. Namely, memory *request*s are sent from PCs to MMs; and memory *response*s, such as data or acknowledgment, are returned from MMs to PCs.

First, we consider a configuration, where both *request*s and *response*s use a single network, and show that deadlock can occur in this arrangement. Next, we discuss four methods to eliminate deadlock, and evaluate their cost and performance effects.

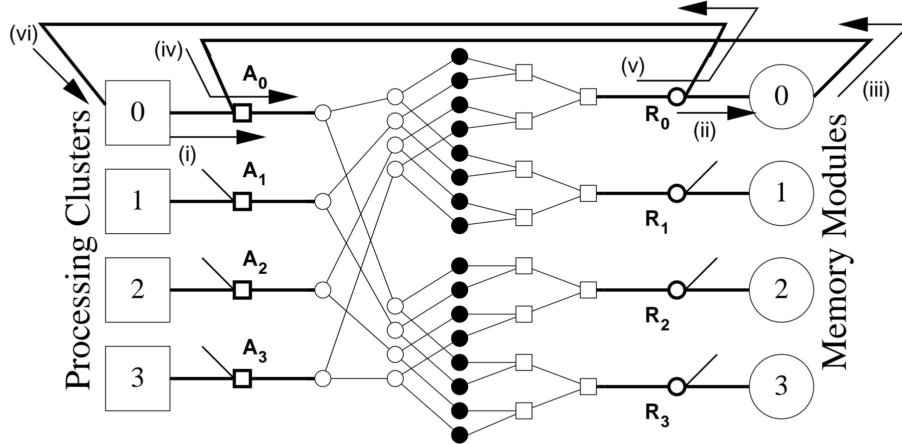


Figure 8.1: Single MoT network in XMT.

8.1.1 Conditions for Deadlock

Without loss of generality we assume that each PC injects at most π flits to the network, until the response of one of the injected messages returns. We also assume that each MM has μ buffers to store incoming messages before processing them. Once a message is processed, it is removed from the buffer, and an appropriate response is injected to the network. We also assume that each routing (fan-out) primitive has enough buffers for β_R flits; and each arbitration (fan-in) primitive has enough buffers for β_A flits per input.

Consider Figure 8.1, where memory requests and responses share the same network. We call this *single network* configuration. In the figure, only connections to and from PC_0 and MM_0 are shown for clarity. Memory access from PC_0 to MM_0 occurs as follows. The request leaves PC_0 as shown in arrow (i), and enters the network after one level of arbitration at node A_0. When it reaches its destination, it passes through one level of routing in R_0, and enters MM_0 (ii). A *response* leaves

MM_0 (iii), and passes through one level of arbitration in A_0 (iv) before entering the network. After the network, it passes through one level of routing in R_0 (v), and enters PC_0 (vi).

A necessary condition for deadlock is cyclical dependencies between network resources [28]. Although the MoT network does not contain such dependencies, they may arise when we consider round-trip memory accesses as described above. For example, with sufficiently many memory accesses from PC_0 to MM_0, all buffers on the path between A_0 and R_0 nodes and all μ buffers in the memory can be occupied with requests. Then, once MM_0 tries to send a response, it cannot enter the network because of the blocked path. More formally, if B_{cycle} represents the total message buffering capacity of the above mentioned cycle, deadlock occurs if all B_{cycle} resources are occupied. B_{cycle} is computed as shown in Equation (8.1). This example shows that deadlock is possible in *single network* configuration.

$$B_{cycle} = \mu + (\log N + 1)(\beta_R + \beta_A) \tag{8.1}$$

In the worst case, all of the N PCs send π requests to one of the MMs, say MM_0, without receiving any response. If the total number of flits, $N\pi$ is less than B_{cycle} (8.2), there will be at least one empty buffer space that will allow the flow of responses. If $N\pi \geq B_{cycle}$, deadlock is possible.

$$N \cdot \pi < \mu + (\log N + 1)(\beta_R + \beta_A) \tag{8.2}$$

8.1.2 Deadlock Prevention Methods for XMT

Typically, deadlock can be avoided by eliminating cycles that are explained in previous section, or preventing the occurrence of them. We consider the following approaches for this purpose.

1. **Increasing** μ. More buffers can be added at each MM. This increases the capacity of deadlock-causing cycles. Such buffers may already be in use in MMs, since some memory requests are sent off-chip, and some record of these requests need to be stored until off-chip response is available.

2. **Limiting** π. By means of software or hardware methods, the injection process from PC to the network can be controlled [61]. If the amount of injected flits is limited, buffers in the cycles fill not fill-up and cause deadlock.

3. **Increasing** β_R **or** β_A. Increasing the number of buffers in a fan-out and fan-in nodes would increase cycle capacity, but incur performance penalties as discussed in Chapter 5.4.

4. **Using virtual channels**. Cycles can be eliminated by assigning *request*s and *response*s to different virtual channels [28]. This approach would increase hardware cost, as well as the complexity of the switch primitives, causing lower performance [79].

5. **Using a second network**. By directing request and response traffic in two different networks in opposite directions, cycles can be eliminated [28]. Figure 8.2 shows this configuration with 4 PCs and 4 MMs.

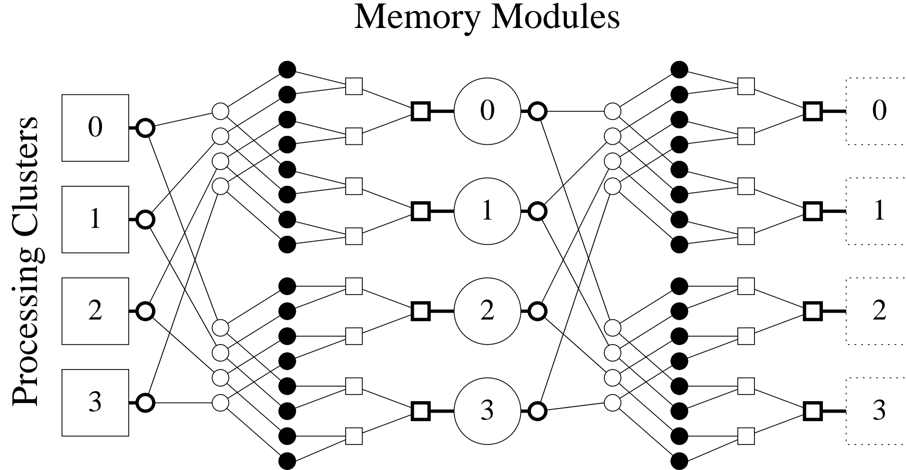


Figure 8.2: Two-network configuration of XMT with 4 PCs and 4 MMs. PCs on the left are also drawn with dotted lines on the right to show the connection of the second network.

8.1.3 Cost of Deadlock Prevention

Most of the proposed methods require additional amounts of buffers. First three methods increase the capacity of deadlock causing cycles, and the last two methods eliminate such cycles completely. We evaluate and compare each method below and summarize in Table 8.1.

Based on (8.2), deadlock can be avoided if μ is increased as shown in (8.3). If $\pi = O(1)$, this approach requires an additional cost of $O(N^2)$ buffers in all N memory modules. Otherwise, the cost is $O(\pi N^2)$. Additional buffers may increase logic complexity of MM and cause performance penalties.

$$\mu > N\pi - (\log N + 1) \cdot (\beta_R + \beta_A) \tag{8.3}$$

In order to prevent deadlock by limiting π, the relation (8.4) shows the necessary condition. This method may be suitable for small systems, or systems that

already imply a limit on π by using blocking memory accesses. However, as N increases π is limited by $O(\log N/N)$, and does not scale well. Furthermore, artificially limiting π may have adverse effects on overall system performance, when functional units in PCs depend on the availability of data. For medium-sizes systems, this can be combined with the first approach to increase memory buffer by less than $O(N)$ per MM as shown in (8.3).

$$\pi < \frac{\mu}{N} + \frac{1}{N}(\log N + 1) \cdot (\beta_R + \beta_A) \tag{8.4}$$

Based on (8.2), deadlock can also be prevented as shown in (8.5). If we rewrite equation (4.9) to reflect variable values of β_R and β_A in MoT area, we obtain that MoT area is $O((\beta_R + \beta_A)N^2)$. Combining this with (8.5), the area cost of this approach becomes $O(\frac{N^3}{\log N}\pi(\beta_R + \beta_A))$. This approach increases the complexity of switch primitives and reduces clock rate. Furthermore, increased area implies longer physical distances for wires. This reduces clock rate further.

$$\frac{N \cdot \pi - \mu}{\log N + 1} < \beta_R + \beta_A \tag{8.5}$$

Virtual channels can be used to break dependency cycles that may cause deadlock [26]. In a plain MoT network, a single physical channel between two switch primitives ends at the input buffer of a primitive. With virtual-channel method, one can split the input buffer into two, and use one of them for requests, and the other one for responses. As a result, requests and responses do not use same resources, and no cyclical dependency is created. Assuming that the additional logic

area in negligible, this method requires double the amount of buffers in a MoT network. Therefore, its hardware cost is the same as the cost of a regular MoT network (Table 8.1). Using virtual channels in switch primitives increases the logic complexity and delay [79], as we already demonstrated in Chapter 5.5.1. Larger area also increases the wire lengths and consequently the wire delays.

Finally, we consider using a second MoT network to avoid deadlock. Memory requests use one network, and responses use the other network. This approach has the same register cost as the virtual channel approach. Essentially, this method duplicates physical channels, as well as the input buffers. As a result, the amount of wires will increase compared to virtual channel approach. In terms of performance, the logic complexity of switch primitives does not increase, as it is the case with virtual channels. On the other hand, wire delays will increase similar to the virtual-channel approach, because the area increase is same.

In terms of hardware and performance, *virtual channels* and *second MoT* seem to be the least expensive methods for XMT context. We assume that the performance loss due to wire delay is same for both approaches, because of same amount of change in area. Second MoT does not have the disadvantage of increased logic delay. Therefore, this approach seems to be the most feasible method.

8.1.4 Summary

We evaluated deadlock conditions of MoT network in XMT context. Similar to some other networks, such as butterfly, MoT network is deadlock free, when it is

Method	**Hardware Cost**	**Performance Cost**
Increasing μ	$O(\pi N^2)$ in MM	Higher MM complexity
Limiting π	None in PC, MM or MoT	Idling units in PC
Increasing $\beta_R + \beta_A$	$O(\frac{N^3}{\log N}\pi(\beta_R + \beta_A))$ in MoT	Longer logic and wire delay
Using virtual channels	$O(N^2)$ buffers in MoT	Longer logic and wire delay
Using a second network	$O(N^2)$ second MoT	Longer wire delay

Table 8.1: Comparison of deadlock avoidance methods.

considered individually. In a parallel processor context, the environment surrounding the network imposes additional constraints. We discussed these constraints, and compared five methods to avoid deadlock. Based on register area and performance costs, using a second network seems to be the least expensive method.

8.2 Application Simulation on XMT

We use a development version of XMT compiler to compile and execute the following applications:

1. **Array Increment** We perform an arithmetic operation (increment by one) on each element of an 256k-long array. The inc-1 program computes one element of result array in one parallel iteration, whereas inc-8 computes 8 elements at once.

2. **Matrix Multiplication** In matmul-1 and matmul-2 we compute the product of 2 64×64 matrices. In one parallel iteration matmul-1 computes one element

of the result matrix, and matmul-2 computes 2 elements. Programs inc-8 and matmul-2 are expected to generate higher traffic than inc-1 and matmul-1 respectively.

3. **Fixed-point FFT** In FFT, we apply a fixed-point FFT implementation on a 64k-long data array, where each parallel iteration computes fixed-point FFT of two points, and intermediate results are stored in the memory.

We tune these programs to generate low or high traffic, without changing the total amount of work required by the data set. In all cases we use the same data set and algorithms.

First, we vary the number of execution units (or hardware threads) connected to each network port based on the available options in the hardware model of [105]. In this specific implementation, one network port can connect to a cluster of 4, 8 or 16 hardware threads. If one hardware thread generates a specific amount of traffic when executing a given piece of code, multiple threads executing the same code will generate a higher amount of traffic.

Second, we vary the number of computed result elements in each parallel thread. We assume that there is a constant overhead for creating each parallel execution thread [105]. As a result, if each thread computes multiple elements of the result set, the average amount of traffic generated during each thread will be larger compared to computing single element per thread.

8.2.1 Application Traffic and Execution Time

We conduct preliminary study on the execution of real life program to demonstrate the effectiveness of the proposed MoT network. The five applications we use are listed in Table 8.2. In inc-1 and inc-8 we perform an arithmetic operation (increment by one) on each element of an 256k-long array. The inc-1 program computes one element of result array in one parallel iteration, whereas inc-8 computes 8 elements at once. In matmul-1 and matmul-2 we compute the product of 2 64×64 matrices. In one parallel iteration matmul-1 computes one element of the result matrix, and matmul-2 computes 2 elements. Programs inc-8 and matmul-2 are expected to generate higher traffic than inc-1 and matmul-1 respectively. In FFT, we apply a fixed-point FFT implementation on a 64k-long data array, where each parallel iteration computes fixed-point FFT of two points, and intermediate results are stored in the memory. The programs are compiled by a development version of XMT compiler.

The FPGA prototype of XMT processor, as described in [105], consists of 64 light-weight processors (hardware threads) grouped in 4 clusters. We configure its hardware model to generate 1024 light-weight processors grouped in 64 clusters. We build two MoT networks with $N = 64$ using Verilog, one form processors to the memory modules and another one in the opposite direction, and integrate them into the XMT processor. We execute the compiled programs and measure the execution time in terms of cycles as reported by verilog simulator. We exclude the time, during which data is uploaded before execution, and the result is downloaded after

execution. We also measure the average traffic rate that enters the interconnection network from processors towards the memory, by tracing the flits at network ports. For comparison, we implement a butterfly network (Replicated Butterfly with $N = 64$, $r = 1$) in Verilog and conduct the same simulation.

Table 8.2 reports the execution time (in 10^3 cycles) and traffic rates (in flits per cycle per port) on two interconnection networks for the same processor. We make the following observations:

1. In average, MoT network accepts more traffic per cycle, for example, 69% more on the tested applications compared to the butterfly network.

2. The execution time of these applications is reduced by approximately the same amount as the increase in traffic rate. This indicates that processor-memory communication strongly impacts the execution time of the tested applications; and the use of a high-throughput MoT network improves execution time.

3. The improvement is significant, even if the corresponding BF traffic is below BF's saturation throughput (Figure 5.13). This could be due to temporary bursts in traffic demand, which are more efficiently handled by MoT.

4. When we increase the traffic rate of same program and data set, (from inc-1 and matmul-1 to inc-8 and matmul-2) the performance of the system with MoT improves, whereas the performance of the system with BF slightly degrades. This could be caused by higher contention due to increased flit conflicts.

App.	inc-1	inc-8	matmul-1	matmul-2	FFT
BF Traffic	0.338	0.384	0.243	0.242	0.082
BF Exec	32.0	32.4	36.2	36.8	3007.3
MoT Traffic	0.527	0.929	0.392	0.458	0.085
MoT Exec	21.5	13.5	23.2	19.8	2934.4
Trf. Ratio	1.49	2.42	1.61	1.89	1.04
Exec. Ratio	0.674	0.412	0.641	0.538	0.964

Table 8.2: Simulation Results for Execution Time and Traffic Rate.

8.3 Layout of XMT ASIC chip

The XMT processor design [104,105] has been advanced from Verilog-HDL description to ASIC layout using standard-cell design methodology [6], and fabricated at 90nm technology [44]. It contains 64 processors, also called hardware threads or Thread Control Units (TCUs), grouped into 4 clusters of 16 TCUs. The chip also includes a Master TCU, the MoT interconnection network, 8 cache modules, and an interface for external memory. The individual components of MoT are outside the scope of the thesis, with the exception of MoT network. This section briefly describes the layout efforts, and performance results.

The final layout of XMT ASIC chip is shown in Figure 8.3. Its components are shown in Figure 8.4:

- **1** Mesh of Trees Interconnection network is placed in the center of the chip, between processing clusters and memory modules.

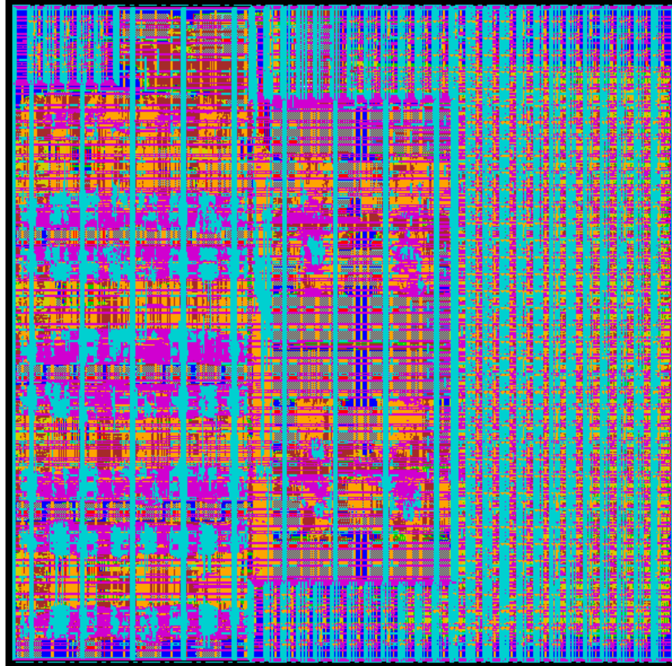


Figure 8.3: $10mm \times 10mm$ layout of XMT ASIC chip with 64 processors.

- **2** Master TCU is placed close to network.
- **3-6** Processing clusters 0, 1, 2, and 3 are placed on one side of the network.
- **7** Phase Locked Loop (PLL) clock generator is placed close to the mid-section of the chip, based on vendor specifications.
- **8-15** Cache modules $0 \ldots 7$ are placed on the right of the network. The network has 4 outputs on the memory side. Each are split into in the space between caches.
- **16** Memory controller and external memory interface, with I/O signal drivers, covers the remaining portion of the chip.

The typical operating conditions for the standard cell library at IBM 90nm

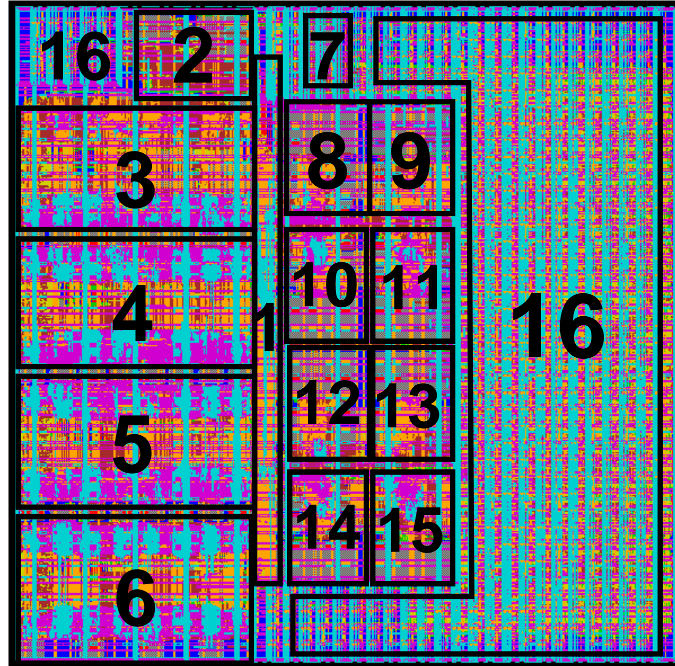


Figure 8.4: Modules of XMT ASIC chip.

technology is 1.2V supply and 25^oC temperature. The The layout results indicate that the slow corner (1.08V, 125^oC) operating frequency of this design is $154MHz$. This value reflects the operating frequency of the core components in the chip; the external memory interface is designed to operate at 1/4 of that frequency, specifically at $38.5MHz$.

Chapter 9

Discussion

In this section we discuss the limitations of our approaches and results, and potential benefits of overcoming them.

9.1 Limiting Factors for Clock Rate

In this section we discuss the factors that limit the clock rate presented in this study. First we discuss three reasons that we observed between different stages of our studies. Next, we discuss a limitation based on a standard method for VLSI circuits.

9.1.1 Clock Rate Decrease Between Development Stages

We observe that the clock rate decreases as we move (1) from synthesis results to layout results; (2) from single primitive results to network results; and (3) from small scale networks to large scale networks.

The main reason for the decrease between synthesis and layout in case (1) is the wire delay, which can be accurately evaluated only after the layout is completed. This is a general issue with integrated circuits, including digital circuits following similar design flows, and has been shown in other network studies [4]. As a related issue, the floorplan also plays a role in determining the clock rate. We used a

partitioned floorplan approach to obtain our earlier published results [8], which is also summarized in Chapter 6. It turns out that higher clock rates can be achieved without partitioning the floorplan, and keeping the aspect ratio as close as possible to a square. To support our reasoning, we performed preliminary experiments with 4, 8 and 16-terminal networks under these new conditions, and we compare clock rates in Table 9.1. One disadvantage of this approach is that 32-terminal and larger networks are too large for us to handle in one piece with the available computing equipment. As a result there are too few data points to extrapolate to larger-scale networks.

Configuration (Number of Terminals)	**4**	**8**	**16**
Clock rate with partitioned and separated floorplan [8]	970	890	680
Clock rate with unpartitioned and square floorplan	1010	1007	854
Improvement	4%	13%	26%

Table 9.1: Performance improvement with square floorplan. Clock rate is measured in MHz.

In case (2), the reason of clock rate decrease between single primitive and network is the changing critical path. In evaluating a single switch primitive, the critical path is contained withing the primitive. On the other hand, in the network, the critical path can be between two consecutive switch primitives. For example, the critical path of arbitration primitive contains 6 levels of logic; and the critical path of a fan-in tree, consisting of the arbitration primitives, contains 7 levels of logic. This issue can be overcome by adding input and output registers to each primitive;

by separating logic operations of each primitive. However, this adds an additional pipeline stage, and increases network latency.

As the network scales up in case (3), there is a fundamental limitation that decreases the clock rate. In order to analyze that limitation, we first focus on switch primitives, and note that the clock rate is determined by the largest sum of logic and wire delay between two consecutive registers, or primitives. Assuming that logic delay remains constant; increasing wire delay will reduce the clock rate. With optimal amount of repeaters, the wire delay increases at the same rate (linearly) as its length [84, 93]. Next, we focus on a network layout with area A. There are multiple paths from inputs to outputs of the network. The physical distance of longest path is $\Omega(\sqrt{A})$. Assuming a square layout in the best case, the longest distance is $\Theta(\sqrt{A})$. If logic delay of all $O(\log N)$ switch primitives on the longest path are equal, this path will be divided into $O(\log N)$ segments with equal distance and equal delay in the best[1] case. This delay will determine the clock rate. Now we consider the best case scenario for clock rate as we scale up number of terminals in MoT. As the MoT network with N terminals and $O(N^2)$ area scales up, the physical length of the longest path will increase as $O(N)$; however, the number of segments will increase as $O(\log N)$. Then, the length of critical segment will increase as $O(\frac{N}{\log N})$. As a result, large scale networks will suffer more from the wire delay effects. A similar decrease in performance can also be observed in results of Table 9.1. Additionally, this effect will also arise, when the network layout is not

[1]In worse cases, a suboptimal segment may have longer delay than others, and that particular segment determines the clock rate.

square. As the layout shape approaches a long and thin rectangle, the longest path approaches to $O(N^2)$. In that case, the longest segment will approach $O(\frac{N^2}{\log N})$ and clock rate will decrease.

Based on the above observation, clock rate can be improved by inserting pipeline stages, or reducing the network area. As shown in Chapter 6, specifically in Table 6.3, a heuristic pipeline insertion approach improves clock rate. On the other hand, the hybrid network approach discussed in Chapter 7 reduces the network area considerably. However, since butterfly primitives have higher logic delays, our layout results with 8-terminal hybrid networks have lower performance compared to regular MoT network, and performance gain from lower area cannot be observed in this case. For larger-scale networks, a pipelined hybrid network may further reduce the effects of wire delay. We outline the desired features of this approach as part of future directions in Section 10.1 (NW.2).

9.1.2 Limitations of Standard-Cell Design Method

Another performance limiting factor is the logic delay of standard cells. In practice, digital VLSI circuits are developed hierarchically. The design process advances from higher levels of abstraction to lower-level modules and components (*top-down design*). The implementation process advances from lowest-level devices to components, modules and the system (*bottom-up implementation*). Since it is not practical to build large systems by manually implementing each transistor, libraries of most commonly used logic functions are developed. These components, called

standard-cells, can be used as building blocks to develop larger systems. One disadvantage of this practice is that the circuit performance is limited by the features and the quality of the available libraries.

We used commercially available standard cell libraries [6] with regular-threshold voltage (RVT). Alternative libraries with low-threshold (LVT) cells provide the same logic functionality with lower delay but increased power consumption. For example, a FO4 delay in the slow process corner[2] of LVT library is $45.4ps$, which is 32% faster than the FO4 delay of RVT library. On the other hand, power consumption of a DFFX1[3] cell in LVT library is $10 - 18\%$ higher compared to the same cell in RVT library. In our studies we used regular cells because the definition of "low" threshold may change depending on the foundry and advances in integration technology.

9.2 Potential Impact of Multi-GHz Operation

In this section, we discuss the difference that improving the clock rate of the MoT network can make for XMT performance, and motivate further research. First, we focus on the general impact, then we present a brief case study with preliminary experiments.

Considering MoT network in isolation, higher clock rates imply higher bandwidth and throughput, and lower latency in terms of wall-clock time. Based on our discussion in Sections 2.3.1 and 4.7.3, the bandwidth of the network is computed as $N \cdot w_c \cdot f_{MoT}$, where N is the number of terminals, w_c is the number of bits in each

[2] Same conditions as RVT library

[3] DFFX1 stands for regular ($\times 1$) sized D flip-flop in the standard cell library.

channel, and f_{MoT} is the operation frequency of the MoT network. For example, a 32-terminal network with 32-bit channels, operating at a hypothetical $1GHz$ frequency, will have a cumulative bandwidth of 1024 *Gbps*. Assuming uniform traffic, this network could reach a cumulative throughput of 986 *Gbps*, which is 96.3% of its bandwidth (Table 5.4). The network latency in terms of cycles does not change, regardless of the frequency value. On the other hand, with high-frequency operation cycles become shorter. Therefore, latency decreases in terms of wall-clock time.

In the context of XMT, the benefits of high-frequency operation is limited by the memory access characteristics of the application, and off-chip memory communication performance. We assume that, in general, off-chip memory bandwidth is less than on-chip bandwidth; and off-chip latency is much larger than on-chip latency. If an application frequently misses the cache, and issues off-chip memory requests, the execution time will be limited by the off-chip access latency and bandwidth. In practical implementations [101, 104, 105], effects of this bottleneck can be reduced, but not completely eliminated, by prefetch operations. On the other hand if an application frequently utilizes the on-chip cache, and rarely requires off-chip memory access, benefits are significant, as we discuss next.

9.2.1 Case Study

We consider the MoT network in XMT context by observing a system with three parts, namely N processing clusters (PC), an N-terminal network (MoT), and N memory modules (MM). Each part can produce (generate) or consume at

most one flit per cycle per network terminal. We assume the operating frequency of MoT network (f_{MoT}) is maximized for a given implementation. We also assume that MMs (f_{MM}) operate at least at the same rate as PCs (f_{PC}), because otherwise MMs would create a significant bottleneck regardless of the network type, size and performance. First, we discuss four interesting cases, then we support our discussion with preliminary experimental results shown in Table 9.2. In the table, case numbers correspond to the following cases.

1. $f_{MoT} = f_{MM} = f_{PC}$. When the frequency of all three components increase at the same rate, more memory requests will be generated per second, more will be transferred to the consumers, and more will be consumed. If all three frequencies are increased by factor of m, the execution may accelerate by up to a factor of m. As a side effect, the m-times increase in frequencies will generate up to m-times off-chip memory traffic, and off-chip access latency increases by m in terms of PC cycles. This may reduce the speed-up, unless the application entirely relies on on-chip memory.

2. $f_{MoT} > f_{MM} = f_{PC}$. When the frequency of MoT network increases by m, but other on-chip components remain same, same amount of memory requests are transferred at a higher rate. Assuming that the production and consumption rates at PCs and MMs remain same, the benefit in this case the reduced latency of memory accesses. From PC's point-of-view, transfers occur up to m-times faster. The network's internal bandwidth increases by m due to increased frequency. However, the PC-to-network and network-to-

MM connection bandwidths remain the same, and they become throughput bottlenecks.

3. $f_{MoT} = f_{MM} > f_{PC}$. When the frequency of MoT is increased by m, together with MMs, the bottleneck at the network-MM connection is eliminated. From network's point-of-view, the PC uses at most $1/m^{th}$ of the available bandwidth. As a result, memory access latency at comparable loads will be lower compared to case (2). However, similar to case (1), this may increase off-chip memory communication.

4. $f_{MoT} > f_{MM} > f_{PC}$. Assume that $f_{MoT} = m_1 \cdot f_{PC}$, $f_{MM} = m_2 \cdot f_{PC}$, and $m_1 > m_2 > 1$. From the network's point-of-view, the bandwidth is capped by MM, at $\frac{m_2}{m_1}$ of the network capacity. PC does not suffer from this cap, since the cap is greater than the maximum generation rate of the PC, namely $\frac{1}{m_1}$ of the network capacity. From the PC's point of view, memory requests travel up to m_2 times faster in the network.

The above discussion shows potential benefits of operating the MoT network at high clock frequencies in XMT context. A detailed analysis requires modification of XMT components, which is beyond the scope of the current study. The cost of such operation also needs to be investigated. We outline a relevant future study direction in Section 10.1 (PP.1).

We support our performance analysis with preliminary experiment results in Table 9.2. We simulated a 32-terminal MoT network with low, medium and high traffic generation rates. We measured latency in network (MoT) cycles and PC

Case — Load	PC gen. rate (w.r.t. MoT)	MM acc. rate (w.r.t. MoT)	Latency MoT cycles	Latency PC cycles
1 — L	0.1	1.0	13.1	13.1
1 — M	0.5	1.0	13.5	13.5
1 — H	0.9	1.0	17.4	17.4
2 — L	0.05	0.5	13.6	6.8
2 — M	0.25	0.5	14.5	7.3
2 — H	0.45	0.5	22.7	11.4
3 — L	0.05	1.0	13.0	6.5
3 — M	0.25	1.0	13.2	6.6
3 — H	0.45	1.0	13.4	6.7
4 — L	0.025	0.5	13.6	3.4
4 — M	0.125	0.5	13.8	3.5
4 — H	0.225	0.5	14.3	3.6

Table 9.2: Benefits of high-frequency network operation. PC generation and MM acceptance rates are given with respect to MoT cycles. Load configurations labeled as L, M and H represent 10% (low), 50% (medium) and 90% (high) of maximum PC generation rate respectively.

cycles. To represent each case, we cap flit generation rate at PCs, and request acceptance rate at MMs. Specifically, we used $m = 2$, $m_1 = 4$, and $m_2 = 2$ in our simulations. Our results show the benefits in network latency as observed from the PC's point-of-view. Recent performance modeling of PRAM-like programming

(on XMT), links the execution performance improved by reducing the round-trip time to memory. As a result our analysis can be used to motivate further studies to optimize XMT architecture.

9.3 Applicability to Other Systems

The features and requirements of the eXplicit Multi-Threading (XMT) architecture motivated and guided specific decisions and optimizations in the MoT network described in this dissertation. However, the MoT network can be used in other systems that require high throughput and low latency at high traffic rates.

In this section, we discuss some features of MoT network that allow integration in other systems without significant changes in the network architecture or functions. We also note that some modifications may come with performance penalties, requiring implementation-specific optimizations.

- **Location**. The MoT network can be placed between different levels of memory hierarchy, for example, between first and second levels of on-chip caches. If the first level cache is able to catch most of the requests, then fewer requests are sent to the second level. As a result, the bandwidth requirement will be different from our current setting. Depending on the required bandwidth, different configuration of hybrid networks may be used. In addition, at different locations in the memory hierarchy, the network may need to carry longer packets. We discuss that issue next.

- **Packet Length**. The MoT network is optimized to carry mostly one-flit

packets. For two-flit store packets, we relate two flits using a single extra bit (chain or glue bit), that triggers a winner-take-all operation in arbitration primitives, as described in Section 5.4.3.3. This method can be used to carry longer packets with more flits, as well as a mixture of long and short packets, without any hardware addition or modification in the network. However, the performance of the network will suffer, since long packets will create head-of-line blocking, and interfere and block others (Section 4.7.6).

Our preliminary experiments show that the maximum throughput of a 64-terminal MoT network reduces from 0.98 fpc to 0.84 fpc per port, when we use 8-flit packets, and keep generation rate at 1.0 fpc. This represents a throughput reduction of 14%. A mixture of long and short packets may improve performance. Furthermore, increasing number of registers in each primitive may reduce the blocking effect and improve throughput at expense of increased area.

- **Data Width**. The width of data channels has no effect on throughput and latency, when measured in fpc and *cycles* respectively. However, it is an important factor in determining the throughput and latency in terms of *Gbps* and ns. Our network evaluations in Chapters 4 and 5 (e.g. Figures 5.13 and 5.14) do not assume a specific data width. Therefore, similar results are expected with any number of bits in terms of data width. Furthermore, for synthesis and layout generation, data width is used as a parameter, which can be easily modified for different instances of MoT.

On the other hand, wider data may cause lower performance after layout; however, such problems are not specific for MoT network, and can affect other networks similarly. For example, the number of bits in the data directly affects the number of flip-flops or latches in the design. This will result in increased area and power figures with respect to presented results. Finally, as we discussed in Section 9.1.1, increased area may also impact clock rate.

- **Tree Radix**. Based on the specific requirements of the XMT system, we used binary trees, with $k = 2$ children per parent, for routing and arbitration. However, for different systems, trees with higher radix, i.e. with $k > 2$ children per parent, can be used. We showed in Section 5.4.2 that the delay from internal registers to output ports increases logarithmically with k. On the other hand, the logarithmic increase of arbitration delay is shown in [91]. If such delay can be tolerated, using trees with higher radix reduces the number of hops between sources and destinations; and reduces latency.

  It is also possible to build trees with varying radix at different levels. For example, the root node of the arbitration tree may be chosen with $k = 2$, and internal nodes can be chosen with $k > 2$. If the logical operation of primitives is preserved, the layout tools need to balance the delays by modifying the placement (wire length and delay) of the network. On the other hand, the operation of an internal node is not as demanding as the job of the root node due to diluted traffic at internal levels. Therefore, the operation of internal nodes can be further optimized to reduce the cycle time.

Chapter 10

Future Directions and Conclusion

This thesis presented the Mesh of Trees network, and evaluated it in an easy-to-program, explicitly multi-threaded parallel processor context. In this concluding chapter we first discuss some possible future directions for research and development. Following that, we present our concluding remarks.

10.1 Future Directions

We present and briefly discuss the following directions to extend this study in the future. We separate them in three main groups: directions towards improving network performance, labeled NW, directions towards improving parallel processor (XMT or another architecture) performance, labeled PP; and a third group, labeled O, representing other directions.

NW.1 **Analysis and evaluation of MoT and hybrid MoT with finite-queue models.** We stated earlier that the classic infinite-queue analysis is not accurate. As a result, we rely on more accurate simulation results for our evaluation. By modeling the MoT network using finite queues in each switch primitive, design space exploration and performance evaluation can be sped-up. Similar to infinite-queue models, finite-queue models can be probabilistic. For example, consider the pipeline primitive as discussed earlier in Section 5.4

with three states, labeled 0, 1 and 2.

Following the "Markovian" queue properties, the steady-state probabilities of each state can be written as shown in (10.1-10.5). This approach can be applied to other primitives, and used to determine the steady state properties of the entire network.

$$P(0) = P(0) \cdot P(r_{AB} = 0) + P(1) \cdot P(r_{AB} = 0 \& ks_{CB} = 1) \quad (10.1)$$

$$\begin{aligned} P(1) = {} & P(0) \cdot P(r_{AB} = 1) + \\ & P(1) \cdot (P(r_{AB} = 1 \& ks_{CB} = 1) + P(r_{AB} = 0 \& ks_{CB} = 0)) + \\ & P(2) \cdot P(ks_{CB} = 1) \end{aligned} \quad (10.2)$$

$$P(2) = P(1) \cdot P(r_{AB} = 1 \& ks_{CB} = 0) + P(2) \cdot P(ks_{CB} = 0) \quad (10.3)$$

$$P(r_{BC} = 1) = P(1) + P(2) \quad (10.4)$$

$$P(ks_{BA} = 1) = P(0) + P(1) \quad (10.5)$$

$$\quad (10.6)$$

NW.2 **Layout optimizations for synchronous and asynchronous implementation, including varying module aspect ratios, and optimal pipelining.** In Section 9.1, we discussed the clock rate limitations due to layout issues, and showed preliminary results of improved layouts in Table 9.1. It is reasonable to assume that in some cases the layout may not be in square shape, depending on other factors such as the size and shape of surrounding components. The network may need to fit a rectangular area, where the length of one side is longer than the other by a factor of x. Assuming that (1) area is

dominated by logic cells, (2) a cell-based placement and routing methodology is followed, and (3) x is not an extreme value such that wire height or width becomes a limiting factor; the area of this rectangle is expected to be same as the square-shaped layout. However, the distances between switch primitives will be different, and that will affect the clock rate and network performance.

It is important to approximately predict the performance earlier in the design. A study that could provide performance prediction methods for layouts of same network with different aspect ratios would be valuable. As a further step, optimal pipelining methods can be applied concurrently with such prediction, in order to reduce performance loss caused by different aspect ratios.

NW.3 **Transistor-level optimization of switch primitives.** For recent technology generations, standard-cell based design flow has been the industry standard for digital VLSI circuits. Commercially available libraries provide layouts and abstracted properties (area, delay, power) of logic gates of different kinds, and different load driving strengths. Design tools use these properties to generate and optimize layouts of larger modules. One shortcoming of this approach is that the tools are limited with the provided cells. Library vendors, constrained by time and labor, supply the same set of most-commonly used and basic cells to all customers. For example, the library used in this study [6] contains 641 total cells with 121 different logic functions. Alternative methods such as *direct transistor level design* [34], *flex cells* [86] or *virtual libraries with transistor-level optimization* [54], show about $10-20\%$ perfor-

mance improvements, up to 23% area improvements, and between $15 - 42\%$ power improvements in various case studies. These methods include generation and transistor-level optimization of custom cells with complex logic functions, which are normally realized using multiple levels of standard cells. For example, the virtual library of [54] includes 15000 total cells with 3500 logic functions that are specifically optimized and used in their case studies. The optimized custom cells are included in the regular design and optimization flow with standard tools.

Considering the repetitive and regular structure of the interconnection network, such methods can be applied to switch primitives in order to improve the overall design quality beyond standard-cell methods.

NW.4 **Improvement and use of reduced-synchrony circuits in a meso-chronous design approach.** In a meso-chronous design approach, components in a design are operated at same clock frequency, however, unlike the regular synchronous approach, the clock signal may have a different phase delay at each component. This approach relaxes some of the constraints on the clock network design on the entire chip. Most recently, the Teraflop project at Intel [99] employs this meso-chronous approach in their design.

Our reduced-synchrony circuits [9] (summarized in Section 5.3.2), incorporates clock signal distribution into logic function the switch primitives of the MoT network. Therefore, it can be modified to support meso-chronous operation of each switch primitive, and consequently, the entire network. We note that such

a modification also needs to consider the current limitations as we discussed earlier in Section 5.3.4.

NW.5 **Reconfigurable-radix networks.** Depending on the bandwidth demand of the system, or an application, the network can be reconfigured by changing radix of routing and/or arbitration trees. The lowest-radix implementation with binary trees, as presented in this dissertation, provides high throughput, with at least $2 \log N$ cycles latency. With higher radix implementations, flits can be transferred in fewer cycles; however, maximum throughput may reduce because of higher contention and lower clock rate.

The radix of routing and arbitration trees can be configured either during the design of the network; or in run-time using reconfigurable hardware techniques. The latter option would have performance overheads related to the reconfiguration methods.

PP.1 **Frequency-island operation of XMT with variable frequencies.** In some Globally Asynchronous Locally Synchronous (GALS) systems, tile-based processors are interconnected using a 2D-mesh network-on-chip (NoC), and groups of tiles are set to operate at different frequencies [73]. This frequency isolation simplifies clock network distribution on the chip. Communication between such frequency-islands is enabled using mixed-clock FIFO buffers [20].

Earlier in Section 9.2, we discussed operating XMT using different frequencies for processors, network and memory modules, and the potential performance benefits. Considering that the boundaries of these three components include

a FIFO operation by definition, mixed-clock interfaces can be built at these boundaries, and the design can be divided in three (or more) frequency islands.

Furthermore, the frequencies can be dynamically modified during execution, depending on factors such as network load, cache misses, and idling of processors. This can dynamically balance the operation speed of processors, network and memory modules; and reduce idle cycles and power consumption.

PP.2 **Architecture optimizations and evaluation of $N \times M$ networks.** In this study we evaluated $N \times N$ networks, in other words, with N sources and N destinations. It is possible that some systems may have differing number of sources (N) and destinations (M). There are multiple methods to support such designs. In order to achieve higher performance, such methods need to be quantitatively evaluated within the context of parallel processors such as XMT.

In the prototype XMT [104], a 4×8 network is realized by splitting the end points of a 4×4 MoT into two. A disadvantage of this approach is that by splitting after the network, two endpoints share the bandwidth of one network port.

Another approach could be to use half of an 8×8 network, where only 4 out of 8 input ports are active. The disadvantage in this case is the increased area cost, and it is not clear if this overhead justifies the potential increase in performance.

Alternatively, a different butterfly-hybrid approach may be applied at two

levels of the destination-end of the network. Here, each root node of a fan-in tree is replaced by two roots connected with their children using a butterfly topology.

O.1 **3D-integration of MoT network.** With the advances in IC technology, it is possible to stack multiple silicon dies on top of each other [78], and provide 10^8 connections per cm^2 between dies [97]. Some recent studies focus on extending on-chip network structures towards stacked chips and demonstrate 3D layouts [60]. Advancing MoT towards this direction, would be worth investigating. An interesting questions is, how to best distribute an MoT network (or XMT processor) over the stacked dies, if $O(N)$ (or $O(\log N)$) dies are stacked on top of each other? Another question regarding performance is, how to tune the MoT network for minimum bandwidth loss between stacks.

O.2 **Fault tolerance in MoT network with reconfigurable or reprogrammable hardware.** As the number of components increase on chips, it is more likely that defects will occur. Under such conditions, it is desirable that the system keeps running, perhaps at a degraded capacity, as opposed to experiencing a complete failure. On-chip networks represent a critical component in single-chip parallel processors or other Systems-on-Chip (SoC), because of their central role to the operation. A small number of faults on the network may prevent otherwise healthy components to communicate with each other. As a result, research on fault tolerance in networks has attracted considerable attention [63,76]. The MoT architecture consists of several modules, which are

identical except for their input and output connections. It is not unreasonable to add reconfigurable or programmable redundant modules, which can replace faulty modules by reconfiguring their connections after manufacturing.

10.2 Conclusion

The aim of this study is to show that Mesh-of-Trees network is a competitive on-chip network, for highly parallel applications on single-chip parallel computing systems with UMA-like globally shared caches. The PRAM-based XMT architecture is a representative of such systems. The interconnection network in such a system is desired to provide processor-memory communication with high-throughput and low-latency, when memory accesses are uniformly distributed over all shared modules.

Tables 10.1 and 10.2 summarize our important results in absolute terms, such as quantitative measures of achieved performance, and relative terms, such as comparisons to other network models, respectively. Our general conclusion is that the MoT network provides the needed performance to XMT architecture, at an acceptable area cost. For better scalability, the MoT-BF hybrid network is a better alternative. As the MoT is optimized for short packets such as memory requests with one or at most two flits, it could incur performance penalties when used with longer packets.

Our analysis and experimental results show that MoT has higher, and better scaling ($O(N)$) bisection bandwidth, compared to mesh and ring topologies. As a result, under uniformly random traffic, MoT reaches higher throughput with same

amount of hardware, compared to those topologies. Furthermore, MoT has lower diameter with respect to these topologies. Therefore, while mesh and ring could be beneficial for traffic between closer units in a memory architecture with Non Uniform Memory Access (NUMA) model, for systems with UMA model MoT is a better alternative.

We also compared MoT with other network architectures, such as butterfly, hypercube, and fat trees. These networks have similar $O(N)$ bandwidth and $O(\log N)$ diameter as MoT. The main problem with those networks is that packets between different sources and destinations may interfere with each other. This will increase latency, and lower performance. MoT network does not suffer performance penalties from this kind of interference. Our analysis and experimental results show that MoT provides higher throughput using same amount of hardware, when all networks operate at same clock rate. For example, comparing 64-terminal networks, MoT throughput is 104% higher than a 2D-Mesh, 22% higher than a hypercube and 9% higher than butterfly with virtual channels.

The MoT network is built with less complicated switch primitives, compared to networks with virtual-channel router circuits. Virtual channels are buffers that can be used to improve network performance. However, they also increase switch complexity and logic delay in switching circuits. Increased logic delay will be reflected as longer clock periods, and lower throughput when measured in terms of bits per second. Comparing switch complexity in terms of technology-independent delay units, the fastest virtual-channel switch with 2 ports and 2 virtual channels is 21% slower than the slowest switch primitive of MoT network. Similarly, a 2D-

Mesh switch with 5 ports and 2 virtual channels is 46% slower. On the other hand, replicated butterfly networks, which we built with similar design principles as MoT, are 5% slower than MoT switch primitives.

We generated layouts of various MoT configurations using a standard-cell based design at 90nm technology. The operating frequency of a pipelined 32-terminal slow-process is reported as $764MHz$ at slow process-voltage-temperature (PVT) corner, where logic delay of gates are 1.92× slower compared to a typical PVT corner. In other words, at typical operating corner, the clock rate for the same configuration could reach $1.4GHz$. Furthermore, the network can be realized using low-threshold (LVT) gates. These gates run faster, but also consume more power. Depending on the overall power budget of the system, LVT gates can improve performance. According to data books [6][1] , slow and typical operation frequency of the above 32-terminal network with LVT gates could reach $1.1GHz$ and $1.9GHz$ respectively. Based on this layout study, we fabricated and tested a prototype chip with 8-terminal MoT network. Our test results were inconclusive, because the fabricated chips were defective, and it was not possible to read the internal signals.

We proposed an area improvement to the MoT network, by replacing parts of the MoT with butterfly (BF) networks of small scale. The MoT-h-BF hybrid networks with h levels of hybridization use less hardware area, but provide performance close to MoT. For example, when operated at same clock rate, a 64-terminal MoT-1-BF network uses 34% fewer registers than MoT, and provides 0.5% less throughput. As we compare different levels of hybridization with regular butterfly networks with

[1]FO4 delay for 90nm LVT library is 45.4ps for slow, and 26.4ps for typical operating corners.

virtual channels or replicated butterfly networks, MoT-h-BF networks provide better throughput with same number of registers. As we compare the layouts of 8-terminal networks, MoT-1-BF has 22% higher layout-accurate throughput compared to a replicated butterfly with comparable area cost.

Finally, we integrated the MoT network into XMT architecture. We evaluated alternatives for avoiding high-level deadlock conditions, and concluded that using two separate networks is better alternative in terms of area and performance costs. We used a Verilog-HDL model of XMT processor to simulate some applications and compared the throughput and execution time to a similar configuration with butterfly network. In evaluated applications we observed an average speedup of 55% with MoT network. We prepared the layout of an $10mm \times 10mm$ XMT chip with a 4-input-4-output MoT network. The layout was released for manufacturing in the 2nd quarter of 2008.

Item	**Metric**	**Value**	**Unit**	**Stage**	**Notes**
MoT-64	Throughput	0.98	fpc/port	simulation [10]	flits per cycle per port
MoT-64	Latency	21.6	cycles	simulation [10]	average, traffic load at 90% of capacity
MoT-32p	Frequency	764	MHz	layout [8]	RVT gates, slow PVT
MoT-32p	Throughput	747	Gbps	layout [8]	avg., cum., 32-bit flits, RVT, slow PVT
MoT-32p	Throughput	1.9	Tbps	layout [8]	avg., cum., 32-bit flits, LVT, typical PVT, projection
MoT-32p	Latency	22.6	cycles	RTL-verilog [8]	avg., traffic load at 90% of capacity
MoT-16	Frequency	680	MHz	layout [8]	RVT gates, slow PVT
MoT-16	Frequency	854	MHz	layout	square layout, RVT gates, slow PVT
MoT-1-BF-64	Throughput	0.97	fpc/port	simulation [11]	per cycle per port
MoT-1-BF-64	Latency	18.4	cycles	simulation [11]	average, traffic load at 90% of capacity
MoT-1-BF-64	Register savings	34%	-	computation [11]	-
Single Primitive	Logic delay	8.98	FO4	synthesis	-

Table 10.1: Summary of important results in absulute terms (quantitative measurements).

Item	Metric	Value	Compared with	Value	Notes
MoT	Bandwidth	$O(N)$	2D-Mesh	$O(\sqrt{N})$	-
MoT	Bandwidth	$O(N)$	Ring	$O(1)$	-
MoT-64	Max Tput	0.98 fpc	2D-Mesh	0.5 fpc	104% improvement at comparable area
MoT-64	Max Tput	0.98 fpc	Hypercube	0.8 fpc	22% improvement at comparable area
MoT-64	Max Tput	0.98 fpc	Butterfly	0.9 fpc	9% improvement at comparable area
MoT	Switch Delay	8.98 FO4	2D-Mesh 5-ports, 2 VC	13.1 FO4	2D-Mesh 46% slower.
MoT	Switch Delay	8.98 FO4	Butterfly 2-ports, 2 VC	10.9 FO4	Butterfly 21% slower.
MoT	Switch Delay	8.98 FO4	Replicated Butterfly 2-ports	9.4 FO4	Replicated Butterfly 5% slower.

Table 10.2: Summary of important results in relative terms (comparison to other networks).

Appendix A

MoT Network in XMT Architecture

Our high-level synthesizer generates a network module to be used in HDL description of XMT. In accordance with the specifications in [104], we generate two MoT networks. One network (CM_Network) carries packets from processing clusters to memory modules; the other network (MC_Network) carries memory responses from memory modules to processing clusters. Figure A.1 shows both of these networks combined into one block labeled **IN**.

Additionally, we have two balanced binary tree networks between the *Master TCU* of the XMT processor and memory modules. The network labeled as **MX_BBT** carries requests from Master TCU to memory modules, and the network labeled **MX_BBT_INV** carries the responses in the reverse direction.

The label and description of signals in synchronous MoT implementation as shown in Figure A.1, are given below. To distinguish enumerated cluster and memory modules, names contain a 2-digit suffix such as `_00`, that represents the number of corresponding component. Each signal bundle consists of *Data*, *Request*, and *Acknowledgment* signals. The following list shows the label of the *Data* signal. Labels for *Request* and *Acknowledgment* signals of the same bundle can be obtained by replacing `Data` with `Req` and `Ack` respectively.

1 `wClsInputDataCluster` Flits from processing clusters to memory modules

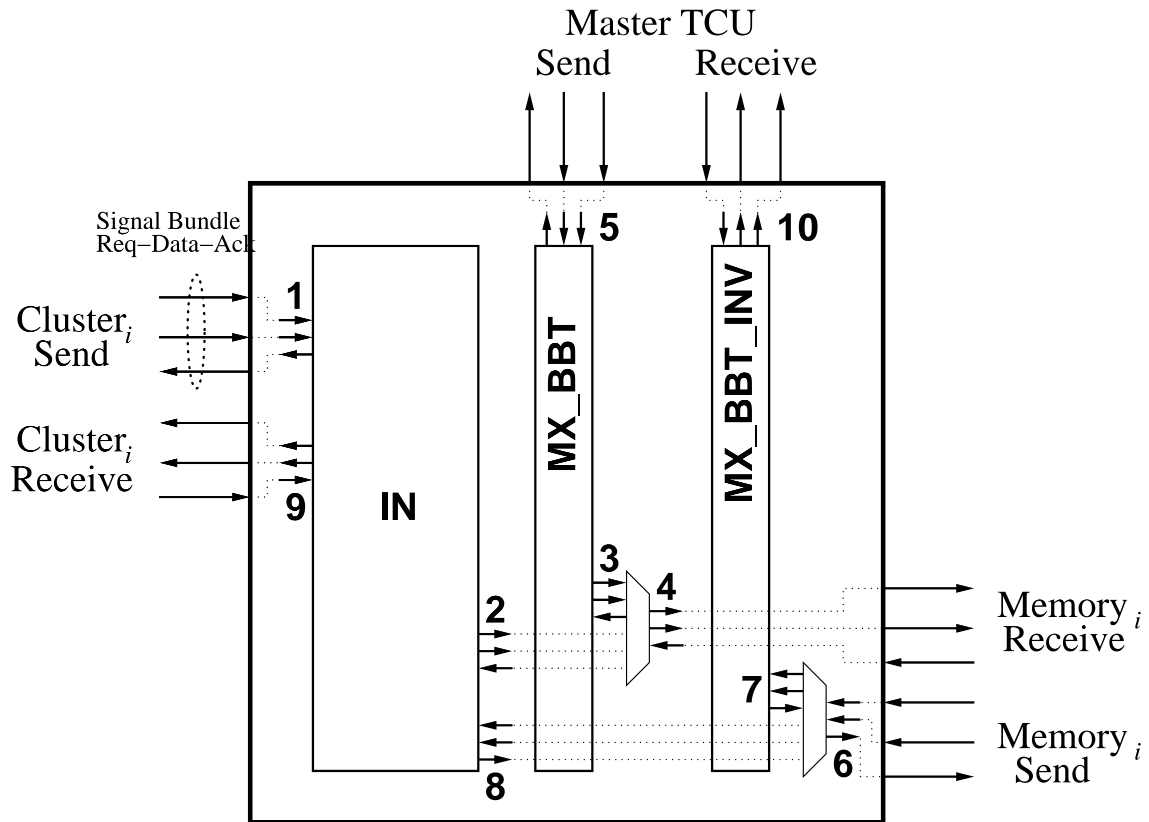


Figure A.1: Block diagram of synchronous network in XMT.

(PC-to-MM) enter the icn.v module.

2 `wClsOutputDataCache` PC-to-MM flits leave the core interconnection network, and enter an arbitration primitive with signal 3.

3 `wMXOutputDataCache` Flits from Master TCU to memory modules (MTCU-to-MM) leave the balanced binary tree module MX_BBT and enter an arbitration primitive with signal 2.

4 `wOutputDataCache` The result of arbitration between signals 2 and 3 leaves the icn.v module to arrive at its destined memory module.

5 `wMXInputDataMX` MTCU-to-MM flits enter icn.v and MX_BBT.

6 `wInputDataCache` Response flits from MM enter icn.v and arrive at a routing block. If their destination is one of the regular clusters, they are routed to signal 8, if their destination is the Master TCU, they are routed to signal 7.

7 `wMXInputDataCache` MM-to-MTCU flits enter the MX_BBT_INV module, which arbitrates and carries them to the Master TCU.

8 `wClsInputDataCache` MM-to-PC flits enter the interconnection network.

9 `wClsOutputDataCluster` MM-to-PC flits leave inerconnection network and icn.v module. Here, we note that the acknowledgment signal in this bundle is hardwired as "logic 1", meaning that the clusters always accept any incoming flit. This is in accordance with the specification shown in [104].

10 `wMXOutputDataMX` MM-to-MTCU flits leave the icn.v module.

The asynchronous implementation contains multiple interfaces between synchronous and asynchronous domains. A block diagram is shown in Figure A.2. Blocks labeled as $S \rightarrow A$ and $A \rightarrow S$ represent synchronous-to-asynchronous and asynchronous-to-synchronous interfaces respectively. Signals labeled 1-10 have the same name and description as described above in the synchronous implementation. Below, we list the additional signals

11 `wClsIfInputDataCluster`. PC-to-MM flits leave a synchronous-to-asynchronous interface and enter asynchronous interconnection network.

12 `wClsIfOutputDataCache`. PC-to-MM flits leave asynchronous network and enter an asynchronous-to-synchronous interface.

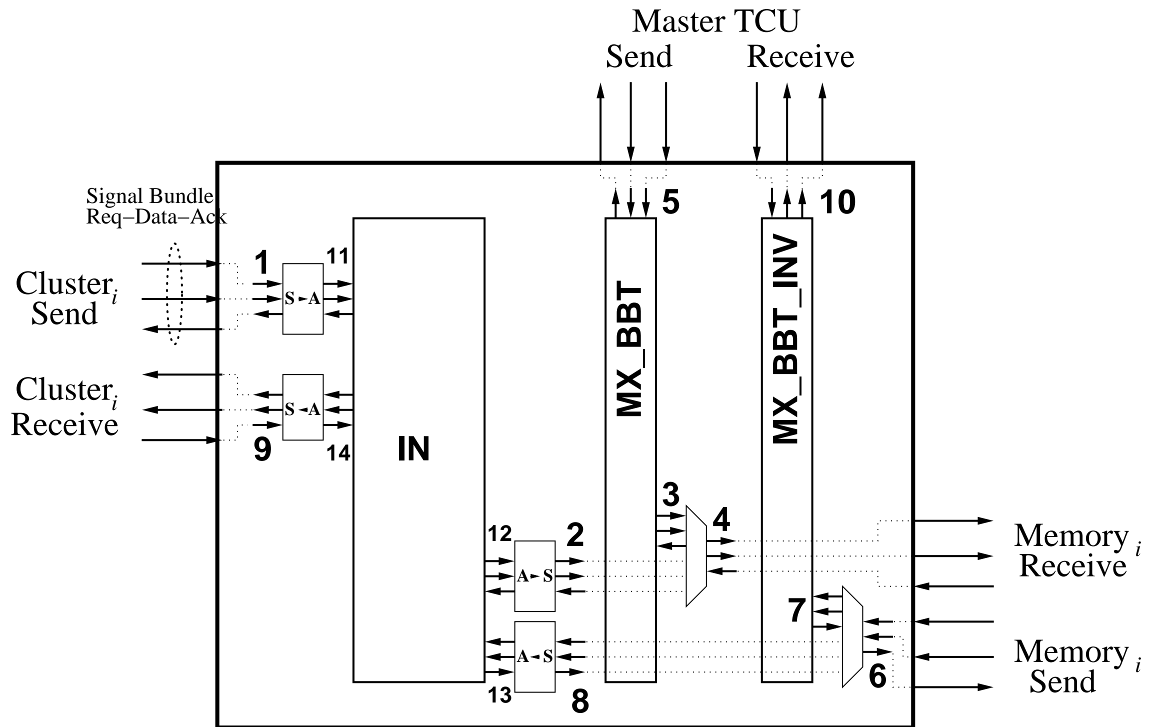


Figure A.2: Block diagram of asynchronous network in XMT.

13 `wClsIfInputDataCache`. MM-to-PC flits leave synchronous-to-asynchronous interface and enter asynchronous interconnection network.

14 `wClsIfOutputDataCluster`. MM-to-PC flits leave asynchronous network and enter asynchronous-to-synchronous interface.

Bibliography

[1] F. Abel, C. Minkenberg, I. Iliadis, T. Engbersen, M. Gusat, F. Gramsamer, and R. Luijten. Design issues in next-generation merchant switch fabrics. *Networking, IEEE/ACM Transactions on*, 15(6):1603–1615, Dec. 2007.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. Int. Conf. On Supercomputing*, pages 1–6, 1990.

[3] F. Angiolini, P. Meloni, S. Carta, L. Benini, and L. Raffo. Contrasting a NoC and a Traditional Interconnect Fabric with Layout Awareness. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, 6-10 March 2006.

[4] F. Angiolini, P. Meloni, S. M. Carta, L. Raffo, and L. Benini. A Layout-Aware Analysis of Networks-on-Chip and Traditional Interconnects for MP-SoCs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(3):421–434, March 2007.

[5] ARM Limited. *Advanced Microcontroller Bus Architecture Specification*, rev 2.0 edition, 1999.

[6] ARM Physical IP Inc. *90nm CMOS9SF LVT 1.2V SAGE-XTM v3 Standard Cell Library Databook*, rev 2.0 edition, May 2006. www.arm.com.

[7] P. Bach, M. Braun, A. Formella, et al. Building the 4 processor SB-PRAM prototype. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, volume 5, pages 14–23, Jan. 1997.

[8] A. O. Balkan, M. N. Horak, G. Qu, and U. Vishkin. Layout-Accurate Design and Implementation of a High-Throughput Interconnection Network for Single-Chip Parallel Processing. In *Proc. IEEE Symp. on High Performance Interconnection Networks (Hot Interconnects)*, Stanford University, CA, August 2007.

[9] A. O. Balkan, G. Qu, and U. Vishkin. Arbitrate-and-Move Primitives for High Throughput On-Chip Interconnection Networks. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, volume II, pages 441–444, Vancouver, May 2004.

[10] A. O. Balkan, G. Qu, and U. Vishkin. A Mesh-of-Trees Interconnection Network for Single-Chip Parallel Processing. In *Proceedings of the Application-Specific Systems, Architectures and Processors (ASAP)*, pages 73 – 80, 2006.

[11] A. O. Balkan, G. Qu, and U. Vishkin. An area-efficient high-throughput hybrid interconnection network for single-chip parallel processing. In *Proc. IEEE/ACM Design Automation Conference (DAC)*, Anaheim, CA, June 2008. IEEE/ACM.

[12] A. O. Balkan and U. Vishkin. Programmer's Manual for XMTC Language, XMTC Compiler and XMT Simulator. Technical Report 2005-45, UMIACS, 2006.

[13] W. Bein, L. Larmore, J. Shields, C., and I. Sudborough. Fixed layer embeddings of binary trees. In *Parallel Architectures, Algorithms and Networks, 2002. I-SPAN '02. Proceedings. International Symposium on*, pages 248–253, 22-24 May 2002.

[14] V. E. Beneš. Rearrangeable Three Stage Connecting Networks. *Bell System Technical Journal*, 41:1481–1492, 1962.

[15] S. N. Bhatt and F. T. Leighton. A Framework for Solving VLSI Graph Layout Problems. Technical Report MIT/LCS/TR-305, Massachusetts Institute of Technology, 1983.

[16] L. Bononi and N. Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2d mesh. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 2, page 6pp., 6-10 March 2006.

[17] Cadence Design Systems Inc. *Design For Test in Encounter RTL Compiler*, March 2007. Product Version 6.2.2.

[18] L. P. Carloni, K. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A Methodology for Correct-by-Construction Latency Insensitive Design. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 301 – 315, 1999.

[19] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059 – 1076, September 2001.

[20] T. Chelcea and S. Nowick. Robust interfaces for mixed-timing systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(8):857–873, Aug. 2004.

[21] W.-M. Chen, G.-H. Chen, and D. Hsu. Combinatorial properties of mesh of trees. In *Parallel Architectures, Algorithms and Networks, 2000. I-SPAN 2000. Proceedings. International Symposium on*, pages 134–139, 7-9 Dec. 2000.

[22] P. Cocchini. Concurrent Flip-Flop and Repeater Insertion for High Performance Integrated Circuits. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 268–273, 2002.

[23] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra. Spidergon: a novel on-chip communication network. In *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, page 15, 16-18 Nov. 2004.

[24] D. E. Culler and J. P. Singh. *Parallel Computer Architecture*. Morgan Kaufmann, 1999.

[25] J. N. Daigle. *Queueing Theory with applications to Packet Telecommunication*. Springer, 2005.

[26] W. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *Computers, IEEE Transactions on*, C-36(5):547–553, May 1987.

[27] W. J. Dally. Virtual-Channel Flow Control. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):194–205, Mar. 1992.

[28] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Fransisco, CA, 2004.

[29] A. DeHon. Compact, Multilayer Layout for Butterfly Fat-Tree. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 206–215, 2000.

[30] A. DeHon and R. Rubin. Design of FPGA interconnect for multilevel metallization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(10):1038–1050, Oct. 2004.

[31] Y. Dinitz, S. Even, R. Kupershtok, and M. Zapolotsky. Some Compact Layouts of the Butterfly. In *SPAA '99: Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 54–63, 1999.

[32] M. Drinic, D. Kirovski, S. Megerian, and M. Potkonjak. Latency-Guided On-Chip Bus-Network Design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(12):2663–2673, Dec. 2006.

[33] G. R. Goke and G. J. Lipovski. Banyan Networks for Partitioning Multiprocessor Systems. In *Proc. 1st Annu. Symp. Computer Architecture*, pages 21–28, 1973.

[34] P. Gopalakrishnan and R. A. Rutenbar. Direct transistor-level layout for digital blocks. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 577–584, Piscataway, NJ, USA, 2001. IEEE Press.

[35] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer–Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. Comput.*, pages 175–189, Feb. 1983.

[36] P. Gratz, C. Kim, K. Sankaralingam, H. Hanson, P. Shivakumar, S. Keckler, and D. Burger. On-chip interconnection networks of the trips chip. *Micro, IEEE*, 27(5):41–50, Sept.-Oct. 2007.

[37] C. Grecu, P. P. Pande, A. Ivanov, and R. Saleh. Structured Interconnect Architecture: A Solution for the Non-Scalability of Bus-Based SoCs. In *Proceedings of the Great Lakes Symposium on VLSI*, pages 192 – 195, 2004.

[38] R. I. Greenberg and L. Guan. On the Area of Hypercube Layouts. *Information Processing Letters*, 84:41–46, 2002.

[39] S. Hassoun, C. J. Alpert, and M. Thiagarajan. Optimal Buffered Routing Path Constructions for Single and Multiple Clock Domain Systems. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 247 – 253, 2002.

[40] J. Henkel, W. Wolf, and S. Chakradhar. On-chip networks: a scalable, communication-centric embedded system design paradigm. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 845–851, 2004.

[41] W. H. Ho and T. M. Pinkston. A Methodology for Designing Efficient On-Chip Interconnects on Well-Behaved Communication Patterns. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 377–388, 2003.

[42] M. N. Horak. Asynchronous implementation of mesh-of-trees network for explicit multi-threading parallel architecture. Master's thesis, University of Maryland, 2008.

[43] C. K. Hung, M. Hamdi, and C. Tsui. Design and Implementation of High-Speed Arbiter for Large Scale VOQ Crossbar Switches. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 2, pages 308–311, 2003.

[44] IBM Corp. *CMOS 9SF Technology Design Manual*, September 2007.

[45] IEEE Computer Society. *IEEE Standard SystemC Language Reference Manual*, March 2006. IEEE Std 1666-2005.

[46] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. *Micro, IEEE*, 26(3):10–23, May-June 2006.

[47] S. Konstantinidou. The selective extra-stage butterfly. *VLSI, IEEE Transactions of*, 1(2):167–171, June 1993.

[48] G. Kornaros and Y. Papaefstathiou. A buffered crossbar-based chip interconnection architecture supporting quality of service. In *Proc. 2007 3rd Southern Conference on Programmable Logic, 2007. SPL '07.*, pages 51–56, 28-26 Feb. 2007.

[49] C. P. Kruskal and M. Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. *Computers, IEEE Transactions on*, 32(12):1091–1098, December 1983.

[50] D. J. Kuck. A Survey of Parallel Machine Organization and Programming. *Computing Surveys*, pages 29–59, 1977.

[51] A. Kumar, L.-S. Peh, P. Kundu, and N. Jha. Toward ideal on-chip communication using express virtual channels. *IEEE Micro*, 28(1):80–90, 2008.

[52] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha. Express virtual channels: Towards the ideal interconnection fabric. In *International Symposium on Computer Architecture (ISCA), 2007*, June 2007.

[53] V. Kumar and S. Reddy. Augmented Shuffle-Exchange Multistage Interconnection Networks. *Computer*, 20(6):30–40, June 1987.

[54] C. Lazzari, C. Santos, A. Ziesemer, L. Anghel, and R. Reist. Efficient timing closure with a transistor level design flow. In *Very Large Scale Integration, 2007. VLSI - SoC 2007. IFIP International Conference on*, pages 312–315, Oct. 2007.

[55] K. Lee, S.-J. Lee, and H.-J. Yoo. A distributed crossbar switch scheduler for on-chip networks. *Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003*, pages 671–674, Sept. 2003.

[56] F. T. Leighton. New Lower Bound Techniques for VLSI. In *Proc. Of the 22nd IEEE Symposium on Foundations of Computer Science*, pages 1–12, 1981.

[57] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[58] C. E. Leiserson. Fat Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, Oct. 1985.

[59] L. Lloyd, K. Heron, A. M. Koelmans, and A. V. Yakovlev. Asynchronous Microprocessors: From High Level Model to FPGA Implementation, 1997.

[60] I. Loi, F. Angiolini, and L. Benini. Developing mesochronous synchronizers to enable 3d nocs. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 1414–1419, 10-14 March 2008.

[61] P. Lopez, P. Lopez, J. Martinez, and J. Duato. Dril: dynamically reduced message injection limitation mechanism for wormhole networks. In J. Martinez, editor, *Proc. International Conference on Parallel Processing*, pages 535–542, 1998.

[62] R. Lu, G. Zhong, C. Koh, and K. Chao. Flip-Flop and Repeater Insertion for Early Interconnect Planning. In *DATE '02: Proceedings of the Conference on Design, Automation and Test in Europe*, page 690, 2002.

[63] R. Marculescu. Networks-on-chip: the quest for on-chip fault-tolerant communication. In *VLSI, 2003. Proceedings. IEEE Computer Society Annual Symposium on*, pages 8–12, Feb. 2003.

[64] K. Mehlhorn and U. Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica*, 21:339–374, 1984.

[65] M. Mehmet-Ali, M. Youssefi, and H. Nguyen. The performance analysis and implementation of an input access scheme in a high-speed packet switch. *Communications, IEEE Transactions on*, 42(12):3189–3199, Dec 1994.

[66] D. Mitra and R. A. Cieslak. Randomized Parallel Communications on an Extension of the Omega Network. *Journal of the Association for Computing Machinery*, 34(4):802–824, October 1987.

[67] C. Molnar and I. Jones. Simple circuits that work for complicated reasons. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 138–149, 2000.

[68] MOSIS. http://mosis.org/cgi-bin/params/tsmc-018/t29b_mm_non_epi-params.txt.

[69] S. Murali, D. Atienza, P. Meloni, S. Carta, L. Benini, G. De Micheli, and L. Raffo. Synthesis of predictable networks-on-chip-based interconnect architectures for chip multiprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(8):869–880, Aug. 2007.

[70] S. Murali, L. Benini, and G. De Micheli. An Application-Specific Design Methodology for On-Chip Crossbar Generation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(7):1283–1296, July 2007.

[71] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Towards a First Vertical Prototyping of an Extremely Fine-Grained Parallel Programming Approach. *Theory of Computer Systems*, 2003. Special Issue of SPAA 2001.

[72] J. Nuzman. Memory Subsystem Design for Explicit Multithreading Architectures. Master's thesis, University of Maryland, 2003.

[73] U. Ogras, R. Marculescu, P. Choudhary, and D. Marculescu. Voltage-frequency island partitioning for gals-based networks-on-chip. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 110–115, June 2007.

[74] P. P. Pande, C. Grecu, A. Ivanov, and R. Saleh. Design of a Switch for Network on Chip Applications. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages V–217 – V–220 vol.5, 2003.

[75] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Evaluation of MP-SoC Interconnect Architectures: A Case Study. In *IEEE International Workshop on System-On-Chip for Real-Time Applications*, pages 253 – 356, 2004.

[76] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. Das. Exploring fault-tolerant network-on-chip architectures. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 93–104, 2006.

[77] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Bmsyn: Bus matrix communication architecture synthesis for mpsoc. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(8):1454–1464, Aug. 2007.

[78] R. Patti. Three-dimensional integrated circuits and the future of system-on-chip designs. *Proceedings of the IEEE*, 94(6):1214–1224, June 2006.

[79] L.-S. Peh and W. J. Dally. A Delay Model and Speculative Architecture for Pipelined Routers. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 255–266, 2001.

[80] F. Petrini and M. Vanneschi. Performance Analysis of Wormhole Routed K-Ary N-Trees. *International Journal of Foundations of Computer Science*, 9(2):157–178, 1998.

[81] F. Petrot and D. Hommais. A Generic Programmable Arbiter with Default Master Grant. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 5, pages 749–752, 2000.

[82] F. P. Preparata and J. Vuillemin. The cube-connected cycles: a versatile network for parallel computation. *Commun. ACM*, 24(5):300–309, 1981.

[83] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. De Micheli, and L. Benini. Bringing nocs to 65 nm. *Micro, IEEE*, 27(5):75–85, Sept.-Oct. 2007.

[84] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits, A Design Perspective*. Prentice Hall, second edition, 2003.

[85] T. G. Robertazzi. *Networks and Grids*. Springer, 2007.

[86] R. Roy, D. Bhattacharya, and V. Boppana. Transistor-level optimization of digital designs with flex cells. *Computer*, 38(2):53–61, Feb. 2005.

[87] J. T. Schwartz. The Burroughs FMP Machine. Ultracomputer Note 5, Courant Institute, NYU, New York, NY, 1980.

[88] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*, 1999. Available online: http://public.itrs.net/Files/1999_SIA_Roadmap/Home.htm.

[89] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*, 2003. Available online: http://public.itrs.net/Files/2003ITRS/Home2003.htm.

[90] M. Shams, J. Ebergen, and M. Elmasry. A Comparison of CMOS Implementations of an Asynchronous Circuits Primitive: The C-Element. In *ISLPED '96: Proceedings of the 1996 international symposium on Low power electronics and design*, pages 93–96, Piscataway, NJ, USA, 1996. IEEE Press.

[91] E. S. Shin, V. J. M. III, and G. F. Riley. Round-Robin Arbiter Design and Generation. In *Proc.* 15th *International Symposium on System Synthesis*, pages 243–248, 2002.

[92] J. Sparsø and S. Furber, editors. *Principles of Asynchronous Design*. Kluwer Academic Publishers, 2001.

[93] S. Srinivasaraghavan and W. Burleson. Interconnect effort - a unification of repeater insertion and logical effort. In *VLSI, 2003. Proceedings. IEEE Computer Society Annual Symposium on*, pages 55–61, 20-21 Feb. 2003.

[94] I. Sutherland. Micropipelines. *Communications of the ACM*, June 1989. Turing Award Lecture.

[95] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agrawal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2003.

[96] C. D. Thompson. *A Complexity Theory for VLSI*. PhD thesis, Carnegie Mellon University, 1979.

[97] A. W. Topol, J. D. C. La Tulipe, L. Shi, D. J. Frank, K. Bernstein, S. E. Steen, A. Kumar, G. U. Singco, A. M. Young, K. W. Guarini, and M. Ieong. Three-dimensional integrated circuits. *IBM Journal of Research and Development*, 50(4/5):491–506, July/September 2006.

[98] E. Upfal. An O(log N) Deterministic Packet-Routing Scheme. *Journal of the Association for Computing Machinery*, 39(1):55–70, January 1992.

[99] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, Jan. 2008.

[100] U. Vishkin. Tutorial on how to develop PRAM-like programs and run them on the FPGA prototype. ICS 2007, June 2007.

[101] U. Vishkin, G. Caragea, and B. Lee. *Handbook on Parallel Computing: Models, Algorithms, and Applications*, chapter Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. CRC Press, 2008. Ed: S. Rajasekaran and J. Reif;.

[102] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit Multi Threading (XMT) Bridging Models for Instruction Parallelism. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 140–151. ACM, 1998.

[103] U. Vishkin and J. Nuzman. Circuit architecture for reduced-synchrony on-chip interconnect. US Patent 6,768,336.

[104] X. Wen. *Hardware Design, Prototyping and Studies of the Explicit Multi-Threading (XMT) Paradigm.* PhD thesis, University of Maryland, 2008.

[105] X. Wen and U. Vishkin. FPGA-Based Prototype of a PRAM-on-Chip Processor. In *Proc. ACM Computing Frontiers*, Ischia, Italy, May 2008.

[106] T. Wu, C.-Y. Tsui, and M. Hamdi. A 2 gb/s 256*256 cmos crossbar switch fabric core design using pipelined mux. *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, 2:II–568–II–571 vol.2, 2002.

[107] T. T. Ye and G. D. Micheli. Physical Planning for On-Chip Multiprocessor Networks and Switch Fabrics. In *Proceedings of the Application-Specific Systems, Architectures and Processors (ASAP)*, pages 97 – 107, 2003.

[108] C.-H. Yeh. Optimal Layout for Butterfly Networks in Multilayer VLSI. In *International Conference on Parallel Processing (ICPP)*, pages 379 – 388, 2003.

[109] C.-H. Yeh, B. Parhami, E. A. Varvarigos, and H. Lee. VLSI Layout and Packaging of Butterfly Networks. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 196–205, 2000.

[110] J. Yuan and C. Svensson. New Single-Clock CMOS Latches and Flipflops with Improved Speed and Power Savings. *IEEE J. Solid-State Circuits*, 32(1):62–69, January 1997.

[111] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. Gao. A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, 25-29 April 2006.

[112] S. Q. Zheng, M. Yang, J. Blanton, P. Golla, and D. Verchere. A Simple and Fast Parallel Round Robin Arbiter for High-Speed Switch Control and Scheduling. In *Proc.* 45th *Midwest Symposium on Circuits and Systems*, volume 2, pages 671–674, 2002.