Time-Multiplexed FPGA Overlay Networks on Chip

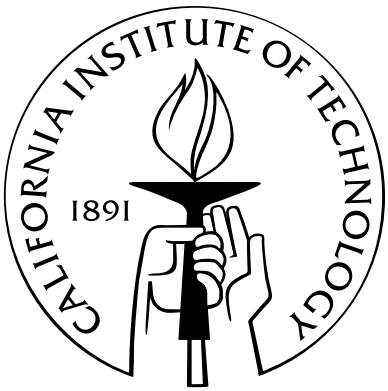
Thesis by

Nikil Mehta

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science



California Institute of Technology

Pasadena, California

2006

(Submitted May 31, 2006)

Acknowledgements

First and foremost I would like to thank my advisor André DeHon for his guidance and support throughout this project. Without his motivation and help this work would not have been possible. He is the best mentor a student could ask for.

Second, and equally important, I must thank Nachiket Kapre for the countless hours he devoted to completing this project with me. His (arguably better) thesis is required reading for those who wish to have a deeper understand of the material covered here. Half of the work presented here is a result of his effort and determination, while the other half is a result of my attempt to match his dedication.

Many thanks go to Rafi Rubin for the incredible amount of support code, software automation, and verbal abuse that he provided during this work. I need to thank Michael deLorimier for his help in partitioning and node decomposition. I would also like to thank the others in the IC Group (Helia Naeimi, Dominic Rizzo, Benjamin Gojman, Concetta Pilotto) for their advice, support, and constant enthusiasm.

I would like to thank those from industry that attended workshops on the GraphMachine project and provided invaluable suggestions: Shepard Siegel, Michael Baxter, Steve Trimberger, Stephen Neuendorffer, Kees Vissers, and many others.

Finally, I want to thank Katie Shilton for reading this work and helping me ensure that people outside of computer science can understand (some of) it. In the words of John Ellis, most graduate students have dark, depressed moments at some point in the game, and Katie helped me persevere through mine.

Abstract

How do we design a communication network for processing elements (PEs) on a single chip that minimizes application communication time and area? In designing such a network it is essential to use a network communication pattern that matches application communication and area requirements. This report characterizes the design space of a particular communication pattern for networks on chip: Time-Multiplexed Interconnect. In contrast to more commonly used packet-switched networks, which route communication dynamically, time-multiplexed networks schedule all possible communication prior to runtime with an offline router. We describe how to build well engineered, highly scalable time-multiplexed FPGA networks in terms of topology selection, routing algorithm design and hardware design that operate on a Xilinx XC2V6000-4 at 166MHz. To benchmark our networks we use real, communication rich applications instead of generating synthetic traffic. We show that over all areas (10K–10M slices) and over all applications the best "one topology fits all" is Butterfly Fat Tree (BFT) with $c = 1, p = 0.5$, which requires, in the worst case, 6.1× as many cycles to route communication than the optimal topology. We compare time-multiplexing to packet-switching, and show that on average, over all applications for all equivalent topologies, online packet-switched communication requires 1.5× as many cycles to route as offline time-multiplexed scheduling. When applying designs to equivalent area, for areas <100K slices packet-switching typically outperforms time-multiplexing, but at >100K slices packet-switching requires up to 3.4× as many cycles to route as time-multiplexing in the worst case. Finally, for equivalent, large networks (>100 PEs) time-multiplexing outperforms packet-switching for communication loads where greater than 10% of all logical links are active. This demonstrates that well designed time-multiplexed FPGA overlay networks can deliver performance and area efficiency exceeding that of packet-switched networks.

Contents

List of Tables

List of Figures

Glossary of Terms

Activity Factor: Percentage of active edges in a communication graph. When representing communication as a graph, where nodes correspond to actors and edges correspond to messages that must be sent, all messages sent corresponds to an activity factor of 100%.

ASIC: Application Specific Integrated Circuit. An integrated circuit that is designed to perform a specific task. ASIC circuits are typically smaller and faster than equivalent FPGA circuits. However, such a circuit cannot be altered after manufacturing and is costly to design.

BFT: Butterfly Fat Tree. A binary tree has connections between levels equal to the number of nodes at a given level. In a Butterfly Fat Tree the number of connections between levels can increase at higher levels of the tree (*i.e.* it becomes "fatter" at the top). The exact "fatness" of the tree is parameterizable by the Rent parameter p.

Bisection: When separating a graph into two parts, the bisection of the graph is the number of edges that cross the two sides. This is an important value in two contexts. First, when the graph is a circuit to be laid out in two-dimensional VLSI, it represents the minimal cross-sectional area required for the wires that connect the two halves of the circuit. Second, when the graph is a network, the bisection of the network represents the total number of physical links between the two halves of the network, and therefore the total number of messages that can be sent between the two halves at one time. This value represents the available bandwidth of the network.

ConceptNet: A common-sense reasoning knowledge base represented as a graph, where nodes represent concepts and edges represent semantic relationships.

Context Memory: A memory that stores exactly what a switch or PE in a time-multiplexed network should do on every cycle. That is, it stores the exact hardware configuration of the element which dictates how messages are routed through the element. Since every element requires a context memory, and each context memory stores a configuration for every communication cycle, the total area consumed by all context memories in a network can potentially be very large.

FSM: Finite State Machine. A model of behavior composed of states, transitions between states, and actions performed when entering a state.

FPGA: Field Programmable Gate Array. A device which contains programmable logic and interconnect. By programming the device one can implement arbitrary digital logic. Reprogrammability allows designs to be changed, unlike ASICs.

FPGA Overlay Network: The notion of placing a logical network over the physical network of an FPGA. The physical FPGA network refers to the wires and programmable switches that connect programmable logic. By programming these switches and logic, one can create a circuit that is itself a network, implemented on top of the physical FPGA network.

GraphStep: A system architecture for sparse graph algorithms. This describes both a way to represent sparse graph algorithms and a process for implementing a hardware system. In the GraphStep model each graph node is an actor that can communicate with the neighboring nodes connected to its edges.

LUT: Look Up Table. The basic unit of logic in an FPGA which maps a set of k (typically 4) input values to a single output value. A LUT is programmed to implement any k input function. Upon receiving one of the 2^k input permutations, instead of actually computing the correct output, the solution is looked up directly in a table.

NoC: Network on chip. Components in a system on chip (SoC) communicate with each other via this network.

PE: Processing Element. Generally used to refer to some circuit that performs computation.

Packet-Switching: A network communication pattern where the path of a message is dynamically determined by switches at runtime (*e.g.* the Internet).

Rent's Rule: $IO = cN^p$. An empirical relationship that relates the number of external signal connections of a block of logic (IO) to the number of logic gates/nodes in the block (N). This relationship applies to many systems such as digital circuits and multicomputers.

Router: A software tool that schedules the communication of a time-multiplexed network prior to runtime. This tool routes all messages in space and schedules them in time on physical network resources. The schedule computed by the router is stored in context memory.

Runtime Reconfiguration: The capability of an FPGA device to reconfigure while the device is performing a computation.

Self Message: A message where the source and destination is the same processing element. As communication in the GraphStep system architecture is specified between nodes, and multiple nodes may reside in a single PE, messages may be sent from and to nodes in the same PE.

SMVM: Sparse Matrix Vector Multiply. A multiplication operation performed between a sparse matrix (a matrix containing primarily zeros) and a vector. This operation is used in many popular iterative algorithms, such as Conjugate Gradient and `GMRES`.

Slice: A unit of area in a Xilinx FPGA. For the Virtex2 family of parts, one slice is equivalent to two 4 input LUTs, each with output registers.

SoC: System on Chip. The idea of integrating all components of a computer system on a single chip. This provides several general benefits such as reduced cost and increased system performance.

SRL16: A 16-bit shift register implemented using the configuration bits of a 4-input LUT. These shift registers can be cascaded together to form larger shift registers.

Time-Multiplexing: A network communication pattern where communication between processing elements is scheduled before actually taking place. Messages are multiplexed (*i.e.* interleaved) in time over a physical link. The exact timing of the interleaving of messages is calculated by a router.

Chapter 1

Introduction

1.1 Thesis

For applications where communication between processing elements can be scheduled offline, well designed time-multiplexed FPGA overlay networks on chip can deliver performance and area efficiency exceeding that of online, packet-switched networks.

1.2 Motivation and Scope

One of the earliest and most obvious techniques developed to increase the performance of computing systems is the simultaneous use of multiple processors to solve a single problem. How quickly a multiprocessor system can solve a problem depends on three fundamental components: the nature of the problem and solution algorithm, the computational rate of each processor, and the communication time between processors. For many parallel systems the primary limiter on total performance is communication time, not the performance of an individual processor. In order for processors to work together to solve a computation, they must communicate with each other quickly and efficiently.

The rate at which processors can communicate is determined by the effectiveness of the network that connects them. The quality of a network can be measured in many ways:

- **Latency:** How long does it take for one processor to communicate with another?
- **Bandwidth:** How much communication can the network support at one time?
- **Area:** How much area (silicon, PCB, etc.) does the network consume?
- **Energy:** How much energy does the network consume?
- **Reliability:** Can the network tolerate faults, defects, or other errors?
- **Flexibility:** Can the network support any type of communication?

- **Binding Time:** Must communication be known before runtime?
- **Scalability:** How do these characteristics change as the size of the network grows?

The scope of this report is to evaluate networks in terms of performance (latency, bandwidth) and area efficiency over a large range of network sizes. We do not attempt to address energy or reliability concerns.

How do we design a scalable network that minimizes both communication time and area? We will explore in detail the primary design decisions involved in constructing such a network:

- **Communication Pattern:** What is the overall model for how messages are transported over the entire network? (Section 1.2.2)
- **Routing Algorithm:** How are messages scheduled/negotiated on individual network resources (*i.e.* processors, switches)? (Chapter 4)
- **Network Topology:** How are network resources connected together? (Chapters 3 and 6)
- **Hardware Design:** How are network resources designed at the circuit level? (Chapters 3 and 8)

The most fundamental design decision is the choice of communication pattern, or switching style of the network. This report focuses on one particular network communication pattern: Time-Multiplexed Interconnect. We attempt to characterize how Time-Multiplexed Interconnect compares to other communication patterns (in particular, Packet-Switched Interconnect), and how to best design a time-multiplexed network in terms of routing algorithm, network topology, and hardware design.

While certain trends presented in this work are relevant to all types of multiprocessor networks, we limit our scope to one specific type of multiprocessor network: the FPGA overlay network on chip (NoC).

1.2.1 FPGA Networks on Chip

Networks for connecting large numbers of processors in computing systems have been studied by the parallel computing community for decades [51, 6, 55, 30, 37, 18]. Researchers have done extensive work in exploring switching styles, routing algorithms, network topologies, and hardware design in multiprocessor systems.

These systems typically assume one processor per single physical chip. With recent increases in silicon capacity it is now possible to fit several processing elements (PEs) on a single chip and connect these components via a network on chip (NoC) [7]. Modern chips have enough capacity for 10s–100s of PEs on chip with the promise of 1000s of PEs in the future, enabling single chip systems large enough to solve significant problems. A number of network architectures have been proposed for

NoCs, borrowing heavily from previously developed architectures for parallel computing. However, NoCs have constraints that differ significantly from those of multiprocessors:

- **Two-Dimensional Layouts:** Multiprocessor systems allows multiple chips to be connected in any configuration in three-dimensional space. However, for single chip solutions, current silicon manufacturing technology allows a only single layer of transistors on a chip. While multiple layer technologies are being developed, current NoC designs must be placed in two-dimensional layouts.

- **Quadratic Unbuffered Wire Delays:** The time required for a signal to propagate over an inter-processor wire in a multiprocessor system is roughly linear in the length of the wire. The delay of an unbuffered wire in a integrated circuit is quadratic in the length of the wire.

- **Fewer Pin Constraints:** The number of input/output pins to a processor in a multiprocessor system is limited by packaging technology, the size of physical wires on a circuit board, and the energy required to drive them. I/O can increase significantly for NoCs by using wires on the same scale as the processing element itself.

Little work has been done to characterize the tradeoffs in network design under these new constraints. Therefore, it is essential to reexamine network architectures under an appropriate cost model.

One model that is particularly intriguing to NoC designers desiring flexibility is the Field Programmable Gate Array (FPGA). FPGAs provide a general reconfigurable fabric of logic upon which arbitrary digital hardware can be created quickly and changed easily. Instead of spending a significant amount of time and money implementing a single design on an ASIC or custom chip, users can create and test many different designs on an FPGA. NoC designers in particular can create an NoC in hardware by mapping or **overlaying** a logical network on top of the FPGA reconfigurable network. The flexibility provided by FPGAs enables techniques such as application-specific NoC synthesis, runtime reconfigurable NoCs, and rapid design space exploration and testing of NoCs. Despite these advantages over an ASIC or custom NoC model, the FPGA overlay NoC model is relatively new and unexplored. We will therefore focus our exploration of time-multiplexed networks on the FPGA overlay NoC cost model.

1.2.2 Network Communication Patterns

An fundamental choice to make in designing network interconnect is the selection of a communication pattern, or switching style. To begin to understand where time-multiplexing belongs in the space of switching styles, Table 1.1 rounds up the relative strengths and weaknesses of four commonly used

Characteristics	Spatially Configurable	Time-Multiplexed	Packet-Switched	Circuit-Switched
Communication known	early (compile time)		late (runtime)	
Communication predictability	high		low	
Communication throughput compared to physical link throughput	>	<	<	<
Message latency compared to message length	n/a	n/a	>	<
Physical link utilization	high		low	
Switch logic area	low	low	high	high
Switch memory area	low	high	modest	low
Relative message latency	lowest	low	highest	moderate

Table 1.1: Role of Various Communication Patterns

communication patterns. Each pattern performs optimally under a unique set of application communication and hardware conditions; our goal is to quantify these conditions for time-multiplexing. The basic idea behind each pattern is explained as follows:

Spatially Configurable Interconnect Spatially Configurable Interconnect consists of dedicating pre-configured physical paths between communicating PEs. These paths are dedicated in the sense that they can only be used for messages sent from the source PE to the sink PE on the path; no other pairs of PEs can share links in this path. To use this pattern the communication between PEs must be *known in advance*. For this pattern to be efficient, the application must require that we move data between PEs at a rate *comparable to or higher than* the physical link throughput (*i.e.* links should be not be underutilized, otherwise dedicating a link is unnecessary).

Figure 1.1 shows a spatially configured network where messages are routed through dedicated links. Each message sent in this figure (from PEs $A \rightarrow D$, $B \rightarrow D$, $C \rightarrow A$) propagates on its own pre-configured physical link. The programmable switches in the network have a local configuration state, and we set that state to provide a direct path of physical links between sending and receiving PEs. Once set, the source PE simply places data on its output and the data propagates over the path from the source to the sink with no additional delays beyond physical switching delays. Consequently, communication latency is minimized. Further, switches can be very simple with minimal logic and memory area.

An everyday example of Spatially Configurable Interconnect is FPGA interconnect, where programmable logic (corresponding to PEs in Figure 1.1) is connected via wires and programmable switches (corresponding to switches in Figure 1.1).

Time-Multiplexed Interconnect Instead of dedicating links, Time-Multiplexed Interconnect consists of sharing (*i.e.* multiplexing in time) physical links between communicating PEs in a completely predetermined manner. An offline **router** is responsible for computing this schedule prior

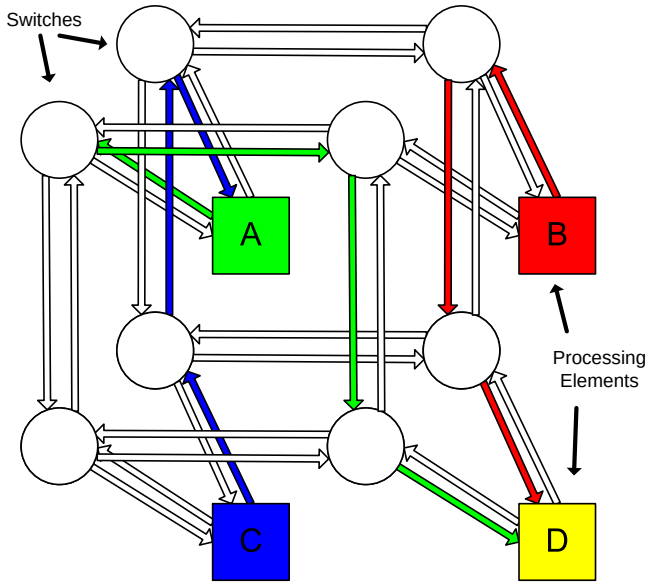


Figure 1.1: Spatially Configurable Interconnect

to runtime. Like Spatially Configurable Interconnect, to use this pattern the communication between PEs must be *known in advance*. Unlike Spatially Configurable Interconnect, for this pattern to be efficient the application must require that we move data between PEs at a rate *significantly lower than* the physical link throughput (*i.e.* since links are shared they cannot be used heavily by pairs of PEs).

Figure 1.2 shows a time-multiplexed network where messages are routed through pre-programmed, shared links. Each message sent in this figure (from PEs $A \rightarrow D$, $B \rightarrow D$, $C \rightarrow A$) propagates on a physical link that is shared in time. The programmable switches in the network have an FSM or **context memory** which is pre-programmed by the router to give it a distinct switch configuration on each time-step. This program is executed repeatedly (*i.e.* a cyclic schedule) to route messages between sources and sinks. We can see in Figure 1.2 that because time-multiplexing can schedule communication offline, network resources are used efficiently and messages do not collide. Communication latency is low as data is routed from link-to-link without additional delays, but may be higher than Spatially Configured Interconnect since switches must change state on every time-step. Switches can be simple with minimal logic area. However, since each switch stores a program in context memory, if the length of the communication is long, the program and the associated context memory can become very large. To illustrate how large this can be: networks we examine contain hundreds of switches and may take tens of thousands of cycles to route communications. Therefore the total number of context memory entries in the entire network can be on the order of millions.

A critical feature to note here is that since time-multiplexing schedules communication prior to runtime, it must schedule *all possible communication*. That is, if only a subset of PEs are communicating and this is not known prior to runtime (*e.g.* it is data dependent), the time-multiplexed

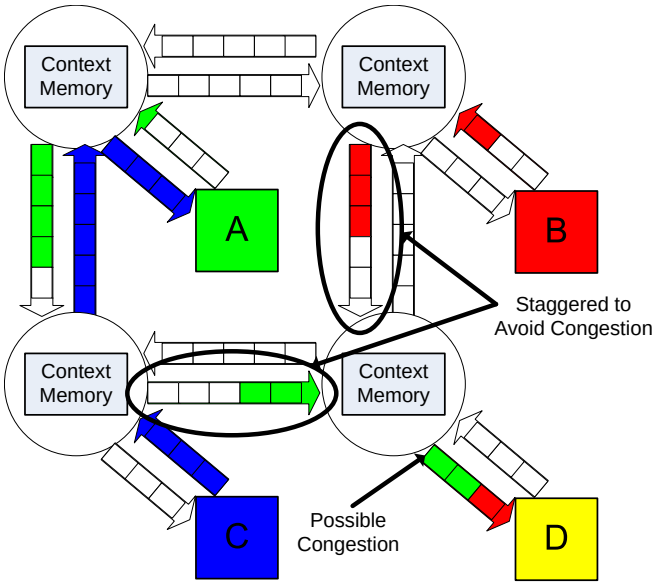


Figure 1.2: Time-Multiplexed Interconnect

router is forced to schedule non-essential communication. Therefore, at runtime PEs that are not actually sending messages will have time-slots allocated in the network in the event that they ever do send messages. For low number of PEs actually communicating at runtime, this could require more cycles to route than necessary.

An everyday example of Time-Multiplexed Interconnect is the setup of train tracks, where trains share tracks in time. The paths of trains and the configuration of switches between tracks are scheduled ahead of time to avoid conflicts.

Circuit-Switched Interconnect Both Spatially-Configured Interconnect and Time-Multiplexed Interconnect require all communication between PEs to be known in advance. However, for some applications communication is not known prior to runtime. Circuit-Switched Interconnect allows communication to be routed dynamically at runtime, where physical paths between communicating PEs are dynamically dedicated for the length of a single communication. Source PEs dynamically request paths to sink PEs at runtime, and the network allocates a dedicated path if one is available. This is useful when the communication pattern between PEs is *not known in advance*. For this pattern to be efficient, communication must be unpredictable such that we move data between PEs at a rate *significantly lower than* the physical link throughput (*i.e.* since links are shared they cannot be used heavily by pairs of PEs). Additionally, the latency of the network should be small in comparison to the length of the message, as the setup time required to allocate and dedicate a path should not be greater than the length of time that the path is used.

Figure 1.3 shows a circuit-switched network where messages are dynamically routed through dedicated paths. Each message sent in this figure (from PEs $A \rightarrow D$, $B \rightarrow D$, $C \rightarrow A$) propa-

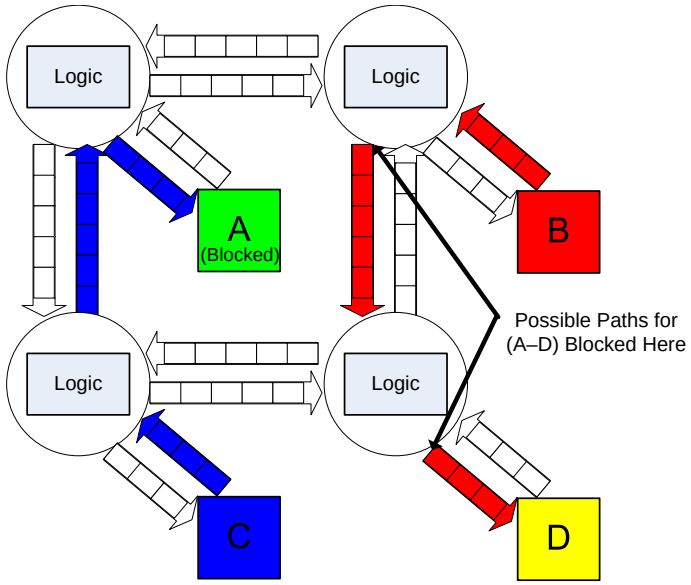


Figure 1.3: Circuit-Switched Interconnect

gates on a physical link that is shared in time. The switches dynamically accept route requests and setup a path to the destination if one is available. We see that no route form $A \rightarrow D$ is available, so A must wait for communication between $B \rightarrow D$ to cease. Since the switches must make online switching decisions, they are more complex, and hence larger and slower than switches in Spatially-Configured Interconnect and Time-Multiplexed Interconnect. Unlike Time-Multiplexed Interconnect no configuration memory is required in the switches, and unlike Packet-Switched Interconnect no memory is required to provide data buffering in the network. Further, since switching decisions are made based only on instantaneous, local information, switch utilization is lower than Time-Multiplexed Interconnect, whose configuration can be globally coordinated offline.

An everyday example of Circuit-Switched Interconnect is the telephone network, where calls are dynamically allocated to physical connections that are dedicated only for the length of the communication.

Packet-Switched Interconnect Like Circuit-Switched Interconnect, Packet-Switched Interconnect is capable of routing communication *not known in advance*, where communicating PEs send packets of data that are individually negotiated through shared physical links the network. Like Circuit-Switched Interconnect, for this pattern to be efficient communication must be unpredictable such that we move data between PEs at a rate *significantly lower than* the physical link throughput (*i.e.* since links are shared they cannot be used heavily by pairs of PEs). However, unlike Circuit-Switched Interconnect, the latency of the network should be large in comparison to the length of the message (otherwise circuit-switching is more efficient).

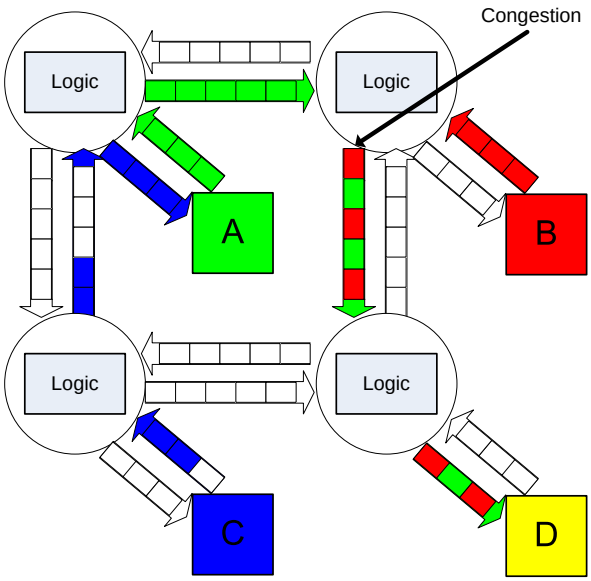


Figure 1.4: Packet-Switched Interconnect

Figure 1.3 shows a packet-switched network where messages are dynamically routed through shared links. Each message sent in this figure (from PEs $A \rightarrow D$, $B \rightarrow D$, $C \rightarrow A$) propagates on a physical link that is shared in time. Communication takes place via packets, where each packet contains a tag with the destination PE. The switches read the tag and locally determine the direction to send the packet in. Because switches make local, greedy decisions, messages can collide and congest switches. We see that both messages from $A \rightarrow D$ and $B \rightarrow D$ are routed through the same physical links at the same time, creating a bottleneck. Each switch contains a Queue with backpressure to accommodate resource limitations in the network. Since the switches must make online switching decisions, packet switches are more complex, and hence larger and slower than switches in Spatially-Configured Interconnect and Time-Multiplexed Interconnect. Unlike Time-Multiplexed Interconnect no configuration memory is required in the switches; however, some memory is required to provide the data queues. Further, since switching decisions are made based only on instantaneous, local information, switch utilization is lower than Time-Multiplexed Interconnect whose configuration can be globally coordinated offline.

A critical feature to note here is that unlike Time-Multiplexed Interconnect, when routing packets we only need to support instantaneous traffic demands. Time-Multiplexed Interconnect must schedule resources for PEs that may never communicate; Packet-Switched Interconnect only needs to route messages for PEs that are actually communicating. This may result in a net reduction in communication time despite the fact that physical resource utilization is lower than Time-Multiplexed Interconnect.

An everyday example of Packet-Switched Interconnect is the Internet.

1.2.3 Time-Multiplexed vs. Packet-Switched Interconnect

Chapter 7 will focus on comparing TIME-MULTIPLEXED INTERCONNECT to PACKET-SWITCHED INTERCONNECT in the context of choosing an appropriate communication pattern. SPATIALLY CONFIGURED INTERCONNECT is inefficient for applications that underutilize physical link throughput and will not be prioritized for this report. CIRCUIT-SWITCHED INTERCONNECT is efficient only for larger messages on shorter networks, and is not an appropriate solution for the kinds of small message applications we focus on here. See Chapter 9 for a discussion of exploring these switching styles in the future.

To illustrate the kinds of tradeoffs between time-multiplexing and packet-switching we will examine, consider the following example. Packet-switching typically requires larger switchboxes due to buffering and logic required for dynamic decision making. Time-multiplexed switchboxes are typically smaller in terms of raw logic, but if the total number of communication cycles is large, switch context memory may require significant area. Packet-switched switchboxes may allow us to fit an 8 PE network on a single fixed-size chip. With comparatively smaller switches we could fit a 16 PE time-multiplexed network as long as routing could be completed in 30K cycles. Here time-multiplexing can use more PEs in parallel to solve a computation and can therefore provide higher performance than packet-switching. However, if routing takes more than 30K cycles, time-multiplexed switch area increases and we can fit only 8 PEs. Furthermore, at more than 100K cycles we can only fit 4 PEs. Now packet-switching enables more parallelism and can outperform time-multiplexing. Consequently, there are operating ranges in which each network is preferred.

To select wisely between packet-switching and time-multiplexing for FPGA overlay NoCs, we need to develop analytic area and time models to ground the general trends of Table 1.1 into quantitative, empirical tradeoffs. In particular, we want to quantify the following three questions:

- For equivalent topologies and communication loads, what is the quantitative difference between the offline, global routing of time-multiplexing and the online, local routing of packet-switching? (Section 7.2)
- For an equivalent, fixed area capacity and equivalent communication loads, what is the performance difference between well engineered time-multiplexed and packet-switched networks? (Section 7.3)
- For communication loads that only require routing a subset of all possible communication, what is the cost of routing all possible communication for time-multiplexed interconnect? (Section 7.4)

1.3 Organization

The rest of this report is organized as follows. Chapter 2 further motivates this study by introducing prior work in NoCs and time-multiplexed networks. Chapter 3 explains how to design the hardware of a time-multiplexed network and introduces the topologies that we explore. Chapter 4 describes our routing algorithm and how to measure its efficiency. Chapter 5 introduces the real (*i.e.* not synthetic) communication workloads we use to benchmark our networks. Chapter 6 shows how to choose an optimal network topology. Chapter 7 compares time-multiplexing with packet-switching. Chapter 8 explains how to further optimize time-multiplexed networks by compressing context memory. Chapters 9 and 10 conclude by suggesting future work and reviewing results.

Chapter 2

Prior Work

To motivate our study of time-multiplexed networks, we first describe prior work in networks on chip, focusing on efforts in topology selection that are relevant to time-multiplexed networks on chip. We then summarize prior work specific to time-multiplexing and the comparison of time-multiplexing to packet-switching.

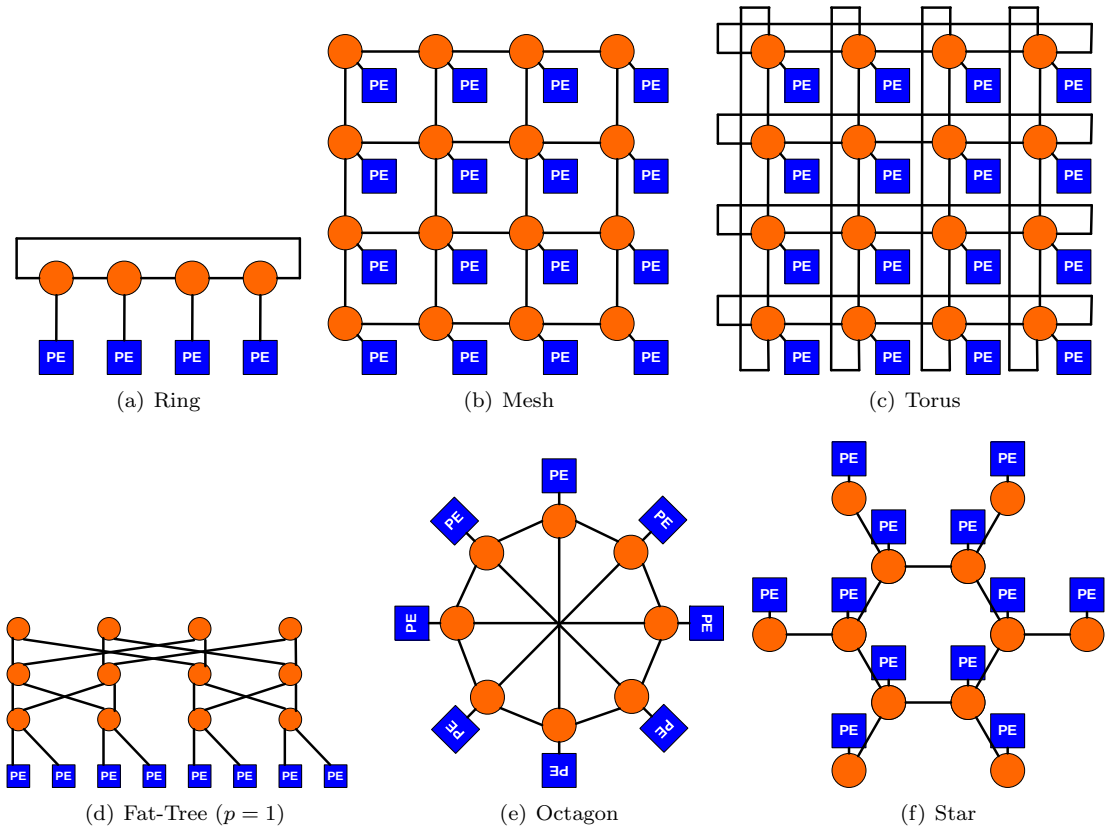
2.1 Networks on Chip

While some researchers have examined issues in designing general time-multiplexed systems, nearly all NoC projects have focused on packet-switched networks. Network topology design has been the focus of much of this research. Findings about performance and area scaling culled from topology design research are relevant across communication patterns, as we will see in Chapter 7.

Most prior work in NoC topology design has assumed a packet-switched network under an ASIC cost model. Several NoCs have been studied for various topologies such as the ring [58], mesh [31,43], torus [13], fat-tree [1, 49], octagon [29], and star [34] (see Figure 2.1). These projects typically evaluate scaling of topologies over 10s of interconnected PEs without evaluating tradeoffs with respect to other topologies. Pande *et al.* [49] do compare different topologies for larger networks. However, they do not examine how these networks scale, as they examine only fixed-size networks of 256 PEs.

Additional work has attempted to provide a design methodology and automatic synthesis tools for evaluating and generating topologies for ASIC NoCs [45,21,23] . Murali *et al.* [45] provide a tool which explores several different application-specific topologies, but they do not examine the effects of scaling NoC topologies. Rather than characterizing topological area and performance for small systems or for a single system size, this work attempts to compare and explore the scalability of different network topologies from 2–2048 PEs.

Some recent work has begun to examine NoCs under an FPGA overlay cost model [41, 44, 53, 46], as recent increases in FPGA capacity have made it possible to map NoCs to FPGAs. While



(a) Ring (b) Mesh (c) Torus

(d) Fat-Tree ($p = 1$) (e) Octagon (f) Star

Figure 2.1: Previously Studied Network on Chip Topologies

promising, this work has ignored time-multiplexing and has avoided comparing the performance of network topologies. We examine several large-scale topologies and are the first to characterize the tradeoffs between topologies for NoCs on FPGAs.

Finally, nearly all prior work has used synthetic traffic for evaluating tradeoffs in NoC design. Select papers [60] have examined real world applications mapped to NoCs without attempting to use them to characterize the design space of NoC architecture. We evaluate our networks with a variety of real world applications 5.2. These applications provide diverse communication patterns which heavily load our network and stress the performance tradeoffs between different network architectures.

2.2 Time-Multiplexed Networks

The idea of time-division multiplexing (TDM) information was invented in World War II for the purpose of encrypting radio transmissions. Bell Labs then popularized the idea by developing the first T1 voice channel system, which interleaves multiple voice signals on a single line in a pre-scheduled manner. The T1 communications standard and its successors have been in use by telephony carriers for years. Additionally, many modern wireless communications protocols use a type of TDM, called

TDMA (time-division multiple access) where many users share the same radio frequency in time. TDMA is used extensively in the Global System for Mobile Communications (GSM), satellite, and even local area network systems.

In the computing world, the distributed parallel processing community has focused much research on the time-multiplexing of communication between processing elements. Several of these projects used concepts similar to those we implement in our network, but with a slightly different focus. Instead of using switches with context memory, many projects used FSMs to coordinate communication between multiple chips (iWarp [9] and NuMesh [54]) or between PEs integrated onto a single chip (Synchroscalar [48]). Other projects used time-multiplexing at the logic-level for implementing circuits [17, 8, 26, 14, 59]. In these designs time-multiplexed context memory is built into logic and switches of the native FPGA or simulation-engine architecture.

The project with a model and focus most similar to ours is Virtual Wires [4]. Instead of examining the time-multiplexing of resources connecting processing elements in an ASIC or embedded in an FPGA, Virtual Wires examined the time-multiplexing of resources overlayed on top of an FPGA. Virtual Wires attempts to overcome pin limitations by time sharing each physical wire among logical wires and pipelining those connections at a high frequency. To schedule the logical wires on the physical wires, they developed a greedy spatial and temporal scheduler [52], demonstrating a significant increase in usable I/O bandwidth. Instead of just time-multiplexing chip I/O communication by overlaying logic on an FPGA, our work attempts to time-multiplex communication over an entire network overlayed on an FPGA. We use a similar space-time greedy scheduler as the basis of our own routing algorithm.

2.3 Time-Multiplexed vs. Packet-Switched Networks

Some researchers have begun to explore the tradeoff between static scheduling and dynamic routing. The RAW microprocessor [61] utilized a static scheduler to compile streams of computations across PEs, but also supported dynamic hardware to allow for unresolvable events such as memory stalls and interrupts. The designers discovered that a combination of static scheduling techniques and software routines handled dynamic events more efficiently than dedicated dynamic hardware. The tradeoffs between static scheduling and dynamic routing were not quantified beyond this observation. The aSoC architecture [38] took this comparison a step further. aSoC utilized a statically scheduled communications processor, complete with context memory, to network together PEs and compared it to dynamic routing models. They found that static scheduling moderately increased performance at the cost of increased area in context memory, but they did not quantify this increase in area. aSoC supported only 16 PEs and a light traffic load where PE processing time generally dominated communication time. Rather than simply examining one system size or configuration,

this work attempts to broadly explore the design space tradeoffs between static scheduling and dynamic routing, and quantifies when either time-multiplexing or packet-switching is preferred under various applications conditions. Our exploration ranges from 2–2048 PEs and includes traffic loads which can often dominate node processing times.

Chapter 3

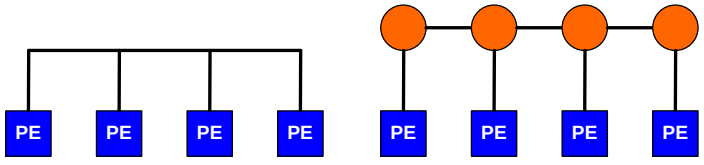
Time-Multiplexed Network Design

To design a physical time-multiplexed network, we must first select a network topology that describes how to connect PEs. Next, we must determine how to construct the individual elements of that topology in hardware. Here we discuss the topologies that we explore in Chapter 6 and their hardware design.

3.1 Network Topologies

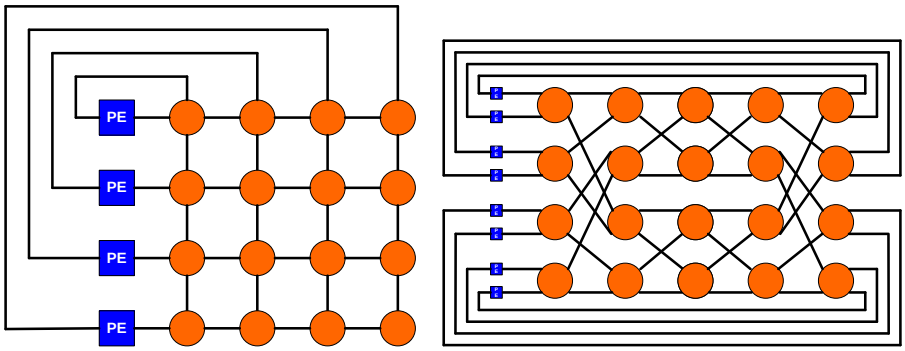
The most commonly used topology for NoCs is the shared bus [3, 25, 62] (Figure 3.1(a)). Only one device can send a message on a bus at one time; therefore, devices on a bus must arbitrate for control. Assuming no segmentation, a bus can only handle a single unique network send and receive per cycle, sent from the single PE controlling the bus. Multiple receives are possible but only for non-unique data that fans out to multiple PEs. Applications with large numbers of PEs communicating in parallel must send and receive significantly more than one message per cycle. For the applications we consider, hundreds of sends and receives per cycle are common (see Figure 5.1 for an example from our workload set). In these situations the bus will be forced to serialize communication and will become bandwidth bottlenecked.

For communication heavy applications it is necessary to avoid the bandwidth limitations of a bus and move to scalable networks. A small modification to the bus to allow increased injection of messages is the addition of pipelined segmentation, creating a ring topology (Figure 3.1(b)). We examine rings as representatives of the best possible performance that a bus can achieve.

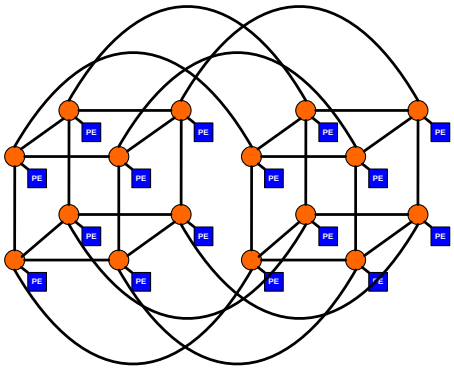


(a) Bus (b) Ring (no wraparound)

Figure 3.1: $p = 0$ Network Topologies



(a) Crossbar (b) Beneš Network



(c) Hypercube

Figure 3.2: $p = 1$ Network Topologies

One extreme alternative to the limited, serialized connectivity of the bus is the selection of a topology which supports *any* connectivity between all PEs. Common multiprocessor topologies such as crossbars, multistage networks (*e.g.* Beneš [5], Clos [11]), and hypercubes all represent networks that provide fully connectivity between PEs (Figure 3.2). Previously studied topologies for NoCs such as octagons and stars provide similar connectivity (Figure 2.1). These topologies have high bandwidth due to their large bisection and hence do not become serialization bottlenecked as quickly as a bus or ring. However, fully connected topologies are prohibitively expensive in terms of area in two-dimensional VLSI layout due to large numbers of wires crossing the bisection [57]. This high area cost is exacerbated when scaled to 1000s of PEs.

Additionally, Landman [32] observes that typical designs do not even require this kind of full connectivity. Using **Rent's rule**, one can characterize the wiring requirements (or connectivity) of a circuit as follows:

$$IO = cN^p \tag{3.1}$$

Here, N is the number of nodes in a circuit, IO is the number of input and output signals to the

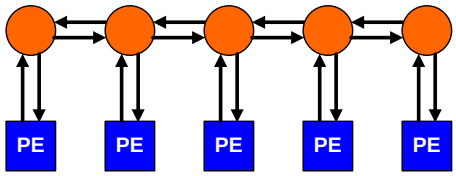


Figure 3.3: Ring (5 PEs, $w = 1$)

circuit, and c and p are parameters that are design specific. The Rent parameters c and $0 < p < 1$ describe the level of connectivity in a circuit, where c is a constant factor and p is an asymptotic factor. The intuition behind this equation is that for a given block of N nodes, the number of wires entering and exiting the block can be no less than a constant ($IO = c$, where $p = 0$), and no greater than linear in N ($IO = cN$, where $p = 1$). Rent's rule can be applied recursively to blocks within a design, where c and p for the entire design are the values that best fit Eq. 3.1 over many blocks.

Buses and rings are $p = 0$ topologies, while fully connected topologies are $p = 1$. Landman found that most real designs are characterized by a Rent parameter of $0.5 < p < 0.75$. Therefore, most designs operate in a limited bisection region with $p < 1$, but do require more bisection than $p = 0$. Therefore, for most designs a bus does not provide enough interconnect, while fully connected topologies are area inefficient in that they provide too much interconnect. All topologies can be characterized by the tunable Rent parameter p, and we can attempt to design a network with a p that will match application requirements. We will see that over our range of applications $p = 0$ networks can quickly become bisection limited, while $p = 1$ networks avoid bandwidth bottlenecks but at a significant cost in area (Section 6.2).

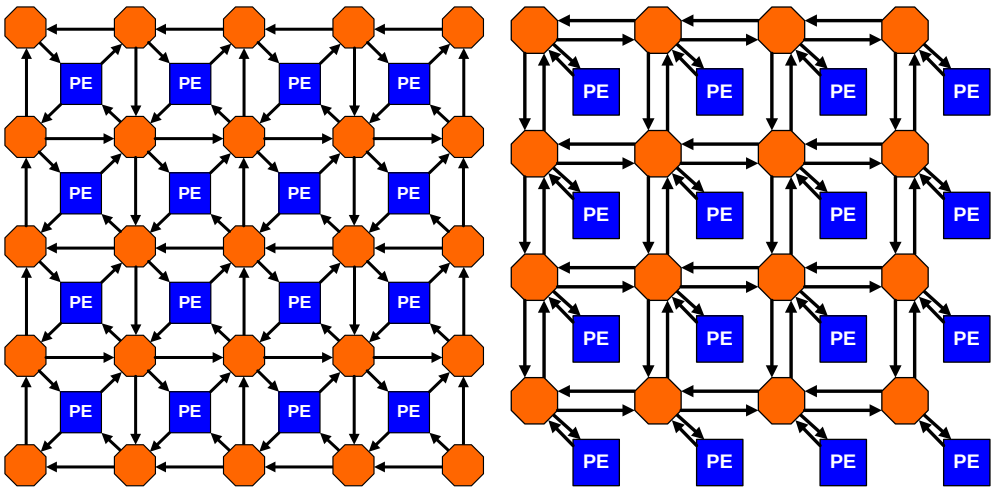
To explore the design space of limited bisection networks, we focus of exploration of network topologies to rings, meshes and Butterfly Fat Trees (BFTs) [37, 20] over a range of configurations.

3.1.1 Ring

Figure 3.3 shows the bidirectional ring topology we examine. Rings can be parameterized by their channel width w, equivalent to the Rent parameter c, which specifies the number of rows of switches in the ring. Each channel is independent and can be thought of as a separate, identical rings of switches, each attached to the row of PEs. We chose not to build wraparound wires (unlike the ring shown in Figure 2.1(a)) as to increase layout simplicity. We examine rings with $w = 1, 2$. Since $p = 0$ for the ring, we consider the ring as a representative of how well one can do with a small amount of interconnect that can be varied by the constant factor w.

3.1.2 Mesh

Figure 3.4 shows the two types of mesh topologies we examine: directional and bidirectional. Like rings, meshes can be parameterized by their channel width w, equivalent to the Rent parameter c,



(a) Directional (b) Bidirectional

Figure 3.4: Mesh (4×4, $w = 1$)

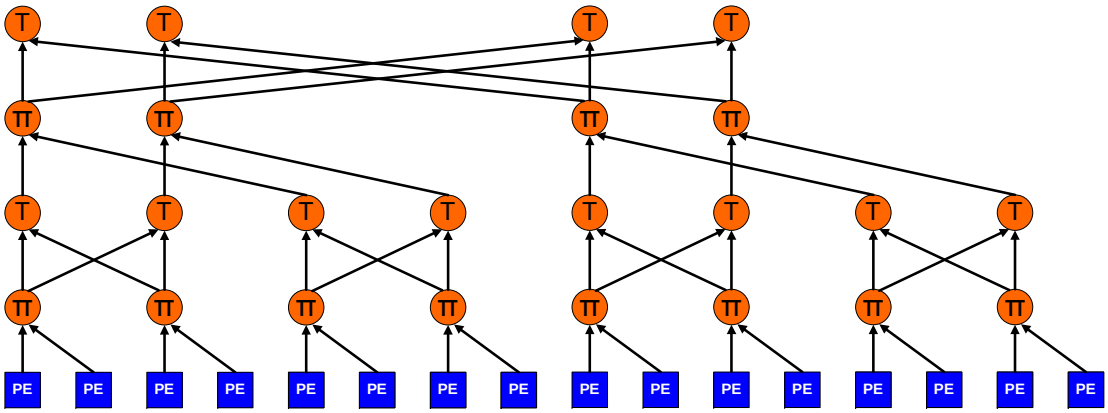
which specifies the number of rows of switches in each horizontal and vertical channel. Each channel is an independent mesh that can be thought of as a separate plane of switches, each attached to the array of PEs. We will examine meshes with $w = 1, 2$. We note that $p = 0.5$ for the mesh, representing a moderate amount of interconnect that can be varied by the constant factor w.

We choose to tune channel width instead of dimensionality for meshes. Previously, higher dimensional meshes such as as hypercubes (Figure 3.2(c)), or more generally k-ary n-cubes, were shown to be less efficient than two-dimensional meshes due to pin constraints [12]. As NoCs are not I/O limited like multiprocessor networks, it may be worth revisiting higher dimensional mesh topologies in the future under an appropriately revised cost model.

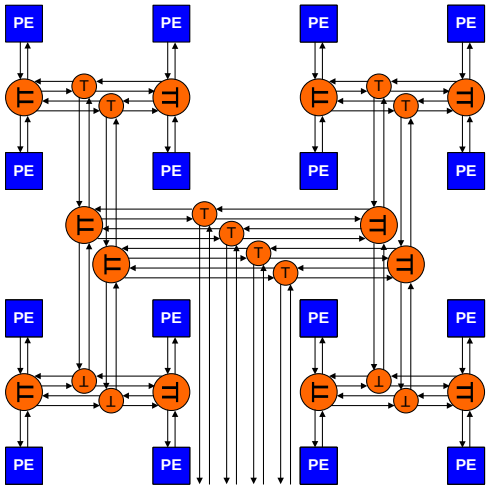
3.1.3 Butterfly Fat Tree

Figure 3.5 shows the Butterfly Fat Tree (BFT) [37,20], from both logical and layout perspectives. The BFT resembles a binary tree with two modifications at higher levels of the tree. First, connections can cross the tree before reaching the root, allowing routes to exploit locality by turning across the tree at lower levels. Second, connections can increase in number (*i.e.* become "fatter") at higher levels, increasing total network bandwidth.

BFTs can be parameterized around the base channel width c and wire growth rate p, allowing a single topology to be fully tuned in terms of its Rent parameters. Like w for the ring and mesh, c indicates the number of parallel, non-interacting planes of switches. The Rent parameter p specifically denotes which of the two types of BFT switches are used at each level of the tree: the Π or T switch. The Π switch is a bandwidth preserving switch, where two inputs enter from the bottom of the switch and two outputs exit from the top. The T switch is a bandwidth reducing switch, where two inputs enter from the bottom and only one output exits from the top. A $p = 0$



(a) Logical View



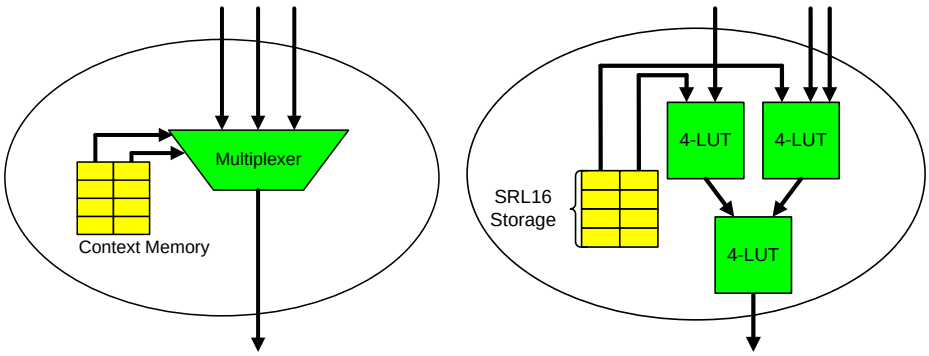
(b) Layout View

Figure 3.5: BFT (16 PEs, $c = 1, p = 0.5$)

BFT indicates no growth of wires, or all T switches. Recall that $p = 0$ in Rent's rule indicates $IO = c$, or that when dividing a BFT recursively into blocks, each block has a constant number of links entering and exiting the block. A $p = 1$ BFT is entirely made of Π switches, such that in each recursive block the number of links entering and exiting that block is linear in the number of elements in the block ($IO = cN$). Figure 3.5 shows a $p = 0.5$ tree, where Π and T switches alternate at each level. We examine BFTs with $c = 1, 2$ and $p = 0, 0.5, 0.67, 1$.

3.2 Hardware Design

We now examine the hardware design of the switches, input/output blocks, and pipelined interconnect of our network (processing element design is discussed in Section 5.5). In designing the hardware for our network, we have four main goals: simplicity, scalability, high performance, and area efficiency. In order to make the network as simple as possible, we construct all switches and I/O blocks out of multiple instances of a single, parameterizable element: the time-multiplexed



(a) General Block Diagram (b) FPGA Block Diagram

Figure 3.6: Time-Multiplexed Merge Primitive (3-input)

merge. The simplicity of the merge block allows it to both run at a high frequency and consume a small amount of logic area. Additionally, by decomposing all switches into simple merge blocks, we can easily pipeline our switches heavily and thus run the entire network at a high frequency. This technique scales well to very large network sizes.

While we present switch and I/O block designs as general circuits that can be constructed on any substrate, the specific area and performance model we show data for is based on the Xilinx XC2V6000-4 FPGA. All area (slices), latency (cycles), and frequency (MHz) numbers presented in this section are based on synthesizing, placing and routing designs on a Xilinx XC2V6000-4 (see Section 5.6 for more details).

3.2.1 Hardware Parameters

Each hardware element in our network is parameterizable in terms of the bit width of data. If the number of bits in a message exceeds the datawidth of network links and switches, messages can be broken up into pieces or "flits" to be routed separately. This provides a way for PEs to send arbitrarily long messages over a network with a fixed datawidth. For the applications we consider, all messages sent within a single application are of equal size (16 or 64 bits). We choose to synthesize networks with datawidths equal to the size of a single message, routing all messages as single flits. Routing multiple flits would decrease the size of the switch logic, as the logic could be sized to process shorter data. However, routing multiple flits increases the total number of communication cycles due to the increased number of flits that must be routed. This in turn increases the total number of bits required for context memory, which dominates switch area. For the small message applications that we focus on here, multiple flit messages are inefficient.

Primitive	Area (Slices)			Latency	Speed
	Logic	Context	Total	(Cycles)	(MHz)
Merge2	8	32	40	1	219
Merge3	24	64	88	1	204
Merge4	24	64	88	1	204

Table 3.1: Time-Multiplexed Merge Primitive (16-bit, 1024-deep)

3.2.2 Merge Primitive

The time-multiplexed network is composed solely from the time-multiplexed merge primitive. A merge has n inputs and a single output, where each input and output can be multiple bits wide. Figure 3.6 shows a sample 3-input merge primitive, implemented both at the block diagram and FPGA circuit level. Merges are simply constructed out of n-input multiplexers controlled by context memory. Context memory controls which merge input is routed to the output on that cycle, corresponding to routing decisions computed offline. Because of this simple design, the latency to route through a merge is a few LUT delays. Since merges need to store routing decisions for every cycle, for large cycle counts context memory can dominate the area of the merge primitive, and in turn the entire network. The following equations describe the area in slices required for time-multiplexed merge primitives as a function of datawidth and context depth, where $n = 2, 3, 4$:

$$Merge2_{area} = \frac{ContextDepth}{32} + \frac{DataWidth}{2} \tag{3.2}$$

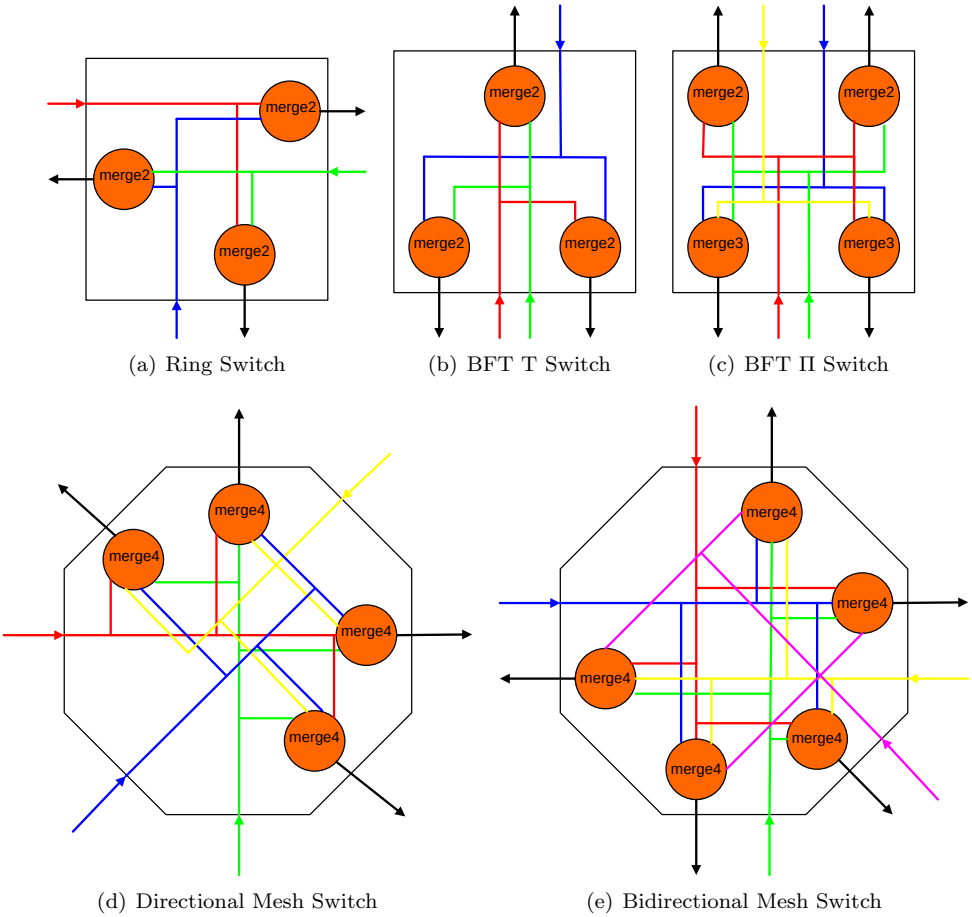
$$Merge3_{area} = 2 \times \frac{ContextDepth}{32} + 3 \times \frac{DataWidth}{2} \tag{3.3}$$

$$Merge4_{area} = 2 \times \frac{ContextDepth}{32} + 3 \times \frac{DataWidth}{2} \tag{3.4}$$

Context memories are implemented with SRL16s [63] for compact storage, where each slice holds a single bit of context memory that is 32 cycles deep. We see that a Merge2 requires 1 bit of context (corresponding to $\frac{ContextDepth}{32}$ slices) to control a single two-input multiplexer (implemented in $\frac{DataWidth}{2}$ slices). The Merge3 and Merge4 however require 2 bits of context to control three-input and four-input multiplexers. Table 3.1 shows the area, latency, and speed of each synthesized merge with a datawidth of 16-bits and with 1024 cycles of context. We further discuss the area model for context memory and techniques to reduce context area in Chapter 8.

3.2.3 Switches

Figure 3.7 shows how we construct each switch for each topology solely out of merge primitives. Table 3.2 shows the area, latency, and speed of each switch for a 16-bit network with 1024 cycles of context. When calculating the area of switches used in the actual network, we ignore the area required for merges connected to outputs that are unused. This is applicable to switches on the edge



(a) Ring Switch (b) BFT T Switch (c) BFT Π Switch

(d) Directional Mesh Switch (e) Bidirectional Mesh Switch

Figure 3.7: Time-Multiplexed Switches

Switch	**Area** (Slices)			**Latency**	**Speed**
	Logic	**Context**	**Total**	(Cycles)	(MHz)
Ring	24	96	120	1	219
Directional Mesh	96	256	352	1	204
Bidirectional Mesh	120	320	440	1	204
BFT Π	64	192	256	1	204
BFT T	24	96	120	1	219

Table 3.2: Time-Multiplexed Switches (16-bit, 1024-deep)

of a mesh, the end of a ring, and the top of a BFT.

3.2.4 I/O Blocks

In order for messages to exit the PE and enter the network, and for messages to exit the network and enter the PE, we construct network I/O blocks to allow the PE to interface with the network. The input block consists of a single multiple input merge, where several incoming network links are able to connect to the PE. The output block simply allows the PE to fanout its output messages to each outgoing network link. While this may cause the network to route spurious messages, each

PE contains a single bit of context memory to indicate when a valid, scheduled message arrives (See Section 5.5).

The area, latency, and speed of an input block is identical to the merge that implements it (Eqs. 3.2, 3.3, 3.4 and Table 3.1). Output blocks require no logic or area. As previously mentioned, for increased efficiency we set the datawidth of all elements in our network and the datawidth of the PE to be the same to allow the routing of single flit messages. This avoids the need to deserialize and serialize data in incoming and outgoing messages respectively, in an attempt to match the PE datawidth to the network datawidth.

3.2.5 Pipelined Interconnect

One important consideration in simulating our network is how to model the latency and speed of switches for very large networks (*i.e.* 1000s of PEs). Large networks require large chips, and we are not guaranteed that the entire network will be able to place and route at > 200 MHz on an arbitrarily large chip. Because of the simplicity of the logic in each merge block, we assume that a time-multiplexed merge can place and route at > 200 MHz, while maintaining a single cycle of latency on an arbitrarily large chip. However, wires that connect switches together may be very long and must be pipelined for all topologies to support high frequency operation.

As the wire distance to cross a chip (corner to corner) is identical for all topologies, we must ensure that for each topology the number of cycles required to cross a chip via only wires is the same. To calculate the required number of registers to pipeline all wires in our network for > 200 MHz operation, we perform the following calculations.

Maximum Length Unpipelined Wire We first compute the maximum length of an unpipelined wire for 200 MHz operation. We observe that when placing two primitives on opposite ends of of a Xilinx XC2V6000-4 (edge to edge), to achieve 200 MHz performance we must pipeline the wires connecting the two primitives with 3 registers each. As the XC2V6000-4 contains 32K slices (*i.e.* 180×180), this corresponds to adding one pipeline register per 60 slices. Therefore, we assume that the maximum length of an unpipelined wire is ≈ 60 slices.

Wires in our system connect PEs to switches and switches to switches. Switches are small enough such that internal wires do not need to be pipelined. PE to switch connections are typically very short and only require 1 cycle of pipelining. Switch to switch connections comprise the longest wires in the system and we must calculate the length and pipelining for each of these wires. We observe that our largest PE (SMVM) is 4000 slices, or roughly 64×64 slices in area. This PE is significantly larger than switches in our system (Table 3.2) and will dominate total area. Therefore, in the worst case we can approximate crossing a N PE system laid out in 2D as traveling the distance of $2 \times \sqrt{N}$ PE length wires. As the PE is 64 slices long, a wire crossing the length of this PE requires at least 1

pipeline register; but because routes in FPGA may take slower wires we assume 2 pipeline registers.

Mesh Latency For an N PE bidirectional mesh we can see that a worst case route (corner to corner) must cross the distance of $2 \times \left(\sqrt{N-1}\right)$ PEs. Given that the number of cycles to cross a PE is 2, we express the mesh wire pipelining requirement as follows:

$$\begin{aligned} Mesh_{wire\_cycles} &= 2 \times \left(\sqrt{N}-1\right) \times (t_{wire}) \\ &= 2 \times \left(\sqrt{N-1}\right) \times (2) \\ &= 4\sqrt{N-1} \\ &\approx 4\sqrt{N} \end{aligned} \tag{3.5}$$

In the worst case the total number of cycles to traverse the mesh is the sum of the number of cycles to traverse both switches and wires:

$$\begin{aligned} Mesh_{switch\_cycles} &= \left(2 \times \sqrt{N}-1\right) \times (t_{switch}) \\ &= \left(2 \times \sqrt{N}-1\right) \times (1) \\ &= 2\sqrt{N}-2 \\ &\approx 2\sqrt{N} \end{aligned} \tag{3.6}$$

$$\begin{aligned} Mesh_{latency} &= Mesh_{wire\_cycles} + Mesh_{switch\_cycles} \\ &= \left(4\sqrt{N}\right) + \left(2\sqrt{N}\right) \end{aligned} \tag{3.7}$$

We make similar latency assumptions for the ring topology.

BFT Latency We observe that for the BFT the length of switch to switch wires doubles after every two stages of the tree (Figure 3.5(b)). We can express the general BFT wire pipelining requirement as follows, where h is the height of a wire in the tree:

$$BFT_{wire\_cycles} = 2 \times \sum_{h=0}^{\log(N)-1} t_{wire_h} \tag{3.8}$$

We must select a sequence of t_{wire_h} such that $BFT_{wire\_cycles} = Mesh_{wire\_cycles}$:

$$\begin{aligned} BFT_{wire\_cycles} &= 2 \times \sum_{h=0}^{\log(N)-1} 2^{\lceil \frac{h+1}{2} \rceil} \\ &= 2 \times \left(1 + 1 + 2 + 2 + 4 \cdots \frac{\sqrt{N}}{2} + \frac{\sqrt{N}}{2}\right) \\ &= 4 \times \left(1 + 2 + 4 \cdots \frac{\sqrt{N}}{2}\right) \\ &= 4\sqrt{N} \end{aligned} \tag{3.9}$$

We can further express the total latency required to traverse both switches and wires as:

$$\begin{aligned} BFT_{switch\_cycles} &= 2 \times \log N \times t_{switch} \\ &= 2 \log N \end{aligned} \tag{3.10}$$

$$\begin{aligned} BFT_{latency} &= BFT_{wire\_cycles} + BFT_{switch\_cycles} \\ &= \left(4\sqrt{N}\right) + (2 \log N) \end{aligned} \tag{3.11}$$

We see that wire pipeline cycles for the mesh and BFT are asymptotically the same ($4\sqrt{N}$). However, the switch latency of a BFT is asymptotically lower than that of a mesh ($2 \log N < 2\sqrt{N}$).

Chapter 4

Time-Multiplexed Router Design

In order for PEs to communicate on a time-multiplexed network, an offline router must schedule all communication prior to runtime. Here we discuss the nature of the routing problem, our greedy routing algorithm, and how to measure our algorithm's efficiency.

4.1 The Routing Problem

The informal goal of time-multiplexed routing for NoCs is: given a static communication load, find a schedule that allows messages to travel from source to sink without using the same resources at the same time (*i.e.* congesting resources). Two types of messages must be routed: self messages and external messages (See Chapter 5 for more details on our application message model). Self messages (*i.e.* messages where the source and destination of the message is the same PE) do not need to traverse the network, but must access local PE resources as necessary. External messages must also access local PE resources, and then must exit the source PE, access network resources by traversing network links and switches, and enter the sink PE. In addition to correctness (*i.e.* all messages routed), several optimization targets can be defined, such as minimizing total communication cycles, router runtime, and energy. Our router will focus on minimizing total communication cycles.

4.1.1 Multicommodity Flow

Time-multiplexed routing is a specific instance of generalized multicommodity flow [35,24], typically used to formulate the problem of spatial routing. Time-multiplexing introduces the additional constraint that messages must also be routed in time; we state this version of multicommodity flow as follows. Directed resource graph $G = (V, E)$ represents the resources in the network and their interconnectivity. In this graph nodes correspond to PEs and network link segments, while edges correspond to the allowable connections between pairs of nodes via switchboxes and input/output blocks. Each edge $e \in E$ has an associated nonnegative capacity in time denoted by $c(e, t)$, which represents the maximum amount of flow that can pass through that edge on cycle t. For time-

multiplexed routing $\forall e, c(e,t) = 1$. Additionally, each resource r has an associated delay c_r, which represents the latency through that resource (Table 3.2).

Additionally, there is a set of $k > 1$ triplets (s_i, t_i, d_i), termed commodities, corresponding to messages that need to be routed. For each commodity i the variables s_i, t_i represent source and sink nodes in the resource graph, while d_i represents a positive demand. For time-multiplexed routing $\forall i, d_i = 1$. The general objective is to route all commodities in time while satisfying demand, obeying capacities, and obeying latencies experienced when traveling through a resource. Formally, we can express the solution as a set of flows through all edges in time, where $f_i(e,t)$ represents the amount of commodity i passing through edge e on cycle t. This solution set corresponds to message assignments for each resource. The set of flows must obey the following constraints:

1. Flow is conserved: $\forall v$, incoming flow at time t = outgoing flow at time $t + c_r$, $\forall i$ and $\forall t$.
2. Capacities are obeyed: $\forall e, \sum_i f_i(e,t) \leq c(e,t)$.
3. Demands are satisfied: $\forall i$, d_i units of commodity i must flow from s_i to t_i.
4. Latencies are obeyed: When a messages enters resource r on cycle t, it must exit that resource on cycle $t + c_r$.

Time-multiplexed networks require the additional constraint that all flows are integral (specifically = 1), as a message flit is an atomic unit. While general multicommodity flow can be formulated as a linear program (LP), integer multicommodity flow is a mixed integer program (MIP), and is therefore NP-complete [19].

The optimization target for our time-multiplexed router is to minimize total communication time. Therefore, our optimization problem is to satisfy the above constraints of time-multiplexed routing while minimizing the the total number of cycles t.

4.1.2 Single-Source Shortest Path

A critical subproblem to be solved in conventional heuristics for time-multiplexed routing is the routing of a single message. We formulate this problem as finding the minimum weight path from a source to a sink: Given a weighted resource graph $G = (V, E)$, and source and sink nodes s, t, find the minimum weight path connecting s and t. Each resource has a weight in time that represents the state of that resource in a given cycle. The weight of resource r at cycle t can be a function of values such as the occupancy of a resource (*i.e.* if a resource is occupied it should cost more to use), the cycle t (*i.e.* routes should be encouraged to use resources earlier in time), etc. The exact weight function depends on the router algorithm used as we will discuss in the following section.

4.2 Greedy Algorithm

One common heuristic algorithm for solving multicommodity flow, typically used in FPGA spatial routing, is Pathfinder [42]. Pathfinder, based on Lee's maze routing algorithm [33], attempts to route each commodity one by one while allowing flow through a resource to exceed its capacity. Routes that violate capacity constraints are later ripped up and re-routed until no routes exceed resource capacities. Each route is found with a shortest path algorithm where resources are weighted based on congestion and the history of congestion. Congestion is simply an integer representing the number of routes occupying a given resource (*i.e.* the amount of flow through a resource). History can be expressed in several ways, but most simply it is a running sum of congestion over the course of the algorithm. Therefore, for Pathfinder the representation of the weight of resource r in cycle t is as follows:

$$\text{weight}(r,t) = 1 + \alpha \times (\text{occupany}(r,t) \times \text{history}(r,t)) \tag{4.1}$$

Where α is some constant. Pathfinder has been shown to produce high quality routes in a moderately short amount of time. However, the amount of state needed to represent every resource in the graph is very large, and can significantly increase algorithm memory usage and runtime. Additionally, Pathfinder may require a significant amount of ripping up and re-routing of routes, further increasing runtime. Finally, Pathfinder is not guaranteed to converge. We found that for purposes, under certain conditions a simple greedy router performed close to the optimal number of total communication cycles while routing significantly faster than Pathfinder. In Section 4.4 we we evaluate the quality and runtime of our router and provide examples that indicate that for most cases, the extra quality obtained by Pathfinder may not be worth the additional runtime and resource cost. In cases where that extra quality is desired, our greedy router complements a Pathfinder approach by providing an achievable lowerbound on performance.

Our solution to the routing problem is a greedy algorithm, similar to that of [52]. Instead of allowing routes to occupy congested resources, we avoid congested resources and therefore avoid ripping up and re-routing routes. Therefore, the weight of each resource is simply:

$$\text{weight}_{r,t} = 1 \tag{4.2}$$

Pseudocode for our algorithm is shown in Figure 4.2. The list of messages to route is initially sorted randomly, as we found that there is no discernible benefit to routing longer routes first or last. Each message is routed one at a time with a fast A* shortest path algorithm (*e.g.* [56]) shown in Figure 4.3. A* works by spreading along the resource graph in time and by placing candidate network links in time to examine on a priority queue. Initially, output links from the source are

Algorithm 4.2.1: GREEDYROUTE($message\_list$)

sort $message\_list$ (randomly)

$total\_cycles \leftarrow 0$
for each $message \in message\_list$
  $path \leftarrow$ FINDSHORTESTPATH($message.source, message.sink, 0, total\_cycles + 1$)
  $total\_cycles \leftarrow \max(path.cycles, total\_cycles)$
return ($total\_cycles$)

Figure 4.1: Time-Multiplexed Greedy Routing Algorithm

placed on the priority queue. We ensure that these links are not occupied and that the resources (*e.g.* memory) of the source are not occupied, ensuring sure that the source is ready to send a message. The algorithm then enters a loop where candidate links in time are examined one by one by popping the top link off the priority queue. If one of those links in time terminates in a node that is the sink, and the sink is not busy, then the algorithm is done. Otherwise, we examine the non-sink node and place its output links in time on the priority queue and continue. We ensure that these output links are not occupied and that they occur at a time = $input\_link.time + node.latency$ in order to account for switch latencies.

The advantage of A* is the method in which it sorts links in time in the priority queue, shown in Figure 4.4. The queue is sorted by three variables: *weight*, *guess*, and *time*. *weight* is equivalent to $weight_{r,t} = 1$ and represents the number of links in time traversed thus far to reach the current link in time, *i.e.* the length of the path. *guess* is a heuristic that A* uses to speed up search time. *guess* is a guess as to how many links are left until the sink. This is calculated on a topology specific basis. Together, $weight + guess$ represent the estimated total length of a given path. Links in time that minimize this value are pushed to the top of the priority queue and thus examined first. If $weight + guess$ are equal for two links in time, then the link that occurs earlier in time is examined first. The end result of this sorting technique is that each message is routed greedily on the shortest available path in space and earliest possible slot in time.

Self messages do not require an A* shortest path search. These messages are simply scheduled on the PE in the earliest possible slot in time when resources (*i.e.* memory) are available.

4.2.1 Communication Constraints

For some applications, messages may have constraints on when they can be routed. That is, before it is possible to send out a particular message, another message may need to be received first. To understand when these constraints occur for our applications, see Section 5.4.

To deal with these constraints, we make the following simple change to our router. When iterating through the list of routes, we only route those messages whose constraints are satisfied.

Algorithm 4.2.2: FINDSHORTESTPATH($source, sink, start\_time, end\_time$)

```
// start search with source's links over time
for time ← start_time to end_time
   for each link_in_time ∈ source.get_outputs(time)
      if !link_in_time.occupied() and !source.occupied(time)
         link_in_time.weight ← 1
         priority_queue.push(link_in_time)

// examine links on priority queue
while !priority_queue.empty()
   link_in_time ← priority_queue.pop()
   time ← link_in_time.time
   weight ← link_in_time.weight
   node ← link_in_time.sink

   // reached sink
   if node = sink and !sink.occupied(time)
     return (link_in_time.path())

   // keep searching
   for each link_in_time ∈ node.get_outputs(time)
      if !link_in_time.occupied()
         link_in_time.weight ← weight + 1
         priority_queue.push(link_in_time)
```

Figure 4.2: A* Shortest Path Algorithm

Algorithm 4.2.3: PRIORITYQUEUECOMPARE($link\_in\_time\_a, link\_in\_time\_b$)

```
// sort by smallest (distance traveled + distance left)
if (link_in_time_a.weight + link_in_time_a.guess) <
   (link_in_time_b.weight + link_in_time_b.guess)
  return (−1)
 else if (link_in_time_a.weight + link_in_time_a.guess) >
         (link_in_time_b.weight + link_in_time_b.guess)
  return (1)

// then sort by earliest in time
 else if link_in_time_a.time < link_in_time_b.time
  return (−1)
 else if link_in_time_a.time > link_in_time_b.time
  return (1)
else
  return (0)
```

Figure 4.3: A* Priority Queue Sorting Algorithm

Topology Size (PEs)	Runtime (min:sec)
2	16:31
4	7:20
8	3:41
16	2:52
32	1:33
64	0:59
128	0:48
256	0:48
512	1:23
1024	2:23
2048	5:02

Table 4.1: Time-Multiplexed Router Runtime (BFT $c = 1, p = 0.5$ for `fidap035`)

When encountering a message whose constraints are not satisfied, we place that constrained message at the end of the route list to be routed later. Once the PE receives the incoming messages that the constrained message depends on, the router is able to route that message. Therefore in Figure 4.2, instead of setting *start_time* to 0, we set it to the time at which the last dependent message is received. For our applications we ensure that there are no cyclic dependencies, guaranteeing that all constraints for all routes will eventually be satisfied. These constraints limit the flexibility of the router and may cause the router to make non-optimal decisions because of its greedy nature. In Section 5.4.1 we go into further detail on the impact of these constraints on router quality.

4.2.2 Runtime

The primary optimization target of our router is to minimize communication time, not algorithm runtime. To ease router development we implement our router in Java (see Section 5.6 for more infrastructure details). Router runtime varies greatly depending on the size of the application graph, the size of the network topology, and machine speed and load. Runtimes for our largest application graph (SMVM fidap035, see Section 5.2.2) on a representative BFT with $c = 1, p = 0.5$ are shown in Table 4.1. All runs were performed on an Intel Xeon 3.6 GHz processor with 4 GB of RAM using Sun Java v1.5.0_06.

4.3 Lowerbound Calculation

Now that we have an algorithm for routing messages on our time-multiplexed network, we can attempt to measure its quality by comparing the number of cycles it takes to route to the lowerbound number of cycles based on physical topology constraints. We identify several quantitative network characteristics which bound the number of cycles required for communication:

4.3.1 PE Serialization

PE Serialization refers to the number of cycles required for a PE to process all messages. We engineer the datapath of our PEs to handle one external input and output message in one cycle each. For more details on our PE implementation see Section 5.5. A self message requires two cycles as it can be thought of as both an input and output message for the same PE. We can bound the number of cycles spent on incoming and outgoing message processing as follows:

$$T_{input} = N_{input} + N_{self} \tag{4.3}$$

$$T_{output} = N_{output} + N_{self} \tag{4.4}$$

While the datapath can handle single input and output messages simultaneously, memory bandwidth limitations may restrict the PE may to handle either an input or output message per cycle, depending on message memory access requirements. We can define the cumulative serialization bound as either one of the following equations, based on memory limitations:

$$\begin{aligned} T_{serialization} &= \max\left(N_{input}, N_{output}\right) \\ &\text{or} \\ T_{serialization} &= N_{input} + N_{output} \end{aligned} \tag{4.5}$$

For the applications we consider, input and output messages can be processed simultaneously, so the equation $T_{serialization} = \max\left(N_{input}, N_{output}\right)$ applies.

Serialization is a function of the number of PEs in a network. For a fixed communication load (*i.e.* fixed number of messages), more PEs translates to less message processing per PE, and a lower bound on serialization cycles. Topologies which fit more PEs in a given area will have the lowest values of $T_{serialization}$.

4.3.2 Network Bisection

Network bisection refers to the available bandwidth of a network, which dictates how many messages the network can support at a single time. When dividing a network into two halves, the number of links crossing between the two halves is called the bisection, or top cut. This indicates the maximum number of message flits that can cross the network in a single cycle. If the number of message flits is greater than the top cut, then communication is bottlenecked and must be serialized across the bisection:

$$T_{cut} = \left\lceil \frac{N_{flits}}{N_{topcut}} \right\rceil \tag{4.6}$$

The top-level bisection may not be the largest serial bottleneck in the network. Hence, we need to recursively bisect the network and identify the most limiting of cuts (T_{cut_i}):

$$T_{bisection} = \max_{\text{all cuts } i} (T_{cut_i}) \tag{4.7}$$

Network bisection is a function of the amount of interconnect in a topology. Topologies with richer interconnect, indicated by a Rent parameter p closer to 1, will have the lowest values of $T_{bisection}$.

4.3.3 Network Latency

Network latency refers to the number of cycles needed to cross the entire network. If the network is sufficiently large, several cycles may be required for a route to traverse the network:

$$T_{latency} = \max_{\text{all routes } i} (latency_i) \tag{4.8}$$

Network latency is a function of the size of a network and the latency of each switch. Short networks with few, low latency switches on the longest path will have the lowest value of $T_{latency}$.

4.3.4 Lowerbound

Thus, the lower bound on performance of a topology is as follows:

$$T_{cycles} = \max(T_{serialization}, T_{bisection}, T_{latency}) \tag{4.9}$$

As an illustrative example, in Figure 4.4 we show how the performance of a topology can be explained in terms of its lowerbounds. Here we plot lowerbound performance and actual performance vs. number of PEs for a BFT $c = 1, p = 0.5$ for the `gemat11` graph (see Section 5.2.2). Initially the performance of the BFT is dominated by input and output serialization until 64 PEs. At low numbers of PEs most cycles are dedicated towards serialized processing at the PEs. As we increase the number of PEs the number of messages in the network increases (see Figure 5.1 for an example). Since this is a limited bisection BFT, performance is subsequently limited by bisection until 1024 PEs. When the size of the network increases beyond 1024 PEs, communication becomes latency dominated.

4.4 Quality

To measure the quality of our router, for each experiment performed we simply compared the number of communication cycles required to route to the number of lowerbound communication cycles. We

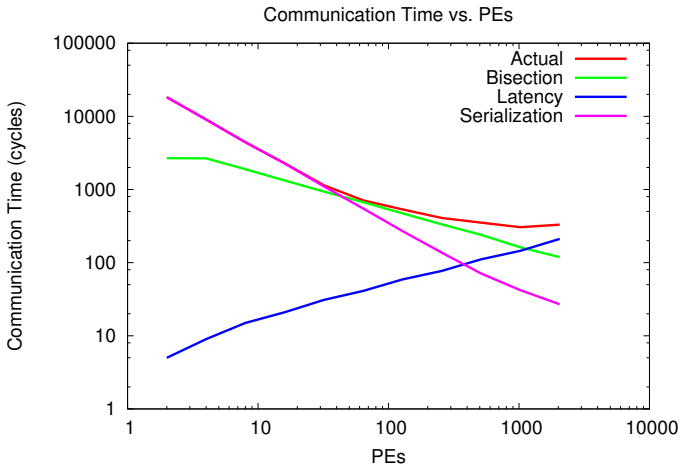


Figure 4.4: Communication Time vs. PEs (BFT $c = 1, p = 0.5$ for `gemat11`)

present quality results for a representative graph in our application set, SMVM `gemat11` (see Section 5.2.1). Figure 4.5 plots the ratio of actual to lowerbound cycles as a function of PEs over a mix of the best performing topologies, while Figure 4.6 plots the absolute difference in cycles.

In Figure 4.5 we see that our router routes in under 3.6× the number of lowerbound cycles. For topologies other than the directional mesh, our router achieves within 2× the number of lowerbound cycles. We note that these results are consistent across all graphs for all applications. There are several possibilities for the gap in route quality between the directional mesh and other topologies. One possibility is that the directional mesh provides far less routing flexibility than the bidirectional mesh, meaning that it may be easier for the greedy router to make non-optimal routing decisions. This would cause the quality ratio to increase.

Another possibility is that mesh lowerbounds in general are not asymptotically tight, meaning that their lowerbounds are too low. Routes in a mesh are free to take many different non-optimal paths to the destination. Taking a path that is not the shortest in distance can increase the amount of traffic passing through a cut beyond what is calculated as a lowerbound. This would cause ratio of actual cycles to lowerbound cycles to increase. Regardless, we are unsure of the exact reason for this gap, and will look to examine this further in future work.

While the quality ratio may be upwards of 2× for most topologies, Figure 4.6 shows that the absolute difference in cycles is minimal for most topologies. We see that the absolute difference in cycles is large for the ring and directional mesh. The difference in cycles for the ring can be attributed to the fact that the ring in general requires a very large number of cycles to route because it is heavily bisection and latency limited. The quality difference for the the directional mesh can be attributed to the reasons mentioned above. For all other topologies, we see that the absolute difference in cycles is < 150. We again note that these results are consistent across all graphs for all applications.

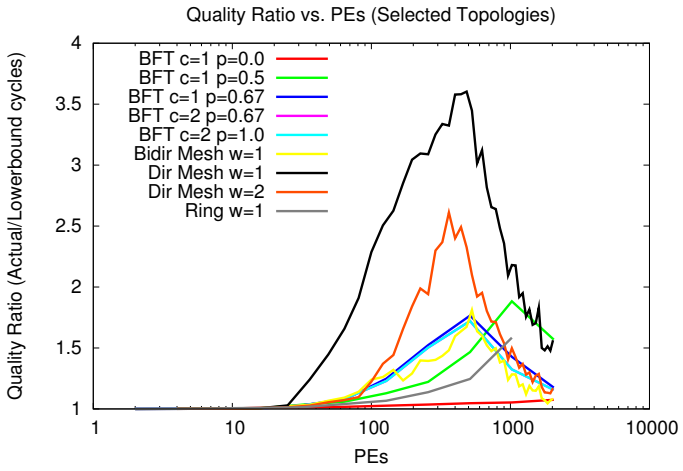


Figure 4.5: Quality Ratio vs. PEs (`gemat11`)

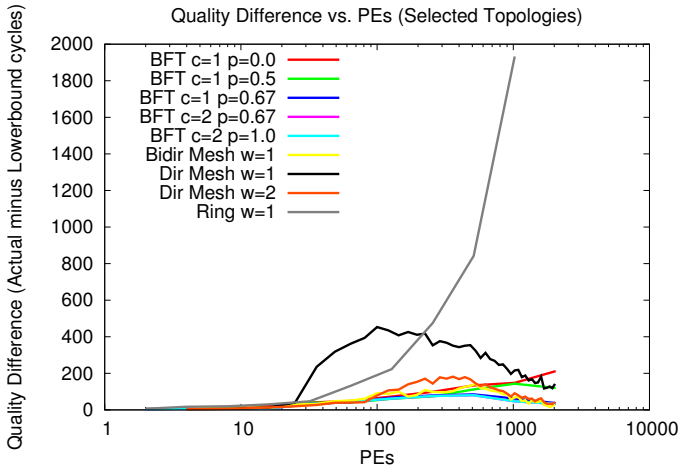


Figure 4.6: Quality Difference vs. PEs (`gemat11`)

For most topologies we see that our router provides adequate quality in reasonable runtimes given the size and complexity of our topologies and workloads. Later we will see that BFTs and directional meshes provide the best performance for time-multiplexed networks (Chapter 6). For BFTs our router provides good results when considering both quality ratio and quality difference. For directional meshes, we cannot conclude yet as to whether or not our router lacks quality or if the lowerbounds for the directional mesh are too low. We plan to explore comparing our router to Pathfinder to help quantify the tradeoffs in quality and runtime. Our greedy router will aid in this exploration as it provides and achievable lowerbound for Pathfinder.

Chapter 5

Communication Workloads

Nearly all prior work on NoC design has used synthetic or lightly loaded traffic to evaluate networks. We believe that it is of critical importance to benchmark NoCs with heavy communication workloads generated from real applications. Here we discuss the details behind the three applications we examine and how we map them to our time-multiplexed networks.

5.1 The GraphStep System Architecture

We examine communication workloads for applications mapped to the GraphStep system architecture [16]. The GraphStep system architecture is a high-level model for capturing graph algorithms, abstracted from a detailed hardware implementation. GraphStep provides a representation and a discipline for designing and implementing sparse graph algorithms. In GraphStep the sparse graph is explicitly represented and computation takes place under an actor model. Each node is an actor that communicates to neighboring nodes (connected via edges) through message passing. The evaluation model is a three phased Receive-Update-Send sequence, termed a GraphStep, where nodes are barrier synchronized on the end of the Update phase. We model the communication of the Receive and Send phases on our network where graph nodes send messages to other graph nodes. We measure network performance as the total number of cycles required to route the workload of a single GraphStep.

In our particular FPGA implementation, multiple graph nodes are physically placed on each PE where they are time-multiplexed. Since messages are sent between graph nodes, messages can be sent from a given PE to the same PE (**self messages**). The PEs consist of specialized processing logic coupled with small, local, high bandwidth on-chip memories (*e.g.* Xilinx BlockRAMs [63]). PEs are heavily pipelined and are therefore capable of sending and receiving messages in a single cycle. This allows applications to potentially send and receive messages on every cycle from every PE in the network. See Figure 5.5 and Section 5.5 for more details.

GraphStep allows us to examine applications that inject large amounts of traffic into the net-

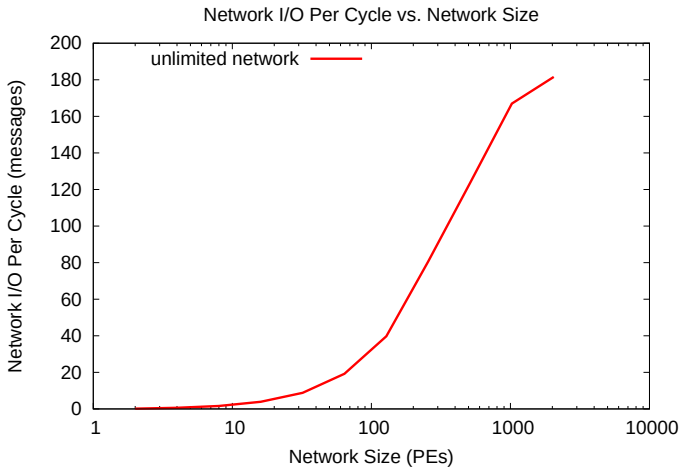


Figure 5.1: Network I/O per Cycle vs. Network Size (`small`)

work. To illustrate the importance of examining applications with high communication requirements mapped to large-scale networks, consider Figure 5.1. Here we plot network I/O messages per cycle as a function of PEs on a network with no bandwidth limitations, given a ConceptNet [40] (Section 5.2.1) communication load. Small networks ($<$ 100 PEs) require only 1–10 network sends or receives per cycle, as most cycles are dedicated to serialized processing at PEs. Larger networks ($>$ 100 PEs) require up to 180 network sends or receives per cycle. Consequently, examining network topologies for small networks or for applications with light communication requirements will not load networks enough to distinguish tradeoffs in network design. It is essential to examine both large-scale networks and applications which stress those networks in order to fully characterize the performance differences between network architectures.

5.2 Applications

We map three applications to the GraphStep system architecture. Each application computes over a static, sparse graph, and thus the communication between graph nodes can be determined and scheduled before runtime.

5.2.1 ConceptNet Spreading Activation Step

ConceptNet [40] is common-sense reasoning knowledge base represented as a graph. Nodes represent nouns or noun-verb pairs (e.g. "cookie", "eat cookie") and edges represent semantic relationships (e.g. "used for", "is a"). Applications of ConceptNet include topic-jisting, analogy-making, text summarization, and semantic prediction.

A key step in the ConceptNet algorithm is spreading activation. In spreading activation, an initial set of graph nodes (corresponding to the set of keywords for a particular query) are activated.

A single step of spreading activation involves sending an activation potential from each node to its neighbors along its edges, activating those neighbors in turn. As the computation proceeds, larger portions of the graph become activated. The percentage of active edges (**activity factor**) over the whole graph depends on the initial query and what step of the spreading activation is being performed. As the time-multiplexed network must compute schedules allowing for all possible communication taking place, we perform our time-multiplexed experiments at 100% activity. We run the packet-switched experiments at 100% activity and for a range of activity factors between 1%–100% which correspond to consecutive steps in selected queries.

ConceptNet consists of four distinct sets of predicates, each of differing breadth and accuracy. The experiments described here were performed on the smallest, highest quality subset consisting of 14,000 nodes and 27,000 edges. This minimal subset exercises the core algorithm while containing simulation run times.

5.2.2 Sparse Matrix-Vector Multiply

Iterative Sparse Matrix-Vector Multiply (SMVM) is used in several iterative numerical routines (*e.g.* Conjugate Gradient, GMRES), where a large sparse matrix (*i.e.* a matrix with many zeroes) is multiplied by a vector. We use a parallel implementation of this algorithm based on deLormier *et al.* [15], which parallelizes Compressed Sparse Row (CSR) SMVM. CSR SMVM is a set of dot products between the vector and matrix rows which are partitioned across multiple PEs. During an iteration of the compute stage, each PE computes the dot product between its assigned rows and the vector, resulting in a vector which must be sent to other PEs for use in the next iteration. For this implementation graph nodes represent matrix rows while edges represent required communication. We map representative matrices from the Matrix Market suite [47] to generate workloads for our experiments.

5.2.3 Bellman-Ford Shortest Path

The goal of Bellman-Ford is to find the shortest (*i.e.* least weight) path from a single node to all other nodes on a weighted graph. The Bellman-Ford algorithm is used in several CAD applications such as single source shortest paths (*e.g.* FPGA Routing [42]), finding negative edge weight cycles (*e.g.* Retiming [36]), and slack propagation (*e.g.* Static Timing Analysis) in circuit graphs. Bellman-Ford operates by performing a relaxation operation on all edges of the graph in each GraphStep until quiescence (*i.e.* no more messages left). A relaxation of an edge consists of examining the sink node of the edge and computing the minimum the node's data value and the edge's data value. We run Bellman-Ford over representative ISPD98 [2] benchmark graphs to produce communication workloads that capture the structure of circuit netlists.

5.3 Application Characteristics

An understanding of application communication characteristics can aid network designers in selecting an appropriate network architecture and size. Characteristics of the application graphs used for our experiment are shown in Table 5.1. In Section 4.3 we identified performance limiting characteristics of network topologies; here, we identify three key performance limiting characteristics of application graphs.

5.3.1 Total Edges

The total number of messages (*i.e.* graph edges) generated by an application dictates how serialized (Eq. 4.5) that application is. That is, for an application graph with a large number of edges mapped to a topology with a small number of PEs, each PE will need to process many messages. When this ratio of edges to PEs is large enough, total communication time will be dictated by serialized processing of self messags at the PEs. We observe that when the ratio of number of application edges to number of PEs is on the order of 100, the application is typically serialization limited. When this ratio is smaller, network bottlenecks such as bisection (Eq. 4.7) and latency (Eq. 4.8) begin to dominate PE serialization. Here, the total number of messages injected into the network (Figure 5.1) is large enough load the network more heavily than serialized self message processing at each PE. For the network sizes we examine (2–2048 PEs), large application graph (80–220K edges) are not affected by bisection or latency until the largest PE counts. Hence, we do not examine workloads beyond 220K messages, as topologies do not differentiate in performance under these conditions (*i.e.* $\frac{220K}{2048} > 100$).

5.3.2 Node Degree

As network size grows and the number of PEs increase, at a large enough network size a PE may contain a single graph node. This PE must process all messages (*i.e.* graph edges) of this node serially (Eq. 4.5). Therefore, total system performance will be limited by the number of edges of the largest degree node. Applications with large fanin or fanout nodes can therefore become performance limited at large PE counts when a single large fanin-fanout graph node resides on a PE. For the ConcepetNet `small` graph, this bottleneck effect is significant enough to dominate performance for all network sizes. As this is an important issue in mapping graphs to the GraphStep system architecture, we implemented a decomposition strategy that reduces the degree of large nodes by creating fanin and fanout trees. See Section 5.4 for an overview of our decomposition strategy. We decompose only the ConcepetNet `small` graph since the maximum degree of nodes in all other graphs was not large enough to dominate performance. After decomposing `small` all graphs have a degree limit of ≤ 250 (Table 5.1).

Graph	Nodes	Edges	Max Degree		Rent	
			Fanin	Fanout	c_{graph}	p_{graph}
ConceptNet						
`small`	15026	27745	63	64	33	0.5
SMVM						
`add20`	2395	17319	124	124	21	0.6
`bcsstk11`	1473	17857	27	30	89	0.2
`rdb32001`	3200	18880	6	6	24	0.5
`gemat11`	4929	33185	27	28	23	0.8
`utm5940`	5940	83842	30	20	300	0.2
`fidap035`	19716	218308	18	18	222	0.0
Bellman-Ford						
`ibm01`	12752	36455	33	93	28	0.3
`ibm05`	29347	97862	9	109	25	0.6

Table 5.1: Application Graphs

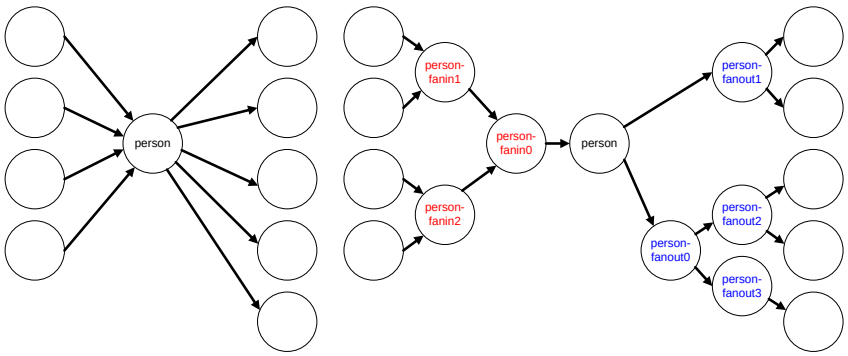
5.3.3 Rent Parameter

To describe the richness and locality of physical network interconnect we used the Rent parameter p (Eq. 3.1). As Rent's rule can be applied to any connected graph structure, for application graphs we can similarly define an intrinsic Rent parameter p_{graph} to describe the communication locality of the application graph itself. p_{graph} is independent of the actual placement of the graph and describes the communication required by the application. We can compute this parameter by counting the number of messages (*i.e.* graph edges) that cross the bisection recursively at each level of a recursive partition, similar to calculating p for network topologies. Applications with higher p have non-local communication and require networks with sufficient interconnect capable of supporting this communication. We examine application graphs with over wide range of p_{graph}.

5.4 Node Decomposition

Node decomposition decreases the size of large degree nodes, making it easier to balance communication. As PEs serially process the edges of resident nodes, very large degree nodes will limit total system performance unless we can decrease their degree. We decrease node degree by breaking up nodes into multiple, low degree nodes connected in fanin and fanout trees. Figure 5.2 shows an example node decomposition.

Given a large node, we construct k-arity bounded fanin and fanout trees to deal with input and output edges. We use $k = 64$ for decomposing `small`. Each message sent on a logical edge to a large fanin node is received by a low degree node at the leaf of a fanin tree. That leaf node waits to receive all input messages and sends a combined message up the tree. Messages sent on output edges are handled in a similar way by nodes in the fanout tree. We observe that for messages where operations performed on edge data are associative, such as ConceptNet, data in messages can be combined and split arbitrarily without affecting correctness. Therefore, by creating extra, low degree nodes that



(a) Large Degree Node in ConceptNet (b) Decomposed Node in ConceptNet (degree = 3)

Figure 5.2: Node Decomposition Strategy

can be placed in different PEs, we can potentially decrease total PE serialization by $max\_degree - k$ messages while ensuring semantic correctness.

However, we must guarantee that all nodes in fanout and fanin trees receive all messages from all predecessors before sending messages to successors. Scheduling all communication prior to runtime with our time-multiplexed router gives us the freedom to deal with this limitation easily. For each node in a fanout-fanin tree we build a list of constraints that contains all predecessor edges. Our router only routes nodes whose constraints have been satisfied (*i.e.* all dependent messages have been received), ensuring that outgoing messages are scheduled in a time-slot after incoming messages. See Section 4.2.1 for more details on router constraint handling, and Section 5.4.1 for the impact on router quality.

To partition graphs with large nodes, we first remove nodes with degree $> k$. We then partition the graph (Section 5.6) and decompose large nodes separately. We then introduce decomposed nodes into the partition and repartition to integrate the new nodes.

5.4.1 Router Quality Impact

Here we provide some preliminary results comparing the performance of decomposition and the quality of routing with and without constraints. Figure 5.3 plots time-multiplexed communication time vs. PEs for ConceptNet `small` over a mix of the best performing topologies. Specifically, in Figure 5.3(a) we plot communication time for an undecomposed graph (requiring no routing constraints); in Figure 5.3(b) we plot communication time for a decomposed graph but without routing with constraints (semantically incorrect, but shown to help understand router quality); in Figure 5.3(c) we plot communication time for a decomposed graph routed with constraints. We see that in Figure 5.3(a), without decomposition performance is limited to over 2000 cycles due to the degree of one very large node (in this case, the "person" node). After introducing decomposition, we see that communication time can decrease by an order of magnitude to 200 cycles. We see that

communication time is roughly equivalent for a decomposed graph when routing without constraints (Figure 5.3(b)) or with constraints (Figure 5.3(c)), implying that total performance and router quality is not affected by constraints.

Figure 5.4 confirms this trend by plotting router quality (ratio of actual to lowerbound cycles) as a function of PEs for the same set of `small` graphs. When routing without decomposition (Figure 5.4(a)) we see that router quality for the undecomposed `small` graph is very high, as it is easy for the router to route in conditions where performance is dominated by a single large degree node. When routing with decomposition we see that router quality without constraints (Figure 5.4(b)) is virtually identical to router quality with constraints (Figure 5.4(c)).

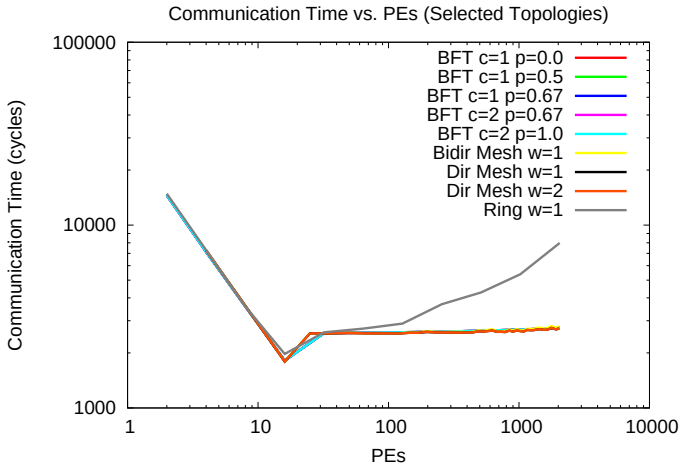
5.5 Processing Elements

We design custom processing elements for each application to model GraphStep computation. The basic structure of the PE is shown in Figure 5.5.

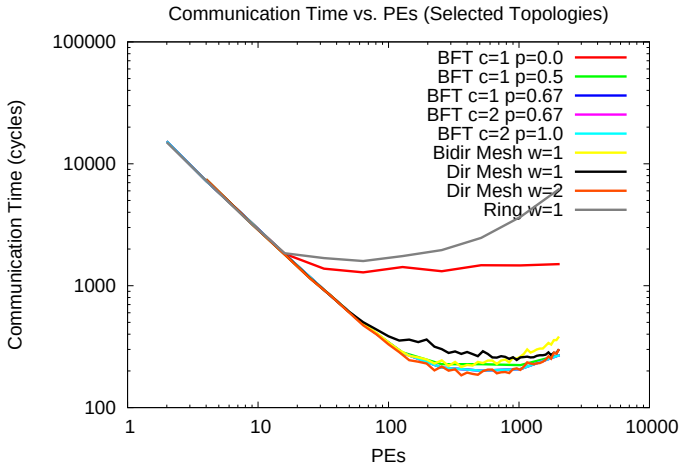
The GraphStep evaluation model is a three phased Receive-Update-Send sequence, where each PE computes in the Update phase and sends and receives messages in overlapped Send and Receive phases. In the Receive phase the PE must store incoming messages in memory, while in the Send phase the PE must read outgoing messages from memory. In the Update phase the PE must read each node from memory, read each incoming edge on a given node to process received data, and then write data to outgoing edges to be sent in the next phase. Additional data may need to be written to the node. Therefore, each PE must perform $O(incoming\_edges)$ writes in Receive, $O(outgoing\_edges)$ reads in Send, and $O(2 \times nodes + incoming\_edges + outgoing\_edges)$ memory operations in the Update phase.

To provide graph storage we create separate logical memories for nodes, incoming edges, and outgoing edges. These memories may be implemented phyiscally as separate or shared memories. We provide storage with dual ported BlockRAMs [63], where PEs receive as many as the number of BlockRAMs divided by PEs, but are guaranteed to receive at least one BlockRAM each. We structure the nodes and edges in the memories such that all memory accesses (node read, incoming edge reads, outgoing edge writes, node write) in the Update phase are sequential, eliminating the need for address computations. We do store a single bit of read/write enable for both node memory operations and edge memory operations in compact SRL16 storage [63] (32 bits per slice), necessary for indicating when to read and write in the Update phase. We account for these two bits in our area model as follows:

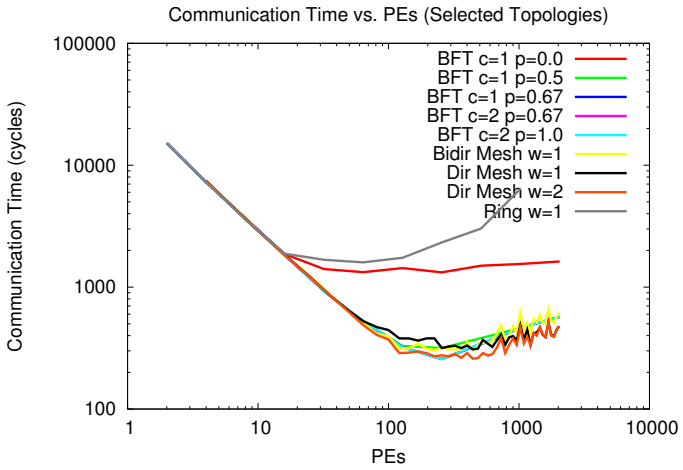
$$PE_{UpdateContext} = 2 \times \frac{nodes + incoming\_edges + outgoing\_edges}{32} \tag{5.1}$$



(a) Communication Time vs. PEs (`small`, undecomposed, no constraints)

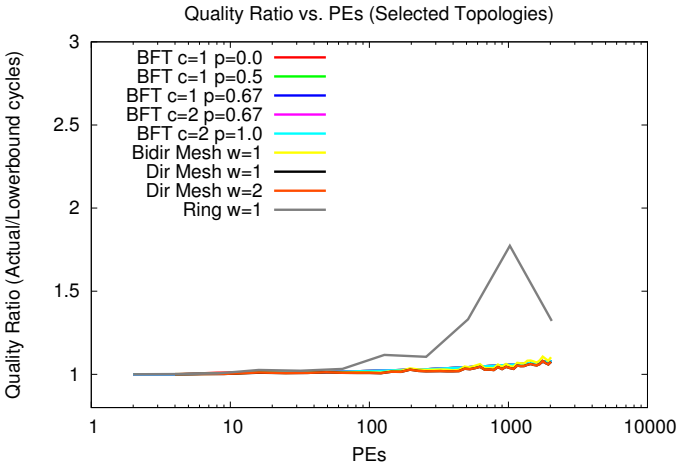


(b) Communication Time vs. PEs (`small`, decomposed, no constraints)

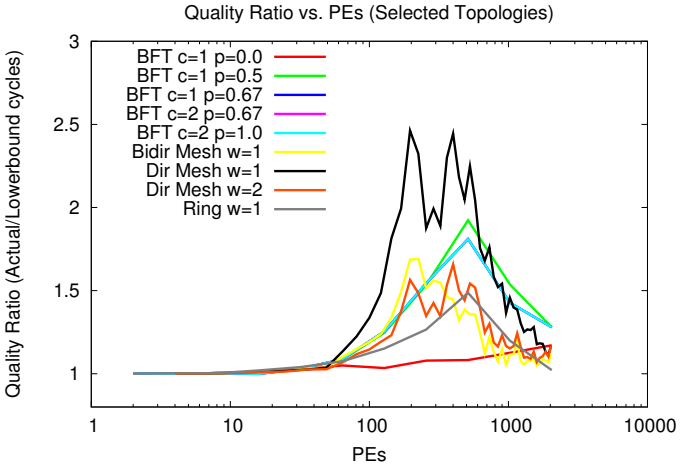


(c) Communication Time vs. PEs (`small`, decomposed, constraints)

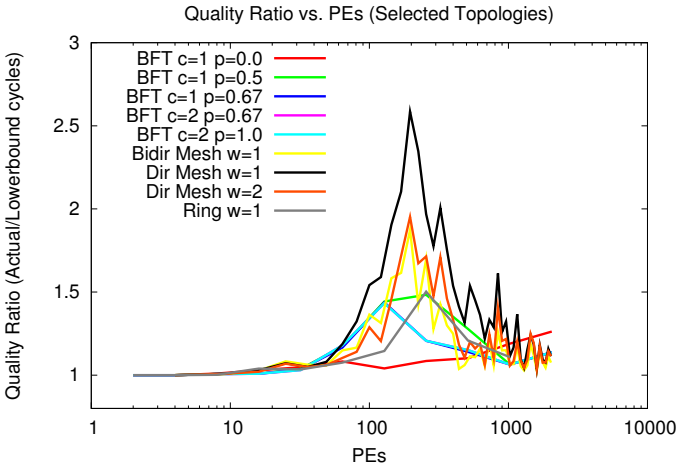
Figure 5.3: Communication Time vs. PEs with and without Decomposition (`small`)



(a) Quality Ratio vs. PEs (`small`, undecomposed, no constraints)



(b) Quality Ratio vs. PEs (`small`, decomposed, no constraints)



(c) Quality Ratio vs. PEs (`small`, decomposed, constraints)

Figure 5.4: Quality Ratio vs. PEs with and without Decomposition (`small`)

Application	Datawidth	Logic Area (slices)	Frequency (MHz)
ConceptNet	16	200	166
SMVM	64	4000	166
Bellman-Ford	16	200	166

Table 5.2: GraphStep Application PEs

To handle the sending and receiving of messages for the Send and Receive phases, each PE contains input and output packet handlers. These handlers, like the compute pipeline, are pipelined to allow the sending and receiving of a single message per cycle. The input handler is responsible for writing data from an incoming packet to memory, while the output handler is responsible for reading data for an outgoing packet. Memory access, like network resources, is completely pre-scheduled by the time-multiplexed router. To indicate when the handlers should read and write to memory a valid bit for each handler is computed by offline router for each cycle, equivalent to read and write enables. The valid bit for the input packet handler also serves to indicate when a valid incoming messages is arriving, to avoid handling spurious messages. A third valid bit signals the processing of self messages. These bits are stored in SRL16s and are accounted for in our area model as follows:

$$PE_{ValidContext} = 3 \times \frac{contextDepth}{32} \tag{5.2}$$

To provide the address at which the handlers read and write, we store the schedule of all read-/write addresses for both incoming and outgoing edge memory. We only need to store a single address for each edge, as the incoming and outgoing valid bits indicate when to read and write on each cycle. We also store these in SRL16s:

$$PE_{ReceiveContext} = \log_2(incoming\_edges) \times \frac{incoming\_edges}{32} \tag{5.3}$$

$$PE_{SendContext} = \log_2(outgoing\_edges) \times \frac{outgoing\_edges}{32} \tag{5.4}$$

The area required for PE in slices is therefore:

$$PE_{area} = PE_{logic} + PE_{SendContext} + PE_{ReceiveContext} + PE_{UpdateContext} + PE_{ValidContext} \tag{5.5}$$

The compute pipeline is unique to each application. For both ConceptNet and Bellman-Ford the pipeline must handle addition and min/max calculations. The SMVM pipeline requires double precision addition and multiplication. For increased speed we implement multipliers using pipelined LUT-level logic instead of the embedded 18x18 multipliers.

Datawidth, area, and frequency for each PE is shown in Table 5.2. Area figures represent only base PE logic area; area may increase significantly depending on graph size, partition balance, and context depth (Eq. 5.5). Each application uses a PE datawidth and network datawidth equal to the

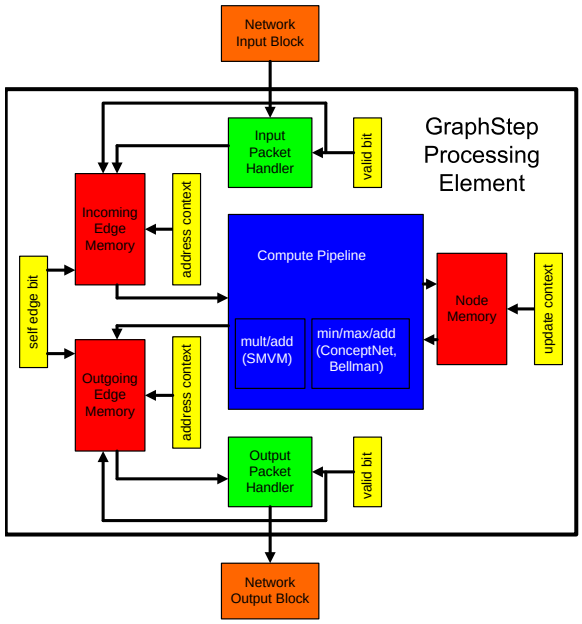


Figure 5.5: GraphStep PE

size of a single message. We note that the area of the SMVM PE logic is significant due to the use of LUT-level logic to implement double precision multipliers. We also note that each PE operates at 166 MHz, below the maximum operating frequency of the network (> 200 MHz). Therefore, total system performance is currently limited by the critical path of the PE datapath and not the network, which is capable of running at much higher speeds.

5.6 Toolflow

To evaluate the performance of our networks, we constructed a Java-based infrastructure to simulate the packet-switched network with cycle accuracy [27, 28] and to compute schedules for the time-multiplexed network (Chapter 4). We map applications to our networks using a partitioner and placer based on MLPart v5.2.14 which is part of the UMPack [10] package. While we ensure that single chip logic and interconnect resources are sufficient to map our applications, we assume that application graphs can be mapped to the available on-chip BRAMs.

We use our Java infrastructure to generate a structural VHDL netlist for a given network configuration. We create VHDL for each PE, implementing the 3 phase algorithm of the GraphStep system architecture [16]. Since we decompose our switchboxes into merge primitives, we can pipeline and optimize at the level of these individual primitives for high performance (Table 3.1). We demonstrate 166 MHz performance for a sample topology (BFT with $c = 1, p = 0.5$ and 8 PEs) on a XC2V6000-4 without context memory. We synthesize, place, and route the entire VHDL design using the Synplicity Compiler v8.0 and the Xilinx ISE v8.1i to obtain hardware operating frequency and slice count results.

Chapter 6

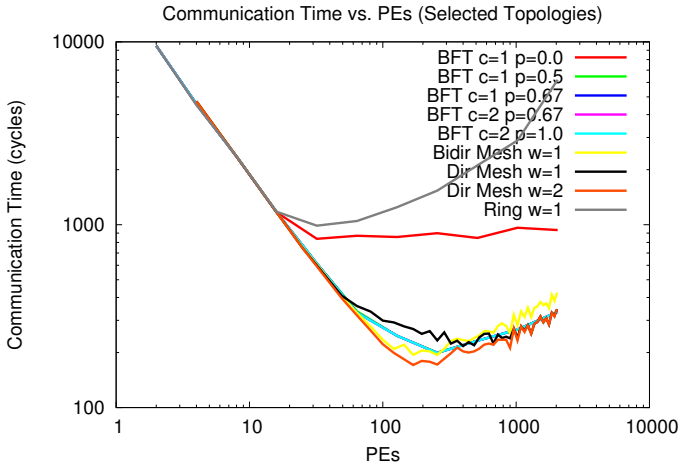
Time-Multiplexed Topology Selection

We present three quantitative comparisons to explore the issues in designing large-scale time-multiplexed network topologies. First, we examine how communication time scales with the size of a topology in terms of number of PEs. Second, we examine the size of a topology in terms of actual area and re-evaluate performance scaling. Third, we attempt to choose a single topology that is robust over all areas and for all applications, providing the best worst-case performance.

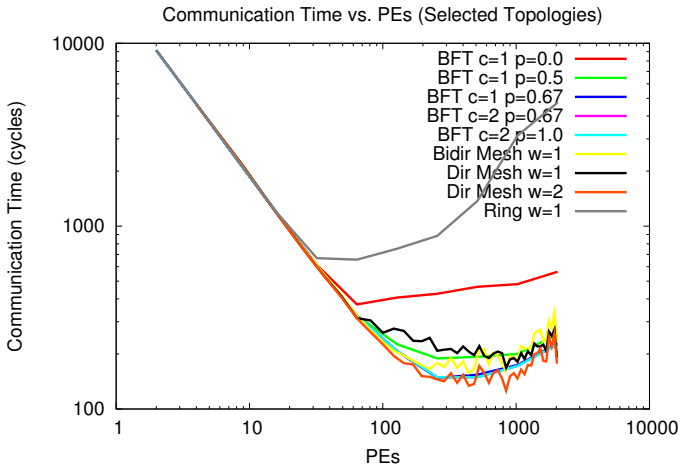
6.1 PE Scaling

Section 4.3 listed several physical characteristics of network topologies that bound performance. As these bounds depend on the physical structure of a topology, they aid in demonstrating inherent differences in topological performance. However, these differences in performance are often not visible until very large numbers of PEs. To explore the relationship between the number of PEs and topological performance, in Figures 6.1, 6.2, 6.3 we plot communication cycles as a function of PEs for small (17K–18K edges), medium (27K–36K edges), and large (80-220k edges) graphs in our application set. In each graph we plot the best performing configurations of the BFT ($c = 1$ with $p = 0, 0.5, 0.67$; $c = 2$ with $p = 0.67, 1$), directional mesh ($w = 1, 2$), bidirectional mesh ($w = 1$), and ring ($w = 1$).

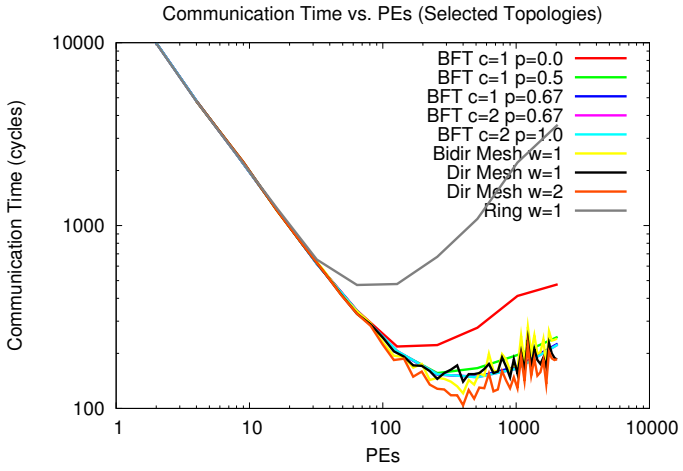
We describe the general trends present in each graph by examining the `bcsstk11` graph as representative example (Figure 6.1(b)). `bcsstk11` contains 18K messages; therefore, we expect to it to be serialization limited until around 100 PEs. We see that at low PE counts (< 10 PEs) there is no significant performance difference between topologies. Here, most PEs are processing self messages and the network is lightly loaded (see Figure 5.1). This motivates us to further increase PE counts to understand when topologies begin to differ significantly. As PE count increases (10–100 PEs), most topologies are still serialization limited, but the ring and BFTs with $p = 0$ begin to bisection



(a) `add20` (SMVM)

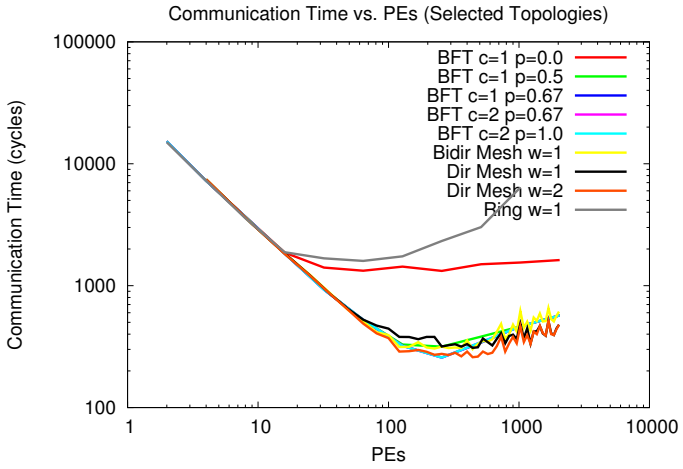


(b) `bcsstk11` (SMVM)

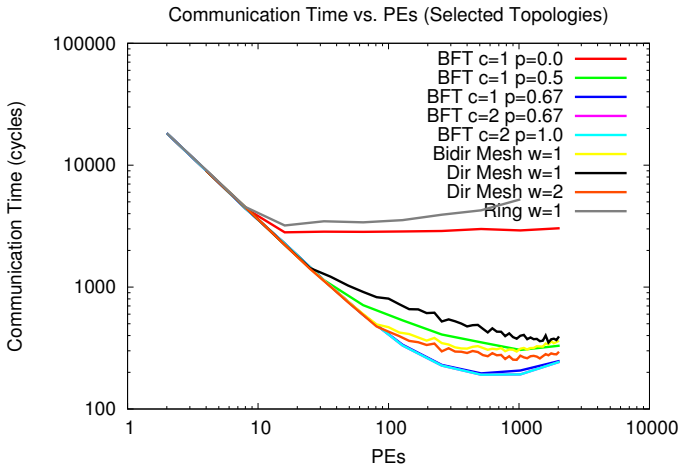


(c) `rdb32001` (SMVM)

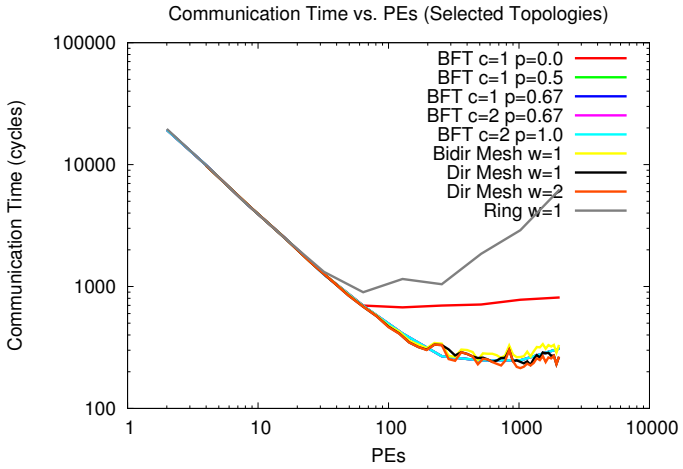
Figure 6.1: Communication Time vs. PEs (Small Graphs)



(a) `small` (ConceptNet)

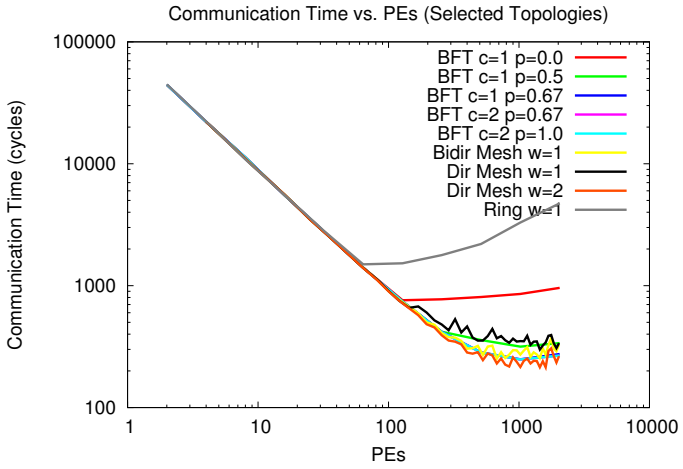


(b) `gemat11` (SMVM)

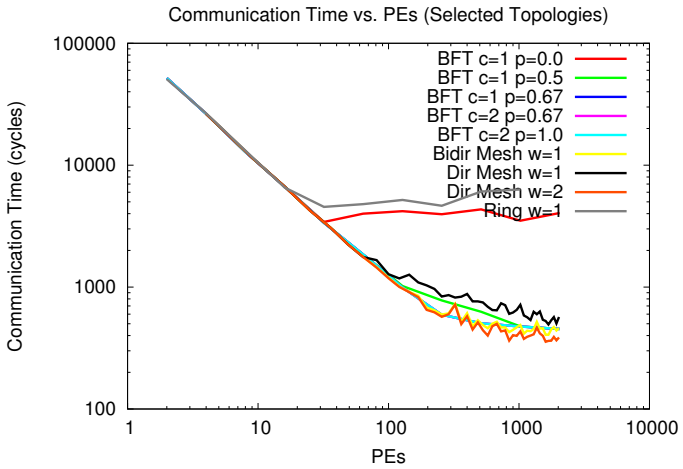


(c) `ibm01` (Bellman-Ford)

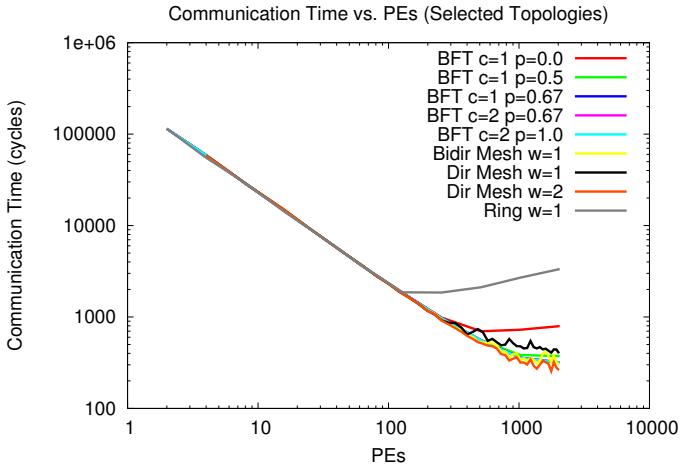
Figure 6.2: Communication Time vs. PEs (Medium Graphs)



(a) `utm5940` (SMVM)



(b) `ibm05` (Bellman-Ford)



(c) `fidap035` (SMVM)

Figure 6.3: Communication Time vs. PEs (Large Graphs)

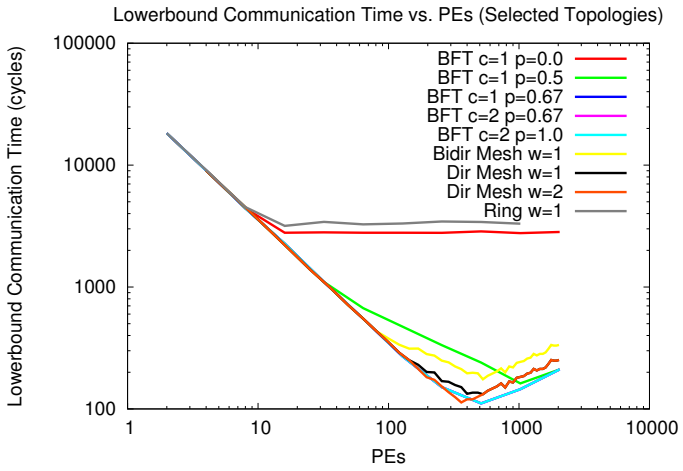


Figure 6.4: Lowerbound Communication Time vs. PEs (`gemat11`)

bottleneck (Eq. 4.7). In this range, PEs are still processing many self messages; however, PEs inject enough messages into the network to limit the performance of low bandwidth topologies. These topologies have a constant number of channels crossing the bisection and are consequently bisection limited. As PE count increases further (100-1000 PEs), as expected we see nearly all topologies leave the serialization limited regime. Rings are severely affected by latency, and the number of cycles required to route all messages increases significantly. As we examine rings as representatives of the best possible performance a bus can achieve, we see that buses cannot compete with the performance provided by scalable networks. Meshes with $w = 1$ are not affected by latency but are bisection limited, and BFTs with $p \leq 0.5$ are similarly bisection limited. We see that BFTs with $p = 0.67, p = 1$ and meshes with $w = 2$ achieve the best performance as they provide needed interconnect. At very high PE counts (>1000 PEs), all topologies are limited due to latency; here, increasing the number of PEs actually increases communication time. We note that these effects can only be seen when the ratio of application size to PE count is > 100.

These trends are consistent across all application graphs. In general we see that meshes with $w = 2$ and BFTs with $p = 0.67, 1$ provide the best performance, with their relative performance being nearly equal. For the `gemat11` graph we see that BFTs provide better performance than meshes at higher PE counts. `gemat11` (Figure 6.2(b) is a graph with highly non-local communication ($p_{graph} = 0.8$), and the $p = 0.5$ of a mesh does not provide enough interconnect to achieve high performance. For the large sized graphs shown in Figure 6.3, all topologies are serialization limited until PE counts larger than medium and small graphs. This is due to the large number of edges in these graphs; the graph edge to PE ratio indicates that all topologies will be serialization bottlenecked until 100s–1000s of PEs. We could attempt to increase PE count beyond 1000s of PEs; however, such topologies will be latency limited and may actually degrade performance.

To help us further understand why topologies perform differently at higher PE counts relative

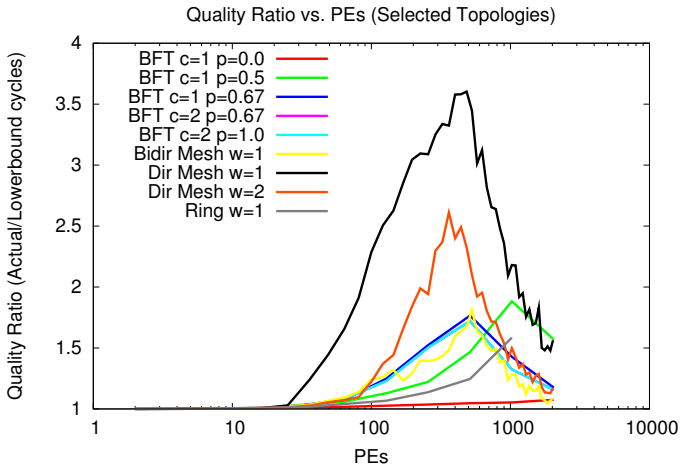


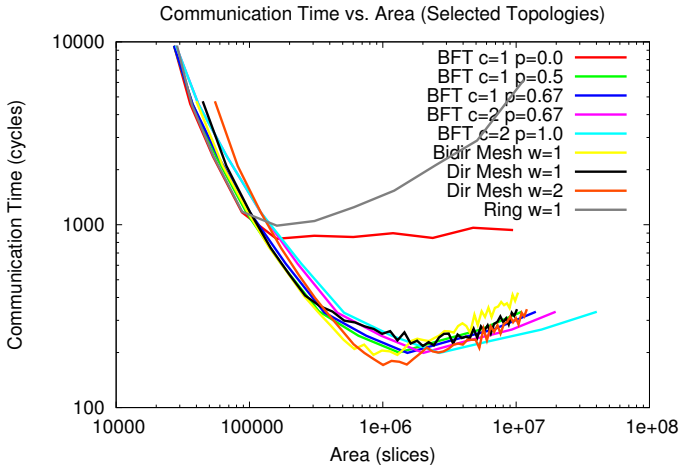
Figure 6.5: Quality Ratio vs. PEs for (`gemat11`)

to application size, in Figure 6.4 we plot the lowerbound performance (Section 4.3) of `gemat11` and in Figure 6.5 we plot the ratio of achieved performance to lowerbound performance . This helps us understand the difference between the inherent lowerbound performance of a topology and the achievable performance by the time-multiplexed router. We see in Figure 6.4 that lowerbound performance is close to actual performance for ring, bidirectional mesh, and BFT topologies shown in Figure 6.2(b); Figure 6.5 confirms this trend. We do see that the lowerbound for directional meshes is noticeably lower than actual performance. As noted in Section 4.4, the quality of the router is lower for directional meshes; this difference can possibly be attributed to the fact that mesh lowerbounds are not asymptotically tight, and that directional meshes are more difficult to route as they are less deterministic than BFTs and rings. Regardless, we observe that the best achievable performance across all PEs for all applications is obtained by BFTs and directional meshes.

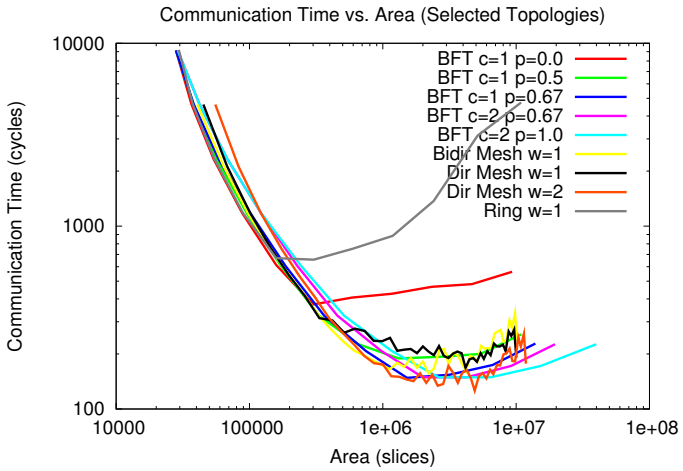
6.2 Area Scaling

In the previous section we did not consider the area requirements of our topologies. Topologies with equivalent PE counts may have very different area requirements as the number of and size of switches varies significantly across topologies (Table 3.2). Additionally, application PEs have varying sizes (Table 5.2). Both PEs and switches use the same area resources on FPGAs; therefore, we can tradeoff area between compute and interconnect to obtain increased performance when either is needed. To explore this tradeoff, in Figures 6.6, 6.7, 6.8 we plot communication cycles as a function of PEs for small (17K–18K edges), medium (27K–36K edges), and large (80-220k edges) graphs in our application set, across the best performing topologies.

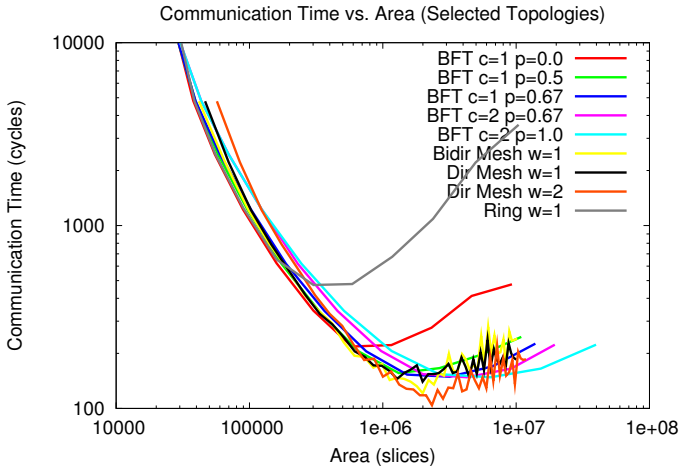
In Section 6.1 we observed that at low PE counts all topologies provide the same performance, and at higher PE counts high bandwidth meshes and BFTs provide the best performance. We see



(a) `add20` (SMVM)

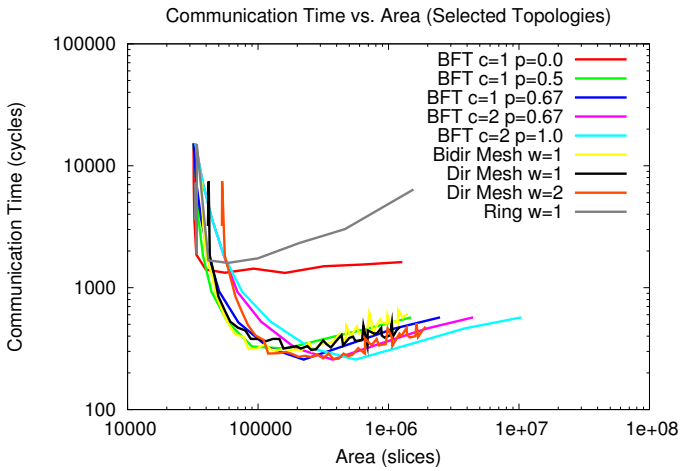


(b) `bcsstk11` (SMVM)

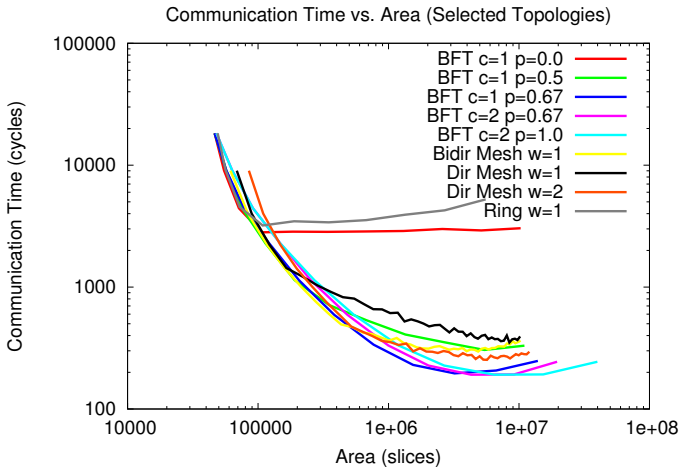


(c) `rdb32001` (SMVM)

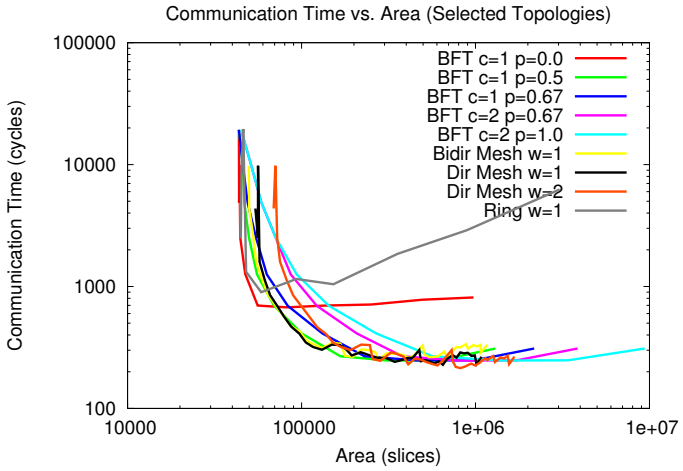
Figure 6.6: Communication Time vs. Area (Small Graphs)



(a) `small` (ConceptNet)

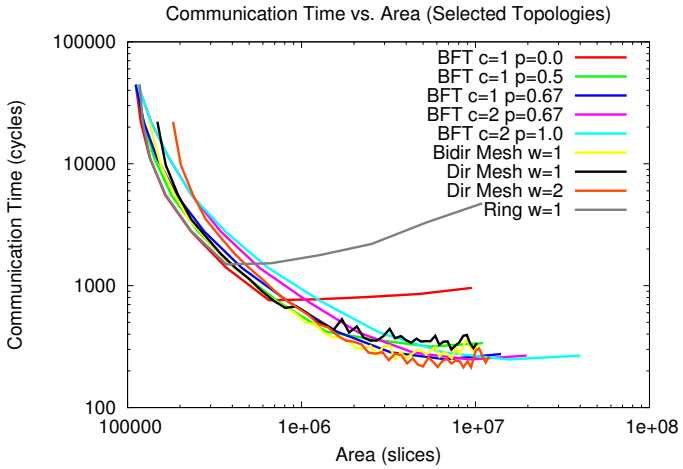


(b) `gemat11` (SMVM)

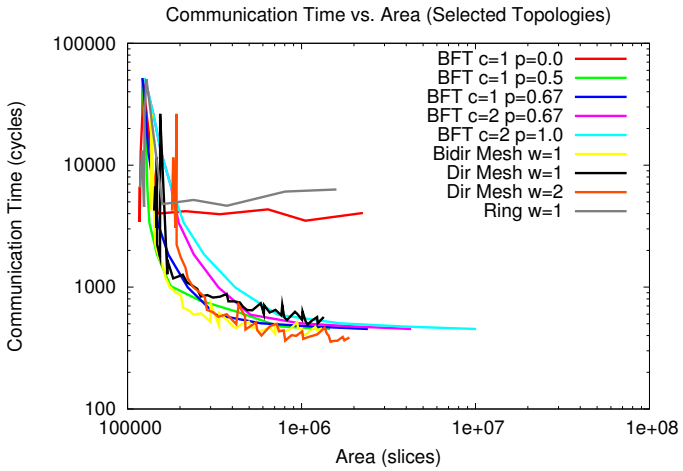


(c) `ibm01` (Bellman-Ford)

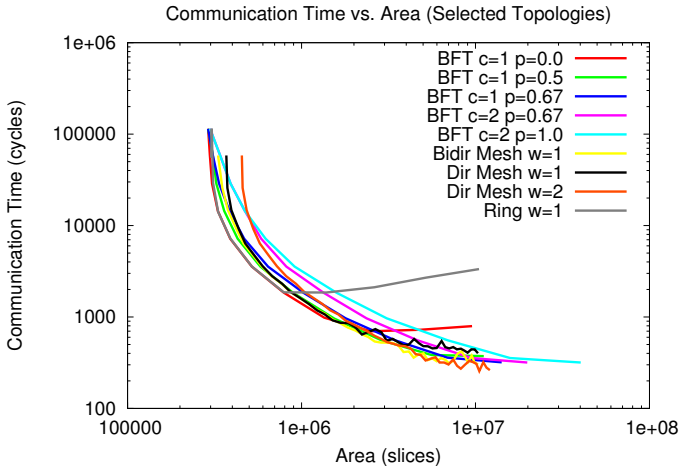
Figure 6.7: Communication Time vs. Area (Medium Graphs)



(a) `utm5940` (SMVM)



(b) `ibm05` (Bellman-Ford)



(c) `fidap035` (SMVM)

Figure 6.8: Communication Time vs. Area (Large Graphs)

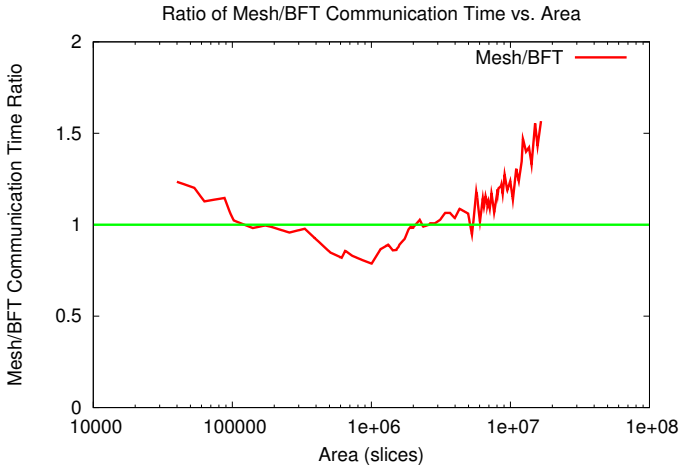
very similar trends here when considering area, but with some slight differences.

Initially, performance of all topologies is serialization limited between 10K–100K slices for small and medium graphs, and 100K–1M slices for large graphs. The serialization region for larger graphs extends beyond that of smaller graphs since larger graphs contain more edges and remain serialized at larger areas (Section 5.3.1). When only considering the number of PEs, in the serialization limited region all topologies yield the same performance. When considering area, topologies with less area allocated to interconnect and more area allocated to PEs provide slightly better performance in a given area. We can see across all graphs that a BFT with $c = 1, p = 0$ or a ring $w = 1$ provide more PEs and enough interconnect to get the best performance.

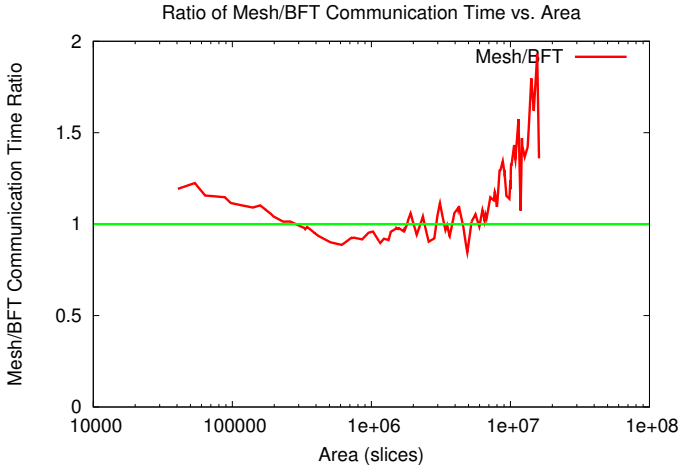
At larger slice counts (100K–1M slices for small and medium graphs, 1M–10M slices for large graphs), the network communication load increases and low bisection topologies that previously provided the best performance begin to bottleneck. We generally see here that topologies with slightly more area in interconnect (directional mesh $w = 1$, BFT $c = 1, p = 0.5$) perform best. For even larger slice counts (1M–10M slices for small and medium graphs, 10M–100M slices for large graphs), the largest bisection topologies (directional mesh $w = 2$, BFT $c = 2, p = 1$) yield the best performance. These topologies tradeoff area in PEs for area in interconnect to increase performance. We see that increased area in interconnect does not always provide increased performance; for large graphs (in particular `fidap035`, Figure 6.8(c)) performance continues to remain serialized, even for the largest areas considered. However, we generally observe that as we increase area, topologies that provide more interconnect achieve better performance.

We can make a few additional qualitative observations. First, we see that no time-multiplexed network for our applications can fit in an area below 30K slices. This is due to the large overhead required for context memory (See Section 8.2) in both the switches and PEs. For a discussion on the implications of the large areas that we consider, and when such areas might be attainable on a single chip, see Section 9.2.3.

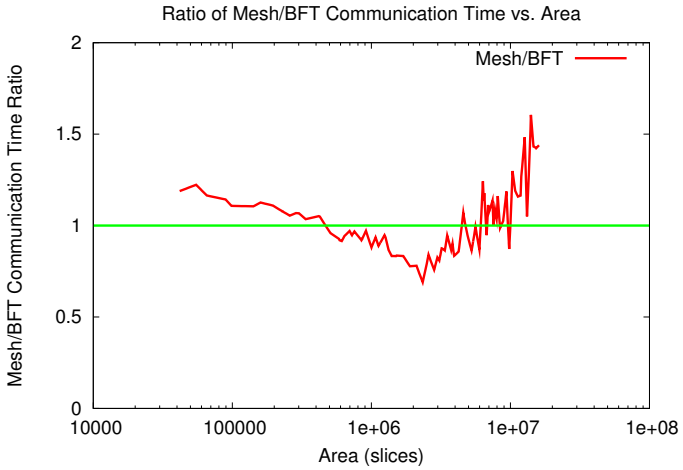
Second, we see that BFTs and meshes provide very similar performance and area efficiency. To help quantify which of these two topologies perform better in a given area, Figures 6.9 6.9 6.9 plot the ratio of optimal mesh performance to optimal BFT performance as a function of area. We see that aside from a very points at very small areas, across all areas the difference between the optimal mesh and optimal BFT is no more than a factor of 2. We generally see that the best mesh requires up to 2× as many cycles to route as the best BFT at smaller areas. For intermediate areas the best BFT requires 1.5× as many cycles to route as the best mesh. Finally, at larger areas the BFT regains its performance advantage, with the mesh requiring up to again 2× as many cycles to route.



(a) `add20` (SMVM)

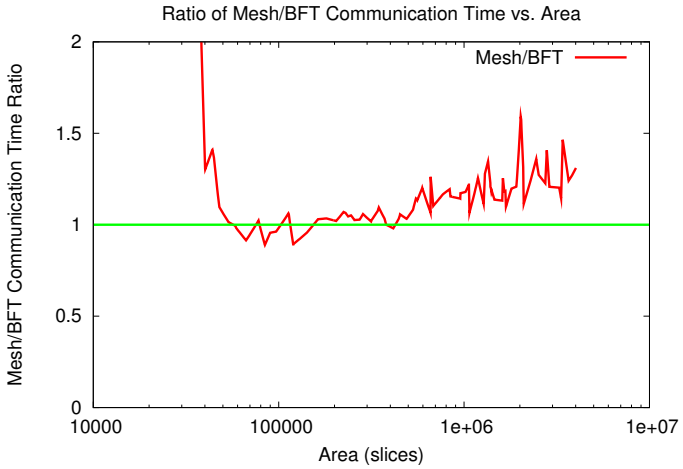


(b) `bcsstk11` (SMVM)

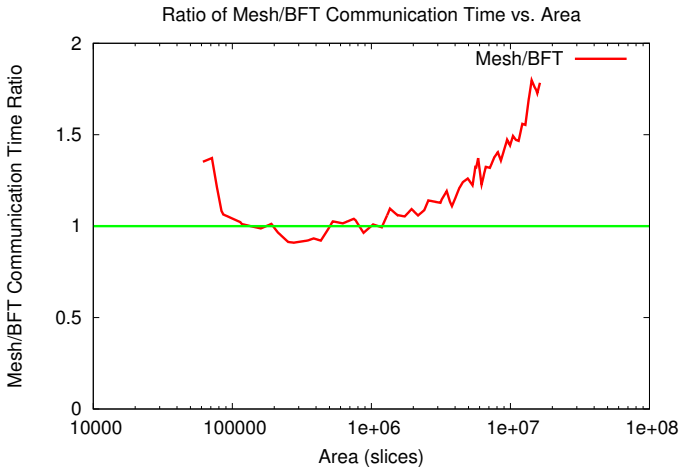


(c) `rdb32001` (SMVM)

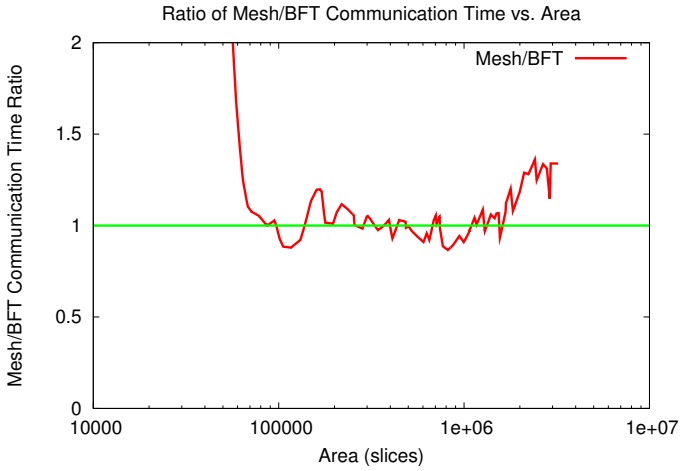
Figure 6.9: Ratio of Mesh/BFT Communication Time vs. Area (Small Graphs)



(a) `small` (ConceptNet)

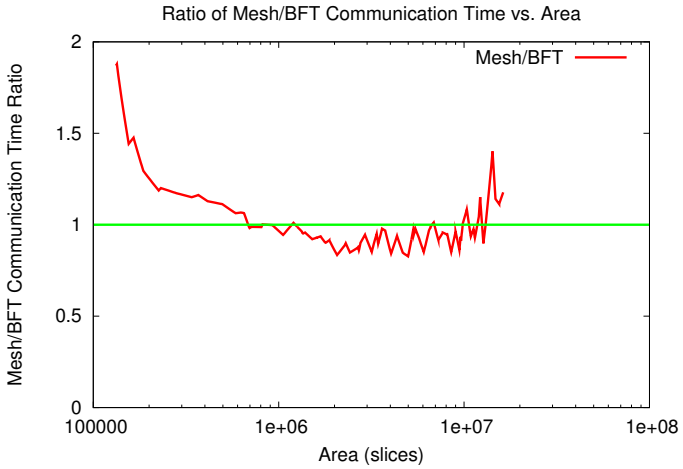


(b) `gemat11` (SMVM)

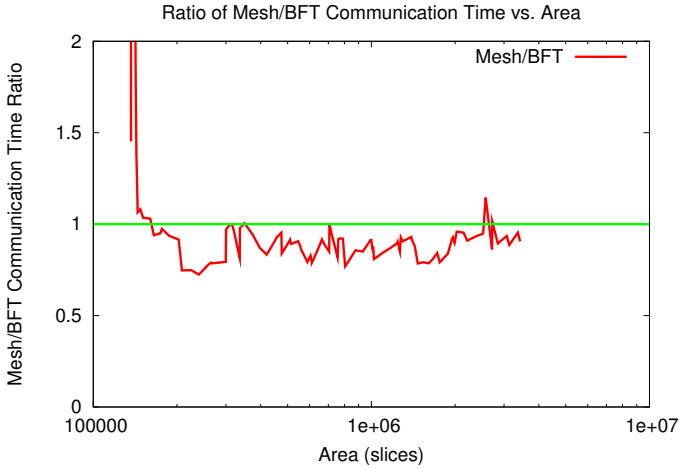


(c) `ibm01` (Bellman-Ford)

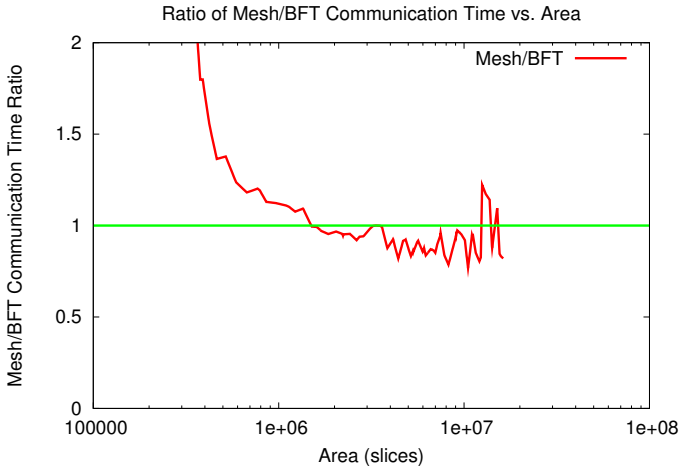
Figure 6.10: Ratio of Mesh/BFT Communication Time vs. Area (Medium Graphs)



(a) `utm5940` (SMVM)



(b) `ibm05` (Bellman-Ford)



(c) `fidap035` (SMVM)

Figure 6.11: Ratio of Mesh/BFT Communication Time vs. Area (Large Graphs)

Topology	Performance Ratio to Optimal Topology
BFT $c = 1, p = 0.0$	15.9
BFT $c = 1, p = 0.5$	**6.1**
BFT $c = 1, p = 0.67$	8.3
BFT $c = 1, p = 1.0$	8.3
BFT $c = 2, p = 0.0$	8.7
BFT $c = 2, p = 0.5$	11.6
BFT $c = 2, p = 0.67$	11.6
BFT $c = 2, p = 1.0$	11.5
Directional Mesh $w = 1$	9.2
Directional Mesh $w = 2$	18.9
Bidirectional Mesh $w = 1$	33.3
Bidirectional Mesh $w = 2$	16.3
Bidirectional Mesh $w = 4$	29.3
Ring $w = 1$	29.2
Ring $w = 2$	32.6
Ring $w = 4$	31.8

Table 6.1: Worst Case Topology Performance (All Areas and Applications)

6.3 Optimal Topology

We have seen that different configurations of BFTs and meshes perform optimally for different areas and applications. For small areas or for applications with a small p_{graph}, a topology with less area in interconnect provides the best performance. However, at large areas or for applications with a large p_{graph}, topologies with more are in interconnect are preferred. This indicates that designing a network with a fixed topology configuration could be inefficient over a range of applications and network sizes [37]. To help quantify the cost of a "one topology fits all" methodology, we compute the ratio of actual performance of each topology to the performance of the best topology at a given area. This ratio is always ≥ 1, where lower values indicate best worst case performance and topological robustness. In Table 6.1 we tabulate the worst case ratios for each topology, computed over all areas and all applications. We see that the most robust topology, a BFT with $c = 1, p = 0.5$, requires in the worst cast as many as $6.1\times$ as many cycles to route as the best topology. Many other topologies require roughly up to an order of magnitude or more cycles to route than the best topology. This indicates that there is a non-trivial cost in selecting a single topology for all areas and applications, motivating application-specific topology design.

Chapter 7

Time-Multiplexed vs. Packet-Switched Networks

We present three quantitative comparisons to help characterize the tradeoffs between packet-switching and time-multiplexing. To review, the goals of this chapter are to answer the following questions posed in Section 1.2.3:

- For equivalent topologies and communication loads, what is the quantitative difference between the offline, global routing of time-multiplexing and the online, local routing of packet-switching?
- For an equivalent, fixed area capacity and equivalent communication loads, what is the performance difference between well engineered time-multiplexed and packet-switched networks?
- For communication loads that only require routing a subset of all possible communication, what is the cost of routing all possible communication for time-multiplexed interconnect?

To answer these questions we perform the following experiments. First, we compare the performance difference between offline and online scheduling by routing identical topologies for identical 100% activity communication loads. Second, we consider the impact of area and compare the performance difference between packet-switching and time-multiplexing for identical, fixed area capacities. Third, we compare performance while varying the activity factor of our communication loads for various network sizes.

7.1 Packet-Switched Implementation

We provide a brief overview of the packet-switched implementation used for comparison (see [27, 28] for more details).

While the time-multiplexed network routes all communication offline with a router, the packet-switched network routes all communication online via intelligent switches. Instead of using context

Primitive	Area (Slices)			Latency (Cycles)	Speed (MHz)
	Queue		Total		
	Ctrl	Buffer			
16-deep Buffer					
Split2	30	33	80	2	218
Merge2	60	66	154	2	200
Split3	30	33	88	2	217
Merge3	90	99	254	2	200
Split4	30	33	96	2	212
Merge4	120	132	340	2	223
1-deep Buffer					
Split2	0	8	23	2	269
Merge2	0	16	58	2	262
Split3	0	8	30	2	269
Merge3	0	24	94	2	252
Split4	0	8	32	2	265
Merge4	0	32	115	2	252

Table 7.1: Packet-Switched Split and Merge Primitives (32-bit)

Switch	IO (Ports)	Area (Slices)	Latency (Cycles)	Speed (MHz)
Ring	3	243	4	200
Directional Mesh Island-Style	4	324	8	200
Directional Mesh Fast X-Y Style	4	972	4 (fast), 8(slow)	200
Bidirectional Mesh Arity-4 DOR	5	603	4	200
Bidirectional Mesh Arity-4 WSF	5	660	4	200
Bidirectional Mesh Fast X-Y Style	5	1215	4 (fast), 8(slow)	200
Bidirectional Mesh Virtual-Channels with Duato	5	-	8	200
BFT T-Switch	3	243	4	200
BFT Π-Switch	3	410	4	200

Table 7.2: Packet-Switched Switches (32-bit, 1-deep buffers)

memory to determine how to route a packet in the switch on a given cycle, switches dynamically compute routing decisions based on the switch routing algorithm and the header of the packet. Packets contain 16-bit headers corresponding to network addresses that each switch reads and interprets in order to determine the direction in which to send the packet. Like the time-multiplexed network we route single flit packets; this requires wider networks to accommodate for the 16-bits of header in each packet (32-bit networks for ConceptNet and Bellman-Ford, 80-bit networks for SMVM).

We examine the same network topologies for the packet-switched network as the time-multiplexed network: ring 3.1.1, mesh 3.1.2, and BFT 3.1.3. Much like the time-multiplexed network, switches in the packet-switched network are composed out of primitives. In addition to the merge primitives the packet-switched network uses the split primitive. These primitives interface with each other using data-presence and back-pressure based flow control. To deal with potential network blocking, we design our primitives with input queues to handle the buffering of packets. We sized our queues to be 1-deep as we found this provided the optimal performance and area efficiency.

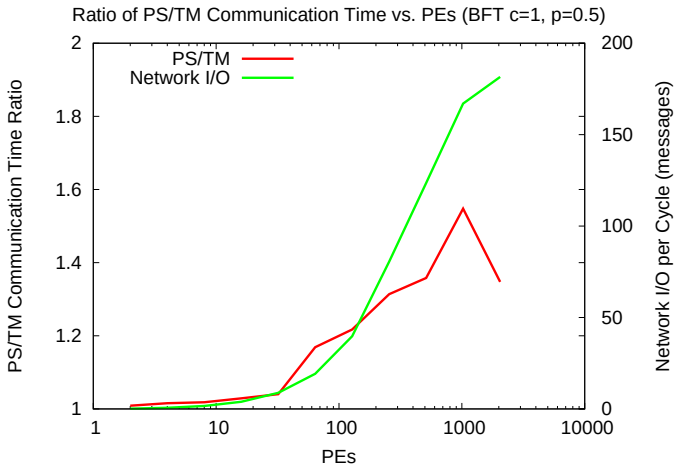


Figure 7.1: Ratio of PS/TM Communication Time vs. PEs (`gemat11`)

The split primitive has one input, two outputs and a routing function which selects one of the two outputs for the incoming packet. The split primitive computes the routing decision in a single cycle based on the destination address of the packet. This routing function determines the path that packets will take on the topology. We use several types of routing functions across topologies: trivial routing for rings, adaptive routing for BFTs, and implementations of dimension ordered, west side first, and Duato's routing algorithms for the mesh.

The merge primitive has two inputs and one output. Packets arriving on the two inputs must compete for a single output. A simple scheme of arbitration would be to select from the two input ports in a round robin fashion. We use a more adaptive scheme that selects a packet based on FIFO occupancies of the input queues.

Tables 7.1 and 7.2 round up the area, latency and speed required for packet-switched primitives and switchboxes.

7.2 Offline vs. Online Routing

To characterize the inherent performance difference between offline and online routing, we route 100% activity communication loads on equivalent time-multiplexed and packet-switched topologies, measuring the total number of communication cycles required to route. An example comparison for a BFT with $c = 1$ and $p = 0.5$ for the `gemat11` benchmark is shown in Figure 7.1. Network I/O per cycle (as in Figure 5.1) is also shown on the same graph. At low numbers of PEs (<32) we see that offline and online routing produce nearly equivalent cycle counts. Small numbers of PEs induce a light communication load (1–10 messages per cycle) and little offline/online differentiation. As the number of PEs increase (>32) the communication load increases (10s–100s of messages per cycle), and we begin to see offline routing outperform online routing. Offline routing is able to take

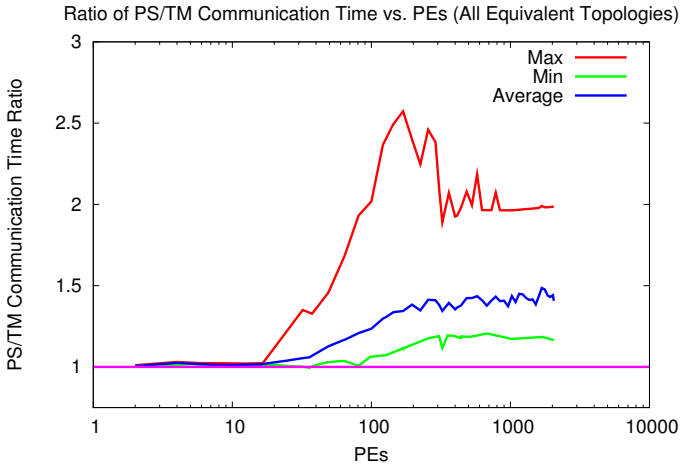
advantage of global information to make optimal routing decisions on larger networks, while online routing is limited to making local decisions which affect the overall quality of route. As a result, in this example online routing requires up to 1.5× as many cycles to route as offline routing for larger networks.

In Figures 7.2, 7.3, 7.4 we plot the ratio of packet-switched to time-multiplexed communication time as a function of PEs across all graphs in our application set: small (17K–18K edges), medium (27K–36K edges), and large (80-220k edges). Like the previous example, the goal of these graphs is to compare the inherent difference in offline and online routing for identical topological structures. Therefore, for these graphs we compare time-multiplexed and packet-switched networks for all equivalent topologies (*e.g.* packet-switched BFT $c = 1, p = 0.5$ vs. the same time-multiplexed topology). For each PE count we compute the ratio of time-multiplexed to packet-switched communication time for a single topology, and then we compute the maximum, minimum, and average ratio across all topologies. For topologies where there is a single time-multiplexed implementation but several equivalent packet-switched implementations (*e.g.* bidirectional mesh $w = 1$ with either dimension-ordered routing or west side first), we select the best packet-switched implementation for comparison.

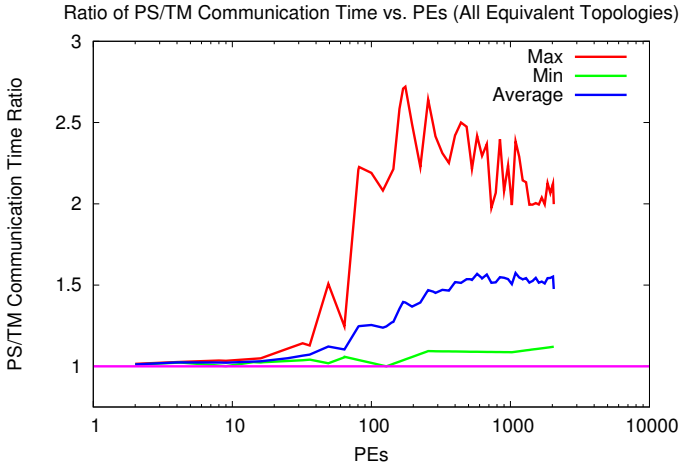
In these graphs we see very similar results across all applications. At low PE counts packet-switching and time-multiplexing route in the same number of cycles. In the worst case (the minimum ratio), across all PE counts time-multiplexing still routes in roughly the same number of cycles as packet-switching. For a very small number of points we see that packet-switching routes in fewer cycles than time-multiplexing (Figure 7.3(c) for example). For these points we note that the absolute difference in cycles is very small (typically 1-10 cycles) and therefore the performance difference is negligible. In the best case (the maximum ratio), at high PE counts packet-switching requires 2.8× as many cycles to route as time-multiplexing (Figure 7.4(a)). On average, across all benchmarks and all equivalent topologies at high PE counts, we see that packet-switching requires 1.5× as many cycles to route as time-multiplexing.

7.3 Normalized Area Comparison

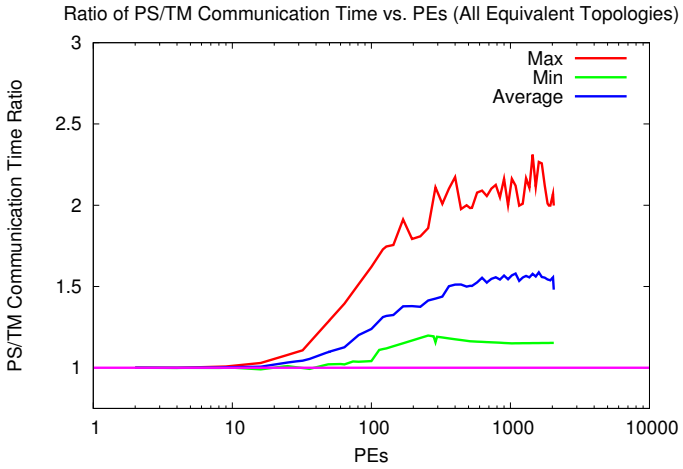
To fully characterize the performance difference between our packet-switched and time-multiplexed networks we must also consider the area they consume. For equivalent topologies at the same PE count, packet-switched and time-multiplexed networks may use significantly different amounts of area due to differences in switch sizes and the amount time-multiplexed context memory required. For example, packet-switched BFT T switchboxes for ConceptNet (32-bits) require 243 slices of logic and buffering. Time-multiplexed BFT T switchboxes for ConceptNet (16-bits) require only 24 slices of logic but $3 \times \frac{ContextDepth}{32}$ slices of context. Therefore, if time-multiplexing requires > 2592 cycles



(a) `add20` (SMVM)

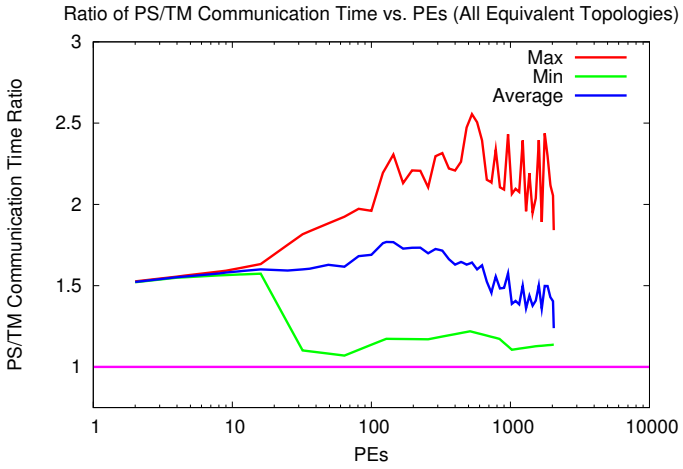


(b) `bcsstk11` (SMVM)

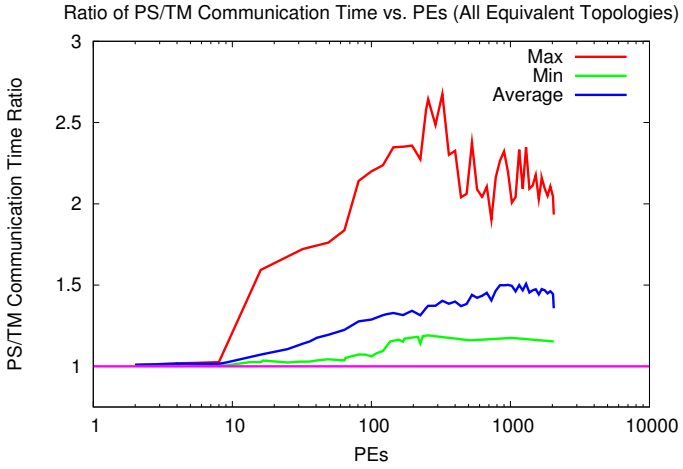


(c) `rdb32001` (SMVM)

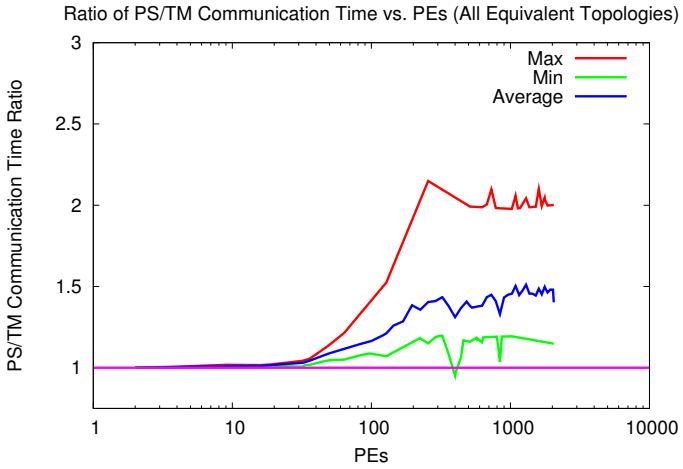
Figure 7.2: Ratio of PS/TM Communication Time vs. PEs (Small Graphs)



(a) `small` (ConceptNet)

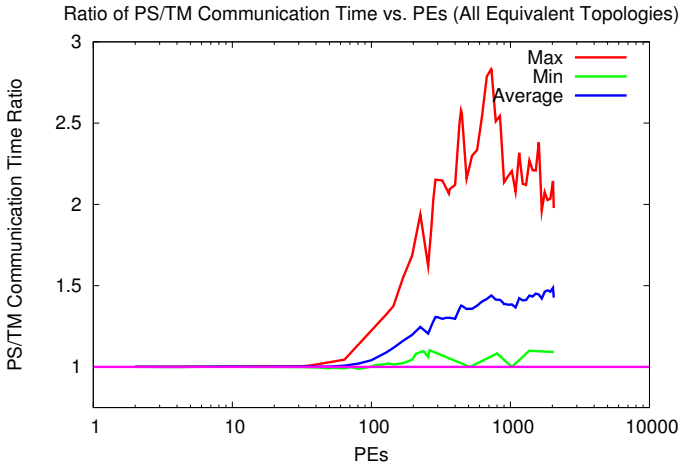


(b) `gemat11` (SMVM)

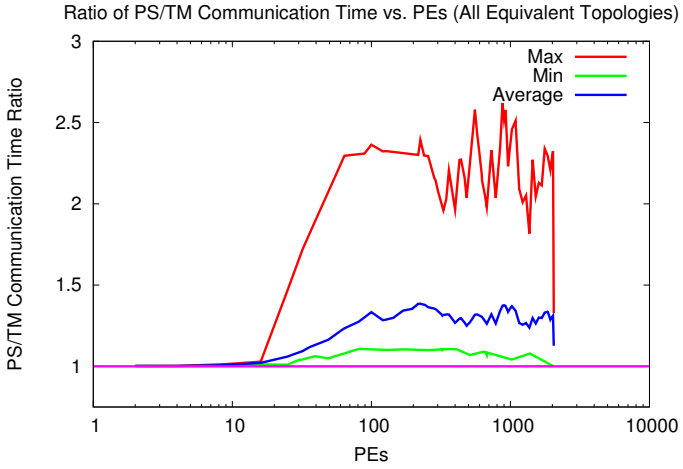


(c) `ibm01` (Bellman-Ford)

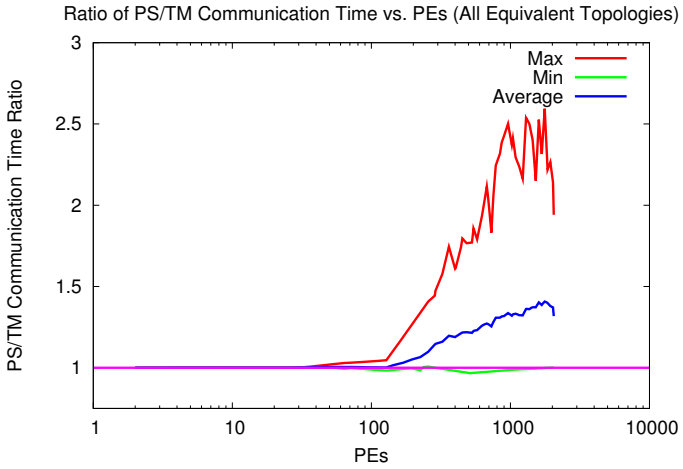
Figure 7.3: Ratio of PS/TM Communication Time vs. PEs (Medium Graphs)



(a) `utm5940` (SMVM)



(b) `ibm05` (Bellman-Ford)



(c) `fidap035` (SMVM)

Figure 7.4: Ratio of PS/TM Communication Time vs. PEs (Large Graphs)

to route communication, time-multiplexed BFT T switchboxes will be larger than their equivalent packet-switched implementations.

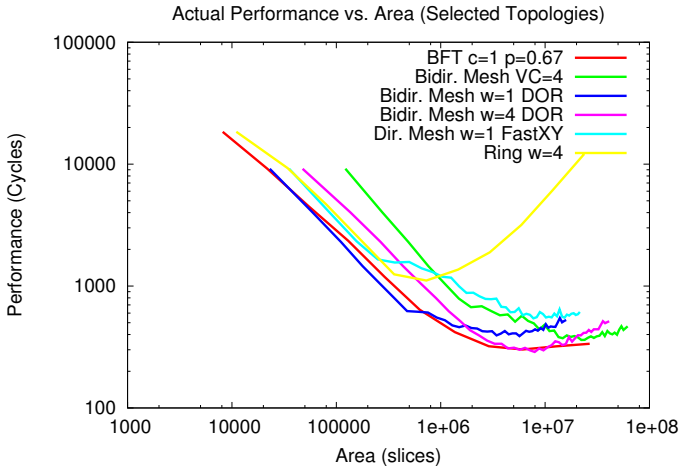
Figure 7.3 illustrates this tradeoff more generally with the `gemat11` benchmark as an example. In Figures 7.5(a) and 7.5(b) we plot communication time at 100% activity as a function of area for both packet-switching and time-multiplexing. In these graphs we show a mix of the best performing topologies across all areas. As described in the previous chapter (Chapter 6, Section 6.2), we see that in general as we increase area, topologies that provide more interconnect achieve better performance. To see how time-multiplexing and packet-switching compare directly, in Figure 7.5(c) we plot the composite best performance of time-multiplexing and packet-switching. To compare performance independent of topology (*i.e.* only for the best topologies at each area point), we obtain Figure 7.5(c) by taking the minimum over all topologies in Figures 7.5(a) and 7.5(b).

We observe that for small areas (8K–50K slices) it is not possible to implement a time-multiplexed network due to area constraints; however, packet-switching is feasible. As area increases and time-multiplexing becomes possible (50K–100K slices), time-multiplexing requires more cycles to route than packet-switching. In these areas time-multiplexing can only fit small numbers of PEs, resulting in higher cycle counts. However, as area increases (100K–10M slices), we see that time-multiplexing routes in fewer cycles than packet-switching. Time-multiplexing can fit more PEs, decreasing serialization and reducing cycle counts. Combined with time-multiplexing's advantage of routing all communication offline, we see that time-multiplexing can outperform packet-switching. Finally, above 10M slices we see this performance advantage decrease until around 40M slices, where the performance of packet-switching equals time-multiplexing. These areas correspond to the largest time-multiplexed networks (2048 PEs), where latency limits performance enough for packet-switching to close the performance gap. Above 40M slices packet-switched scaling continues, where the largest packet-switched networks require more area than the largest time-multiplexed networks.

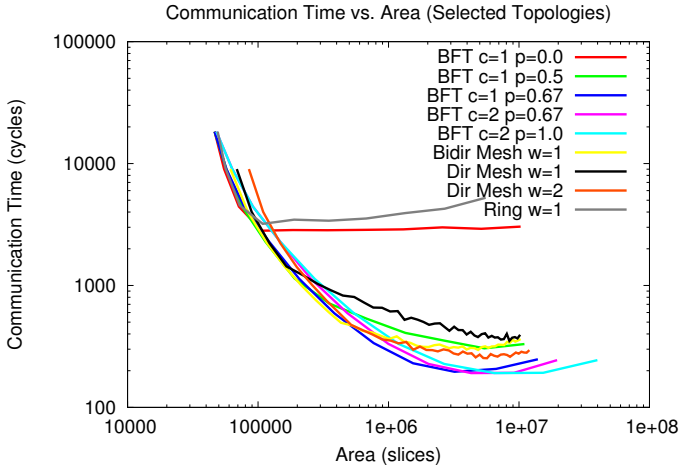
We note that the range of areas that we examine is extremely large (up to almost $100M$ slices), far exceeding the capacity of current chips. For a discussion on the implications of the large areas that we consider, and when such areas might be attainable on a single chip, see Section 9.2.3.

To quantify this tradeoff, Figures 7.6, 7.7, 7.8 show the ratio of packet-switched to time-multiplexed communication time as a function of area over these optimal topology points, for all graphs in our application set: small (17K–18K edges), medium (27K–36K edges), and large (80-220k edges). We can obtain these curves by dividing the time-multiplexed and packet-switched curves in Figure 7.5(c) for each application graph. In these figures we also show the individual area points at which time-multiplexing and packet-switching yield the best absolute performance, to help clarify the area required for optimal performance.

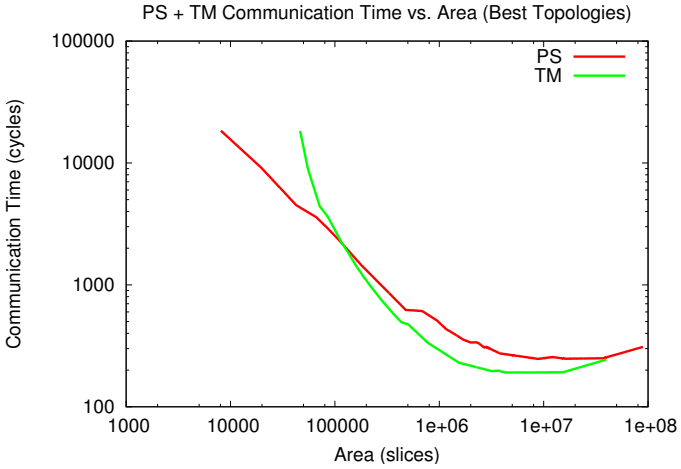
We see very similar results across all applications. At low areas the ratio of packet-switched to time-multiplexed communication time is below 1, indicating superior packet-switching performance.



(a) Packet-Switched Communication Time vs. Area (`gemat11`)

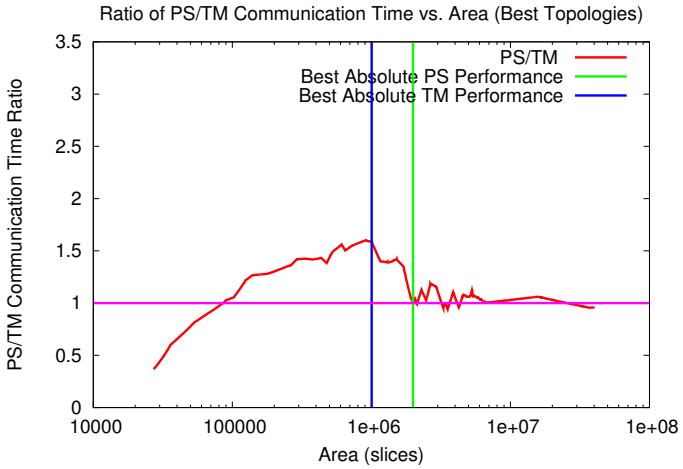


(b) Time-Multiplexed Communication Time vs. Area (`gemat11`)

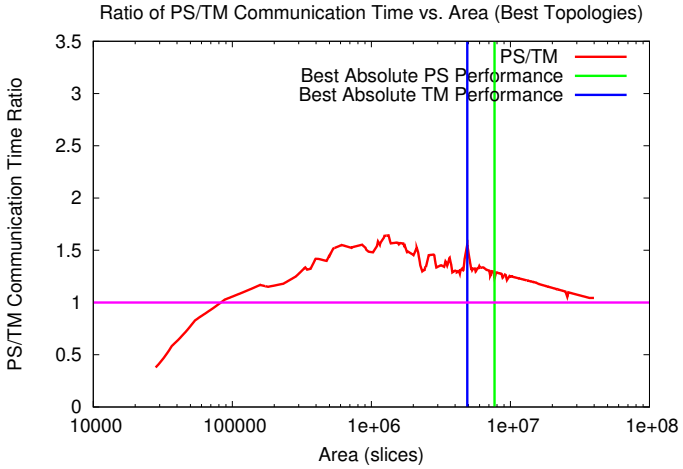


(c) PS and TM Communication Time vs. Area for Best Topologies (`gemat11`)

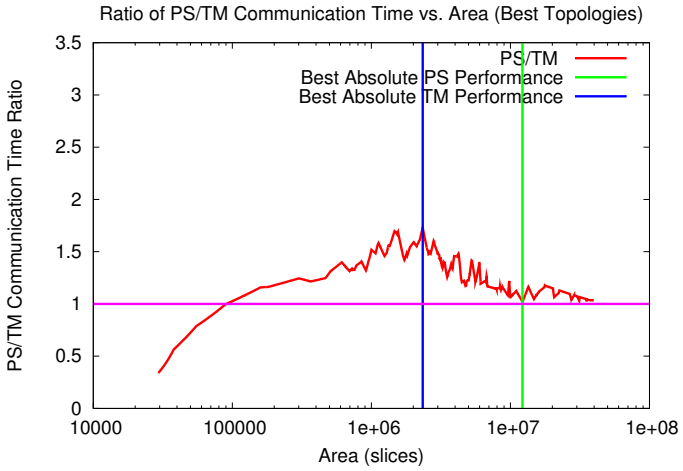
Figure 7.5: Time-Multiplexed and Packet-Switched Communication Time vs. Area (`gemat11`)



(a) `add20` (SMVM)

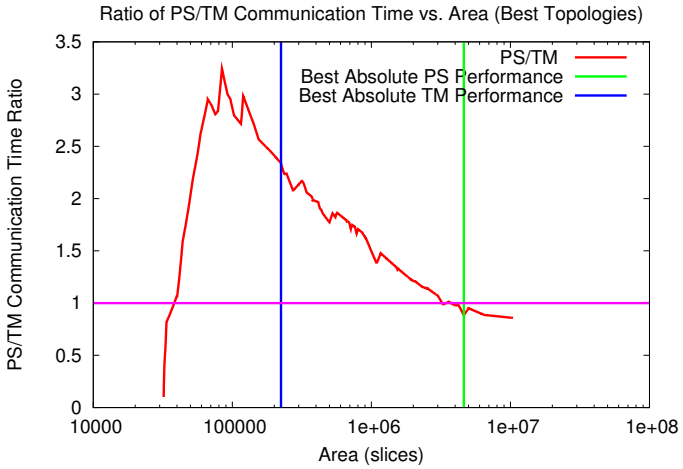


(b) `bcsstk11` (SMVM)

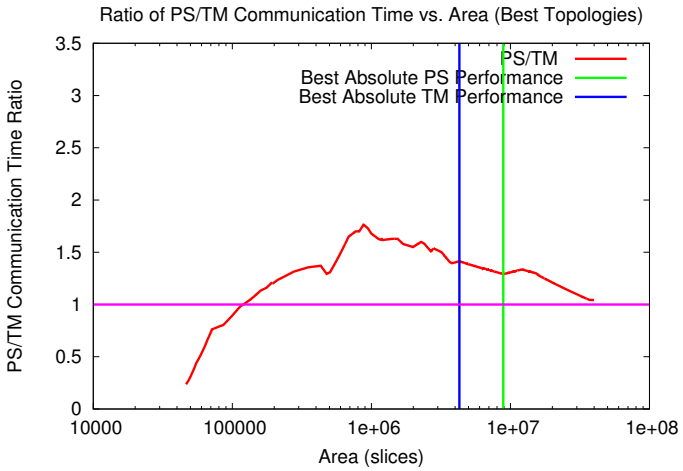


(c) `rdb32001` (SMVM)

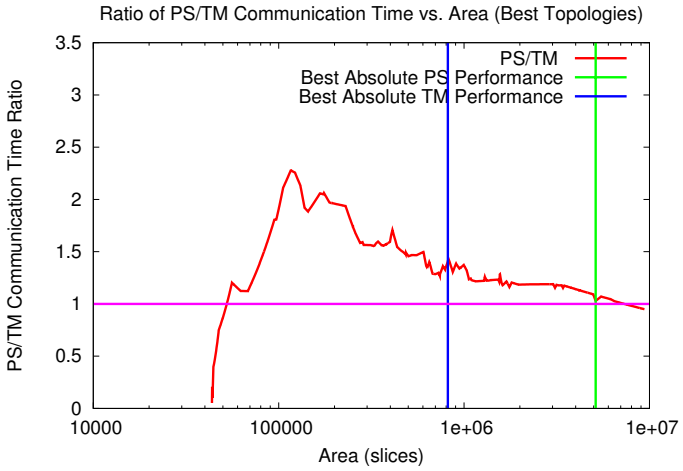
Figure 7.6: Ratio of PS/TM Communication Time vs. Area (Small Graphs)



(a) `small` (ConceptNet)

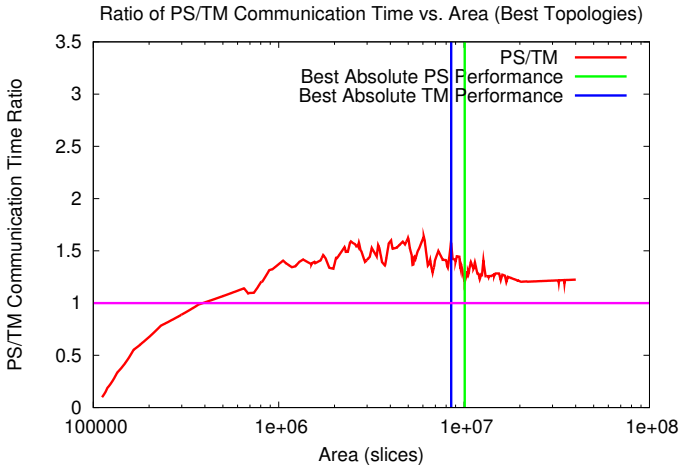


(b) `gemat11` (SMVM)

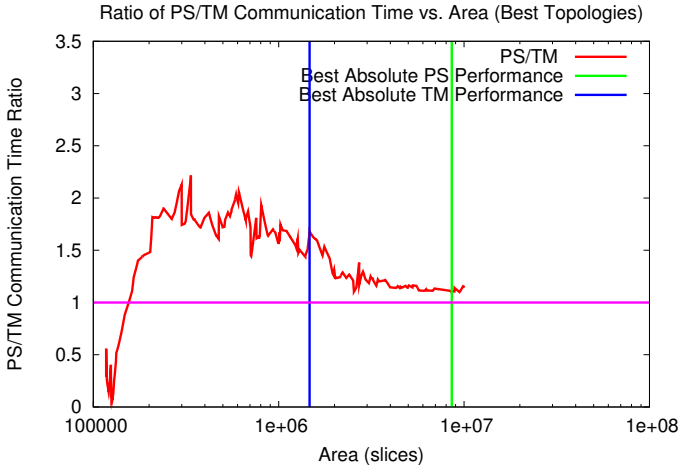


(c) `ibm01` (Bellman-Ford)

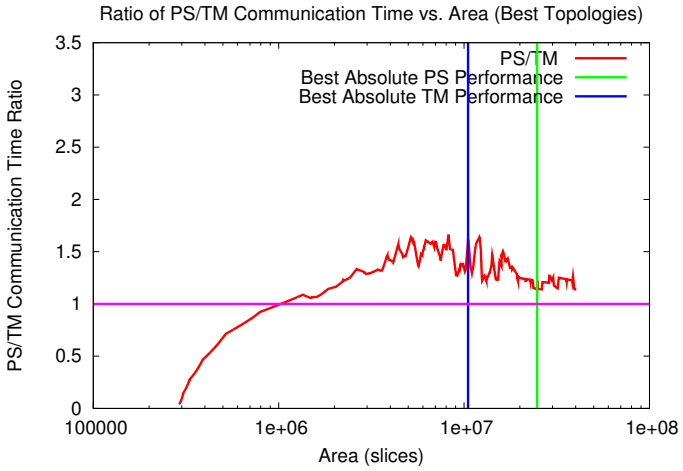
Figure 7.7: Ratio of PS/TM Communication Time vs. Area (Medium Graphs)



(a) `utm5940` (SMVM)



(b) `ibm05` (Bellman-Ford)



(c) `fidap035` (SMVM)

Figure 7.8: Ratio of PS/TM Communication Time vs. Area (Large Graphs)

For some applications at low areas this ratio approaches 0 (*e.g.* Figure 7.8(c)), indicative of packet-switching vastly outperforming time-multiplexing. As area increases, time-multiplexing outperforms packet-switching, with packet-switching requiring typically 1.5× as many cycles to route as time-multiplexing. In the best case (Figure 7.7(a)) packet-switching requires 3.4× as many cycles. As we increase area even further packet-switching is able to close the performance gap, and in some cases outperform time-multiplexing at very large areas (*e.g.* Figure 7.7(a) at 4M slices).

For all application graphs other than `small`, we see that the optimal time-multiplexed performance requires around 1M slices or more. The optimal packet-switched performance requires typically a factor of 2–10× additional area than time-multiplexing.

7.4 Cost of Routing All Possible Communication

Thus far we have compared packet-switching and time-multiplexing assuming 100% communication loads. Given 100% communication and identical networks, time-multiplexing outperforms packet-switching. However, many applications do not exhibit 100% communication, such as ConceptNet Spreading Activation (Section 5.2.1) and Bellman-Ford Shortest Path (Section 5.2.3). For example, it make take spreading activation several steps until all edges of the graph are activated; for this section we consider the early steps of real queries in spreading activation over the `small` graph. For these early steps where the activity factor is less than 100%, time-multiplexing must still route all possible communication while packet-switching only needs to route those edges that are active. At some activity factor less than 100% packet-switching should be able to outperform time-multiplexing given the same network topology.

To demonstrate where this activity crossover point can be, Figure 7.9 plots packet-switched and time-multplexed communication time as a function of activity factor for a BFT with $c = 1, p = 0.5$ and 256 PEs. Here we see that the performance of time-multiplexing is constant over all activity factors. At 100% activity packet-switching requires 1.5× as many cycles to route as time-multiplexing. However, as we decrease the activity factor we see that the number of cycles required for packet-switching decreases. At 9% activity and below packet-switching routes in fewer cycles than time-multiplexing, with time-multiplexing requiring up to 1.5× as many cycles as packet-switching in the worst case. It is this crossover point of 9% activity that we are interested in determining for all network sizes.

Figure 7.10 characterizes this crossover point across a range of PEs for `small`. At each PE count we select the optimal packet-switched and time-multiplexed topology and determine their activity crossover point. We see that at low PE counts (2–100) time-multiplexing routes 100% activity in the same number of cycles that packet-switching routes 100-60% activity or less. For these small network sizes the cost of routing all possible communication in time-multiplexing is high: packet-

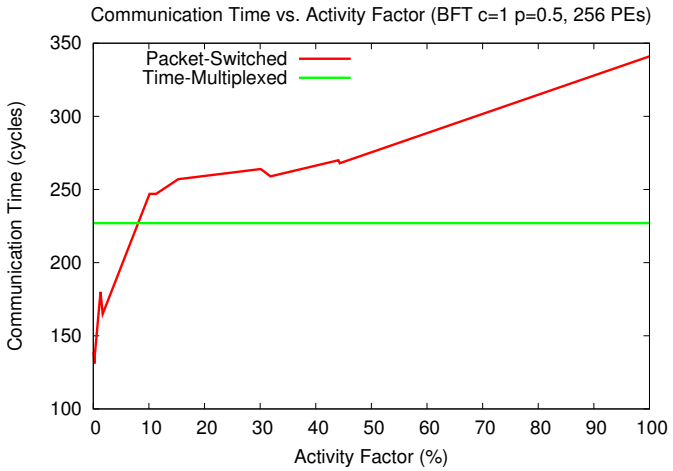


Figure 7.9: Communication Time vs. Activity Factor (`small`)

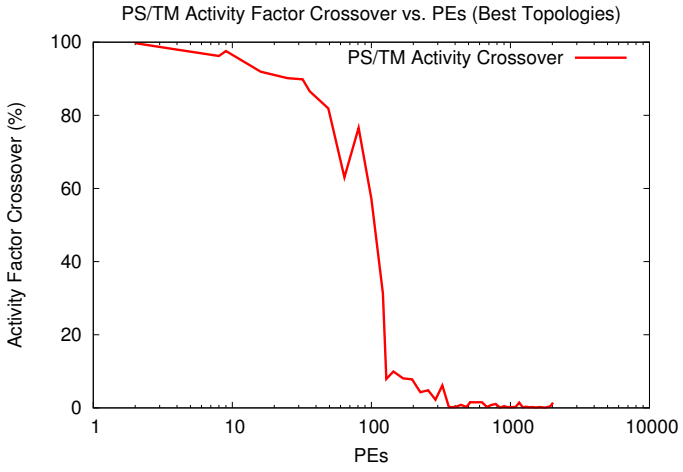


Figure 7.10: PS/TM Activity Factor Crossover vs. PEs (`small`)

switching can route nearly the same amount of communication for the same PE count, and is able to route less than 100% activity in fewer cycles. As we increase PE count above 100, we see that the activity crossover point drops abruptly to under 10%. For larger network sizes we can conclude that the cost of routing all possible communication is low. Only for activity factors under 10% can packet-switching outperform time-multiplexing. Therefore, for large networks (>100 PEs) offline routing can outperform online routing for communication loads above 10% activity.

Chapter 8

Context Memory Compression

One of the most significant disadvantages of time-multiplexed networks is the large amount of area needed for context memory. Each element in the network must store some set of configuration bits for every cycle to specify element behavior on that cycle. Every programmable switch in the network requires context memory to store switch configuration (Eqs. 3.2, 3.3, 3.4), and each PE requires context memory to indicate when to read and write to memory and at which address (Eq. 5.5).

In this chapter we provide data from preliminary attempts to reduce context memory area, both by altering our router algorithm and by changing the method of context storage.

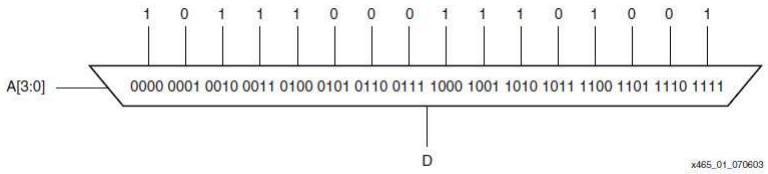
8.1 SRL16 Area Model

We store all bits of context memory in compact SRL16 storage [63], where one slice can hold 32 bits. SRL16s are simply LUTs converted into shift registers, where each LUT SRAM configuration bit is used as memory storage. These bits can be chained together to form a 16-bit shift register. We map a single bit of n cycle deep context to a chain of $\lceil \frac{n}{16} \rceil$ SRL16s, or $\lceil \frac{n}{32} \rceil$ slices. On each cycle of operation all SRL16s in the time-multiplexed network shift to provide the appropriate configuration bits for that cycle. Figure 8.1 shows how a LUT can be used as SRL16 storage, and how multiple SRL16s can be cascaded together to form arbitrarily large memories. The SRL16 is the most compact form of storage apart from BlockRAMs on Xilinx parts.

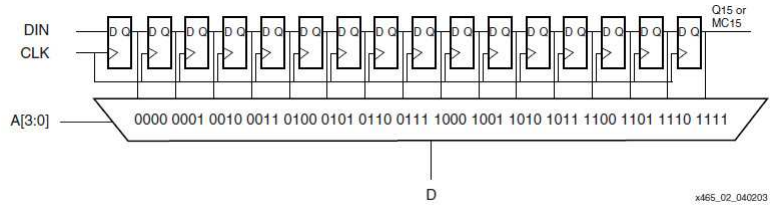
8.2 Percent Area in Context

Area in our cost model is split into three basic parts: logic, context, and pipelined interconnect. Context area is used in both switches and PEs, and depends on the following:

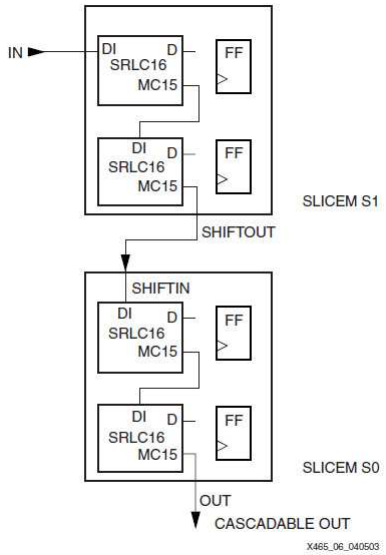
- **Switch Context:** This depends the type of switch (*i.e.* the number of merge units and the width of each unit) and the number of cycles required to route (Eqs. 3.2, 3.3, 3.4).



(a) Xilinx LUT Model



(b) Xilinx SRL16 Model

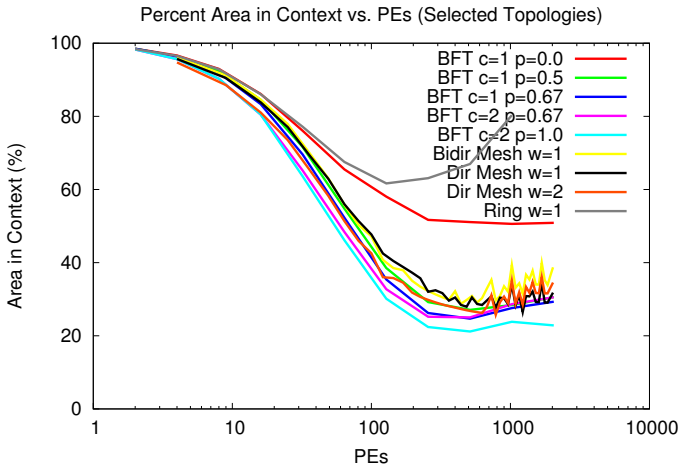


(c) Xilinx SRL16 Cascade

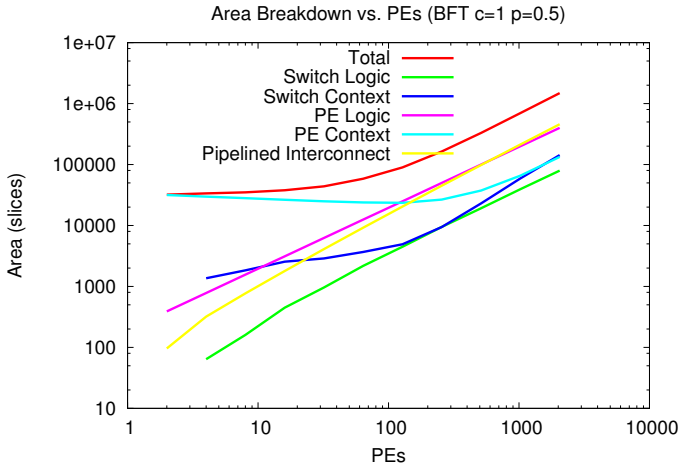
Figure 8.1: Xilinx SRL16 Structure [63]

- **PE Context:** This depends on the number of cycles required to route (Eq. 5.2), but depends much more heavily on the number of nodes and edges allocated to a given PE (Eqs. 5.1, 5.4, 5.3).

To examine how efficient our current model of storage is, we plot the area characteristics of both ConceptNet `small` and SMVM `gemat11` in Figures 8.2 and 8.3 respectively. Specifically, in Figures 8.2(a) and 8.3(a) we plot the percentage of area in context (both switch and PE) as a function of PEs across the best performing topologies. We note that these curves are consistent across all graphs in their respective applications. For `small` we see that at low PE counts (2–16 PEs), 80–100% of area is in context memory. At around 100 PEs area is evenly divided between context and logic/interconnect. At >100 PEs only 20–40% of of area is in context. For `gemat11` we typically see a smaller percentage of total area devoted to context: from 2–10 PEs context area



(a) Percent Area in Context vs. PEs

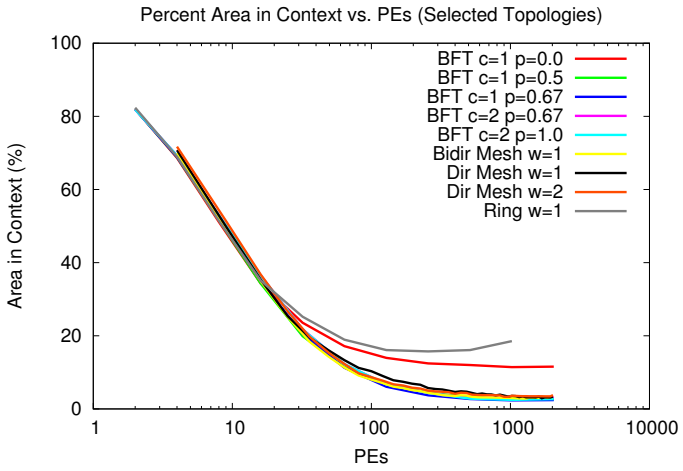


(b) Area Breakdown vs. PEs

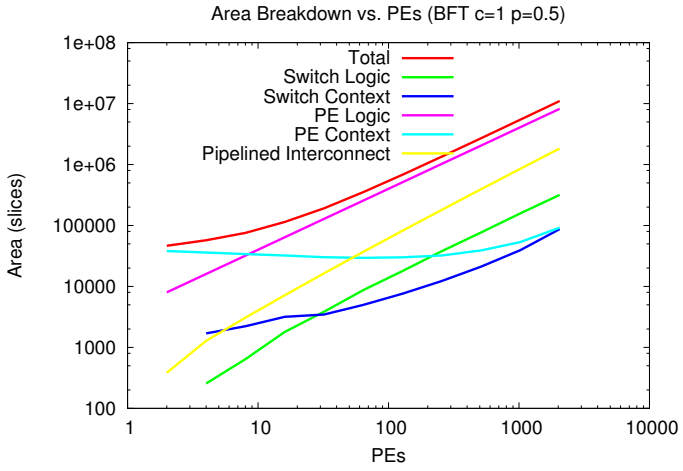
Figure 8.2: Area Characteristics (`small`)

comprises 50% or more of total area, but above 10 PEs context area drops quickly to under 10% of total area. We see that all topologies follow nearly the same curve, with low bisection topologies (ring $w = 1$, BFT $c = 1, p = 0$) requiring the most area in context.

To understand the breakdown of area in terms of switch logic, switch context, PE logic, PE context, and pipelined interconnect, in Figures 8.2(b) and 8.3(b) we plot individual components of area as a function of PEs for `small` and `gemat11`. Here we focus on a single BFT $c = 1, p = 0.5$ topology as all topologies exhibit similar breakdowns. For `small` we see that from 1–100 PEs area is dominated by PE context, specifically by the context required for memory addressing. Above 100 PEs both PE logic and pipelined interconnect begin to dominate equally. For `gemat11`, we see that from 1–8 PEs area is dominated by PE context, but above 8 PEs area is dominated solely by PE logic. This is because the SMVM PE is significantly larger than the ConceptNet/Bellman-Ford PEs (4000 vs. 200 slices). Therefore, we observe that attempting to reduce context area in SMVM



(a) Percent Area in Context vs. PEs



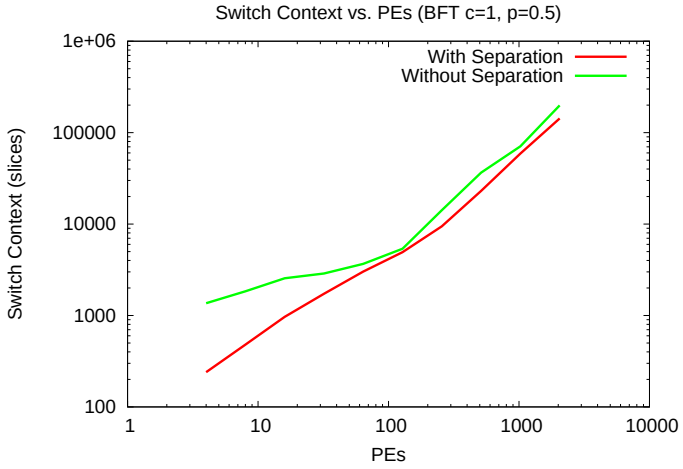
(b) Area Breakdown vs. PEs

Figure 8.3: Area Characteristics (`gemat11`)

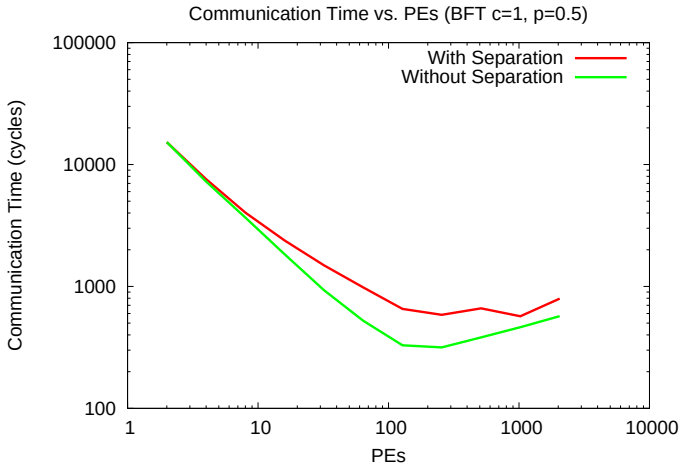
networks will yield diminishing returns. We therefore focus our reduction techniques on `small`.

8.3 Separating External and Self Messages

While switch context area only comprises a small percentage of total area compared to PE context and PE logic, we can make a simple change to our router to experiment in reducing switch context. We make the simple observation that context only needs to be stored in switches for period of time where external messages are routed. That is, if we have a hypothetical communication load where all messages are self messages, no messages will traverse the network and therefore switches require 0 bits of context. For low PE counts self messages account for the vast majority of traffic (Figure 5.1); therefore, we can attempt to route self messages separately from external messages. We will then only need switch context for the number of cycles required to route all external messages separately.



(a) Switch Context Area vs. PEs with and without Separation



(b) Communication Time vs. PEs with and without Separation

Figure 8.4: Routing External and Self Messages Separately (`small`)

In Figure 8.4 we plot the results of routing `small` with and without separation of external and self messages on a BFT $c = 1, p = 0.5$. In Figure 8.4(a) we see that routing with separation decreases total switch context area from 2–100 PEs by almost an order or magnitude in the best case, but above 100 PEs area savings diminish. We note that routing self and external messages separately introduces new constraints to our greedy router which will cause total communication time to change. Figure 8.4(b) compares communication time vs. PEs for routing with and without separation. We see that above 16 PEs the penalty for routing with separation is significant, costing almost $2\times$ as many cycles to route as no separation in the worst case. However, the difference in cycles for <16 PEs is negligible. Therefore, routing with separation reduces switch context area at low PE counts without affecting performance.

8.4 Context Compression with `espresso` and `jedi`

The disadvantage with our current model of storing context in SRL16s is that configuration bits must be stored for all elements in every cycle, regardless if all elements are performing necessary switching operations on every cycle. That is, if a switch is not routing a packet, the value of its merge configuration bits do not matter. If these "don't care" situations occur often with network elements, storing context bits for all elements in every cycle is inefficient.

Instead of storing context information in SRL16s, we make the observation that context memory just provides a simple mapping from cycle number to some number of configuration bits. This mapping could be implemented using combinational logic just as easily as SRL16 memories. This requires that for each network element we synthesize a specialized piece of logic that simply implements the correct mapping function (*i.e.* truth table) from cycle number to configuration bits. The advantage of such an approach is that if there are a large number of "don't cares" or enough structure (*i.e.* runs of 0s and 1s) in the output bits (*i.e.* configuration bits) of the truth table, we can use conventional two-level logic optimization tools to greatly reduce the size of the circuit required to implement the truth table. The area of such a circuit may in fact be smaller than the area of SRL16 storage.

To explore this approach, we show some preliminary data on compressing only switch context for `small` on a BFT with $c = 1, p = 0.5$ and 8 PEs. To perform context compression we create truth tables for each switch which maps cycle number to switch configuration, where we explicitly specify both the input cycle bits the output binary configuration bits. For unused configuration bits we specify don't cares. We run this truth table through the `espresso` two-level boolean optimization tool [50] to reduces the number of pterms (*i.e.* cycle to configuration rows in the truth table). We then create VHDL for each minimized truth table that we synthesize (Section 5.6) to obtain area in slices for the context circuit.

Table 8.4 shows the results of this experiment. We note that switches at height 2 do not contain context memory as they do not need to make switching decisions (*i.e.* they are just pipelined, configured wires). We see that while `espresso` is able to reduce the number of pterms per switch by nearly an order of magnitude, this reduction in pterms does not translate to a significant reduction in synthesized area. Because this BFT configuration is serialization limited, certain switches (*e.g.* (1,1)) are underutilized due to greedy decisions made by the router and their area is reduced by a factor of 4. However, switches that experience more heavy traffic (*e.g.* (2,1)) actually increase in area after synthesis. We see that it is difficult to store context memory in more compact storage than the SRL16.

These results are preliminary and we hope that with changes to our toolflow (using `jedi` for example), and with attempts in reducing PE context in addition to switch context that we may

Switch (x, height)	Before Compression		After Compression	
	Pterms	Area (slices)	Pterms	Area (slices)
0, 0	3674	818	317	766
1, 0	3674	818	333	775
2, 0	3674	818	310	811
3, 0	3674	818	258	601
0, 1	3674	417	338	794
1, 1	3674	417	53	167
2, 1	3674	417	336	844
3, 1	3674	417	50	179
0, 2	0	0	0	0
1, 2	0	0	0	0
Total	29392	4940	1995	4937

Table 8.1: `espresso` Switch Context Compression (BFT $c = 1, p = 0.5$ with 8 PEs for `small`)

eventually see total system area savings through context compression. See Section 9.3 for more comments on future work.

Chapter 9

Future Work

9.1 Communication Patterns

9.1.1 Spatially Configurable and Circuit-Switched Interconnect

Thus far we have compared time-multiplexing to only packet-switching. To fully characterize the design space of time-multiplexing for NoCs, we need to compare it to both spatially configurable interconnect and circuit-switched interconnect. Some work has been done in characterizing the design space of circuit-switched NoCs for FPGAs [22]. For applications that send significantly longer messages than those we have considered thus far, circuit-switching may be useful. For the kinds of sparse-graph algorithms we are in general interested in, we believe that spatially configurable interconnect may not be appropriate. In terms of area, in configurable interconnect a network link must be allocated for all graph nodes that are communicating. Therefore, the number of wires in the system scales according to the number of edges in the application graph. For large graphs this area cost may be insurmountable. In terms of performance, physical configured links offer minimum end to end latency and very high throughput. However, since a message will only be sent on an edge once during a single GraphStep (potentially 100s-1000s of cycles), physical links will only be utilized every 100s-1000s of cycles, dramatically underutilizing their maximum throughput. However, we would like to further quantify these area and performance tradeoffs.

9.1.2 Additional Workloads

Communication patterns and densities can vary greatly between different applications. Exploration of more real applications would help us to better characterize the tradeoffs between packet-switching and time-multiplexing for any given general application. We are particularly interested in mapping larger communication graphs with smaller fanout limitations in order to increase network communication and test the capabilities of our networks.

9.2 Network Topologies

9.2.1 Additional Topologies

Certain topologies that researchers have previously dismissed for multiprocessor networks (*e.g.* the Hypercube, Figure 3.2(c) [12]) may be worth revisiting under the NoC cost model. As NoCs differ from multiprocessor systems (Section 1.2.1), previously dismissed topologies may perform well under an appropriately revised cost model.

9.2.2 Automated Topology Synthesis

Since no single topology performs optimally over all applications and all areas (Section 6.3), ideally, one would want to synthesize a different topology for each application. Therefore, we are motivated to automate the process of topology selection based on a given set of parameters such as application workload characteristics, desired performance, chip area, and technology cost model. Using a reconfigurable FPGA substrate greatly facilitates this application-specific customization.

9.2.3 Additional Cost Models

The specific cost model that we examine is an FPGA overly NoC model for the Xilinx XC2V6000-4. However, the areas that we consider for our single chip networks greatly exceed the maximum capacity of this device (32K slices). As the Virtex2 is a relative outdated part, and new devices provide increases in speed, capacity, and features, extrapolating the performance of our networks to large areas under the XC2V6000-4 cost model is conservative. The question remains, however: at what point will the areas that we consider (from 10K–10M slices) become feasible for a single chip?

In terms of current technology, the recently announced Xilinx Virtex5 [63] provides far greater capacity than the Virtex2. Area in the Virtex2 is measured in slices, where a single slices is equivalent to 2 4-input LUTs with registers. Area in the Virtex5 is also measured in slices, but here a single slice is equivalent to 4 6-input LUTs with register. We can very conservatively assume that a single Virtex5 slice provides equal or greater capacity than two Virtex2 slices. The largest Virtex5 parts provide 50K slices of logic, at least as much capacity as 100K Virtex 2 slices. Therefore, chips with capacities of more than 100K slices are currently available. If chip scaling continues to follow Moore's law (capacity doubles every 24 months), we can expect capacities of 1M and 10M slices in roughly 6–8 years and 14–16 years respectively.

Additionally, we have limited our exploration to an FPGA overlay NoC cost model due to the advantages of FPGA NoCs over ASIC NoC (Section 1.2.1) and the ease of mapping our networks directly to hardware. We would like to revisit our analysis under an ASIC cost model to examine how our conclusions might differ in a revised area-time model for switches and interconnect. This may

provide insight as to the advantages of building FPGA with hardwired support for time-multiplexed interconnect. We would also like to compare the performance of our network to that of previous ASIC networks to examine the advantages in designing a simple, highly pipelined network.

9.2.4 Multiple-Chip Network

Our network design space exploration has thus far been limited to single chip networks. A fairly simple extension of this work would be to model multiple-chip networks, examining similar area-time tradeoffs in communication patterns and topologies (both inter-chip and intra-chip). Preliminary switchboxes to handle offchip traffic have already been designed and tested.

9.3 Context Compression

9.3.1 PE Context

Thus far we have only provided preliminary results for compression switch context. While these results are not yet promising, we should shift our focus to PE context compression as it constitutes a larger percentage of total area.

9.3.2 Compressing with `jedi`

One issue with our approach thus far is that we explicitly specify the configuration bits for switches, limiting the freedom of `espresso` to perform minimization. To increase freedom we could enumerate states for each legal switch configuration and let the tool choose an appropriate minimal encoding. `jedi` [39] allows us to specify output states instead of explicit output bits and then attempts to find an optimal encoding. We have yet to explore this technique but we believe that it may yield improved results over the `espresso` approach.

9.4 Router Improvements

9.4.1 Routing Fanout

Currently our router routes messages assuming point-to-point communication. However, for some applications the same message can be sent to multiple destination PEs (*e.g.* Bellman-Ford). Under these conditions routing messages without taking advantage of fanout can be extremely inefficient for high fanout messages. We expect significant performance gains by routing fanout explicitly.

9.4.2 Pathfinder Quantification

The quality of our router is adequate for the applications that we examine (Section 4.4). To obtain additional quality we plan to explore Pathfinder. Pathfinder yields higher quality results than greedy methods, but at a significant runtime cost with not guarantee of convergence. We plan to explore comparing our router to Pathfinder to help quantify the tradeoffs in quality and runtime. Our greedy router will aid in this exploration as it provides and achievable lowerbound for Pathfinder.

Chapter 10

Conclusions

In this work we characterized the design space of Time-Multiplexed interconnect for FPGA overlay networks on chip and compared it to Packet-Switched Interconnect. We described how to build well engineered, highly scalable time-multiplexed FPGA networks in terms of topology selection, router algorithm design, and hardware design.

We showed how to design networks out of simple, high performance network primitives and pipelined interconnect. These allow us to compose our network hardware elements out of small blocks than can easily be pipelined for high performance. We demonstrated network performance of >200MHz on a Xilinx XC2V6000-4, with a total system performance of 166MHz. We note that total system performance is limited not by network timing but by critical paths within the processing elements.

To benchmark our networks we mapped several real applications with heavy communication requirements (ConceptNet Spreading Activation, Sparse Matrix-Vector Multiply, Bellman-Ford Shortest Path) to the GraphStep system architecture. For each application we selected graphs that span a range of sizes and communication requirements. To route these workloads, we designed a greedy routing algorithm that routes in the worst case within 3.6× as many cycles as the lowerbound, across all topologies, network sizes, and applications.

To determine the best time-multiplexed topology, we performed experiments over rings, meshes, and BFTs over a wide range of network sizes (2–2048 PEs) and areas (10K–10M slices). We demonstrated that performance differences between topologies begin to emerge only at large networks sizes, emphasizing the importance of examining large-scale networks. We showed that in general as area increases, topologies which provide a higher percentage of area in interconnect rather than compute provide the best performance. Both meshes and BFTs of various configurations provide the best performance at large chip sizes, while their relative performance is within a factor of 2×. We showed that over all areas (10K–10M slices) and over all applications the best "one topology fits all" configuration is Butterfly Fat Tree (BFT) with $c = 1, p = 0.5$, which requires, in the worst case, 6.1× as many cycles to route communication than the optimal topology. This indicates that there

is non-trivial cost in selecting a single topology for all areas and applications.

We compared time-multiplexing to packet-switching, and showed that on average, over all applications for all equivalent topologies, online packet-switched communication requires 1.5× as many cycles to route as offline time-multiplexed scheduling. When applying designs to equivalent area, for areas typically <100K slices packet-switching typically outperforms time-multiplexing, but at >100K slices packet-switching requires up to 3.4× as many cycles to route as time-multiplexing in the worst case. Finally, for equivalent large networks (>100 PEs) time-multiplexing outperforms packet-switching when routing communication loads where greater than 10% of all logical links are active. This demonstrates that well designed time-multiplexed FPGA overlay networks can deliver performance and area efficiency exceeding that of packet-switched networks.

Bibliography

[1] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. A. Zeferino. SPIN: a Scalable, Packet Switched, On-chip Micro-network. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*, pages 1128–1129, 2003.

[2] C. J. Alpert. The ISPD98 Circuit Benchmark Suite. In *Proceedings of the International Symposium on Physical Design*, pages 80–85, 1998.

[3] ARM. *AMBA Bus specification*, 1999.

[4] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142–151, April 1993.

[5] V. Beneš. Permutation Groups, Complexes, and Rearrangeable Multistage Connecting Networks. *Bell System Technical Journal*, 43:1619–1640, July 1964.

[6] V. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic.* Academic Press, New York, NY, 1965.

[7] L. Benini and G. D. Micheli. Networks on Chips: A New SOC Paradigm. *IEEE Computer*, 35(1):70–78, 2002.

[8] N. B. Bhat, K. Chaudhary, and E. S. Kuh. Performance-Oriented Fully Routable Dynamic Architecture for a Field Programmable Logic Device. UCB/ERL M93/42, University of California, Berkeley, June 1993.

[9] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of Supercomputing*, pages 330–339, November 1988.

[10] A. Caldwell, A. Kahng, and I. Markov. Improved Algorithms for Hypergraph Bipartitioning. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 661–666, January 2000.

[11] C. Clos. A Study of Non-Blocking Switching Networks. *Bell System Technical Journal*, pages 406–424, March 1953.

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

[13] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Design Automation Conference*, pages 684–689, 2001.

[14] A. DeHon. Reconfigurable Architectures for General-Purpose Computing. AI Technical Report 1586, MIT Artificial Intelligence Laboratory, 545 Technology Sq., Cambridge, MA 02139, October 1996.

[15] M. deLorimier and A. DeHon. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 75–85, February 2005.

[16] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, Jr., and A. DeHon. GraphStep: A System Architecture for Sparse-Graph Algorithms. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006.

[17] M. Denneau. The Yorktown Simulation Engine. In *19th Design Automation Conference*, pages 55–59. IEEE, 1982.

[18] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers, San Francisco, 2003.

[19] S. Even, A. Itai, and A. Shamir. On the Complexity of Timetable and Multicommodity Flow Problems. *SIAM Journal of Computing*, 5(4):691–703, November 1967.

[20] R. I. Greenberg and C. E. Leiserson. *Randomness in Computation*, volume 5 of *Advances in Computing Research*, chapter Randomized Routing on Fat-Trees. JAI Press, 1988. Earlier version MIT/LCS/TM-307.

[21] A. K. Gupta and W. J. Dally. Topology Optimization of Interconnection Networks. *Computer Architecture Letters*, 4, July 2005.

[22] C. R. Hilton. *A Flexible Circuit-Switched Communication Network For FPGA-Based SoC Design*. PhD thesis, Brigham Young University, 2005.

[23] W. H. Ho and T. M. Pinkston. A Design Methodology for Efficient Application-Specific On-Chip Interconnects. *IEEE Transactions On Parallel and Distributed Systems*, 17(2):174–190, February 2006.

[24] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1996.

[25] IBM. *CoreConnect Specification*, 1999.

[26] D. Jones and D. Lewis. A Time-Multiplexed FPGA Architecture for Logic Emulation. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 495–498. IEEE, May 1995.

[27] N. Kapre. Packet-Switched On-Chip FPGA Overlay Networks. Master's thesis, California Institute of Technology, May 2006.

[28] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon. Packet-Switched vs. Time-Multiplexed FPGA Overlay Networks. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006.

[29] F. Karim, A. Nguyen, and S. Dey. An Interconnect Architecture for Networking System on Chips. *IEEE Micro*, 22(5):36–45, September/October 2002.

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

[31] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. berg, K. Tiensyrj, and A. Hemani. A Network on Chip Architecture and Design Methodology. In *International Symposium on VLSI*, pages 117–124, 2002.

[32] B. S. Landman and R. L. Russo. On Pin Versus Block Relationship for Partitions of Logic Circuits. *IEEE Transactions on Computers*, 20:1469–1479, 1971.

[33] C. Lee. An Algorithm for Path Connections and its Applications. *IEEE Transactions on Electronic Computers*, EC-10(2), 1961.

[34] S.-J. Lee, K. Lee, S.-J. Song, and H.-J. Yoo. Packet-Switched On-Chip Interconnection Network for System-On-Chip Applications. *IEEE Transactions on Circuits & Systems*, 52(6):308–312, June 2005.

[35] T. Leighton and S. Rao. Multicommodity Max-Flow Min-Cut Theorems and Their Use in Designing Approximation Algorithms. *ACM*, 46(6):787–832, November 1999.

[36] C. Leiserson, F. Rose, and J. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Third Caltech Conference On VLSI*, March 1983.

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

[38] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier. An Architecture and Compiler for Scalable On-Chip Communication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(7):711, 726 2004.

[39] B. Lin and R. Newton. Synthesis of Multiple-Level Logic from Symbolic High-Level Description Languages. In *Proceedings of the IFIP International Conference on VLSI*, pages 187–196, 1989.

[40] H. Liu and P. Singh. ConceptNet – A Practical Commonsense Reasoning Tool-Kit. *BT Technical Journal*, 22(4):211, October 2004.

[41] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. B. W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Run-Time Support for Heterogeneous Multitasking on Reconfigurable SoCs. *INTEGRATION, The VLSI Journal*, 38(1):107–130, 2004.

[42] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 111–117. ACM, February 1995.

[43] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The Nostrum Backbone - a Communication Protocol Stack for Networks on Chip. In *International Conference on VLSI Design*, 2002.

[44] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip. *INTEGRATION, The VLSI Journal*, 38(1):69–93, 2004.

[45] S. Murali and G. D. Micheli. SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs. In *Proceedings of the ACM/IEEE Design Automation Conference*, 2004.

[46] Nallatech. Simplifying Communication Across DSP Networks. Programmable World, 2003. <`http://www.mactivity.com/xilinx/pw2003/workshops/presos/wsa3_nallatech.pdf`> .

[47] NIST. Matrix Market. <`http://math.nist.gov/MatrixMarket/`> , June 2004. Maintained by: National Institute of Standards and Technology (NIST).

[48] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. W. J. IV, V. Akella, and F. T. Chong. Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor. In *Proceedings of the International Symposium on Computer Architecture*, 2004.

[49] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance Evaluation and Design Trade-offs for Network-on-chip Interconnect Architectures. *IEEE Transactions on Computers*, 54(8):1025–1040, August 2005.

[50] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems*, 6(5):727–751, September 1987.

[51] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, pages 22–33, January 1985.

[52] C. Selvidge, A. Agarwal, M. Dahl, and J. Babb. TIERS: Topology IndependEnt Pipelined Routing and Scheduling for VirtualWire^{TM} Compilation. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 25–31, 1995.

[53] B. Sethuraman, P. Bhattacharya, J. Khan, and R. Vemuri. LiPaR: A LightWeight Parallel Router for FPGA based Networks on Chip. In *Proceedings of the Great Lakes Symposium on VLSI*, 2005.

[54] D. Shoemaker, F. Honore, C. Metcalf, and S. Ward. NuMesh: An architecture optimized for scheduled communication. *Journal of Supercomputing*, 10(3):285–302, 1996.

[55] H. Siegel. *Interconnection Networks for Large-Scale Parallel Processing.* Lexington Books, Lexington, MA, 1985.

[56] R. Tessier. Negotiated A* Routing for FPGAs. In *Proceedings of the 5th Canadian Workshop on Field Programmable Devices*, June 1998.

[57] C. D. Thompson. A Complexity Theory of VLSI. Technical Report CMU-CS-80-140, Department of Computer Science, Carnegie-Mellon University, 1980.

[58] D. S. Tortosa and J. Nurmi. Proteo: A New Approach to Network-on-Chip. In *Proceedings of International Conference on Communication Systems and Networks*, 2002.

[59] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, April 1997.

[60] G. V. Varatkar and R. Marculescu. On-Chip Traffic Modeling and Synthesis for MPEG-2 Video Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(1), January 2004.

[61] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Micro*, 30(9):86–93, September 1997.

[62] D. Wingard. MicroNetwork-Based Integration for SOCs. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 673–677, June 2001.

[63] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *The Programmable Logic Data Book-CD*, 2005.