On-chip Networks for Manycore Architecture

by

Myong Hyon Cho

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author .....
Department of Electrical Engineering and Computer Science
August 20, 2013

Certified by .....
Srinivas Devadas
Edwin Sibley Webster Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

On-chip Networks for Manycore Architecture

by

Myong Hyon Cho

Submitted to the Department of Electrical Engineering and Computer Science
on September 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Over the past decade, increasing the number of cores on a single processor has successfully enabled continued improvements of computer performance. Further scaling these designs to tens and hundreds of cores, however, still presents a number of hard problems, such as scalability, power efficiency and effective programming models.

A key component of manycore systems is the on-chip network, which faces increasing efficiency demands as the number of cores grows. In this thesis, we present three techniques for improving the efficiency of on-chip interconnects. First, we present PROM (Path-based, Randomized, Oblivious, and Minimal routing) and BAN (Bandwidth Adaptive Networks), techniques that offer efficient intercore communication for bandwith-constrained networks. Next, we present ENC (Exclusive Native Context), the first deadlock-free, fine-grained thread migration protocol developed for on-chip networks. ENC demonstrates that a simple and elegant technique in the on-chip network can provide critical functional support for higher-level application and system layers. Finally, we provide a realistic context by sharing our hands-on experience in the physical implementation of the on-chip network for the Execution Migration Machine, an ENC-based 110-core processor fabricated in 45nm ASIC technology.

Thesis Supervisor: Srinivas Devadas
Title: Edwin Sibley Webster Professor

Acknowledgments

This thesis would not be complete without offering my sincere gratitude to those who motivated, guided, assisted, and supported me during my Ph.D. study.

Foremost, words cannot do justice to express my deepest appreciation to Professor Srinivas Devadas. I respect him not only for his intelligence prowess but also for his kindness, patience, and understanding. I feel fortunate to have had the opportunity to study under his guidance; what he showed and taught me will continue to guide me even after graduation.

I am deeply honored to have Professor Joel Emer and Professor Daniel Sanchez on my committee. Professor Emer has always been such an inspiration to me for his continuing achievements in this field. I am also grateful for Professor Sanchez, whose work I have always admired even before he came to MIT. Both professors provided invaluable feedback and suggestions during the development of this thesis.

I cannot thank my fellow students enough. I was able to perform at my best because I knew they were doing the same. While all of the students in the Computation Structures Group at MIT deserve my appreciation, I reserve the most heartfelt gratitude for Keun Sup Shim and Mieszko Lis for being great friends and truthful colleagues. Also, when I was faced with the formidable task of building a 110-core processor, Sunghyun Park, Owen Chen, Chen Sun and Yunsup Lee (at U.C. Berkeley) surprised me by helping me in every possible way they could.

I have to leave a special thanks to Google, because quite honestly, there is no way I could have done this research without it. I also thank 1369 Coffee House, Neptune Oyster, Five Guys, Toscanini's, Boston Symphony Orchestra, and Catalyst, for helping me keep my faith in the belief that my life is so awesome. I would like to extend my special gratitude to Samsung Scholarship for financially supporting the first two years of my Ph.D. program.

Saving the best for last, I would like to thank the most special people in my life. Above all, my wife Yaejin Kim who never let me think that I was alone, gave me the strength to carry on even in the hardest time. Although I cannot find words to

express how deeply grateful I am to her, I hope she will be able to find out as we share the rest of our lives together. I also thank my one month old son, Allen Minjae Cho, who made me realize his dad cannot be a graduate student forever. I am so genuinely grateful to Haram Suh, for always being a truly faithful friend, no matter how far apart. Finally, I would like to send my gratitude to my family in South Korea; my parents, Gieujin Cho and Kyoungho Shin, have never lost their faith in me in all these years, and I owe everything I am to them. I am also much obliged to my parents-in-law, Chungil Kim and Mikyoung Lee, for always believing in me and treating me like their own son.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Chip Multiprocessors: Past, Present and Future

1.1.1 The Birth of Multicore

Until the end of the 20th century, the scaling down of CMOS devices was the driving force behind continued improvement in computer performance [82]. There were two fundamental impacts; first, microprocessors were able to embed more sophisticated functionality with smaller transistors to accelerate complex operations. And second, clock speeds increased by orders of magnitude, thanks to smaller and faster CMOS devices. Dennard's scaling theory [19] showed when transistor dimensions scaled down by a factor of α, the power density remained the same so the required supply voltage decreased by α. Therefore, the power required to maintain the same frequency for the same logic was reduced by α^2, and we were able to increase the frequency for better performance without using more power. Dennard scaling enabled processor performance to improve 52% annually for nearly two decades since 1985 [35].

The relentless scaling down of transistors, per Moore's law [63], still enables more transistors on recent microprocessor designs. Since the 90nm technology node, however, the long-lasting race of clock speed improvements came to an abrupt end. This

was mostly due to leakage current becoming a major contributor to power consumption, and because supply voltage stopped scaling with the channel length [77]. The inability to scale down the supply voltage caused higher energy consumption at higher clock frequency, and power and cooling limitations prohibited further frequency increases.

The continuation of Moore's law and the demise of Dennard's scaling theory forced changes in the old strategy of smaller transistors, lower supply voltage, and higher clock rate [29]. As it became infeasible for single-core performance to benefit from a higher frequency, the microprocessor industry quickly adapted to another strategy: chip multiprocessors (CMPs). As Moore's law continues to hold, manufacturers were able to duplicate previously designed cores on a single die with less effort than increasing the frequency. At the same time, new applications such as image processing, video encoding/decoding, and interactive applications with graphical user interface, have a lot of thread-level parallelism that CMPs can easily exploit to boost the performance. Consequently, multicore has quickly made its way into the mainstream since it was first introduced to the market in 2005 [21, 74, 75].

1.1.2 From Multicore to Manycore

Eight years have passed since Intel shipped the first x86 dual-core processor in 2005, and current trends clearly indicate an era of multicores. In 2013, commercially available x86 desktop processors have up to eight cores per chip, server processors up to ten. Even mobile processors also have up to eight cores on a single chip. The big question is, will the number of cores keep increasing to tens and hundreds?

1.1.2.1 Technical Issues of Manycore Architecture

We are at the best time to ask this question, because more breakthroughs are required than ever before in order to further increase the number of cores in a multicore processor. For example, a few cores can be connected using a totally ordered bus interconnect that enables the implementation of cache coherence with a snooping protocol.

However, a simple common bus is not scalable to more than eight cores [67]. Intel actually replaced its Front-Side Bus (FSB) with the point-to-point Intel QuickPath Interconnect (Intel QPI) for Nehalem [68], which is Intel's first microarchitecture that supports up to eight cores per chip. And as more scalable on-chip networks are used, the lack of ordering complicates the implementation of cache coherence [83].

The implementation of cache coherence is an example of the scalability issues on manycore architecture. Another issue is the gap between the speed of cores and external memory, commonly known as the memory wall [94]. The gap has become significantly wider in multicore as multiple cores share the off-chip memory interface [47]. The power wall also plays an important role as the number of cores grows; due to the fixed power budget, 21% of a chip must be powered off at all times at the 22nm technology node [25]. These issues prevent the maximum theoretical speedup of multicore processors, leaving challenges for researchers in achieving continuing performance improvements.

Innovations are also required in the software development. To fully benefit from large-scale CMPs, applications need a sufficient level of thread-level parallelism. Efficient parallel programming models must be provided to ease the difficulties of writing massively threaded applications. Due to the prevalence of single-threaded applications, manual or automatic extraction of thread-level parallelism from existing programs is also very important [8].

1.1.2.2 Why Manycore Architecture is Still the Most Promising Solution

Despite all these difficult challenges, however, manycore architecture is still the most viable option to maintain the growth in performance. Recall that the number of cores has increased mainly because 1) we could not improve the clock speed without hitting the power limit, but 2) we could still have more transistors in the same area. If an alternative technology can realize higher frequency devices than silicon-based technology, then we might not need manycore processors to get more performance. On the other hand, if it becomes impossible to further scale down transistor dimensions, it would be infeasible to increase the number of cores beyond the current trend. Will

either of these two scenarios be the case in the near future?

As a matter of fact, good progress has been made in the research of non-silicon based technologies in the hope that one such technology would eventually overcome the speed limit of silicon-based computers. For example, field-effect transistors (FETs) that work with a cut-off frequency as high as 100 GHz have been fabricated on a graphene wafer [52]. However, none of the alternative technologies has yet surmounted the hurdles of mass production to match the outstanding technological maturity of silicon processes. As a result, it is expected to take another couple of decades before such technologies reach to the consumer market [15, 30].

The number of transistors on a chip has kept increasing as well. In 2011, Intel integrated 3.1 billion transistors on a $18.2 \times 29.9 mm^2$ die using 32nm technology. One year later, Intel put almost 5 billion transistors on the Xeon Phi coprocessor using 22nm technology [42]. In case of GPUs, NVIDIA crammed 7.1 billion transistors into its GK110 chip using the 28nm node in 2012 [13]. As the current process technology roadmap predicts [28], Moore's law will continue for at least another half a decade, until the point where transistors get four times smaller.

In summary, having more cores still remains a feasible and attractive solution because the reasons why multicore processors came into place still hold. The lack of alternatives urges researchers to address the technical issues of manycore processors and extend its practicality. Section 1.1.2.3 illustrates recent efforts to transit from a few cores to many cores.

1.1.2.3 Examples of Manycore Processors

Intel has long been trying to develop a practical manycore architecture; in 2008, Intel demonstrated its 80-core Polaris research processor with simple VLIW cores and a high-performance 8×10 2-D mesh network [89]. Two years later, the Single-Chip Cloud Computer (SCC) followed, which contained 48 full-fledged P54C Pentium cores [38]. Finally, the 60-core Xeon Phi coprocessor was released to the consumer market in 2012 [12].

Tilera Corporation has focused on manycore general-purpose processors since its

Name	#Cores	Technology	Frequency	Power	Year
Intel Polaris*	80	65nm	4.27GHz	97W	2008
Intel Single-Chip * Cloud Computer (SCC)*	48	45nm	1.0GHz	125W	2010
Intel Xeon E7-8870	10	32nm	2.4GHz	130W	2011
AMD FX-8350	8	32nm	4.0GHz	125W	2012
Tilera TILE-Gx36	36	40nm	1.2GHz	28W	2013
Intel Xeon Phi 7120	61	22nm	1.2GHz	300W	2013
Tilera TILE-Gx72	72	40nm	1.0GHz	N/A	2013

Table 1.1: Recent multicore and manycore processors

founding in 2004. Based on a scalable mesh network [91], Tilera provides processors with various numbers of cores. Recently the company announced that their 72-core TILE-Gx72 processor achieved the highest single-chip performance for Suricata, an open source-based intrusion detection system (IDS) [14].

Table 1.1 shows several examples of recent multicore and manycore processors[1].

1.2 On-chip Network for Manycore Architecture

Solving the issues in Section 1.1.2.1 requires a broad range of system layers to be optimized for manycore architecture. For example, low-power circuit design, scalable memory architecture, and efficient programming models are all important to continue scaling up the number of cores. Among the various approaches, however, the on-chip network is one of the most critical elements to the success of manycore architecture.

In manycore CMPs, the foremost goal of the on-chip network is to provide a scalable solution for the communication between cores. The on-chip network almost always stands at the center of scalability issues, because it is how cores communicate with each other. The on-chip network also worsens the power problem as it contributes a significant portion of the total power consumption. In the Intel's 80-core Polaris processor, for example, the on-chip network consumes 28% of total chip power [37]. In addition, high-level system components may require the on-chip net-

[1]The names of research chips are shown with asterisk marks.

work to support specific mechanisms. For instance, directory-based cache coherence protocols require frequent multicast or broadcast, which is a challenge for the on-chip network to implement efficiently.

Researchers have taken many different approaches to on-chip networks to overcome the challenges of manycore architecture. These approaches can be categorized into circuit-level optimization, network-level optimization, and system-level optimization.

1.2.1 Circuit-level Optimization

Research in this category aims at improving the performance and reducing the cost of data movement through better circuit design. For example, a double-pumped crossbar channel with a location-based channel driver (LBD), which reduces the crossbar hardware cost by half [88], was used in the mesh interconnect of the Intel manycore processor [37]. The ring interconnect used for the Intel Nehalem-EX Xeon microprocessor also exploits circuit-level techniques such as conditional clocking, fully shielded wire routing, etc., to optimize its design [68]. Self-resetting logic repeaters (SRLR) is another example that incorporates circuit techniques to better explore the trade-off between area, power, and performance [69].

1.2.2 Network-level Optimization

The logical and architectural design of the on-chip network plays an essential role in both the functionality and performance of the on-chip network. An extensive range of network topologies have been proposed and examined over the years [18]. Routing [87, 65, 66, 76, 49, 31, 9] is another key factor that determines the characteristics of on-chip communication. This level of optimization also has a significant impact on the power dissipation of the network because the amount of energy consumed by on-chip network is directly related to activity on the network. Therefore, using better communication schemes can result in reducing the power usage as shown in [51].

1.2.3 System-Level Optimization

While most work on on-chip networks focuses on the area/power/performance trade-off in the network itself, an increasing number of researchers have begun to take a totally different approach, for example, embedding additional logic into the on-chip network that is tightly coupled with processing units so the on-chip network can directly support high-level functionality of the system.

One example of such functionality is Quality of Service (QoS) across different application traffic, which is important for performance isolation and differentiated services [50]. Many QoS-capable on-chip networks have been proposed; while early work largely relies on time-sharing of channel resources [32, 60, 7], Globally-Synchronized Frames (GSF) orchestrate source injection based on time windows using a dedicated network for fast synchronization [50]. In Preemptive Virtual Clock (PVC), routers intentionally drop lower-priority packets and send NACK messages back to the sources for later retransmission [34].

Additionally, on-chip networks can alleviate the lack of scalability of directory-based cache coherence protocols. By embedding directories within each router, for example, requests can be redirected to nearby data copies [22]. In another example, each router keeps information that helps to decide whether to invalidate a cache line or send it to a nearby core so it can be used again without going off-chip [23].

1.3 Thesis Organization

First, our network-level research on oblivious routing is described in Chapter 2. We continue in Chapter 3 with the introduction of the bandwidth-adaptive network, another network-level technique that implements a dynamic network topology. Chapter 4 describes the Exclusive Native Context protocol that facilitates fine-grained thread migration, which is a good example of system-level optimization of an on-chip network. Chapter 5 shares our hands-on experience in the physical implementation of the on-chip network for a 110-core processor. Finally, Chapter 6 presents the conclusions of this thesis and summarizes its contributions.

Chapter 2

Oblivious Routing with Path Diversity

2.1 Introduction

The early part of this thesis focuses on network-level optimization techniques. These techniques abstract the on-chip network from other system components and aim to improve the performance of the network under general traffic patterns. In this approach, the choice of routing algorithm is a particularly important matter since routing is one of the key factors that determines the performance of a network [18]. This chapter will discuss oblivious routing schemes for on-chip networks, present a solution, Path-based, Randomized, Oblivious, Minimal (PROM) routing [10], and show how it improves the diversity of routes and provides better throughput across a class of traffic patterns.

2.1.1 Oblivious vs. Adaptive Routing

Routing algorithms can be classified into two categories: oblivious and adaptive. Oblivious routing algorithms choose a route without regard to the state of the network. Adaptive algorithms, on the other hand, determine what path a packet takes based on network congestion. Because oblivious routing cannot avoid network con-

gestion dynamically, it may have lower worst-case and average-case throughput than adaptive routing. However, its low-complexity implementation often outweighs any potential loss in performance because an on-chip network is usually designed within tight power and area budgets [43].

Although many researchers have proposed cost-effective adaptive routing algorithms [3, 16, 26, 31, 33, 40, 48, 84], this chapter focuses on oblivious routing for the following reasons. First, adaptive routing improves the performance only if the network has considerable amount of congestion; on the other hand, when congestion is low oblivious routing performs better than adaptive routing due to the extra logic required by adaptive routing. Because an on-chip network usually provides ample bandwidth relative to demand, it is not easy to justify the implementation cost of adaptive routing.

Furthermore, the cost of adaptive routing is more severe for an on-chip network than for its large-scale counterparts. Many large-scale data networks, such as a wireless network, have unreliable nodes and links. Therefore, it is important for every node to report its status to other nodes so each node can keep track of ever-changing network topology. Because the network nodes are already sharing the network status, adaptive routing can exploit this knowledge to make better routing decisions without additional costs. In contrast, on-chip networks have extremely reliable links among the network nodes so they do not require constant status checking amongst the nodes. Therefore, monitoring the network status for adaptive routing always incurs extra costs in on-chip networks.

2.1.2 Deterministic vs. Path-diverse Oblivious Routing

Deterministic routing is a subset of oblivious routing, which always chooses the same route between the same source-destination pair. Deterministic routing algorithms are widely used in on-chip network designs due to their low-complexity implementation.

Dimension-ordered routing (DOR) is an extremely simple routing algorithm for a broad class of networks that include 2D mesh networks [17]. Packets simply route along one dimension first and then in the next dimension, and no path exceeds the

minimum number of hops, a feature known as "minimal routing". Although it enables low-complexity implementation, the simplicity comes at the cost of poor worst-case and average-case throughput for mesh networks[1].

Path-diverse oblivious routing algorithms attempt to balance channel load by randomly selecting paths between sources and their respective destinations. The Valiant [87] algorithm routes each packet via a random intermediate node. Whenever a packet is injected, the source node randomly chooses an intermediate node for the packet anywhere in the network; the packet then first travels toward the intermediate node using DOR, and, after reaching the intermediate node, continues to the original destination node, also using DOR. Although the Valiant algorithm has provably optimal worst-case throughput, its low average-case throughput and high latency have prevented widespread adoption.

ROMM [65, 66] also routes packets through intermediate nodes, but it reduces latency by confining the intermediate nodes to the minimal routing region. n-phase ROMM is a variant of ROMM that uses $n-1$ different intermediate nodes that divide each route into n different *phases* so as to increase path diversity. Although ROMM outperforms DOR in many cases, the worst-case performance of the most popular (2-phase) variant on 2D meshes and tori has been shown to be significantly worse than optimal [85, 76], and the overhead of n-phase ROMM has hindered real-world use.

O1TURN [76] on a 2D mesh selects one of the DOR routes (XY or YX) uniformly at random, and offers performance roughly equivalent to 2-phase ROMM over standard benchmarks combined with near-optimal worst-case throughput; however, its limited path diversity limits performance on some traffic patterns.

Unlike DOR, each path-diverse algorithm may create dependency cycles amongst its routes, so it requires extra hardware to break those cycles and prevent network-

[1]The worst-case throughput of a routing algorithm on a network is defined as the minimum throughput over all traffic patterns. The average-case throughput is its average throughput over all traffic patterns. Methods have been given to compute worst-case throughput [85] and approximate average-case throughput by using a finite set of random traffic patterns [86]. While these models of worst-case and average-case throughput are important from a theoretical standpoint, they do not model aspects such as head-of-line blocking, and our primary focus here is evaluating performance on a set of benchmarks that have a variety of local and non-local bursty traffic.

	DOR	**O1TURN**	**2-phase ROMM**	**n-phase ROMM**	**Valiant**
Path diversity	None	Minimum	Limited	Fair to Large	Large
#channels used for deadlock prevention	1	2	2	n (2 required)	2
#hops	minimal	minimal	minimal	minimal	non-minimal
Communication overhead in bits per packet	None	None	$log_2 N$	$(n-1) \cdot log_2 N$	$log_2 N$

Table 2.1: Deterministic and Path-diverse Oblivious Routing Algorithms

level deadlock. Additionally, ROMM and Valiant also have some communication overhead, because a packet must contain the information of an intermediate node, or a list of intermediate nodes. Table 2.1 compares DOR and the path-diverse oblivious routing algorithms. Note that n different channels are not strictly required to implement n-phase ROMM; although it was proposed to be used with n channels, our novel virtual channel allocation scheme can work with n-phase ROMM with only 2 channels without network-level deadlock (Section 2.2.3).

We set out to develop a routing scheme with low latency, high average-case throughput, and path diversity for good performance across a wide range of patterns. The PROM family of algorithms we present here is significantly more general than existing oblivious routing schemes with comparable hardware cost (e.g., O1TURN). Like n-phase ROMM, PROM is maximally diverse on an $n \times n$ mesh, but requires less complex routing logic and needs only two virtual channels to ensure deadlock freedom.

In what follows, we describe PROM in Section 2.2, and show how to implement it efficiently on a virtual-channel router in Section 2.3. In Section 2.4, through detailed network simulation, we show that PROM algorithms outperform existing oblivious routing algorithms (DOR, 2-phase ROMM, and O1TURN) on equivalent hardware. We conclude the chapter in Section 2.5.

2.2 Path-based, Randomized, Oblivious, Minimal Routing (PROM)

Given a flow from a source to a destination, PROM routes each packet separately via a path randomly selected from among all minimal paths. The routing decision is made *lazily:* that is, only the next hop (conforming to the minimal-path constraint) is randomly chosen at any given switch, and the remainder of the path is left to the downstream nodes. The local choices form a random distribution over all possible minimal paths, and specific PROM routing algorithms differ according to the distributions from which the random paths are drawn. In the interest of clarity, we first describe a specific instantiation of PROM, and then show how to parametrize it into a family of routing algorithms.

2.2.1 Coin-toss PROM

Figure 2-1 illustrates the choices faced by a packet routed under a PROM scheme where every possible next-hop choice is decided by a fair coin toss. At the source node S, a packet bound for destination D randomly chooses to go north (bold arrow) or east (dotted arrow) with equal probability. At the next node, A, the packet can continue north or turn east (egress south or west is disallowed because the resulting route would no longer be minimal). Finally, at B and subsequent nodes, minimal routing requires the packet to proceed east until it reaches its destination.

Note that routing is oblivious and next-hop routing decisions can be computed locally at each node based on local information and the relative position of the current node to the destination node; nevertheless, the scheme is maximally diverse in the sense that each possible minimal path has a non-zero probability of being chosen. However, the coin-toss variant does *not* choose *paths* with uniform probability. For example, while uniform path selection in Figure 2-1 would result in a probability of $\frac{1}{6}$ for each path, either border path (e.g., $S \rightarrow A \rightarrow B \rightarrow \cdots \rightarrow D$) is chosen with probability $\frac{1}{4}$, while each of the four paths passing through the central node has only

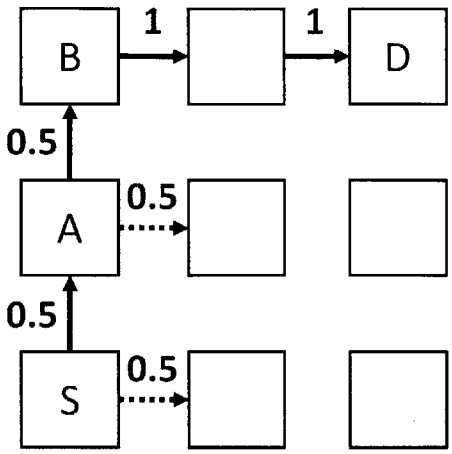


Figure 2-1: Randomized minimal routing in PROM

a $\frac{1}{8}$ chance. In the next section, we show how to parametrize PROM and create a uniform variant.

2.2.2 PROM Variants

Although all the next-hop choices in Figure 2-1 were 50–50 (whenever a choice was possible without leaving the minimum path), the probability of choosing each egress can be varied for each node and even among flows between the same source and destination. On a 2D mesh under minimum-path routing, each packet has at most two choices: continue straight or turn;[2] how these probabilities are set determines the specific instantiation of PROM:

O1TURN-like PROM O1TURN [76] randomly selects between XY and YX routes, i.e., either of the two routes along the edges of the minimal-path box. We can emulate this with PROM by configuring the source node to choose each edge with probability $\frac{1}{2}$ and setting all intermediate nodes to continue straight with probability 1 until a corner of the minimal-path box is reached, turning at the corner, and again continuing straight with probability 1 until the destination.[3]

[2]While PROM also supports other non-minimal schemes, we focus on minimal-path routing.
[3]This slightly differs from O1TURN in virtual channel allocation, as described in Section 2.2.3.

Uniform PROM Uniform PROM weighs the routing probabilities so that each possible minimal *path* has an equal chance of being chosen. Let's suppose that a packet on the way to node D is currently at node S, where x and y indicate the number of hops from S to D along the X and Y dimensions, respectively. When either x or y is zero, the packet is going straight in one direction and S simply moves the packet to the direction of D. If both x and y are positive, on the other hand, S can send the packet either along the X dimension to node S'_x, or the Y dimension to node S'_y. Then, for each of the possible next hops,

$$N_{S'_x \to D} = \frac{\{(x-1)+y\}!}{(x-1)! \cdot y!}$$

$$N_{S'_y \to D} = \frac{\{x+(y-1)\}!}{x! \cdot (y-1)!}$$

where $N_{A \to B}$ represents the number of all minimal paths from node A to node B.

In order that each minimal path has the same probability to be taken, we need to set the probability of choosing S'_x and S'_y proportional to $N_{S'_x \to D}$ and $N_{S'_y \to D}$, respectively. Therefore, we calculate P_x, the probability for S to move the packet along the X dimension as

$$P_x = \frac{N_{S'_x \to D}}{N_{S'_x \to D} + N_{S'_y \to D}} = \frac{x \cdot (x+y-1)!}{x \cdot (x+y-1)! + y \cdot (x+y-1)!} = \frac{x}{x+y}$$

and similarly, $P_y = \frac{y}{x+y}$. In this configuration, PROM is equivalent to n-phase ROMM with each path being chosen at the source with equal probability.[4]

Parametrized PROM The two configurations above are, in fact, two extremes of a continuous family of PROM algorithms parametrized by a single parameter f, as shown in Figure 2-2(b). At the source node, the router forwards the packet towards the destination on either the horizontal link or the vertical link randomly according to the ratio $x + f : y + f$, where x and y are the distances to the destination along the corresponding axes. At intermediate nodes, two possibilities exist: if the packet

[4] again, modulo differences in virtual channel allocation

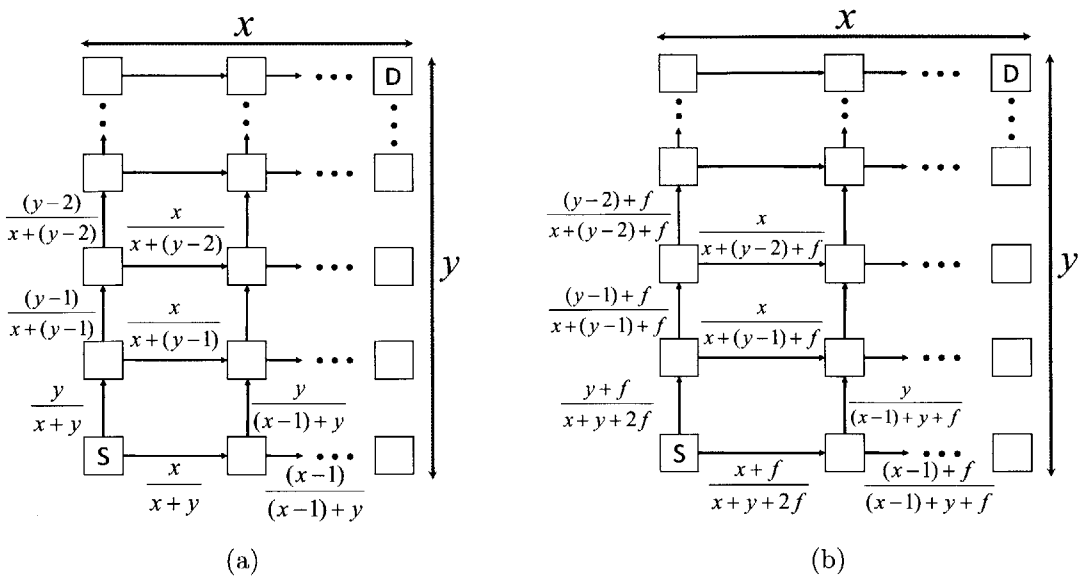
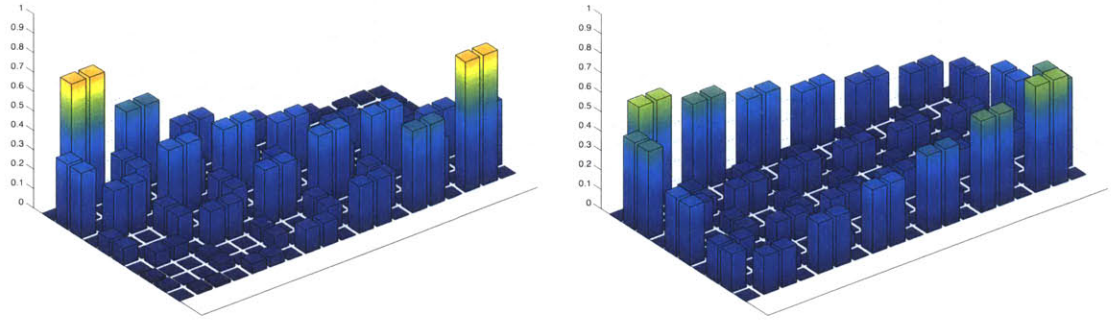


Figure 2-2: Probability functions of uniform PROM(a) and parameterized PROM(b)

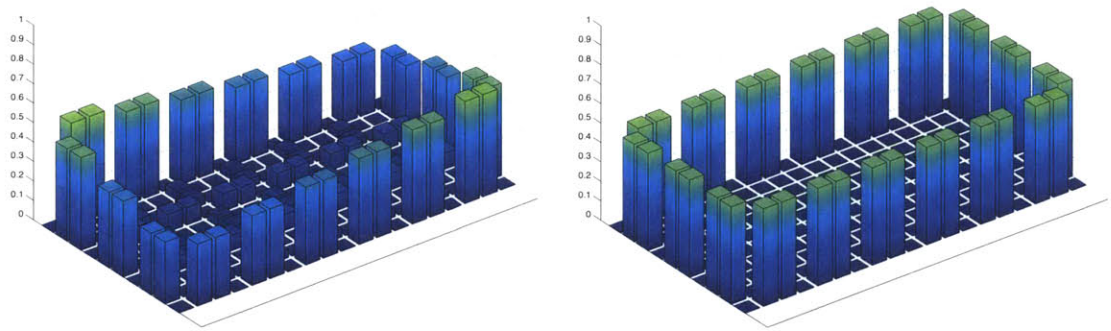
arrived on an X-axis ingress (i.e., from the east or the west), the router uses the ratio of $x + f : y$ in randomly determining the next hop, while if the packet arrived on an Y-axis ingress, it uses the ratio $x : y + f$. Intuitively, PROM is less likely to make extra turns as f grows, and increasing f pushes traffic from the diagonal of the minimal-path rectangle towards the edges (Figure 2-3). Thus, when $f = 0$ (Figure 2-3(a)), we have Uniform PROM, with most traffic near the diagonal, while $f = \infty$ (Figure 2-3(d)) implements the O1TURN variant with traffic routed exclusively along the edges.

Variable Parametrized PROM (PROMV) While more uniform (low f) PROM variants offer more path diversity, they tend to increase congestion around the center of the mesh, as most of the traffic is routed near the diagonal. Meanwhile, rectangle edges are underused especially towards the edges of the mesh, where the only possible traffic comes from the nodes on the edge.

Variable Parametrized PROM (PROMV) addresses this shortcoming by using different values of f for different flows to balance the load across the links. As the minimal-path rectangle between a source–destination pair grows, it becomes more



(a) $f = 0$ (b) $f = 10$



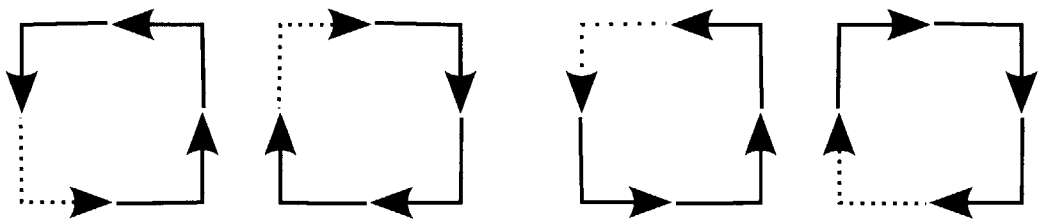
(c) $f = 25$ (d) $f = \infty$

Figure 2-3: Probability distributions of PROM routes with various values of f

likely that other flows within the rectangle compete with traffic between the two nodes. Therefore, PROMV sets the parameter f proportional to the minimal-path rectangle size divided by overall network size so traffic can be routed more toward the boundary when the minimal-path rectangle is large. When x and y are the distance from the source to the destination along the X and Y dimensions and N is the total number of router nodes, f is determined by the following equation:

$$f = f_{max} \cdot \frac{xy}{N} \tag{2.1}$$

The value of f_{max} was fixed to the same value for all our experiments (cf. Section



(a) West-First (rotated 180°) (b) North-Last (rotated 270°)

Figure 2-4: Permitted (solid) and prohibited (dotted) turns in two turn models

2.4). This scheme ensures efficient use of the links at the edges of the mesh and alleviates congestion in the central region of the network.

2.2.3 Virtual Channel Assignment

To provide deadlock freedom for PROM, we invoke the turn model [31], a systematic way of generating deadlock-free routes. Figure 2-4 shows two different turn models that can be used in a 2D mesh: each model disallows two of the eight possible turns, and, when all traffic in a network obeys the turn model, deadlock freedom is guaranteed. For PROM, the key observation[5] is that minimal-path traffic always obeys one of those two turn models: eastbound packets never turn westward, westbound packets never turn eastward, and packets between nodes on the same row or column never turn at all. Thus, westbound and eastbound routes always obey the restrictions of Figures 2-4(a) and 2-4(b), respectively, and preventing eastbound and westbound packets from blocking each other ensures deadlock freedom.

Therefore, PROM uses only two virtual channels for deadlock-free routing; one virtual channel for eastbound packets, and the other for westbound packets. When a packet is injected, the source node S checks the relative position of the destination node D. If D lies to the east of S, the packet is marked to use the first VC of each link on its way; and if D lies to the west of S, the second VC is used for the packet. If S and D are on the same column, the source node may choose any virtual channel because the packet travels straight to D and does not make any turn, conforming

[5]due to Shim et al. [79]

to both turn models. Once the source node chooses a virtual channel, however, the packet should use only that VC along the way.[6]

Although this is sufficient to prevent deadlock in PROM, we can optimize the algorithm to better utilize virtual channels. For example, the first virtual channel in any westbound links are never used in the original algorithm because eastbound packets never travel on westbound links. Therefore, westbound packets can use the first virtual channel on these westbound links without worrying about blocking any eastbound packets. Similarly, eastbound packets may use the second virtual channel on eastbound links; in other words, packets may use any virtual channel while they are going across horizontal links because horizontal links are used by only one type of packets. With this optimization, when a packet is injected at the source node S and travels to the destination node D,

1. if D lies directly north or south of S, the source node chooses one virtual channel that will be used along the route;

2. if the packet travels on horizontal links, any virtual channel can be used on the horizontal links;

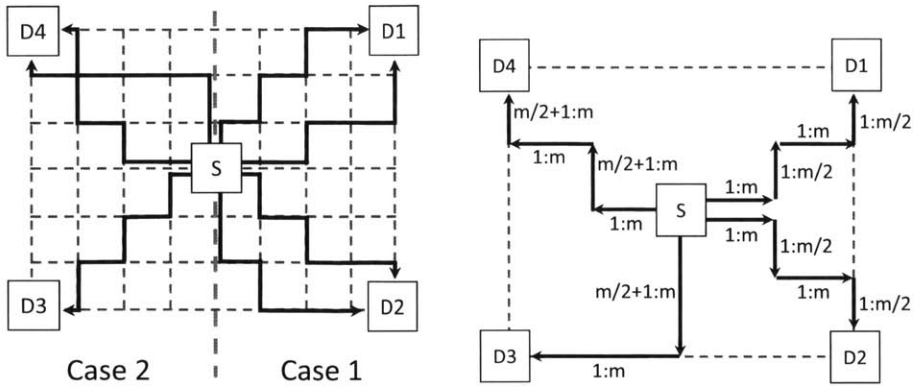
3. if the packet travels on vertical links and D lies to the east of S, the first VC is used on the vertical links;

4. if the packet travels on vertical links and D lies to the west of S, the second VC is used on the vertical links;

(When there are more than two virtual channels, they are split into two sets and assigned similarly). Figure 2-5 illustrates the division between eastbound and westbound traffic and the resulting allocation for m virtual channels.

It is noteworthy that PROM does not *explicitly* implement turn model restrictions, but rather forces routes to be minimal, which automatically restricts possible turns;

[6]If such a packet is allowed to switch VCs along the way, for example, it may block a westbound packet in the second VC of the upstream router, while being blocked by an eastbound packet in the first VC of the downstream router. This effectively makes the eastbound packet block the westbound packet and may cause deadlock.



(a) East- and westbound routes (b) VC set allocation

Figure 2-5: Virtual channel assignment in PROM

thus, we only use the turn model to show that VC allocation is deadlock-free. Also note that the correct virtual channel allocation for a packet can be determined *locally* at each switch, given only the packet's destination (encoded in its flow ID), and which ingress and virtual channel the packet arrived at. For example, any packet arriving from a west-to-east link and turning north or south must be assigned the first VC (or VC set), while any packet arriving from an east-to-west link and turning must get the second VC; finally, traffic arriving from the north or south stays in the same VC it arrived on.

The virtual channel assignment in PROM differs from that of both O1TURN and n-phase ROMM even when the routing behavior itself is identical. While PROM with $f = \infty$ selects VCs based on the overall direction as shown above, O1TURN chooses VCs depending on the initial choice between the XY and YX routes at the source node; because all traffic on a virtual network is either XY or YX, no deadlock results. ROMM, meanwhile, assigns a separate VC to each *phase*; since each phase uses exclusively one type of DOR (say XY), there is no deadlock, but the assignment is inefficient for general n-phase ROMM which uses n VCs where two would suffice.

2.3 Implementation Cost

Other than a randomness source, a requirement common to all randomized algorithms, implementing any of the PROM algorithms requires almost no hardware overhead over a classical oblivious virtual channel router [18]. As with DOR, the possible next-hop nodes can be computed directly from the position of the current node relative to the destination; for example, if the destination lies to the northwest on a 2D mesh, the packet can choose between the northbound and westbound egresses. Similarly, the probability of each egress being chosen (as well as the value of the parameter f in PROMV) only depends on the location of the current node, and on the relative locations of the source and destination node, which usually form part of the packet's flow ID.

As discussed in Section 2.2.3, virtual channel allocation also requires only local information already available in the classical router: namely, the ingress port and ingress VC must be provided to the VC allocator and constrain the choice of available VCs when routing to vertical links, which, at worst, requires simple multiplexer logic. This approach ensures deadlock freedom, and eliminates the need to keep any extra routing information in packets.

The routing header required by most variants of PROM needs only the destination node ID, which is the same as DOR and O1TURN and amounts to $2\log_2(n)$ bits for an $n \times n$ mesh; depending on the implementation chosen, PROMV may require an additional $2\log_2(n)$ bits to encode the source node if it is used in determining the parameter f. In comparison, packets in canonical k-phase ROMM carry the IDs for the destination node as well as the $k-1$ intermediate nodes in the packet, an overhead of $2k\log_2(n)$ bits on an $n \times n$ mesh, although one could imagine a somewhat PROM-like version of ROMM where only the next intermediate node ID (in addition to the destination node ID) is carried with the packet, and the $k+1$st intermediate node is chosen once the packet arrives at the kth intermediate destination.

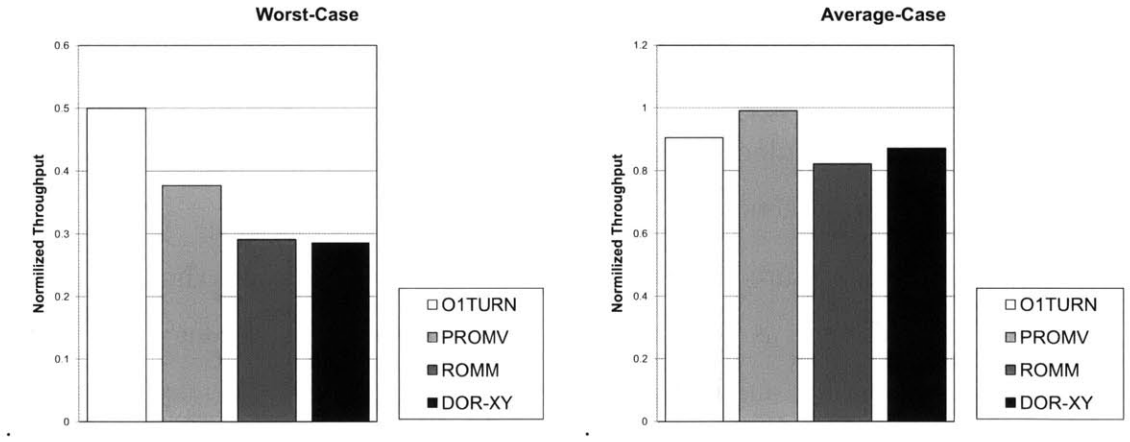
Thus, PROM hardware offers a wide spectrum of routing algorithms at an overhead equivalent to that of O1TURN and smaller than even 2-phase ROMM.

2.4 Experimental Results

To evaluate the potential of PROM algorithms, we compared variable parametrized PROM (PROMV, described in Section 2.2.2) on a 2D mesh against two path-diverse algorithms with comparable hardware requirements, O1TURN and 2-phase ROMM, as well as dimension-order routing (DOR). First, we analytically assessed throughput on worst-case and average-case loads; then, we examined the performance in a realistic router setting through extensive simulation.

2.4.1 Ideal Throughput

To evaluate how evenly the various oblivious routing algorithms distribute network traffic, we analyzed the ideal throughput[7] in the same way as [85] and [86], both for worst-case throughput and for average-case throughput.



(a) Worst-Case (b) Average

Figure 2-6: Ideal balanced throughput of oblivious routing algorithms

On worst-case traffic, shown in Figure 2-6(a), PROMV does significantly better than 2-phase ROMM and DOR, although it does not perform as well as O1TURN (which, in fact, has optimal throughput [76]). On average-case traffic, however,

[7] "ideal" because effects other than network congestion, such as head-of-line blocking, are not considered. In this model, each network flow is assumed to have a constant throughput demand. When a network link is saturated by multiple flows, those flows are *throttled down* by the same ratio, so that their total throughput matches the link bandwidth.

Name	Pattern	Example (b=4)
Bit-complement	$d_i = \neg s_i$	$(d_3, d_2, d_1, d_0) = (\neg s_3, \neg s_2, \neg s_1, \neg s_0)$
Bit-reverse	$d_i = s_{b-i-1}$	$(d_3, d_2, d_1, d_0) = (s_0, s_1, s_2, s_3)$
Shuffle	$d_i = s_{(i-1) \mod b}$	$(d_3, d_2, d_1, d_0) = (s_2, s_1, s_0, s_3)$
Transpose	$d_i = s_{(i+b/2) \mod b}$	$(d_3, d_2, d_1, d_0) = (s_1, s_0, s_3, s_2)$

Table 2.2: Synthetic network traffic patterns

PROMV outperforms the next best algorithm, O1TURN, by 10% (Figure 2-6(b)); PROMV wins in this case because it offers higher path diversity than the other routing schemes and is thus better able to spread traffic load across the network. Indeed, average-case throughput is of more concern to real-world implementations because, while every oblivious routing algorithm is subject to a worst-case scenario traffic pattern, such patterns tend to be artificial and rarely, if ever, arise in real NoC applications.

2.4.2 Simulation Setup

The actual performance on specific on-chip network hardware, however, is not fully described by the ideal-throughput model on balanced traffic. Firstly, both the router architecture and the virtual channel allocation scheme could significantly affect the actual throughput due to unfairness of scheduling and head-of-line blocking issues; secondly, balanced traffic is often not the norm: if network flows are not correlated at all, for example, flows with less network congestion could have more delivered traffic than flows with heavy congestion and traffic would not be balanced.

In order to examine the actual performance on a common router architecture, we performed cycle-accurate simulations of a 2D-mesh on-chip network under a set of standard synthetic traffic patterns, namely *transpose*, *bit-complement*, *shuffle*, and *bit-reverse*. In these traffic patterns, each bit d_i of the b-bit destination address is decided based on the bits of the source address, s_j [18] (See Table 2.2 for the definition of each pattern, and Table 2.3 for other simulation details). One should note that, like the worst-case traffic pattern above, these remain specific and regular traffic patterns and do not reflect all traffic on an arbitrary network; nevertheless, they were designed to

Characteristic	Configuration
Topology	8x8 2D MESH
Routing	PROMV($f_{max} = 1024$), DOR, O1TURN, 2-phase ROMM
Virtul channel allocation	Dynamic, EDVCA
Per-hop latency	1 cycle
Virtual channels per port	8
Flit buffers per VC	8
Average packet length (flits)	8
Traffic workload	bit-complement, bit-reverse, shuffle, transpose
Warmup / Analyzed cycles	20K / 100K

Table 2.3: Simulation details for PROM and other oblivious routing algorithms

simulate traffic produced by real-world applications, and so are often used to evaluate routing algorithm performance.

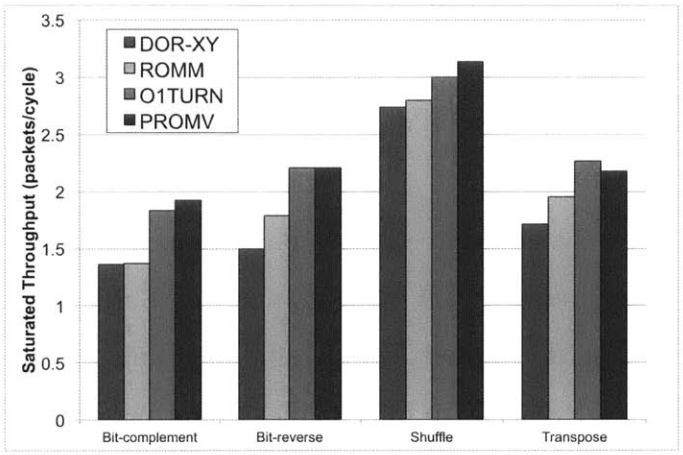
We focus on delivered throughput in our experiments, since we are comparing minimal routing algorithms against each other. We left out Valiant, since it is a non-minimal routing algorithm and because its performance has been shown to be inferior to ROMM and O1TURN [76]. While our experiments included both DOR-XY and DOR-YX routing, we did not see significant differences in the results, and consequently report only DOR-XY results.

Routers in our simulation were configured for 8 virtual channels per port, allocated either in one set (for DOR) or in two sets (for O1TURN, 2-phase ROMM, and PROMV; cf. Section 2.2.3), and then dynamically within each set. Because under dynamic allocation the throughput performance of a network can be severely degraded by head-of-line blocking [79] especially in path-diverse algorithms which present more opportunity for sharing virtual channels among flows, we were concerned that the true performance of PROM and ROMM might be hindered. We therefore repeated all experiments using Exclusive Dynamic Virtual Channel Allocation [53] or Flow-Aware Virtual Channel Allocation [4], dynamic virtual channel allocation techniques which reduce head-of-line blocking by ensuring that flits from a given flow can use only one virtual channel at each ingress port, and report both sets of results. Note that

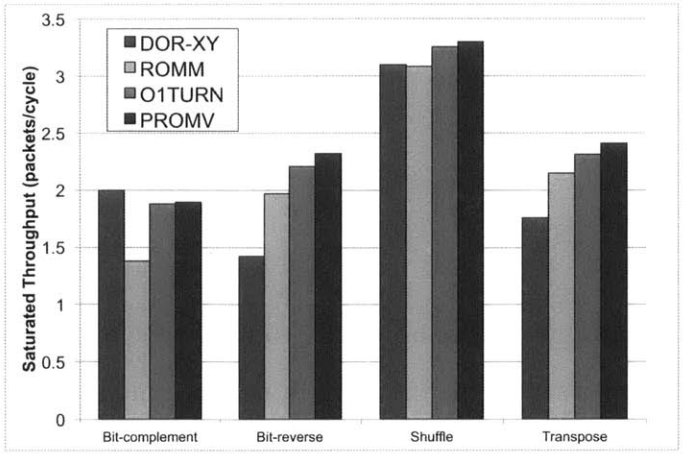
under this allocation scheme multiple flows *can* share the same virtual channel, and therefore they are different from having private channels for each flow, and can be used in routers with one or more virtual channels.

2.4.3 Simulation Results

Under conventional dynamic virtual channel allocation (Figure 2-7(a)), PROMV shows better throughput than ROMM and DOR under all traffic patterns, and slightly better than O1TURN under *bit-complement* and *shuffle*. The throughput of PROMV is the same as O1TURN under *bit-reverse* and worse than O1TURN under *transpose*.



(a) Dynamic VC allocation



(b) Exclusive-dynamic VC allocation

Figure 2-7: Saturated Throughput of oblivious routing algorithms

Using Exclusive Dynamic VC allocation improves results for all routing algorithms (Figure 2-7(b)), and allows PROMV to reach its full potential: on all traffic patterns but *bit-complement*, PROMV performs best. The perfect symmetry of *bit-complement* pattern causes PROMV to have worse ideal throughput than DOR and O1TURN which have perfectly even distribution of traffic load all over the network; in this special case of the perfect symmetry, the worst network congestion increases as some flows are more diversified in PROMV[8].

Note that these results highlight the limitations of analyzing ideal throughput given balanced traffic (cf. Section 2.4.1). For example, while PROMV has better ideal throughput than O1TURN on *transpose*, head-of-line blocking issues allow O1TURN to perform better under conventional dynamic VC allocation; on the other hand, while the perfectly symmetric traffic of *bit-complement* enables O1TURN to have better ideal throughput than PROMV, it is unable to outperform PROMV under either VC allocation regime.

While PROMV does not guarantee better performance under *all* traffic patterns (as exemplified by *bit-complement*), it offers competitive throughput under a variety of traffic patterns because it can distribute traffic load among many network links. Indeed, we would expect PROMV to offer higher performance on most traffic loads because it shows 10% better average-case ideal throughput of balanced traffic (Figure 2-6(b)), which, once the effects of head-of-line blocking are mitigated, begins to more accurately resemble real-world traffic patterns.

2.5 Conclusions

We have presented a parametrizable oblivious routing scheme that includes n-phase ROMM and O1TURN as its extreme instantiations. Intermediate instantiations push traffic either inward or outward in the minimum rectangle defined by the source and destination. The complexity of a PROM router implementation is equivalent

[8]Exclusive Dynamic VC allocation also makes the networks stable [18] (compare Figure 2-8 and Figure 2-9), as it improves the fairness of the routing schemes.

to O1TURN and simpler than 2-phase ROMM, but the scheme enables significantly greater path diversity in routes, thus showing 10% better performance on average in reducing the network congestion under random traffic patterns. The cycle-accurate simulations under a set of synthetic traffic patterns show that PROMV offers competitive throughput performance under various traffic patterns. It is also shown that if the effects of head-of-line blocking are mitigated, the performance benefit of PROMV can be significant.

Going from PROM to PRAM, where A stands for Adaptive is fairly easy. The probabilities of taking the next hop at each node can depend on local network congestion. With parametrized PROM, a local network node can adaptively control the traffic distribution simply and intuitively by adjusting the value of f in its routing decision. This may enable better load balancing especially under bursty traffic and we will investigate this in the future.

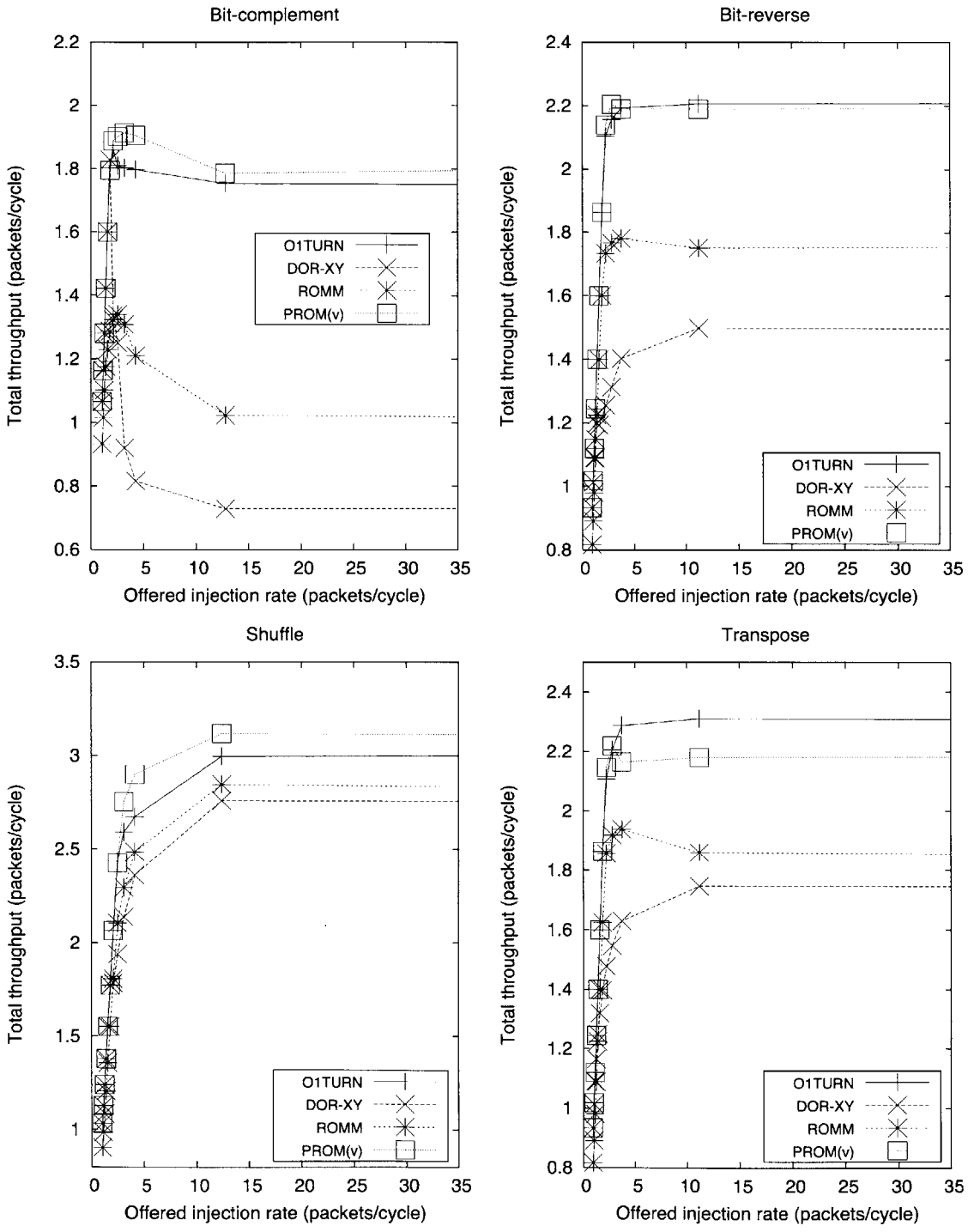


Figure 2-8: Throughput with dynamic VC allocation

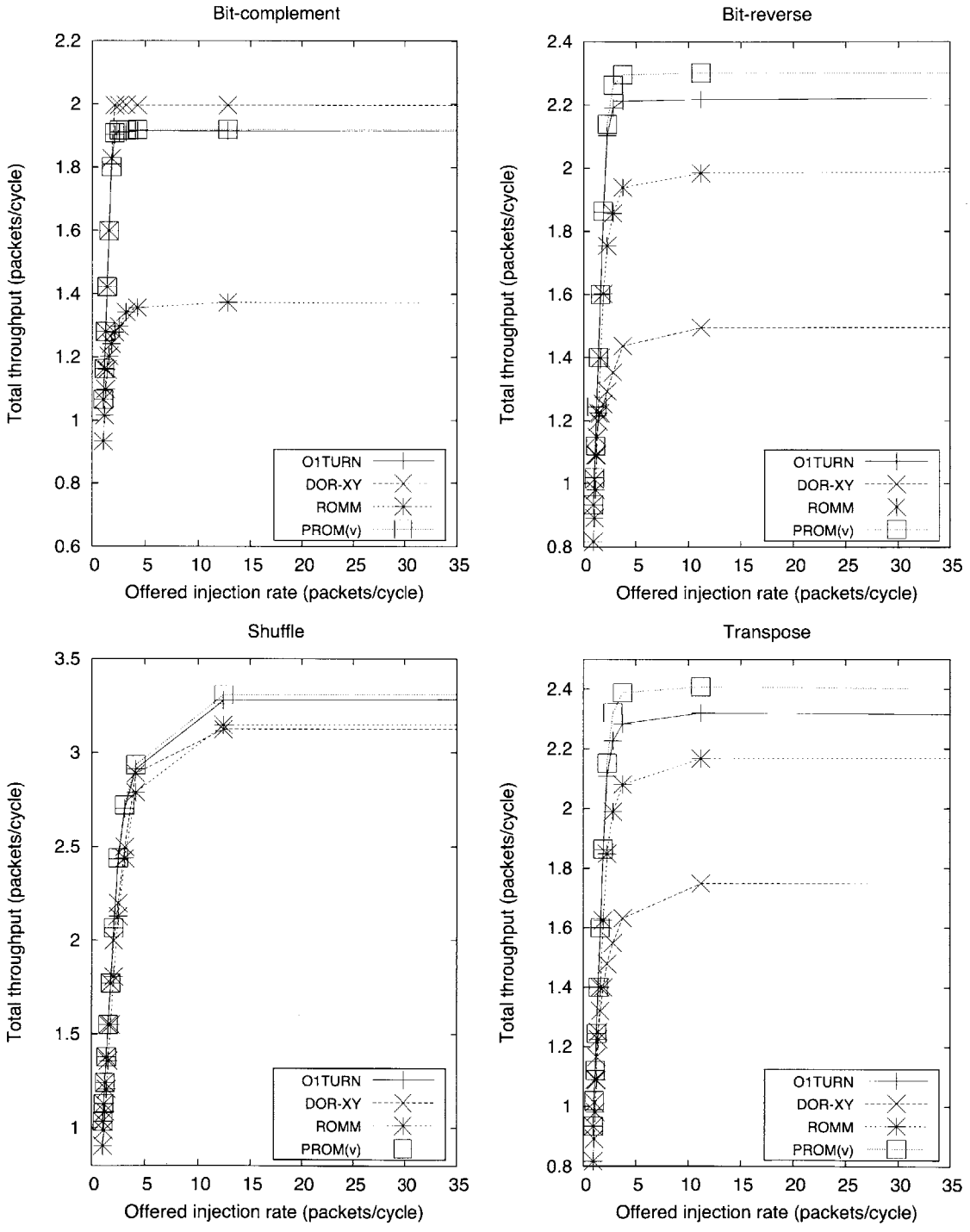


Figure 2-9: Throughput with exclusive-dynamic VC allocation

Chapter 3

Oblivious Routing in On-chip Bandwidth-Adaptive Networks

3.1 Introduction

This chapter presents another network-level optimization technique, Bandwidth-Adaptive Network (BAN). Like PROM (Chapter 2), BAN also aims at performance improvement of oblivious routing. Instead of changing how packets are routed, however, BAN changes the directions of network links adaptively based on network state.

3.1.1 Trade-off between Hardware Complexity and Global Knowledge in Adaptive Routing

As mentioned in Section 2.1.1, adaptive routing algorithms collect network congestion information and thus incur hardware and performance overheads. To alleviate the overheads, many adaptive routing schemes for on-chip networks use only local information of next-hop congestion to select the next egress port. The congestion metrics include the number of free next-hop virtual channels [16], available next-hop buffer size [48], etc.

Using only locally available information significantly reduces the hardware complexity. However, the local nature of the routing choices makes it difficult to make

assertions about, or optimize for, the network as a whole. Greedy and local decisions can actually do more harm than good on global load balance for certain traffic patterns [33]. Therefore some adaptive routing schemes go beyond local congestion data. Regional Congestion Awareness [33] combines local information with congestion reports from a neighborhood several hops wide; because reports from far-away nodes take several cycles to propagate and can be out of date, they are weighted less than reports from close-by nodes. Path-load based routing [84] routes packets along some minimal path and collects congestion statistics for switches along the way; when the destination node decides that congestion has exceeded acceptable limits, it will send an "alert" packet to the source node and cause it to select another, less congested minimal path. Both of these schemes require additional storage to keep the congestion data, and possibly inaccuracies when congestion information is several cycles old.

Researchers continue to search for the optimum balance between hardware complexity and routing performance. For example, DyAD [39] attempts to balance the simplicity of oblivious routing with the congestion-avoiding advantages of adaptive routing by using oblivious routing when traffic is light, and adaptive routing only if network is heavily loaded. Globally Oblivious Adaptive Locally (GOAL) [81] is another example of hybrid approaches where the direction of travel is chosen obliviously and then the packet is adaptively routed.

3.1.2 Oblivious Routing with Adaptive Network Links

Both DyAD and GOAL try to take advantage of oblivious routing techniques to to reduce the overhead of adaptive routing. If it is adaptive routing that causes significant overheads, then why do we not stick to oblivious routing and try to achieve adaptivity in a different way?

This is the fundamental idea of BAN; in BAN, the bisection bandwidth of a network can adapt to changing network conditions, while the routing function always remains oblivious. We describe one implementation of a bandwidth-adaptive network in the form of a two-dimensional mesh with adaptive bidirectional links[1], where the

[1]Bidirectional links have been preferred to as half-duplex links in router literature.

bandwidth of the link in one direction can be increased at the expense of the other direction. Efficient local intelligence is used to appropriately reconfigure each link, and this reconfiguration can be done very rapidly in response to changing traffic demands. Reconfiguration logic compares traffic on either side of a link to determine how to reconfigure each link.

We compare the hardware designs of a unidirectional and bidirectional link and argue that the hardware overhead of implementing bidirectionality and reconfiguration is reasonably small. We then evaluate the performance gains provided by a bandwidth-adaptive network in comparison to a conventional network through detailed network simulation of oblivious routing methods under uniform and bursty traffic, and show that the performance gains are significant.

In Section 3.2, we describe a hardware implementation of an adaptive bidirectional link, and compare it with a conventional unidirectional link. In Section 3.3, we describe schemes that determine the configuration of the adaptive link, i.e., decide which direction is preferred and by how much. The frequency of reconfiguration can be varied. Simulation results comparing oblivious routing on a conventional network against a bandwidth-adaptive network are the subject of Section 3.4. Section 3.5 concludes this chapter.

3.2 Adaptive Bidirectional Link

3.2.1 Conventional Virtual Channel Router

Although bandwidth adaptivity can be introduced independently of network topology and flow control mechanisms, in the interest of clarity we assume a conventional virtual-channel router on a two-dimensional (2-D) mesh network as a baseline.

Figure 3-1 illustrates a conventional virtual-channel router architecture and its operation [18, 64, 70], As shown in the figure, the datapath of the router consists of buffers and a switch. The input buffers store flits waiting to be forwarded to the next hop; each physical channel often has multiple input buffers, which allows flits to flow

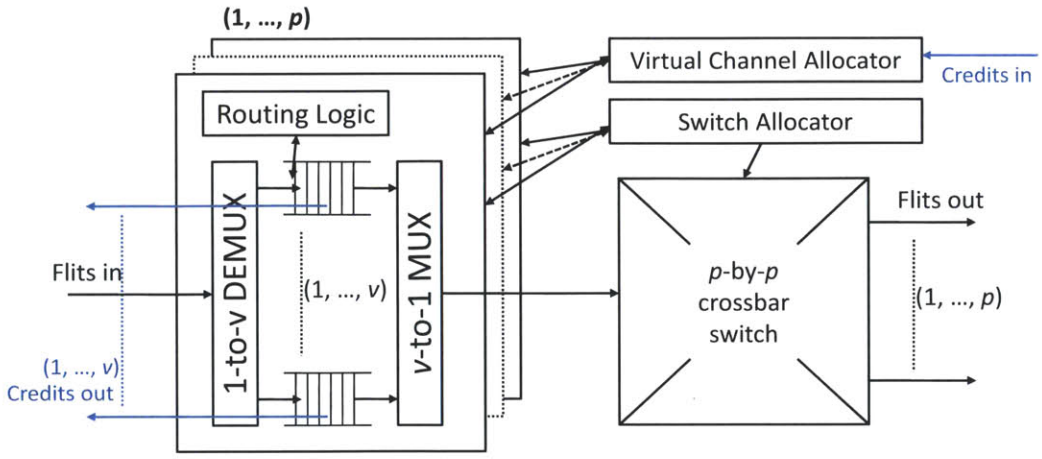


Figure 3-1: Conventional router architecture with p physical channels and v virtual channels per physical channel.

as if there were multiple "virtual" channels. When a flit is ready to move, the switch connects an input buffer to an appropriate output channel. To control the datapath, the router also contains three major control modules: a router, a virtual-channel (VC) allocator, and a switch allocator. These control modules determine the next hop, the next virtual channel, and when a switch is available for each packet/flit.

Routing comprises four steps: routing (RC), virtual-channel allocation (VA), switch allocation (SA), and switch traversal (ST); these are often implemented as four pipeline stages in modern virtual-channel routers. When a head flit (the first flit of a packet) arrives at an input channel, the router stores the flit in the buffer for the allocated virtual channel and determines the next hop node for the packet (RC stage). Given the next hop, the router then allocates a virtual channel in the next hop (VA stage). The next hop node and virtual channel decision is then used for the remaining flits of the given packet, and the relevant virtual channel is exclusively allocated to that packet until the packet transmission completes. Finally, if the next hop can accept the flit, the flit competes for a switch (SA stage), and moves to the output port (ST stage).

3.2.2 Bidirectional Links

In the conventional virtual-channel router shown in Figure 3-1, each output channel is connected to an input buffer in an adjacent router by a unidirectional link; the maximum bandwidth is related to the number of physical wires that constitute the link. In an on-chip 2-D mesh with nearest neighbor connections there will always be two links in close proximity to each other, delivering packets in opposite directions.

We propose to merge the two links between a pair of network nodes into a set of bidirectional links, each of which can be configured to deliver packets in either direction, increasing the bandwidth in one direction at the expense of the other. The links can be driven from two different sources, with local arbitration logic and tristate buffers ensuring that both do not simultaneously drive the same wire.

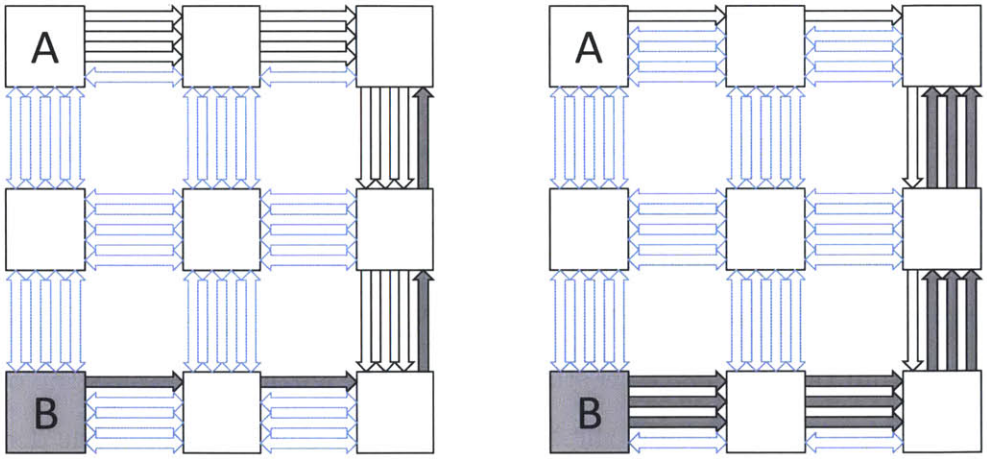


Figure 3-2: Adaptivity of a mesh network with bidirectional links

Figure 3-2 illustrates the adaptivity of a mesh network using bidirectional links. Flow A is generated at the upper left corner and goes to the bottom right corner, while flow B is generated at the bottom left corner and ends at the upper right corner. When one flow becomes dominant, bidirectional links change their directions in order to achieve maximal total throughput. In this way, the network capacity for each flow can be adjusted taking into account flow burstiness without changing routes.

Figure 3-3 shows a bidirectional link connecting two network nodes (for clarity,

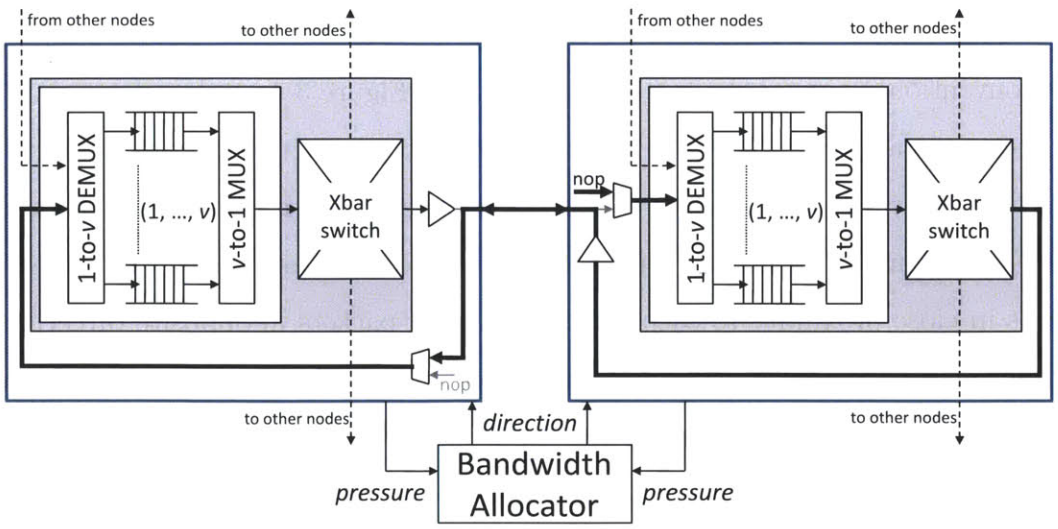


Figure 3-3: Connection between two network nodes through a bidirectional link

only one bidirectional link is shown between the nodes, but multiple bidirectional links can be used to connect the nodes if desired). The bidirectional link can be regarded as a bus with two read ports and two write ports that are interdependent. A bandwidth arbiter governs the direction of a bidirectional link based on *pressure* (Section 3.3) from each node, a value reflecting how much bandwidth a node requires to send flits to the other node. Bold arrows in Figure 3-3 illustrate a case when flits are delivered from right to left; a tri-state buffer in the left node prevents the output of its crossbar switch from driving the bidirectional link, and the right node does not receive flits as the input is being multiplexed. If the link is configured to be in the opposite way, only the left node will drive the link and only the right node will receive flits.

Router logic invalidates the input channel at the driving node so that only the other node will read from the link. The switching of tri-state buffers can be done faster than other pipeline stages in the router so that we can change the direction without dead cycles in which no flits can move in any direction. Note that if a dead cycle is required in a particular implementation, we can minimize performance loss by switching directions relatively infrequently. We discuss this tradeoff in Section 3.4.

While long wires in on-chip networks require repeaters, we focus on a nearest-

neighbor mesh network. As can be seen in Figure 3-3, only a short section of the link is bidirectional. Tri-state buffers are placed immediately to either side of the bidirectional section. This will be true of links connecting to the top and bottom network nodes as well. Therefore, the bi-directional sections do not need repeaters. If a bi-directional link is used to connect faraway nodes in a different network topology, a pair of repeaters with enable signals will be required in place of a conventional repeater on a unidirectional link.

3.2.3 Router Architecture with Bidirectional Links

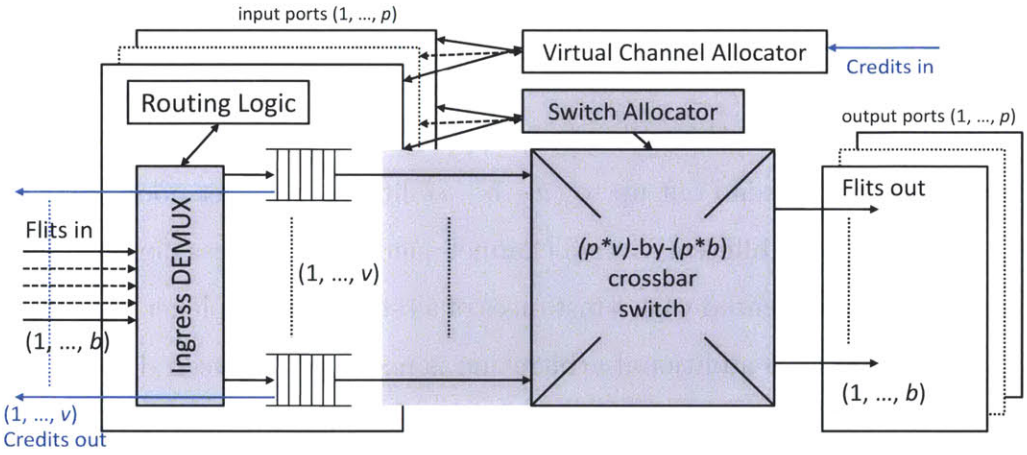


Figure 3-4: Network node architecture with u unidirectional links and b bidirectional links between each of p neighbor nodes and itself.

Figure 3-4 illustrates a network node with b bidirectional links, where each link has a bandwidth of one flit per router cycle; gray blocks highlight modules modified from the baseline architecture shown in Figure 3-1. Adjacent nodes are connected via p ports (for the 2-D mesh we consider here, $p = 4$ at most). At each port, b input channels and b output channels share the b bidirectional links via tri-state buffers: if a given link is configured to be ingressive, its input channel is connected to the link while the output channel is disconnected, and vice versa (output channels are not shown in the figure).

We parametrize architectures with and without bidirectional links by the number of unidirectional links u and the number of bidirectional links b; in this scheme, the

conventional router architecture in Figure 3-1 has $u = 1$ and $b = 0$. We will compare configurations with the same bisection bandwidth. A router with $u = 0$ and $b = 2$ has the same bisection bandwidth as $u = 1$ and $b = 0$. In general, we may have hybrid architectures with some of the links bidirectional and some unidirectional (that is, $u > 0$ and $b > 0$). A (u, b) router with bidirectional links will be compared to a conventional router with $u + b/2$ unidirectional links in each direction; this will be denoted as $(u + b/2, 0)$.

We assume, as in conventional routers, that at most one flit from each virtual channel can be transferred in a given cycle – if there are v virtual channels in the router, then at most v flits can be transferred in one cycle regardless of the bandwidth available. In a (u, b) router, if i out of b bidirectional links are configured to be ingressive at a router node, the node can receive up to $u + i$ flits per cycle from the node across the link and send out up to $(u + b - i)$ flits to the other node. Since each incoming flit will go to a different virtual channel queue,[2] the ingress demultiplexer in Figure 3-4 can be implemented with b instances of a v-to-1 demultiplexer with tri-state buffers at the outputs; no additional arbitration is necessary between demultiplexers because only one of their outputs will drive the input of each virtual channel.

In a bidirectional router architecture, the egress link can be configured to exceed one flit per cycle; consequently, the crossbar switch must be able to consider flits from more than one virtual channel from the same node. In the architecture described so far, the output of each virtual channel is directly connected to the switch and competes for an outgoing link. However, one can use a hierarchical solution where the v virtual channels are multiplexed to a smaller number of switch inputs. The Intel Teraflops has a direct connection of virtual channels to the switch [37]. Most routers have v-to-1 multiplexers that select one virtual channel from each port for each link prior to the crossbar.

In addition, the crossbar switch must now be able to drive all $p \cdot (u + b)$ outgoing links when every bidirectional link is configured as egressive, and there are u unidi-

[2]Recall that once a virtual channel is allocated to a packet at the previous node, other packets cannot use the virtual channel until the current packet completes transmission.

rectional links. Consequently, the router requires a $p \cdot v$-by-$p \cdot (u+b)$ crossbar switch, compared to a $p \cdot v$-by-$p \cdot (u + b/2)$ switch of a conventional $(u + b/2, 0)$ router that has the same bisection bandwidth; this larger switch is the most significant hardware cost of the bidirectional router architecture. If the v virtual channels are multiplexed to reduce the number of inputs of the switch, the number of inputs to the crossbar should be at least equal to the maximum number of outputs in order to fully utilize the bisection bandwidth. In this case, we have a $p \cdot (u + b/2)$-by-$p \cdot (u + b/2)$ crossbar in the $(u + b/2, 0)$ case. In the (u, b) router, we will need a $p \cdot (u + b)$-by-$p \cdot (u + b)$ crossbar. The v virtual channels at each port will be multiplexed into $(u + b)$ inputs to the crossbar.

To evaluate the flexibility and effectiveness of bidirectional links, we compare, in Section 3.4, the performance of bidirectional routers with $(u, b) = (0, 2)$ and $(u, b) = (0, 4)$ against unidirectional routers with $(u, b) = (1, 0)$ and $(u, b) = (2, 0)$, which, respectively, have the same total bandwidth as the bidirectional routers. We also consider a hybrid architecture with $(u, b) = (1, 2)$ which has the same total bandwidth as the $(u, b) = (2, 0)$ and $(u, b) = (0, 4)$ configurations. Table 3.1 summarizes the sizes of hardware components of unidirectional, bidirectional and hybrid router architectures assuming four virtual channels per ingress port (i.e., $v = 4$). There are two cases considered. The numbers in bold correspond to the case where all virtual channels compete for the switch. The numbers in plain text correspond to the case where virtual channels are multiplexed before the switch so the number of inputs to the switch is restricted by the bisection bandwidth. While switch allocation logic grows as the size of crossbar switch increases and bidirectional routers incur the additional cost of the bandwidth allocation logic shown in Figure 3-3, these are insignificant compared to the increased size of the demultiplexer and crossbar.

When virtual channels directly compete for the crossbar, the number of the crossbar input ports remains the same in both the unidirectional case and the bidirectional case. The number of crossbar output ports is the only factor increasing the crossbar size in bidirectional routers $(u, b) = (0, 4)$ and $(1, 2)$ when compared with the unidirectional $(2, 0)$ case; this increase is size is roughly equal to the ratio of the output

Architecture	Ingress Demux	Xbar Switch
(u, b) = (1, 0)	one 1-to-4 demux	4-by-4 or **16-by-4**
(u, b) = (0, 2)	two 1-to-4 demuxes	8-by-8 or **16-by-8**
(u, b) = (2, 0)	two 1-to-4 demuxes	8-by-8 or **16-by-8**
(u, b) = (0, 4)	four 1-to-4 demuxes	16-by-16 or **16-by-16**
(u, b) = (1, 2)	three 1-to-4 demuxes	12-by-12 or **16-by-12**

Table 3.1: Hardware components for 4-VC BAN routers

ports. Considering that a 32×32 crossbar takes approximately 30% of the gate count of a switch [45] with much of the actual area being accounted for by queue memory and wiring which is not part of the gate count, we estimate that a $1.5\times$ increase in crossbar size for the $(1, 2)$ case will increase the area of the node by $< 15\%$. If the queues are smaller, then this number will be larger. Similar numbers are reported in [33].

There is another way to compare the crossbars in the unidirectional and bidirectional cases. It is well known that the size of a $n \times n$ crossbar increases as n^2 (e.g., [93]). We can think of n as $p \cdot (u + b/2) \cdot w$, where w is the bit-width for the unidirectional case. If a bidirectional router's crossbar is $1.5\times$ larger, then one can create an equivalent-size unidirectional crossbar with the same number of links but $\sqrt{1.5}\times$ bit-width, assuming zero buffer sizes. In reality, the buffers will increase by $\sqrt{1.5} = 1.22\times$ due to the bit-width increase, and so the equivalent-size unidirectional crossbar will have a bit-width that is approximately $1.15\times$ of the bidirectional crossbar, assuming typical buffer sizes. This implies the performance of this crossbar in a network will be $1.15\times$ the baseline unidirectional case. As can be seen in Section 3.4, the bidirectional link architecture results in greater gains in performance.

3.3 Bandwidth Allocation in Bidirectional Links

Bidirectional links contain a bandwidth arbiter (Figure 3-3) which governs the direction of the bidirectional links connecting a pair of nodes and attempts to maximize the connection throughput. The locality and simplicity of this logic are key to our

approach: the arbiter makes its decisions based on very simple information local to the nodes it connects.

Each network node tells the arbiter of a given bidirectional link how much *pressure* it wishes to exert on the link; this pressure indicates how much of the available link bandwidth the node expects to be able to use in the next cycle. In our design, each node counts the number of flits ready to be sent out on a given link (i.e., at the head of some virtual channel queue), and sends this as the pressure for that link. The arbiter then configures the links so that the ratio of bandwidths in the two directions approximates the pressure ratio, additionally ensuring that the bandwidth granted does not exceed the free space in the destination node. Consequently, if traffic is heavier in one direction than in the other, more bandwidth will be allocated to that direction.

The arbitration logic considers only the next-hop nodes of the flits at the front of the virtual channel queues and the available buffer space in the destination queues, both of which are local to the two relevant nodes and easy to compute. The arbitration logic itself consists of threshold comparisons and is also negligible in cost.

With one-flit packets, the pressure as defined above exactly reflects the traffic that can be transmitted on the link; it becomes approximate when there are multiple flits per packet, since some of the destination queues with available space may be in the middle of receiving packets and may have been assigned to flows different from the flits about to be transmitted. Although more complex and accurate definitions of pressure are possible, our experience thus far is that this simple logic performs well in practice.

In some cases we may not want arbitration to take place in every cycle; for example, implementations that require a dead cycle after each link direction switch will perform poorly if switching takes place too often. On the other hand, switching too infrequently reduces the adaptivity of the bidirectional network, potentially limiting the benefits for quickly changing traffic and possibly requiring more complex arbitration logic. We explore this tradeoff in Section 3.4.

When analyzing link bandwidth allocation and routing in a bidirectional adaptive

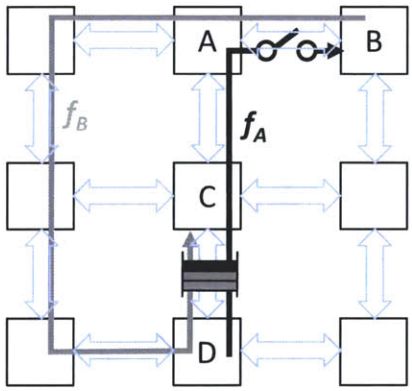


Figure 3-5: Deadlock on deadlock-free routes due to bidirectional links

network, we must take care to avoid additional deadlock due to bidirectional links, which may arise in some routing schemes. Consider, for example, the situation shown in Figure 3-5: a flow f_B travels from node B to node C via node A, and all links connecting A with B are configured in the direction $B \rightarrow A$. Now, if another, smaller flow f_A starts at D and heads for B, it may not exert enough pressure on the $A \rightarrow B$ link to overcome that of f_B, and, with no bandwidth allocated there, may be blocked. The flits of f_A will thus eventually fill the buffers along its path, which might prevent other flows, including f_B, from proceeding: in the figure, f_B shares buffering resources with f_A between nodes C and D, and deadlock results. Note that the deadlock arises only because the bidirectional nature of the link between A and B can cause the connection $A \rightarrow B$ to disappear; since the routes of f_A and f_B obey the west-first turn model [31], deadlock does not arise in the absence of bidirectional links. One easy way to avoid deadlock is to require, in the definition of pressure, that some bandwidth is always available in a given direction if some flits are waiting to be sent in that direction. For example, if there are four bidirectional links and there are eight flits waiting to travel in one direction and one in the opposite direction, BAN assigns three links to the first direction and one to the opposite direction.

3.4 Results and Comparisons

3.4.1 Experimental Setup

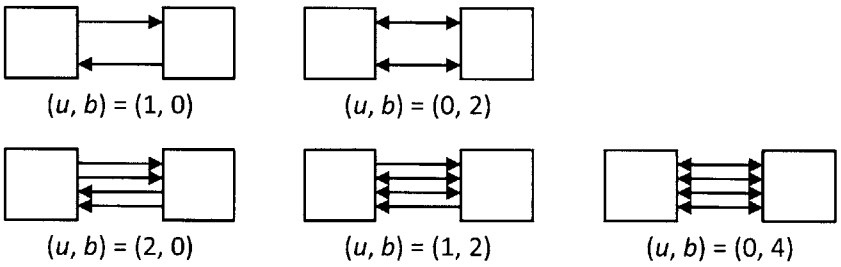


Figure 3-6: Link configurations for BAN evaluation

A cycle-accurate network simulator was used to model the bidirectional router architectures with different combinations of unidirectional (u) and bidirectional (b) links in each connection (Figure 3-6 and Table 3.2 for details). To evaluate performance under general traffic patterns, we employed a set of standard synthetic traffic patterns (*transpose*, *bit-complement*, *shuffle*, and *uniform-random*) both without burstiness and with a two-state Markov Modulated Process (MMP) bursty traffic model [18]. In the MMP model, a source node is in one of the two states, the "on" state or the "off" state, and the injection rate is r_{on} in the on state and 0 in the off state. In every cycle, a source node in the off state switches to the on state with the probability of α, and from the on state to the off state with probability β. Then the source node stays in the on state with the probability of $\frac{\alpha}{\alpha+\beta}$, so the steady-state injection rate r is $\frac{\alpha}{\alpha+\beta} \times r_{on}$. In our experiments, α was set to 30% and β was set to 10%, so that the injection rate during the on state packets during the on state at $\frac{\alpha+\beta}{\alpha} = \frac{4}{3}$ times larger than steady-state injection rates.

For the evaluation of performance under real-world applications, we profiled the network load of an H.264 decoder implemented on an ASIC; we measured how much data is transferred between the modules in the ASIC design, and mapped each module to a network node in such a way that each module is close to other modules that it directly communicates with. We then simulated the resulting traffic pattern on the unidirectional and the bidirectional networks. We also examined several frequencies

Characteristic	Configuration
Topology	8x8 2D MESH
Link configuration	(u, b) = (1,0), (0,2) (2,0), (1,2), (0,4)
Routing	DOR-XY and DOR-YX
VC output multiplexing	None, Matching maximum bandwidth
Per-hop latency	1 cycle
Virtual channels per port	4
Flit buffers per VC	4
Average packet length (flits)	8
Traffic workload	transpose, bit-complement, shuffle, uniform-random profiled H.264 decoder
Burstiness model	Markov modulated process
Warmup cycles	20,000
Analyzed cycles	100,000

Table 3.2: Simulation details for BAN and unidirectional networks

of bandwidth allocation to estimate the impact on architectures where a dead cycle is required to switch the link direction.

Although the bidirectional routing technique applies to various oblivious routing algorithms, we have, for evaluation purposes, focused on Dimension Ordered Routing (DOR), the most widely implemented oblivious routing method. While our experiments included both DOR-XY and DOR-YX routing, we did not see significant differences in the results, and consequently report only DOR-XY results. In all of our experiments, the router was configured for four virtual channels per ingress port under a dynamic virtual channel allocation regimen. The effect of multiplexing virtual channels in front of the crossbar switches was also examined.

3.4.2 Non-bursty Synthetic Traffic

Figure 3-7 shows the throughput in the unidirectional and bidirectional networks under non-bursty traffic. When traffic is consistent, the improvement offered by bidirectional links depends on how symmetric the flows are. On the one extreme, *bit-complement*, which in steady state is entirely symmetric when routed using DOR

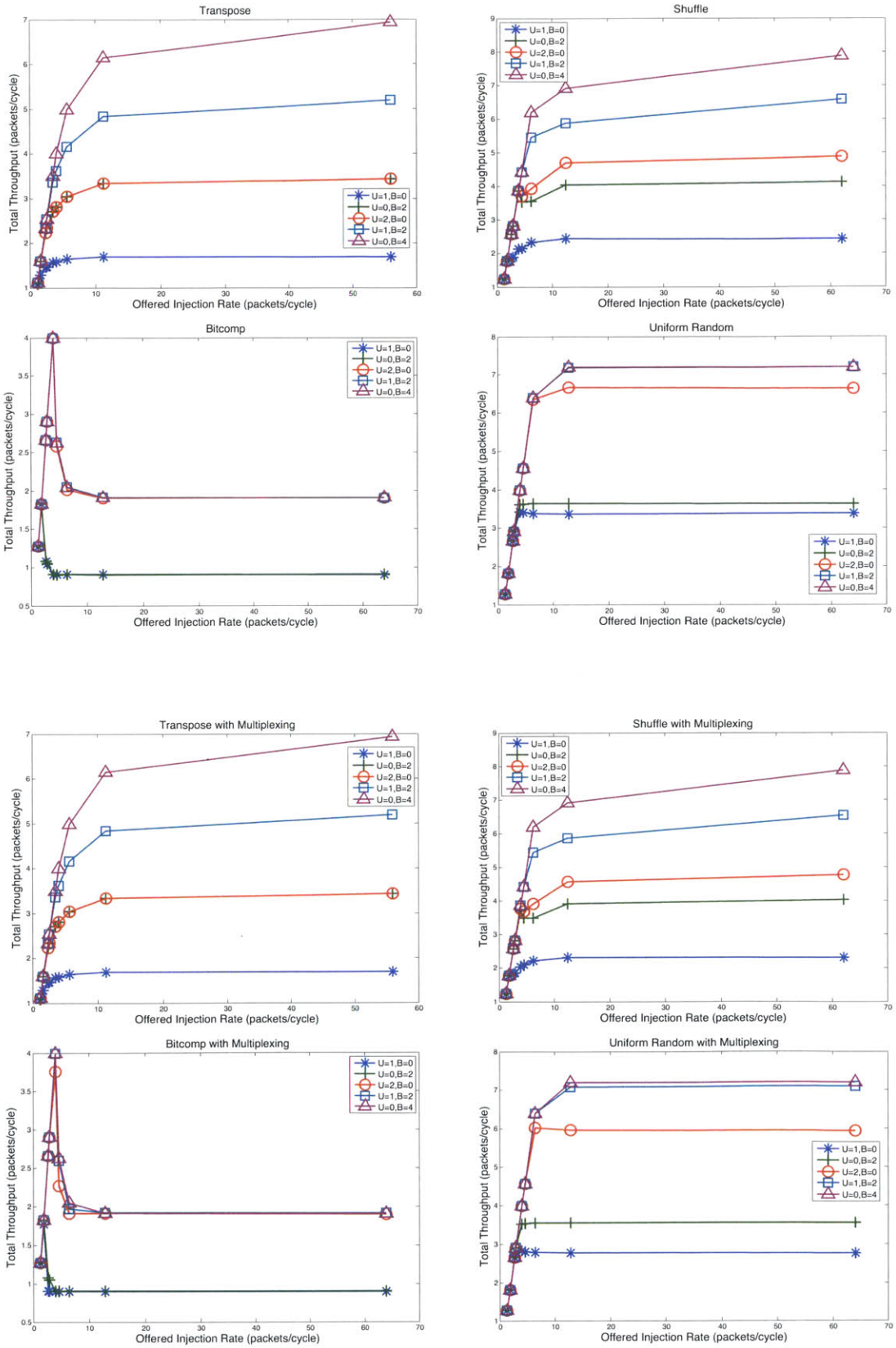


Figure 3-7: Throughput of BAN and unidirectional networks under non-bursty traffic

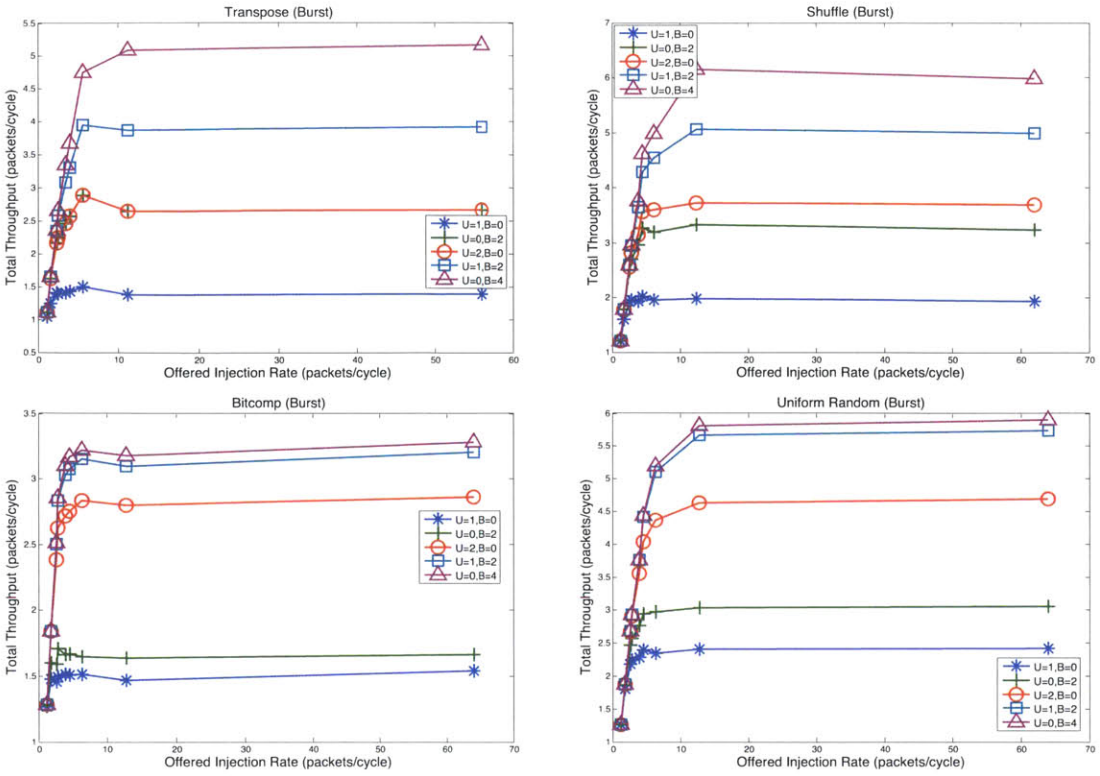


Figure 3-8: Throughput of BAN and unidirectional networks under bursty traffic

and results in equal traffic in each direction on any link, shows no improvement; on the other extreme, in *transpose*, packets move in only one direction over any given link, and bidirectional links improve throughput twofold. *Shuffle* lies between the two extremes, with the bidirectional network outperforming the unidirectional solution by 60% when total bandwidth is equal.

Uniformly random traffic is also symmetric when averaged over a period of time. For very short periods of time, however, the symmetry is imperfect, allowing the bidirectional network to track the traffic shifts as they happen and outperform the unidirectional network throughput by up to 8% without multiplexing virtual channel outputs.

3.4.3 Non-bursty Synthetic Traffic with Multiplexed VC Outputs

If the outputs of virtual channels are multiplexed, the number of inputs to the crossbar switch can be significantly reduced, especially in unidirectional networks. However, the use of multiplexers can limit the flexibility of switch allocation because fewer virtual channels can compete for the switch at a given cycle.

This limited flexibility does not significantly affect performance of *bit-complement*, *transpose* and *shuffle* because packet flow at each network node is in steady-state under these traffic patterns. If packet flow is in steady-state, each port at each network node has the same inflows and outflows of flits, which are bounded by the maximum outgoing bandwidth. Therefore, multiplexing corresponding to the maximum outgoing bandwidth does not affect throughput because we need not connect more virtual channels to the switch than the number of multiplexer outputs.

On the other hand, if the congestion at each link is not in steady-state as in the *uniform-random* example, each port sees a temporal mismatch between inflows and outflows of flits. If all virtual channels can compete for the switch without multiplexers, flits in ingress queues can be quickly pulled out as soon as the link to the next hop becomes less congested. The results show that the unidirectional networks have 10% less throughput under *uniform-random* when multiplexers are used, as they cannot pull out congested flits as fast as networks without multiplexers. Bidirectional networks have more multiplexer outputs than unidirectional networks because their maximum outgoing bandwidth is greater than unidirectional networks. Therefore, the size of crossbar switches of bidirectional networks increases, but they can send out more flits in congested ports than unidirectional networks. Consequently, the bidirectional networks outperform the unidirectional network throughput by up to 20% under *uniform-random* when virtual channel outputs are multiplexed, as shown in Figure 3-7.

3.4.4 Bursty Synthetic Traffic

The temporary nature of bursty traffic allows the bidirectional network to adjust the direction of each link to favor whichever direction is prevalent at the time, and results in throughput improvements across all traffic patterns (Figure 3-8). With bursty traffic, even *bit-complement*, for which the bidirectional network does not win over the unidirectional case without burstiness, shows a 20% improvement in total throughput because its symmetry is broken over short periods of time by the bursts. For the same reason, *shuffle* and *uniform-random* outperform the unidirectional network by 66% and 26% respectively, compared to 60% and 8% in non-bursty mode. Finally, *transpose* performance is the same as for the non-bursty case, because the traffic, if any, still only flows in one direction and requires no changes in link direction after the initial adaptation.

These results were obtained with virtual channels directly competing for the crossbar. We have simulated these examples with multiplexed VC outputs and the results have the same trends as in Figure 3-8, and therefore are not shown here.

3.4.5 Traffic of an H.264 Decoder Application

As illustrated in the example of *transpose* and *bit-complement*, bidirectional networks can significantly improve network performance when network flows are not symmetric. As opposed to synthetic traffic such as *bit-complement*, the traffic patterns in many real applications are not symmetric as data is processed by a sequence of modules. Therefore, bidirectional networks are expected to have significant performance improvement with many real applications. Figure 3-9 illustrates the performance of the bidirectional and the unidirectional networks under traffic patterns profiled from an H.264 decoder application, where the bidirectional network outperforms the unidirectional network by up to 35%. The results correspond to the case where virtual channels directly compete for the crossbar, and are virtually identical to the results with VC multiplexing.

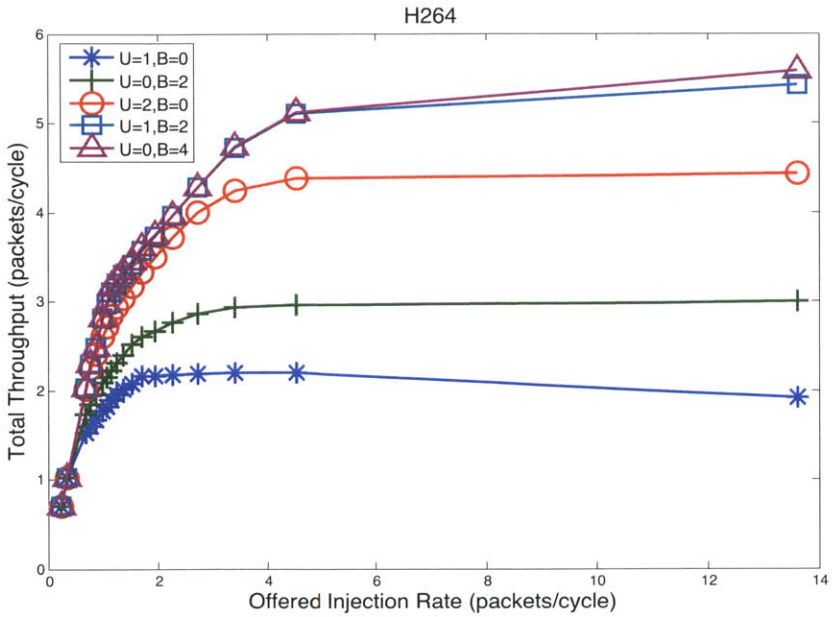


Figure 3-9: Throughput of BAN and unidirectional networks under H.264 decoder traffic

3.4.6 Link Arbitration Frequency

So far, our results have assumed that the bandwidth arbiter may alter the direction of every link on every cycle. While we believe this is realistic, we also considered the possibility that switching directions might require a dead cycle, in which case changing too often could limit the throughput up to 50%. We therefore reduced the arbitration frequency and examined the tradeoff between switching every N cycles

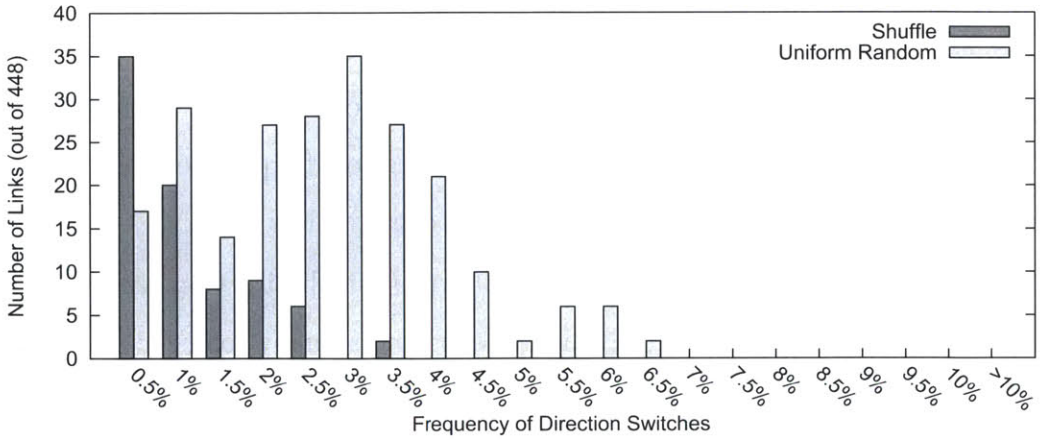


Figure 3-10: Frequency of direction changes on bidirectional links

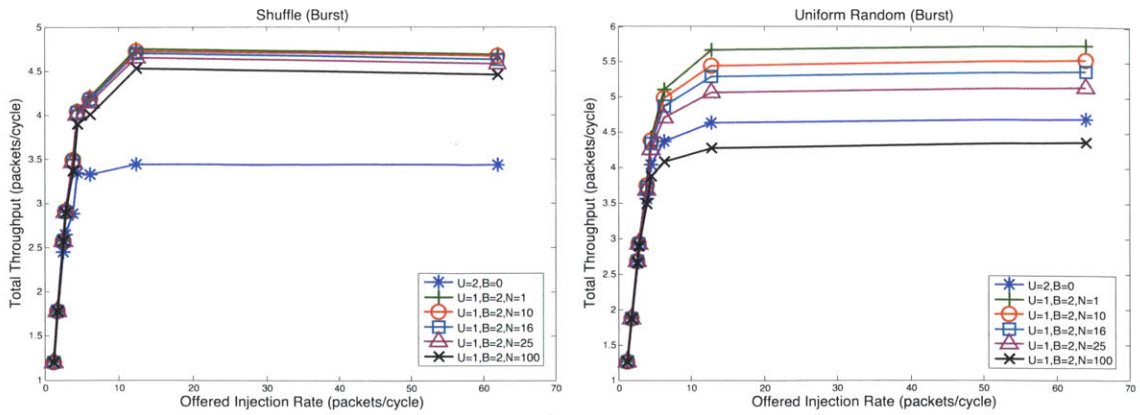


Figure 3-11: BAN performance under bursty traffic with various link arbitration periods (N)

(thereby lessening the impact of a dead cycle to $\frac{1}{N+1}$) and limiting the network's adaptivity to rapid changes in traffic patterns. The results in this section illustrate the relevant tradeoffs.

Figure 3-10 shows how often each bidirectional link actually changes its direction under bursty *shuffle* and *uniform-random* traffic: the x-axis shows how frequently links' directions change and the y-axis shows how many links switch that often. For example, under *shuffle* traffic, only 8% of the bidirectional links change their direction more frequently than once in two hundred cycles. Traffic exhibiting the *uniform-random* pattern, in comparison, is more symmetric than *shuffle*, and so the link directions change more often.

The observation that no link changes its direction more frequently than once in ten cycles led us to investigate how infrequent the link switches could be without significantly affecting performance. In Figure 3-11 we compare the performance of the bidirectional network under different link arbitration frequencies; as expected, throughput decreases when the links are allowed to switch less often.

Even with a switching period as large as 100 cycles, the bidirectional network still significantly outperforms the unidirectional design under many loads (e.g., by more than 20% for *shuffle*). In the case of *uniform-random*, however, the bidirectional network performance trails the unidirectional design when switching is infrequent. This is because, when each link arbitration decision lasts 100 cycles, any temporary benefit

from asymmetric bandwidth allocation is nullified by changes in traffic patterns, and, instead of improving throughput, the asymmetric allocations only serve to throttle down the total throughput compared to the unidirectional router.

Infrequent link switching, therefore, demands a more sophisticated link bandwidth arbiter that bases its decisions on the pressures observed over a period of time rather than on instantaneous measurements. For *uniform-random*, for example, the symmetry of uniform random traffic over time would cause the link bandwidths to be allocated evenly by such an arbiter, allowing it to match the performance of the unidirectional network.

3.5 Conclusions

We have proposed the notion of bandwidth-adaptive networks, given one concrete example of bidirectional links in a 2-D mesh, and evaluated it. Adaptivity is controlled by local pressure that is easily computed. While more comprehensive evaluation should be performed, adaptive bidirectional links provide better performance under both uniform and bursty traffic for the tested benchmarks.

We have focused on a mesh; however, adaptive bidirectional links can clearly be used in other network topologies. In adaptive routing decisions are made on a per-packet basis at each switch. In bandwidth-adaptive networks, decisions are made on a per-link basis. We believe this difference makes bandwidth-adaptivity more amenable to local decision making, though more rigorous analysis is required.

Chapter 4

On-chip Network Support for Fine-grained Thread Migration

4.1 Introduction

Two network-level optimization techniques were introduced in the previous chapters. These techniques aim to improve network performance without significantly increasing complexity. However, network performance is not the only thing that on-chip networks can provide. Because the network is tightly coupled with other components in a manycore system, it is capable of directly supporting system-level or application-level functionality (Section 1.2.3). In this chapter, we present Exclusive Native Context (ENC), the first deadlock-free fine-grained thread migration protocol built on an on-chip network. ENC demonstrates that a simple and elegant technique in an on-chip network can provide critical functional support for the higher-level application and system layers.

4.1.1 Thread Migration on CMPs

In SMP multiprocessor systems and multicore processors, process and thread migration has long been employed to provide load and thermal balancing among the processor cores. Typically, migration is a direct consequence of thread scheduling

and is performed by the operating system (OS) at timeslice granularity; although this approach works well for achieving long-term goals like load balancing, the relatively long periods, expensive OS overheads, and high communication costs have generally rendered fast thread migration impractical [90].

Recently, however, several proposals with various aims have centered on thread migration too fine-grained to be effectively handled via the OS. In the design-for-power domain, rapid thread migration among cores in different voltage/frequency domains has allowed less demanding computation phases to execute on slower cores to improve overall power/performance ratios [72]; in the area of reliability, migrating threads among cores has allowed salvaging of cores which cannot execute some instructions because of manufacturing faults [71]; finally, fast instruction-level thread migration has been used in lieu of coherence protocols or remote accesses to provide memory coherence among per-core caches [46] [55]. The very fine-grained nature of the migrations contemplated in these proposals—a thread must be able to migrate immediately if its next instruction cannot be executed on the current core because of hardware faults [71] or to access data cached in another core [46]—demands fast, hardware-level migration systems with decentralized control, where the decision to migrate can be made *autonomously* by each thread.

4.1.2 Demand for a New Thread Migration Protocol

The design of an efficient fine-grained thread migration protocol has not, however, been addressed in detail. The foremost concern is avoiding deadlock: if a thread context can be blocked by other contexts during migration, there is an additional resource dependency in the system which may cause the system to deadlock. But most studies do not even discuss this possibility: they implicitly rely on expensive, centralized migration protocols to provide deadlock freedom, with overheads that preclude frequent migrations [41], [62], or limit migrations to a core's local neighborhood [78]. Some fine-grain thread migration architectures simply give up on deadlock avoidance and rely on expensive recovery mechanisms (e.g., [59]).

With this in mind, we introduce a novel thread migration protocol called Exclusive

Native Context (ENC). To the best of our knowledge, ENC is the first on-chip network solution to guarantee freedom from deadlock for general fine-grain thread migration without requiring handshaking. Our scheme is simple to implement and does not require any hardware beyond that required for hardware-level migrations; at the same time, it decouples the performance considerations of on-chip network designs from deadlock analysis, freeing architects to consider a wide range of on-chip network designs.

In the rest of this chapter,

- we present ENC, a novel deadlock-free fine-grained thread migration protocol;
- we show how deadlock arises in other migration schemes, and demonstrate that ENC is deadlock-free;
- we show that ENC performance on SPLASH-2 application benchmarks [92] running under a thread-migration architecture [46] is on par with an idealized deadlock-free migration scheme that relies on infinite resources.

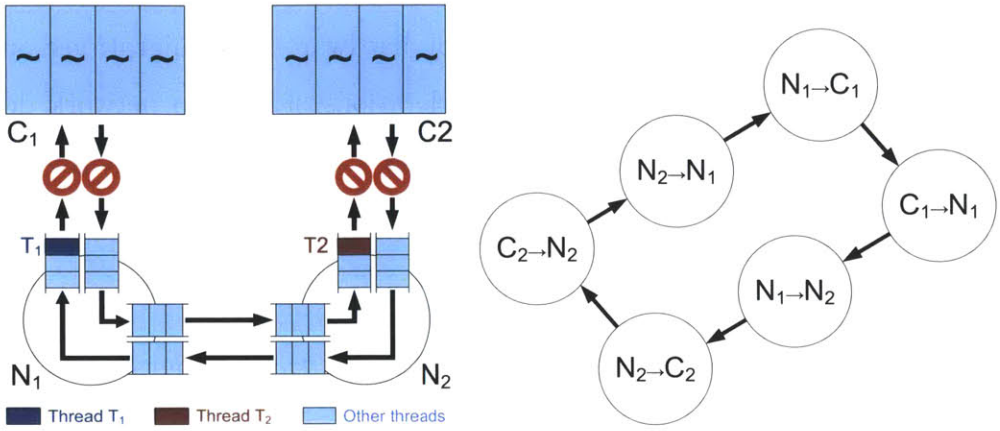
4.2 Deadlock in Thread Migration

4.2.1 Protocol-level Deadlock

Most studies on on-chip networks focus on the network itself and assume that a network packet *dies* soon after it reaches its destination core—for example, the result of a memory load request might simply be written to its destination register. This assumption simplifies deadlock analysis because the dead packet no longer holds any resources that might be needed by other packets, and only *live* packets are involved in deadlock scenarios.

With thread migration, however, the packet carries an execution context, which moves to an execution unit in the core and *occupies* it until it migrates again to a different core. Thus, unless migrations are centrally scheduled such that the migrating context always finds available space at its destination, execution contexts occupying

a core can block contexts arriving over the network, creating additional deadlock conditions that conventional on-chip network deadlock analysis does not consider.



(a) Network Buffers in a Protocol-level Deadlock

(b) Channel Dependency Graph

Figure 4-1: Protocol-level deadlock of fine-grain, autonomous thread migration

For example, suppose a migrating thread T_1 in Figure 4-1(a) is heading to core C_1. Although T_1 arrives at routing node N_1 directly attached to C_1, all the execution units of C_1 are occupied by other threads ($\sim$), and one of them must migrate to another core for T_1 to make progress. But at the same time, thread T_2 has the same problem at core C_2, so the contexts queued behind T_2 are backed up all the way to C_1 and prevent a C_1 thread from leaving. So T_1 cannot make progress, and the contexts queued behind it have backed up all the way to C_2, preventing any of C_2's threads from leaving, and completing the deadlock cycle. Figure 4-1(b) illustrates this deadlock using a channel dependency graph (CDG) [18] where nodes correspond to channels of the on-chip network and edges to dependencies associated with making progress on the network.

We call this type of deadlock a *protocol-level deadlock*, because it is caused by the migration protocol itself rather than the network routing scheme. Previous studies involving rapid thread migration typically either do not discuss protocol-level deadlock, implicitly relying on a centralized deadlock-free migration scheduler [41, 62, 78], using deadlock detection and recovery [59], employing a cache coherence protocol to migrate contexts via the cache and memory hierarchy, effectively providing a very large

Core and Migration	
Core architecture	single-issue, two-way multithreading
The size of a thread context (relative to the size of network flit)	4 flits
Number of threads	64
Number of hotspots	1, 2, 3 and 4
Migration interval	100 cycles
On-chip Network	
Network topology	8-by-8 mesh
Routing algorithms	Dimension-order wormhole routing
Number of virtual channels	2 and 4
The size of network buffer (relative to the size of context)	4 per link or 20 per node
The size of context queue (relative to the size of context)	0, 4 and 8 per core

Table 4.1: Simulation details for synthetic migration patterns with hotspot cores

buffer to store contexts [72], or employing slow handshake-based context swaps [71]. All of these approaches have substantial overheads, motivating the development of an efficient network-level deadlock-free migration protocol.

4.2.2 Evaluation with Synthetic Migration Benchmarks

As a non-deadlock-free migration protocol, we consider the naturally arising SWAP scheme, implicitly assumed by several works: whenever a migrating thread T_1 arrives at a core, it evicts the thread T_2 currently executing there and sends it back to the core where T_1 originated. Although intuitively one might expect that this scheme should not deadlock because T_2 can be evicted into the slot that T_1 came from, this slot is not reserved for T_2 and another thread might migrate there faster, preempting T_2; it is therefore not guaranteed that T_2 will exit the network and deadlock may arise. (Although adding a handshake protocol with extra buffering can make SWAP deadlock-free [71], the resulting scheme is too slow for systems which require frequent migrations).

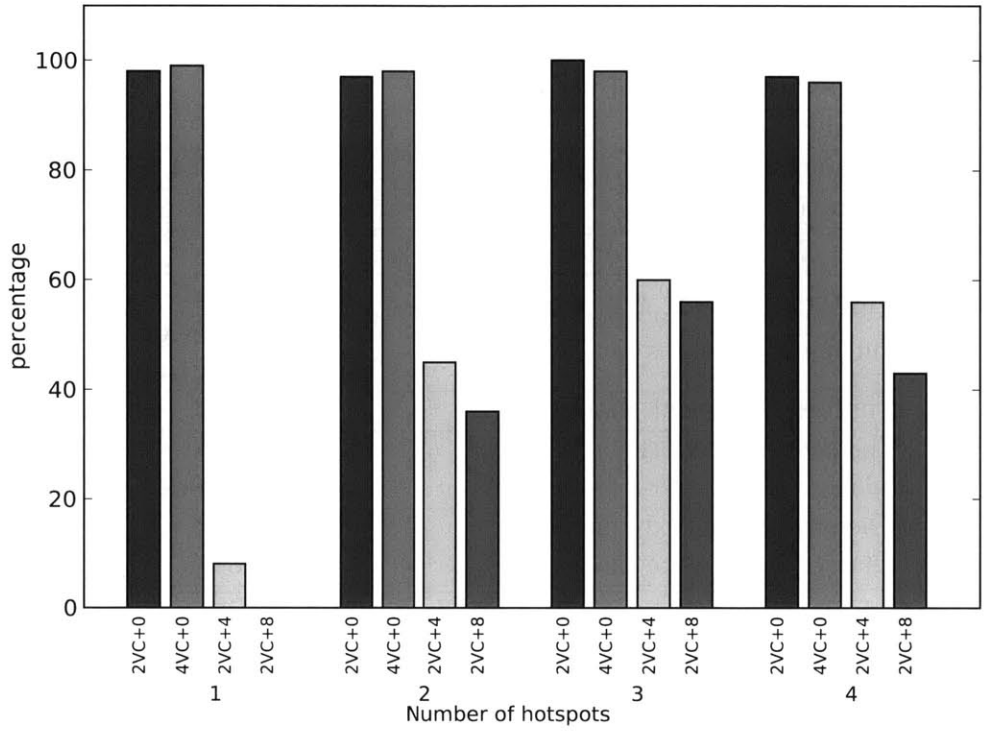


Figure 4-2: Deadlock scenarios with synthetic sequences of fine-grained migrations on 2VC and 4VC networks

In order to examine how often the migration system might deadlock in practice, we used a synthetic migration benchmark where each thread keeps migrating between the initial core where it was spawned and a *hotspot* core. (Since migration typically occurs to access some resource at a core, be it a functional unit or a set of memory locations, such hotspots naturally arise in multithreaded applications). We used varying numbers (one to four) of randomly assigned hotspots, and 64 randomly located threads that made a thousand migrations to destinations randomly chosen among their originating core and the various hotspots every 100 cycles. To stress the migartion framework as in a fine-grain migration system, we chose the migration interval of 100 cycles. We used the cycle-level network-on-chip simulator DARSIM [57], suitably modified with a migration mechanism, to model a 64-core system connected by a 2D mesh interconnect. Each on-chip network router had enough network buffers to hold 4 thread contexts on each link with either 2 or 4 virtual channels; we also examined

the case where each core has a context queue to hold arriving thread contexts when there are no available execution units. We assumed Intel Atom-like x86 cores with execution contexts of 2 Kbits [72] and enough network bandwidth to fit each context in four or eight flits. Table 4.1 summarizes the simulation setup.

Figure 4-2 shows the percentage of runs (out of the 100) that end with deadlock under the SWAP scheme. 2VC+0 and 4VC+0 correspond to networks with 2 and 4 virtual channels, respectively. 2VC+4 and 2VC+8 are 2 virtual-channel networks which have an extra buffer for 4 and 8 thread contexts at each node. Without the extra buffer, nearly all experiments end in deadlock. Further, even though context buffering can *reduce* deadlock, deadlock still occurs at a significant rate for the tested configurations.

The synthetic benchmark results also illustrate that susceptibility to deadlock depends on migration patterns: when there is only one hotspot, the migration patterns across threads are usually not cyclic because each thread just moves back and forth between its own private core and only one shared core; when there are two or more hotspots and threads have more destinations, on the other hand, their paths intersect in more complex ways, making the system more prone to deadlock. Although small context buffers prevent deadlock with some migration patterns, they do not ensure deadlock avoidance because there are still deadlock cases.

4.3 Exclusive Native Context Protocol

ENC takes a network-based approach to provide deadlock freedom. Unlike coarse-grain migration protocols, ENC allows autonomous thread migrations. To enable this, the new thread context may evict one of the thread contexts executing in the destination core, and ENC provides the evicted thread context a *safe path* to another core on which it will never be blocked by other threads that are also in transit.

To provide the all-important safe path for evicted threads, ENC uses a set of policies in core scheduling, routing, and virtual channel allocation.

Each thread is set as a *native context* of one particular core, which reserves a

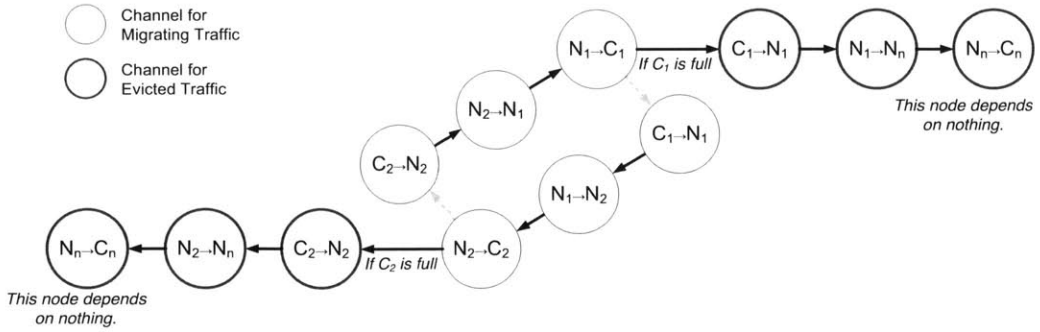


Figure 4-3: Acyclic channel dependency graph of ENC

register file (and other associated context state) for the thread. Other threads cannot use the reserved resource even if it is not being used by the native context. Therefore, a thread will always find an available resource every time it arrives at the core where the thread is a native context. We will refer to this core as the thread's *native core*.

Dedicating resources to native contexts requires some multithreading support in the cores. If a thread may migrate to an arbitrary core which may have a different thread as its native context, the core needs to have an additional register file (i.e., a *guest context*) to accept a non-native thread because the first register file is only available to the *native* context. Additionally, if a core has multiple native contexts, there must be enough resources to hold all of its native contexts simultaneously so no native thread is blocked by other native threads. The number of additional registers depends on the size of context, which varies greatly among different architectures; while the size of a thread context is usually less than 2Kbits for a simple 32-bit RISC core (32 general-purpose registers, plus additional registers such as program counter, possibly including a few TLB entries), complex cores with additional resources such as floating-point registers have more than 2× larger contexts. However, it is a reasonable assumption that an efficient fine-grain, migration-based architecture will require some level of multithreading, in order to prevent performance degradation when multiple threads compete for the resources of the same core.

If an arriving thread is not a native context of the core, it may be temporarily blocked by other non-native threads currently on the same core. The new thread

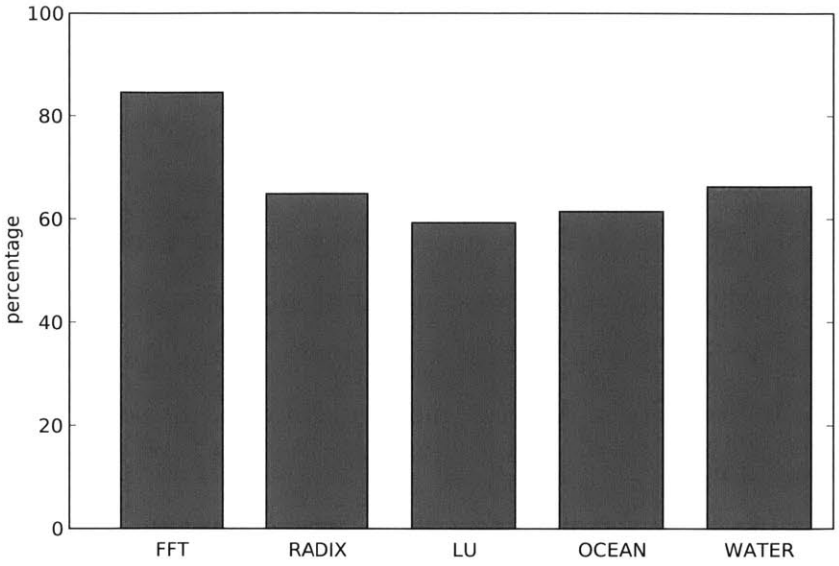


Figure 4-4: The percentage of accesses to a threads *native* core in SPLASH-2 applications

evicts one of the executing non-native threads and takes the released resource. Note that a thread never evicts a native context of the destination core because the resource is usable only by the native context. To prevent livelock, however, a thread is not evicted unless it has executed at least one instruction since it arrived at the current core. That is, an existing thread will be evicted by a new thread only if it has made some progress in its current visit on the core. This is why we say the arriving thread may be temporarily blocked by other non-native threads.

Where should the native core be? In the first-touch data placement policy [58] we assume here, each thread's stack and private data are assigned to be cached in the core where the thread originates. We reasoned, therefore, that most accesses made by a thread will be to its originating core (indeed, Figure 4-4 shows that in the SPLASH-2 benchmarks we used, about 60%–85% of a thread's accesses are to its native core). We therefore select each thread's originating core as its native core.

In what follows, we first describe a basic, straightforward version of ENC, which we term ENC0, and then describe a better-performing optimized version.

4.3.1 The Basic ENC Algorithm (ENC0)

Whenever a thread needs to move from a non-native core to a destination core, ENC0 first sends the thread to its native core which has a dedicated resource for the thread. If the destination core is not the native core, the thread will then move from its native core to the destination core. Therefore, from a network standpoint, a thread movement either ends at its native core or begins from its native core. Since a thread arriving at its native core is guaranteed to be unloaded from the network, any migration is fully unloaded (and therefore momentarily occupies no network resources) somewhere along its path.

To keep the migrations deadlock-free, however, we must also ensure that movements destined for a native core actually get there without being blocked by any other movements; otherwise the native-core movements might never arrive and be unloaded from the network. The most straightforward way of ensuring this is to use two sets of virtual channels, one for to-native-core traffic and the other for from-native-core traffic. If the baseline routing algorithm requires only one virtual channel to prevent network-level deadlock like dimension-order routing, ENC0 requires a minimum of two virtual channels per link to provide protocol-level deadlock avoidance. Note that ENC0 may work with any baseline routing algorithm for a given source-destination pair, such as Valiant [87] or O1TURN [76], both of which require two virtual channels to avoid deadlock. In this case, ENC0 will require four virtual channels.

4.3.2 The Full ENC Algorithm

Although ENC0 is simple and straightforward, it suffers the potential overhead of introducing an intermediate destination for each thread migration: if thread T wishes to move from core A to B, it must first go to N, the native core for T. In some cases, this overhead might be significant: if A and B are close to each other, and N is far away, the move may take much longer than if it had been a direct move.

To reduce this overhead, we can augment the ENC0 algorithm by distinguishing *migrating* traffic and *evicted* traffic: the former consists of threads that wish to mi-

grate on their own because, for example, they wish to access resources in a remote core, while the latter corresponds to the threads that are evicted from a core by another arriving thread.

Whenever a thread is *evicted*, ENC, like ENC0, sends the thread to its native core, which is guaranteed to accept the thread. We will not therefore have a chain of evictions: even if the evicted thread wishes to go to a different core to make progress (e.g., return to the core it was evicted from), it must first visit its native core, get unloaded from the network, and then move again to its desired destination. Unlike ENC0, however, whenever a thread migrates on its own accord, it may go directly to its destination without visiting the home core. (Like ENC0, ENC must guarantee that evicted traffic is never blocked by migrating traffic; as before, this requires two sets of virtual channels). Note that network packets always travel within the same set of virtual channels.

Based on these policies, the ENC migration algorithm works as follows:

1. If a native context has arrived and is waiting on the network, move it to a reserved register file and proceed to Step 3.

2. (a) If a non-native context is waiting on the network and there is an available register file for non-native contexts, move the context to the register file and proceed to Step 3.

   (b) If a non-native context is waiting on the network and all the register files for non-native contexts are full, choose one among the threads that have finished executing an instruction on the core[1] *and* the threads that want to migrate to other cores. Send the chosen thread to its *native* core on the virtual channel set for evicted traffic. Then, advance to the next cycle. (No need for Step 3).

3. Among the threads that want to migrate to other cores, choose one and send it to the desired destination on the virtual channel set for migrating traffic. Then,

[1] No instructions should be in flight.

advance to the next cycle.

This algorithm effectively breaks the cycle of dependency of migrating traffic and evicted traffic. Figure 4-3 illustrates how ENC breaks the cyclic dependency shown in Figure 4-1(b), where C_n denotes the native core of the evicted thread, and N_n its attached router node.

There is a subtlety when a migrating context consists of multiple flits and the core cannot send out an entire context all at once. For example, the core may find no incoming contexts at cycle 0 and start sending out an executing context T_1 to its desired destination, but before T_1 completely leaves the core, a new migrating context, T_2, arrives at the core and is blocked by the remaining flits of T_1. Because T_1 and T_2 are on the same set of virtual channels for migration traffic, a cycle of dependencies may cause a deadlock. To avoid this case, the core must inject migration traffic only if the *whole* context can be moved out from the execution unit so arriving contexts will not be blocked by incomplete migrations; this can easily be implemented by monitoring the available size of the first buffer on the network for migration traffic or by adding an additional outgoing buffer whose size is one context size.

Although both ENC0 and ENC are provably deadlock-free under deadlock-free routing because they eliminate all additional dependencies due to limited context space in cores, we confirmed that they are deadlock-free with the same synthetic benchmarks used in Section 4.2.2. We also simulated an incomplete version of ENC that does *not* consider the aforementioned subtlety and sends out a migrating context if it is possible to push out its first flit. While ENC0 and ENC did not deadlock, deadlocks occurred with the incomplete version because it does not provide a safe path for evicted traffic in the case when a migrating context is being sequentially injected to the network; this illustrates that fine-grained migration is very susceptible to deadlock and migration protocols need to be carefully designed.

4.4 Performance Evaluation

4.4.1 Baseline Protocols and Simulated Migration Patterns

We compared the performance overhead of ENC0 and ENC to the baseline SWAP algorithm described in Section 4.2.2. However, as SWAP can deadlock, in some cases the execution might not finish. Therefore, we also tested SWAPinf, a version of SWAP with an infinite context queue to store migrating thread contexts that arrive at the core; since an arriving context can always be stored in the context queue, SWAPinf never deadlocks. Although impractical to implement, SWAPinf provides a useful baseline for performance comparison. We compared SWAP and SWAPinf to ENC0 and ENC with two virtual channels. The handshake version of SWAP was deemed too slow to be a good baseline for performance comparison.

In order to see how ENC would perform with arbitrary migration patterns, we first used a random sequence of migrations in which each thread may migrate to any core at a fixed interval of 100 cycles. In addition, we also wished to evaluate real applications running under a fine-grained thread-migration architecture. Of the three such architectures described in Section 4.1, we rejected core salvaging [71] and ThreadMotion [72] because the thread's migration patterns do not depend on the application itself but rather on external sources (core restrictions due to hard faults and the chip's thermal environment, respectively), and could conceivably be addressed with synthetic benchmarks. We therefore selected the EM^2 architecture [46], which migrates threads to a given core to access memory exclusively cached in that core; migrations in EM^2 depend intimately on the application's access patterns and are difficult to model using synthetic migration patterns.

We used the same simulation framework as described in Section 4.2.2 to examine how many cycles are spent on migrating thread contexts.

4.4.2 Network-Independent Traces (NITs)

While software simulation provides the most flexibility in the development of many-core architectures, it is severely constrained by simulation time. For this reason, common simulation methods do not faithfully simulate every detail of target systems, to achieve reasonably accurate results in an affordable time. For example, Graphite [61] provides very efficient simulation of a many-core system based on the x86 architecture. However, it has yet to provide faithful simulation of network buffers. Therefore, Graphite simulator does not model the performance degradation due to head-of-line blocking, and moreover, deadlock cannot be observed even if the application being simulated may actually end up in deadlock.

On the other hand, most on-chip network studies use a detailed simulator that accurately emulates the effect of network buffers. However, they use simple traffic generators rather than simulating actual cores in detail. The traffic generator often replays *network traces* captured from application profiling, in order to mimic the traffic pattern of real-world applications. It, however, fails to mimic complex dependency between operations, because most communication in many-core systems depends on the previous communication. For example, a core may need to first receive data from a producer, before it processes the data and sends it to a consumer. Obviously, if the data from the producer arrives later than in profiling due to network congestion, sending processed data to the consumer is also delayed. However, network traces typically only give the absolute time when packets are sent, so the core may send processed data to the consumer prior to it even receiving the data from its producer! In other words, the network-trace approach fails to realistically evaluate application performance, because the timing of packet generation, which depends on on-chip network conditions, is assumed *before* the actual simulation of the network.

It is very important to reflect the behavior of network conditions, because it is critical not only for performance, but also to verify that network conditions don't cause deadlock. Therefore, we use DARSIM [57], a highly configurable, cycle-accurate on-chip network simulator. Instead of using network traces, however, we generate

network-independent traces (NITs) from application profiling. Unlike standard application traces, NITs keep inter-thread dependency information and relative timings instead of absolute packet injection times; the dependencies and relative timings are replayed by an interpreter module added to the network simulator. By replacing absolute timestamps with dependencies and relative timings, NITs allow cores to "respond" to messages from other cores once they have arrived, and solve the consumer-before-producer problem that occurs with network traces.

The NITs we use for EM^2 migration traffic record memory instruction traces of all threads, which indicate the home core of each memory instruction and the number of cycles it takes to execute all non-memory instructions between two successive memory instructions. With these traces and the current location of threads, a simple interpreter can determine whether each memory instruction is accessing memory cached on the current core or on a remote core; on an access to memory cached in a remote core, the interpreter initiates a migration of the corresponding thread. After the thread arrives at the home core and spends the number of cycles specified in the traces for non-memory operations, the interpreter does the same check for the next memory instruction.

The interpreter does not, of course, behave exactly the same as a real core does. For one, it does not consider lock/barrier synchronization among threads; secondly, it ignores possible dependencies of the actual *memory addresses* accessed on network performance (consider, for example, a multithreaded work-queue implemented via message passing: the memory access patterns of the program will clearly depend on the order in which the various tasks arrive in the work queue, which in turn depends on network performance). Nevertheless, NITs allow the system to be simulated in a much more realistic way by using memory traces rather than network traces.

4.4.3 Simulation Methodology

For the evaluation under arbitrary migration patterns, we used a synthetic sequence of migrations for each number of hotspots as in Section 4.2.2. We also chose five applications from the SPLASH-2 benchmark suite to examine application-specific migration

Protocols and Migration Patterns	
Migration Protocols	SWAP, SWAPinf, ENC0 and ENC
Migration Patterns	a random sequence & 5 SPLASH-2 applications: FFT, RADIX, LU (contiguous) WATER (n-squared) OCEAN (contiguous)
Core	
Core architecture	single-issue, two-way multithreading EM^2
The size of a thread context	4, 8 flits
Number of threads	64
On-chip Network	
Network topology	8-by-8 mesh
Routing algorithms	Dimension-order wormhole routing
Number of virtual channels	2
The size of network buffer (relative to the size of context)	4 per link (20 per node)
The size of context queue	∞ for SWAPinf, 0 otherwise

Table 4.2: Simulation details for ENC with random migration pattern and SPLASH-2 applications

patterns, namely FFT, RADIX, LU (contiguous), WATER (n-squared), and OCEAN (contiguous), which we configured to spawn 64 threads in parallel. Then we ran those applications using Pin [2] and Graphite [61], to generate memory instruction traces. Using the traces and the interpreter as described in the previous section, we executed the sequences of memory instructions on DARSIM.

As in Section 4.2.2, we first assumed the context size is 4 flits. However, we also used 8-flit contexts to examine how ENC's performance overhead would change if used with an on-chip network with less bandwidth, or a baseline architecture which has very large thread context size. The remaining simulation setup is similar to Section 4.2.2. Table 4.2 summarizes the simulation setup used for the performance evaluation.

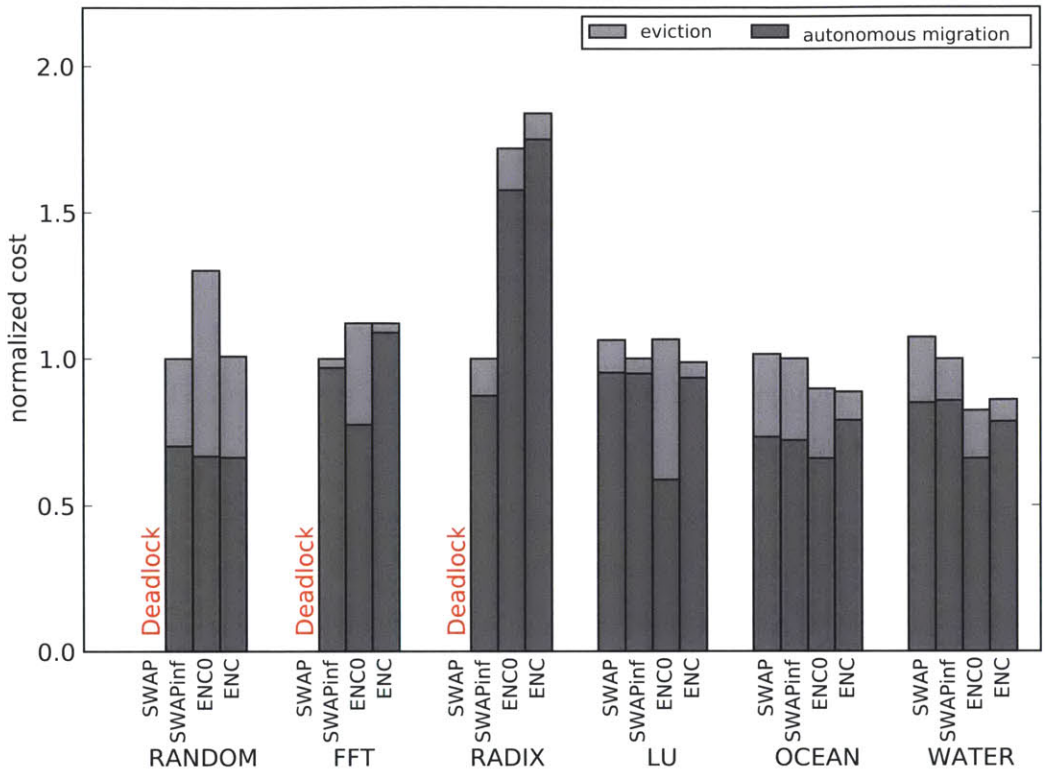


Figure 4-5: Total migration cost of ENC and SWAP with 4-flit contexts

4.4.4 Simulation Results

Figure 4-5 shows the total migration cost in each migration pattern normalized to the cost in SWAPinf when the context size is equivalent to four network flits. Total migration cost is the sum of the number of cycles that each thread spends between when it moves out of a core and when it enters another. First of all, the SWAP algorithm causes deadlock in FFT and RADIX, as well as in RANDOM, when each thread context migrates in 4 network flits. As we will see in Figure 4-8, LU and OCEAN also end up with deadlock with the context size of 8 flits. Our results illustrate that real applications are also prone to deadlock if they are not supported by a deadlock-free migration protocol, as mentioned in Section 4.2.2.

Deadlock does not occur when SWAPinf is used due to the infinite context queue. The maximum number of contexts at any moment in a context queue is smaller

RANDOM	FFT	RADIX	LU	OCEAN	WATER
8	61	60	61	61	61

Table 4.3: Maximum size of context queues in SWAPinf relative to the size of a thread context

in RANDOM than in the application benchmarks because the random migration evenly distributes threads across the cores so there is no heavily congested core (cf. Table 4.3). However, the maximum number of contexts is over 60 for all application benchmarks, which is more than 95% of all threads on the system. This discourages the use of context buffers to avoid deadlock.[2]

Despite the potential overhead of ENC described earlier in this section, both ENC and ENC0 have comparable performance, and are overall 11.7% and 15.5% worse than SWAPinf, respectively. Although ENC0 has relatively large overhead of 30% in total migration cost under the random migration pattern, ENC reduces the overhead to only 0.8%. Under application-specific migration patterns, the performance largely depends on the characteristics of the patterns; while ENC and ENC0 have significantly greater migration costs than SWAPinf under RADIX, they perform much more competitively in most applications, sometimes better as in applications such as WATER and OCEAN. This is because each thread in these applications mostly works on its private data; provided a thread's private data is assigned to its native core, the thread will mostly migrate to the native core (cf. Figure 4-4). Therefore, the native core is not only a safe place to move a context, but also the place where the context most likely makes progress. This is why ENC0 usually has less cost for autonomous migration, but higher eviction costs. Whenever a thread migrates, it needs to be "evicted" to its native core. After eviction, however, the thread need not migrate again if its native core was its migration destination.

The effect of the portion of native cores in total migration destinations can be seen in Figure 4-6, showing total migration distances in hop counts normalized to the SWAPinf case. When the destinations of most migrations are native cores, such as

[2]Note that, however, the maximum size of context buffers from the simulation results is not a necessary condition, but a sufficient condition to prevent deadlock.

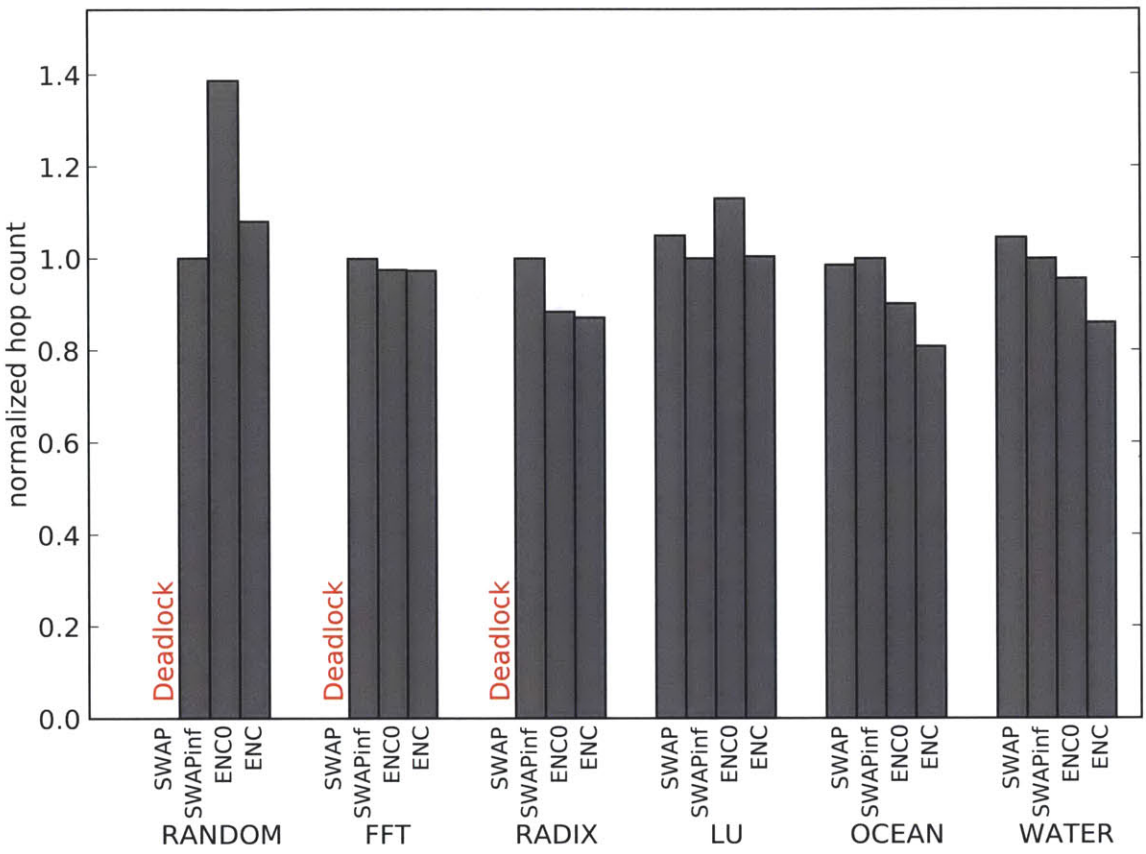


Figure 4-6: Total migration distance of ENC and SWAP for various SPLASH-2 benchmarks.

in FFT, ENC has not much different total migration distance from SWAPinf. When the ratio is lower, such as in LU, the migration distance for ENC is longer because it is more likely for a thread to migrate to non-native cores after it is evicted to its native core. This also explains why ENC has the most overhead in total migration distance under random migrations because the least number of migrations are going to native cores.

Even when the migrating thread's destination is often not its native core, ENC has an overall migration cost similar to SWAPinf as shown in LU, because it is less affected by network congestion than SWAPinf. This is because ENC effectively distributes network traffic over the entire network, by sending out threads to their native cores. Figure 4-7 shows how many cycles are spent on migration due to congestion, normalized to the SWAPinf case. ENC and ENC0 have less congestion costs under

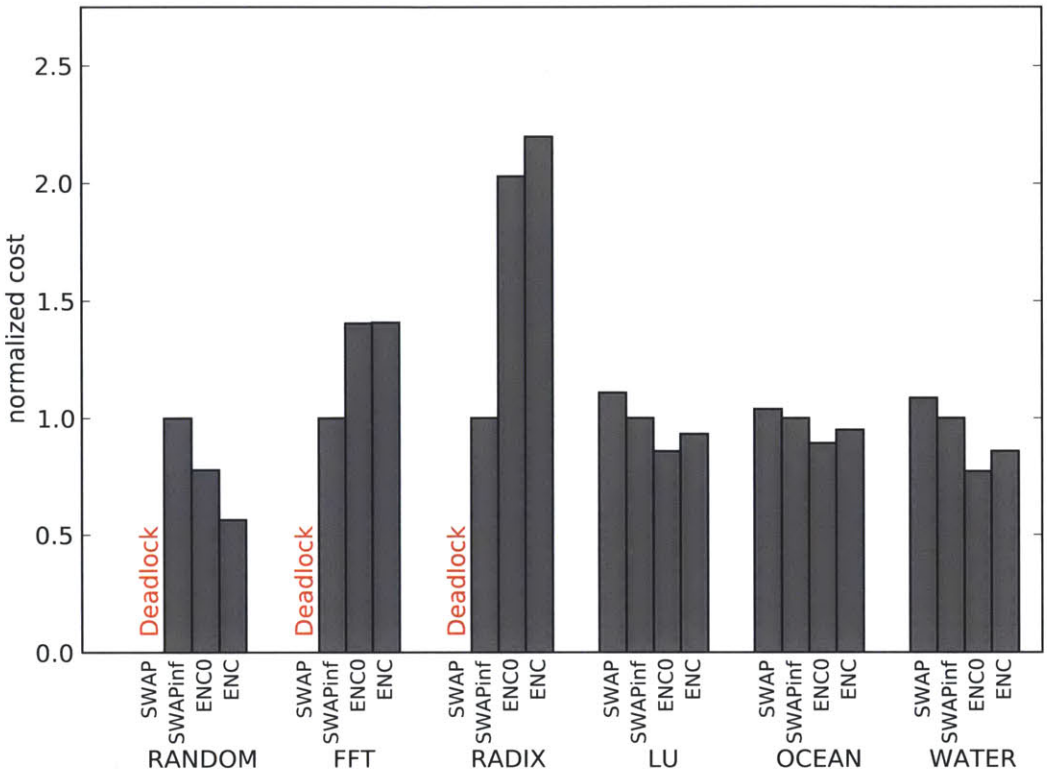


Figure 4-7: Part of migration cost of ENC and SWAP due to congestion

RANDOM, LU, OCEAN, and WATER. This is analogous to the motivation behind the Valiant algorithm [87]. One very distinguishable exception is RADIX; while the migration distances of ENC/ENC0 are similar to SWAPinf because the native-core ratio is relatively high in RADIX, they are penalized to a greater degree by congestion than SWAPinf. This is because other applications either do not cause migrations as frequently as RADIX, or their migration traffic is well distributed because threads usually migrate to nearby destinations only.

If the baseline architecture has a large thread context or an on-chip network has limited bandwidth to support thread migration, each context migrates in more network flits which may affect the network behavior. Figure 4-8 shows the total migration costs when a thread context is the size of eight flits. As the number of flits for a single migration increases, the system sees more congestion. As a result,

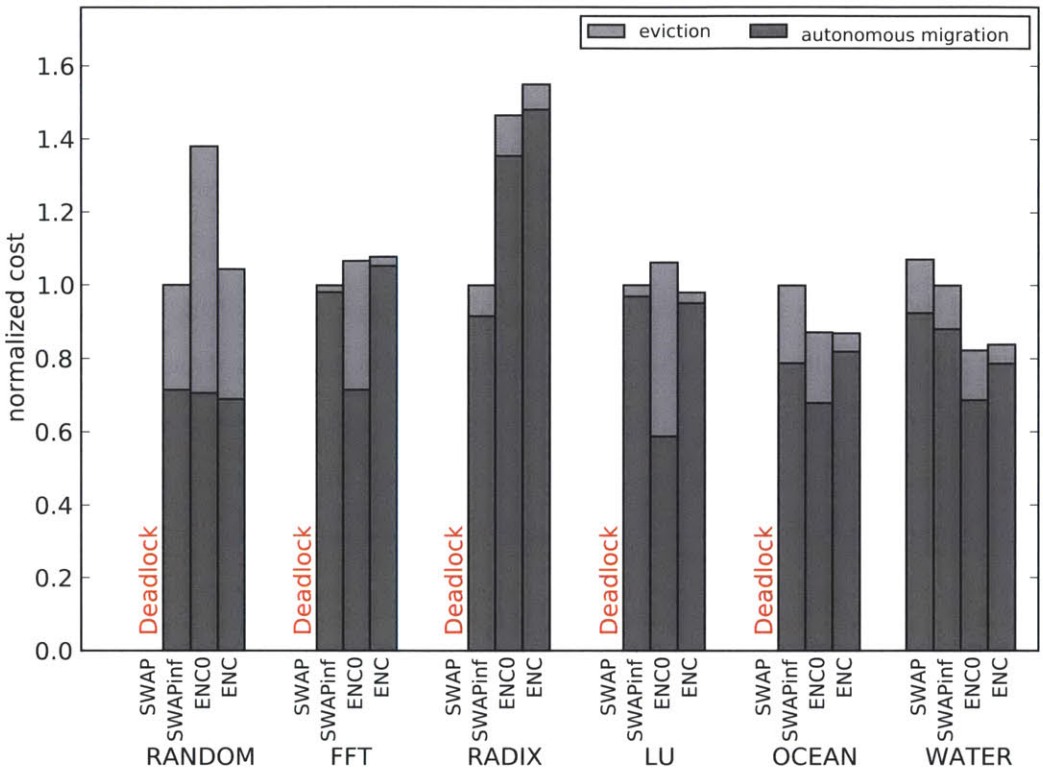


Figure 4-8: Total migration cost of ENC and SWAP with 8-flit contexts

the migration costs increase by 39.2% across the migration patterns and migration protocols. While the relative performance of ENC/ENC0 to SWAPinf does not change much for most migration patterns, the increase in the total migration cost under RADIX is greater with SWAPinf than with ENC/ENC0 as the network becomes saturated with SWAPinf too. Consequently, the overall overhead of ENC and ENC0 with the context size of 8 flits is 6% and 11.1%, respectively. The trends shown in Figure 4-6 and Figure 4-7 also hold with the increased size of thread context.

4.5 Conclusions

We have developed ENC, deadlock-free migration protocol for general fine-grain thread migration. Using ENC, threads can make autonomous decisions on when

and where to migrate; a thread may just start traveling when it needs to migrate, without being scheduled by any global or local arbiter. Therefore, the migration cost is only due to the network latencies in moving thread contexts to destination cores, possibly via native cores.

Compared to a baseline SWAPinf protocol which assumes infinite queues, ENC has an average of 11.7% overhead for overall migration costs under various types of migration patterns. The performance overhead depends on migration patterns, and under most of the synthetic and application-specific migration patterns used in our evaluation ENC shows negligible overhead or performs even better; although ENC may potentially increase the total distance that threads migrate by evicting threads to their native cores, it did not result in higher migration cost in many cases because evicted threads often need to go to the native core anyway, and intermediate destinations can reduce network congestion.

While the performance overhead of ENC remains low in most migration patterns, a baseline SWAP protocol actually ends up with deadlock, not only for synthetic migration sequences but also for real applications. Considering this, ENC is a very compelling mechanism for any architecture that exploits very fine-grain thread migrations and which cannot afford conventional, expensive migration protocols.

Finally, ENC is a flexible protocol that can work with various on-chip networks with different routing algorithms and virtual channel allocation schemes. One can imagine developing various ENC-based on-chip networks optimized for performance under a specific thread migration architecture.

Chapter 5

Physical Implementation of On-chip Network for EM^2

5.1 Introduction

Like many other research projects, PROM, BAN, and ENC all focus on specific target components; PROM on routing, BAN on the network links, and ENC on the migration protocol. In their evaluation, other system layers that are not closely related to the main ideas are represented in a simplified or generalized form. In this way, researchers can concentrate their efforts on the key research problem and the solution can be evaluated not only in one specific design instance but also with many different environments.

This approach, however, makes it hard to take into account every detail in the whole system. For example, architects often face credibility problems if they do not fully discuss related circuit-level issues.

Therefore, building the entire system is a very important experience in computer architecture research because it reveals every issue that might not be obvious at the architectural level. In this chapter, we share our hands-on experience in the physical implementation of the on-chip network for Execution Migration Machine (EM^2), an ENC-based 110-core processor in 45nm ASIC technology.

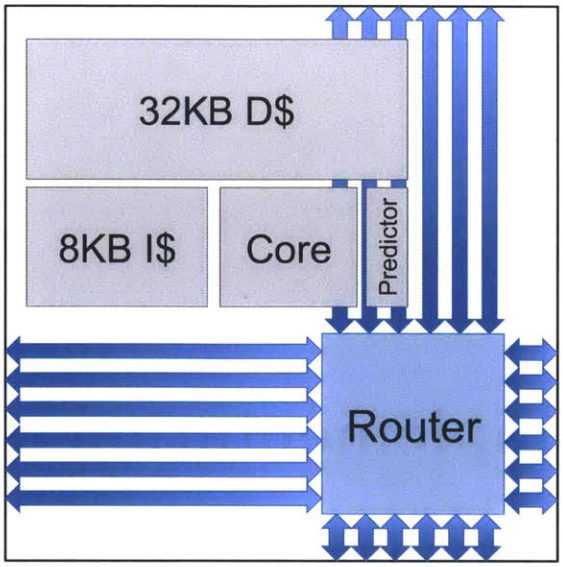


Figure 5-1: EM^2 Tile Architecture

5.2 EM^2 Processor

EM^2 is a large-scale CMP based on fine-grained hardware-level thread migration [54], which implements ENC to facilitate instruction-level thread migration. We have taped out our design in a 110-core CMP, where the chip occupies 100mm^2 in 45nm technology.

5.2.1 Shared Memory Model

The most distinctive feature of EM^2 is its simple and scalable shared memory model based on remote cache access and thread migration [56]. As in traditional NUCA architectures, each address in the system is assigned to a unique core where it may be cached; by allowing data to be cached only at a single location, the architecture scales trivially and properties like sequential consistency are easy to guarantee. To access data cached at a remote core, EM^2 can either send a traditional remote access (RA) request [27], or migrate the execution context to the core that is "home" for that data. Unlike RA-only machines, it can take advantage of available data locality because migrating the execution context allows the thread to make a sequence of local

accesses while it stays at the destination core.

5.2.2 On-chip Network Architecture

EM^2 has three types of on-chip traffic: migration, remote access, and off-chip memory access. Although it is possible for this traffic to share on-chip interconnect channels, this would require suitable arbiters (and possibly deadlock recovery logic), and would significantly expand the state space to be verified. To avoid this, we chose to trade off area for simplicity, and route traffic via six separate channels, which is sufficient to ensure deadlock-free operation [11].

Further, the six channels are implemented as six physically separate on-chip networks, each with its own router in every tile. While using a single network with six virtual channels would have utilized available link bandwidth more efficiently and made inter-tile routing simpler, it would have exponentially increased crossbar size and significantly complicated the allocation logic (the number of inputs grows proportionally to the number of virtual channels and the number of outputs to the total bisection bandwidth between adjacent routers). More significantly, using six identical networks allowed us to verify in isolation the operation of a *single* network, and then safely replicate it six times to form the interconnect, significantly reducing the total verification effort.

While six physical networks would provide enough bandwidth to the chip, it is still very important to minimize the latency because the network latency affects the performance of both migration and RA. To keep the hardware complexity low and achieve single cycle-per-hop delay, EM^2 routers use dimension order routing.

5.2.3 Tile Architecture

Figure 5-1 shows an EM^2 tile that consists of an 8KB instruction cache, a 32KB data cache, a processor core, a migration predictor, and six on-chip network routers [54]. The processor core contains two SMT contexts, one of them can be used only by its native thread. The core also has two hardware stacks, "main" and "auxiliary";

instructions follow a custom stack-machine ISA. The 32KB data cache serves not only memory instructions from the native and guest contexts at the same core, but also RA requests from distant cores. Finally, a hardware migration predictor [80] keeps track of memory access patterns of each thread and decides whether to make RA requests or migrates to the home core.

5.3 Design Goals, Constraints, and Methodology

5.3.1 Goals and Constraints

Scalability was the foremost concern throughout this project. Distributed directory cache coherence protocols (DCC) are not easily scalable to the number of cores [1, 5, 20, 44, 95]. EM^2 provides a simple but flexible solution that scales to the demands of the diverse set of programs running on manycore processors. To prove EM^2 scales beyond DCC, our major design objective was to build a massive-scale multicore processor with more than 100 cores.

The goal of 100 cores or more imposed a fundamental constraint for the project: tight area budget. EM^2 is a 10mm$\times$10mm chip in 45nm ASIC technology, fairly large for a research chip. Each tile, however, has only a small footprint for its process core, caches, and on-chip network router. Therefore, our design process focused on area efficiency, often at the expense of clock speed.

Maintaining a simple design was another important goal, because we planned to finish the entire 110-core chip design and implementation process (RTL, verification, physical design, tapeout) with only 18 man-months of effort. While the simplicity of directoryless memory substrate was the key to meet the tight schedule of the whole project, we also needed to make salient design choices to simplify design and verification.

Vying for simplicity brought an important implication for the I/O design for the chip. There are two common methods used to connect a chip to its packaging: wire bonding and flip chip. In general, wire bonding is widely used for the ICs with up to

600 I/O pins, while flip chip can provide better scalability and electrical advantages for larger designs [24]. We opted for the wire bonding method, because it simplifies the layout process of EM^2 significantly. In wire bonding, wires are attached only to the edges of the chip, so the tile layout need not include solder bumps that complicate the place and route (P&R) process (for the hierarchical design process of EM^2, see Section 5.3.2).

Using the wire bonding technique for EM^2 had a severe impact on its power budget. Wire bonding limits the total number of pins for large chips because the number of pins scales with the length of boundaries, not the area. The EM^2 chip has a total of 476 pins, where 168 pins are signal pins and 308 pins are power pins (for 154 power-ground pairs). The 308 power pins can supply a maximum of 13.286W[1] of power for the entire chip, which is quite low for this size of chip[2]. As will be shown in the following sections, the tight power budget affected the entire design and implementation process significantly.

5.3.2 Methodology

Bluespec [6] is a high-level hardware design language based on synthesizable guarded atomic actions [36]. In Bluespec, each distinct *operation* is described separately (as a "rule"), as opposed to VHDL or Verilog which describes each distinct *hardware elements*. In this way, implementation errors are localized to specific rules, which reduces the scope of each bug-fix and simplifies the verification process significantly. The Bluespec compiler automatically generates the necessary logic that controls how those rules are applied, and converts the design to synthesizable Verilog which can be used for the standard ASIC flow.

We used Synopsys Design Compiler to synthesize the RTL code into gate-level netlists. Despite the tight power budget, we were not able to rely on custom circuit design techniques to scale down the power, due to the limited resources. Instead, we

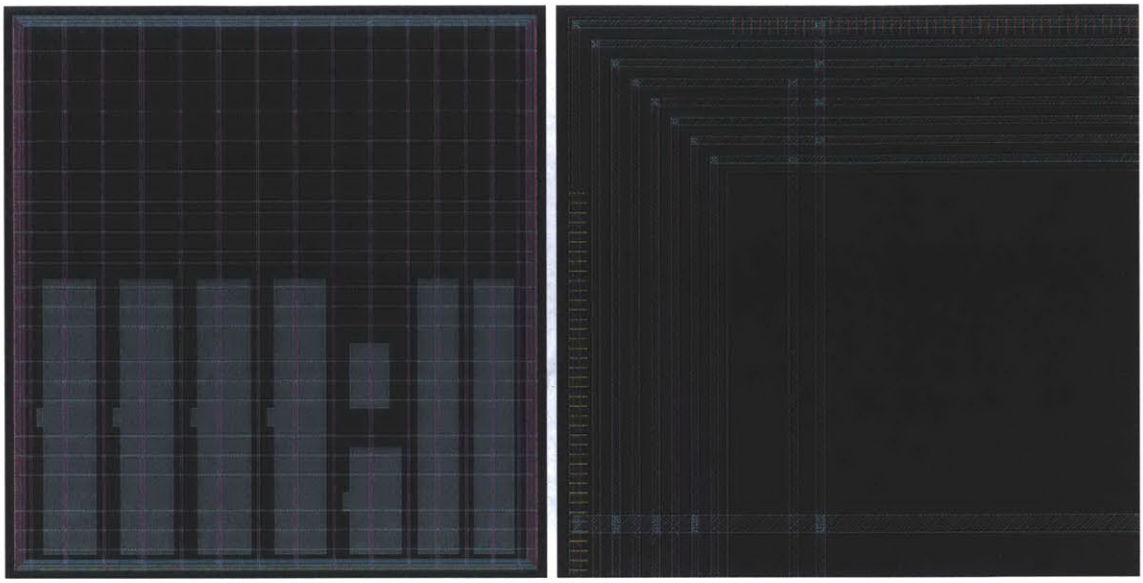
[1] from the maximum DC current constraints of the I/O library for reliable operation

[2] The actual power budget further decreases to 11.37W due to the power grid of the chip. See Section 5.4.2.

compromised performance for power efficiency in two ways. To save leakage power, we switched to the high-voltage threshold (HVT) standard cell library, which reduced the leakage power dissipation by half. To save dynamic power, on the other hand, we used Synopsys Power Compiler for automatic clock gating insertion. Although clock gating effectively lengthened the critical path, it resulted in 5.9× decrease in the average power[3].

Throughout the design process, hierarchical design and floorplanning was essential to exploit the benefit of homogeneous core design and verification. Every tile on the chip has the same RTL and the same layout, except only for the two memory controller (MC) cores which contain additional logic to communicate with external memory. The perfectly homogeneous tile design was duplicated to built an 11×10 array. To integrate as many cores as possible, we took a bottom-up approach; we first build a layout of single tile as compact as possible, and then instantiated the layout for the chip-level design.

[3]from the power reports by Synopsys Design Compiler



(a) (b)

Figure 5-2: EM^2 tile floorplan

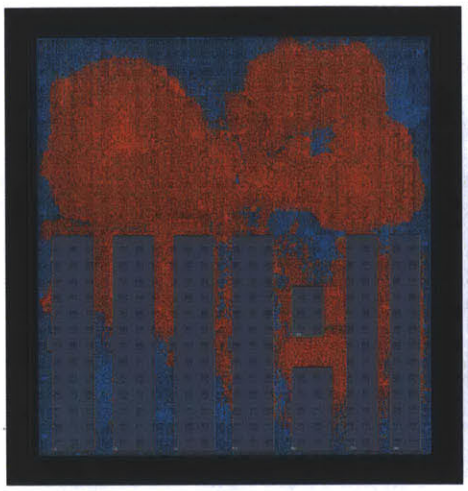
5.4 Physical Design of the 110-core EM^2 Processor

This section illustrates the physical design process of EM^2, highlighting the key engineering issues in manycore processor design. We used Cadence Encounter for the P&R of the complete design.

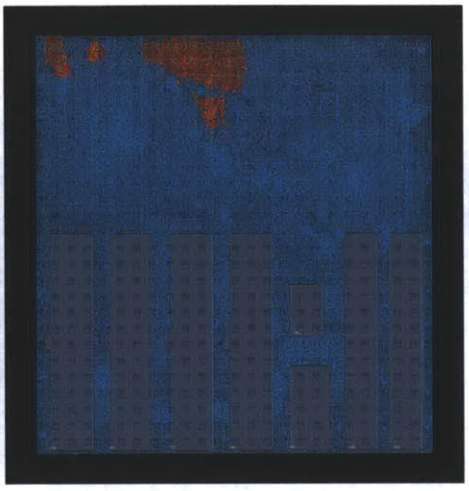
5.4.1 Tile-level Design

Figure 5-2(a) is the floorplan for the EM^2 tile, and Figure 5-2(a) magnifies the view near the upper left corner of the tile. They reveal that only the eight SRAM blocks are manually placed, and other blocks are automatically placed by Encounter. Because the tile is relatively small, all components are flattened to provide the most flexibility for the tool to optimize placement to the finest level.

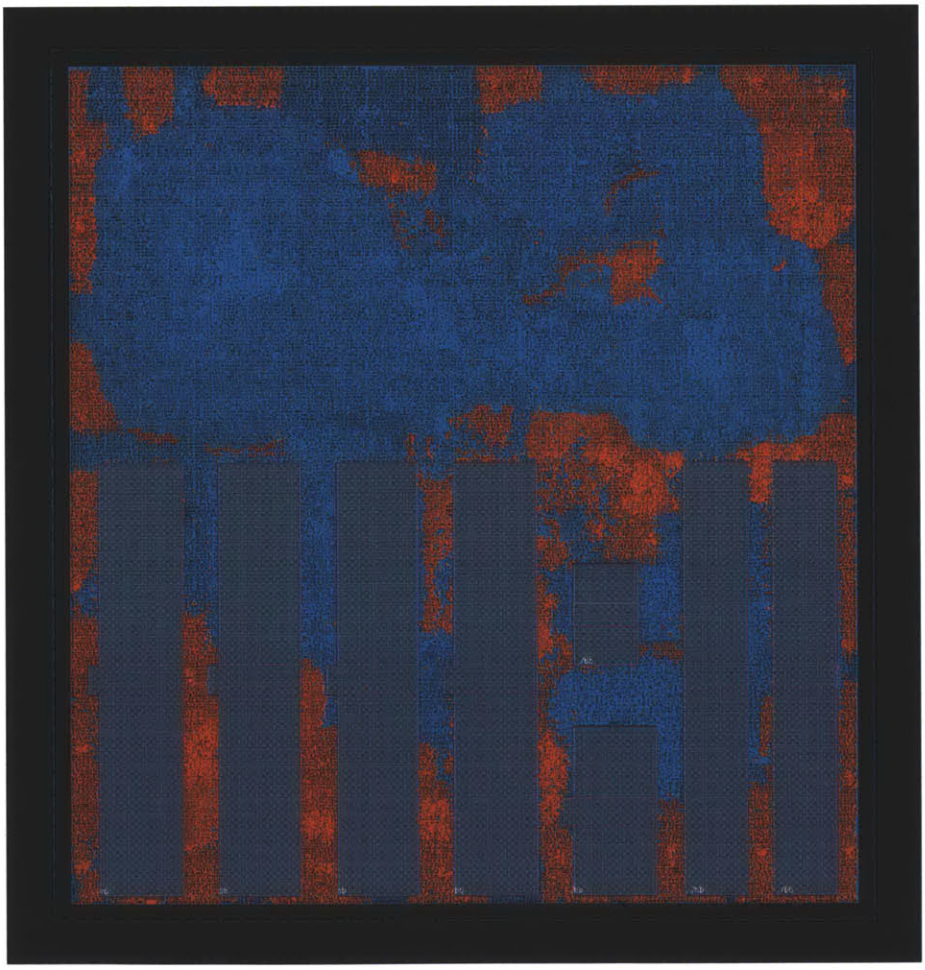
Additionally, Figure 5-2(a) shows the tile pins are manually aligned on the tile boundaries. Because we are following a bottom-up approach, Encounter does not have a chip-level view at this stage so it does not know where these tile pins will connect to. Therefore, these pins are manually placed along the edges in such a way



(a) Processor core



(b) Migration predictor



(c) On-chip network router

Figure 5-3: Placement of the tile components by Encounter

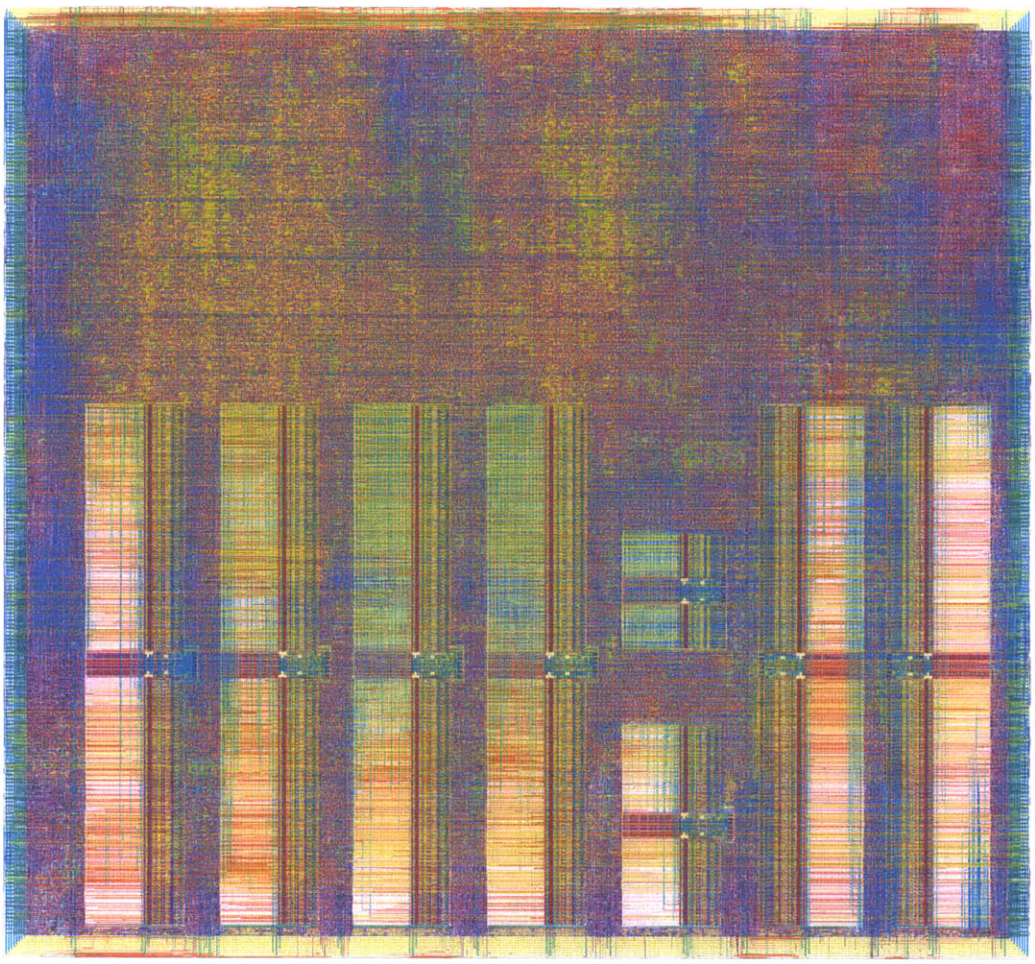


Figure 5-4: EM^2 tile layout

that once tiles are aligned into an array, the pins to be connected will be the closest to each other.

The floorplan view also reveals the power planning of the tile. There are total eight power rings, and horizontal and vertical power stripes are connected to the rings. The power nets are routed down to the lower metal layers and connected to standard cells or SRAM blocks only from the vertical stripes. In Figure 5-2(a), note that every SRAM block has vertical power stripes running over itself, so the power nets can be easily routed down to the SRAM blocks. Also, because the SRAM blocks intersect the horizontal power rails that supply power to the standard cells, a set of vertical power stripes are manually placed in every narrow space between two SRAM

blocks.

Figure 5-3 illustrates the actual placement of each tile component after the P&R process. It is most noticeable in Figure 5-3(c) that the tool was able to optimize the placement efficiently; it placed the ingress buffer of the router close to the tile boundaries, to reduce the wire length and leave a large space in the middle for the processor core logic. Finally, Figure 5-4 shows the final layout of the EM^2 tile design with the area of $0.784mm^2$ ($855\mu m \times 917\mu m$).

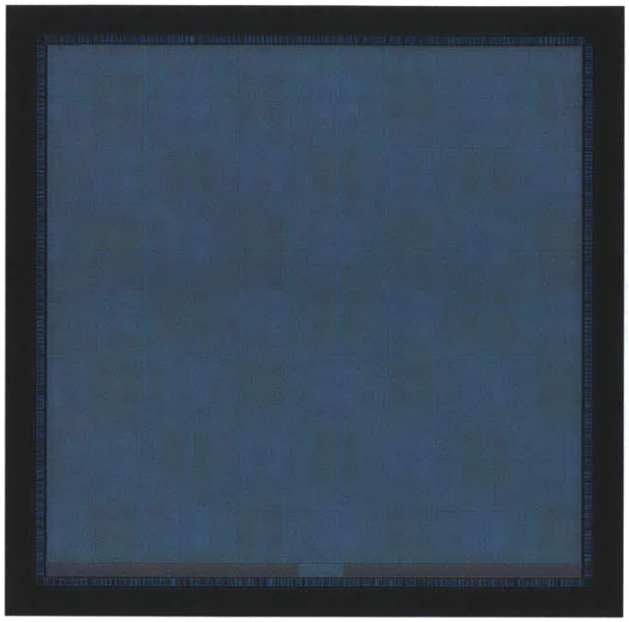


Figure 5-5: EM^2 chip-level floorplan

5.4.2 Chip-level Design

5.4.2.1 Chip-level Floorplanning

Figure 5-5 is the floorplan for the chip-level design. First, the tile layout from Section 5.4.1 is duplicated into an 11×10 array. The small rectangle below the tile array is the clock module, which selects one among the three available clock sources: two external clock sources (single-ended and differential) and one from the PLL output. This module is custom designed except for the PLL block (Figure 5-6).

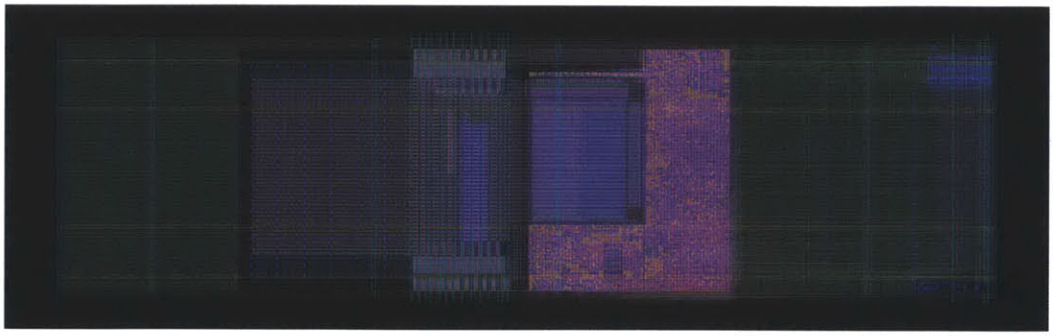
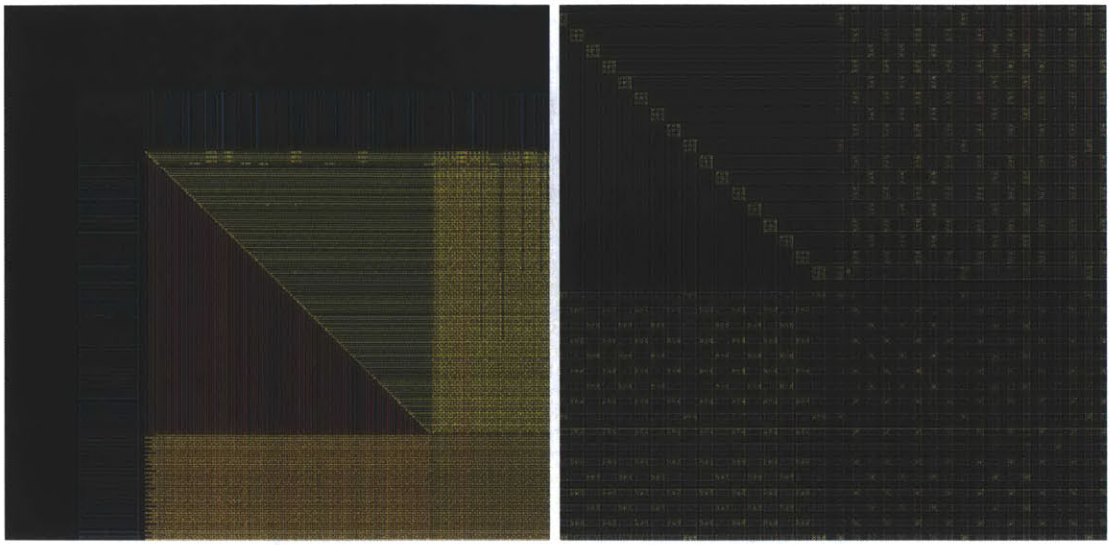


Figure 5-6: EM^2 clock module



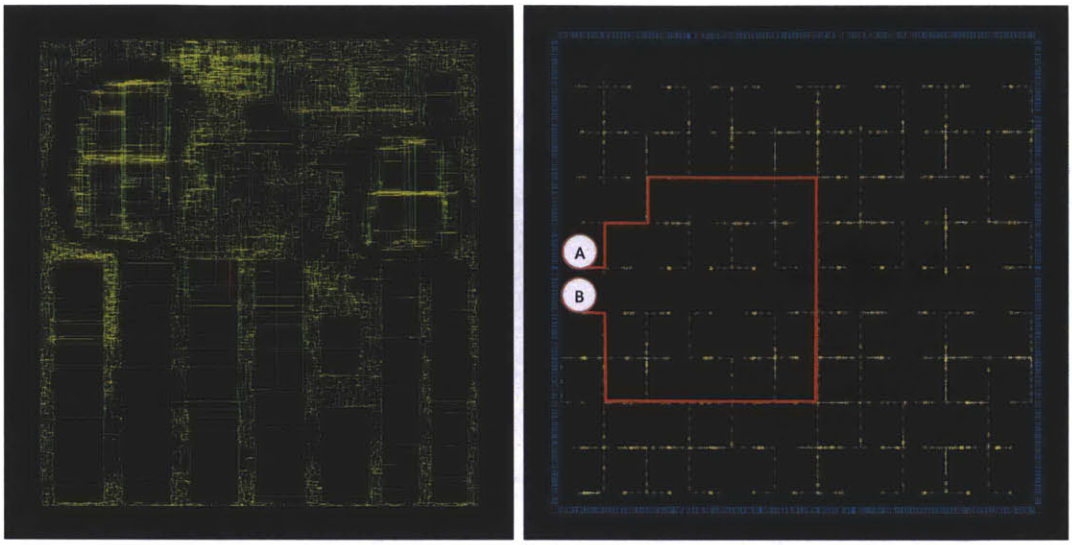
(a) A corner of global power rings (b) Magnified view on the power grid

Figure 5-7: EM^2 global power planning

As mentioned in Section 5.3.1, the EM^2 chip uses the wire bonding technique; the I/O ring outside of the tile array has 476 bonding pads to connect to a package. While wire bonding simplifies the design process, the tiles in the middle are placed too far away from the power pads, exacerbating IR drop issues. Therefore, we took a very conservative approach to global power planning. In order to use as many as wires as possible, the top two metal layers in the design are dedicated for the global power rings and stripes. Figure 5-7 shows the upper left part of the chip, revealing part of the 122 power rings and the dense power grid. All power stripes, both horizontal and vertical, have a width of $2\mu m$ and a pitch of $5\mu m$, covering 64% of the area. The power rings are even denser at $4\mu m$ width and $5.2\mu m$ pitch.

5.4.2.2 Building a Tile Array

Although EM^2 tiles are perfectly homogeneous, it is not trivial to integrate them to a tiled array. The foremost concern is clock skew, which is illustrated in Figure 5-8(a). Suppose that the output of a flip-flop FF_A in tile A is driving the input of FF_B in tile B. Even though FF_A and FF_B are next to each other, tile A and tile B are very distant nodes on the global clock tree, so there could be a significant clock skew



(a) Tile-level (b) Chip-level

Figure 5-8: Clock tree synthesized by Encounter

between FF_A and FF_B. If clock edges arrive at FF_B sooner than FF_A, the output of FF_A begins to change later and FF_B samples the input earlier, so it becomes more difficult to avoid setup-time violations. If clock edges arrive at FF_A earlier, on the other hand, the output of FF_A can change to a new value even before FF_B samples the current value, so it is possible to violate hold-time constraints. The latter case is a more serious problem because while we can eliminate setup-time violations by lowering operating frequency, there is no way to fix hold-time violations after tape-out. In order to fix this problem, a negative-edge flip-flop is inserted between FF_A and FF_B; even if FF_A changes its output before FF_B samples the current value, the output of the negative-edge flip-flop does not change until half cycle later, so hold-time violations can be avoided.

5.4.2.3 Final EM^2 Layout

Figure 5-9 shows the taped out layout of the entire EM^2 chip. The chip area is $10mm \times 10mm$, and the static timing analysis with extracted RC parasitics estimates that the chip works at 105MHz, dissipating $50mW$ at each tile. Note that a number of decoupling capacitors are manually inserted around the clock module.

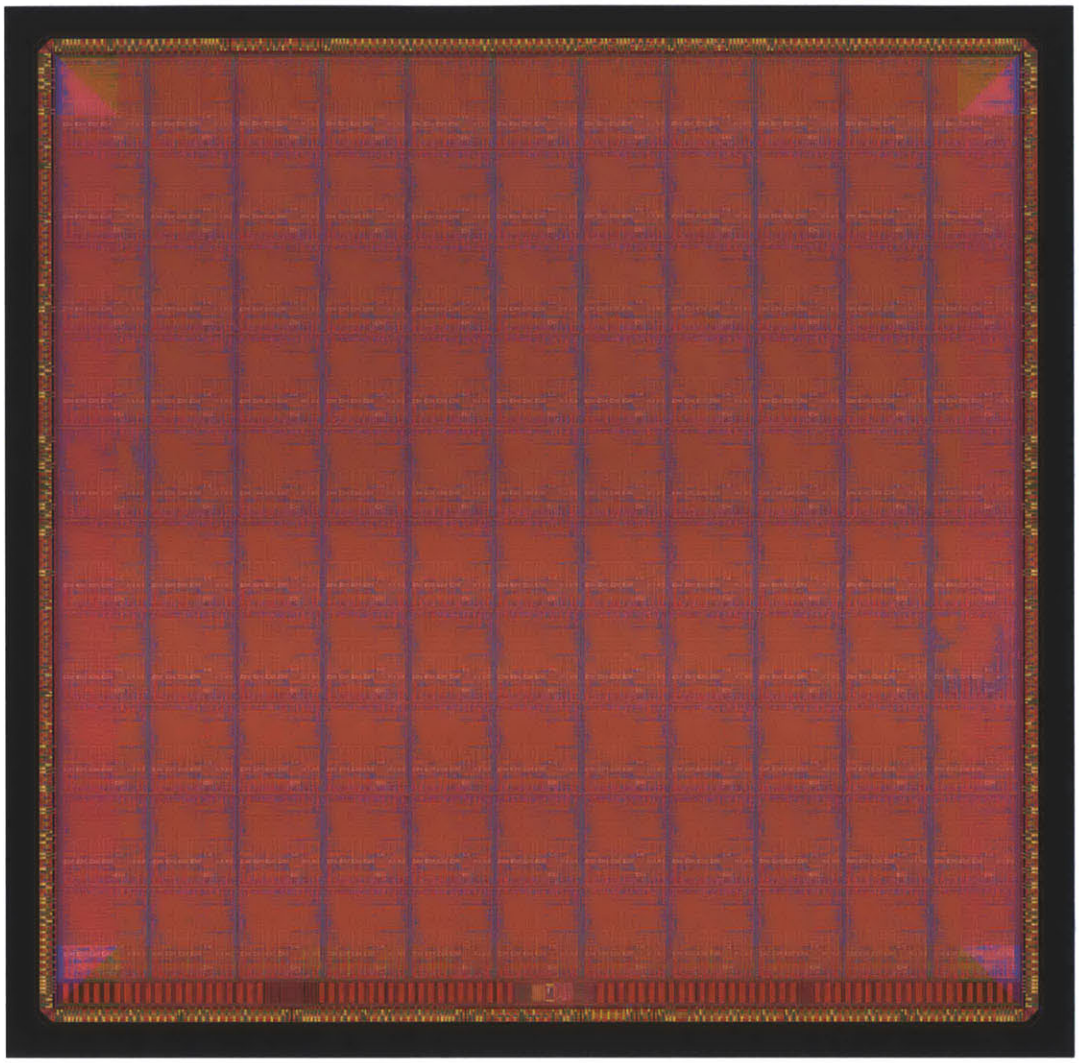
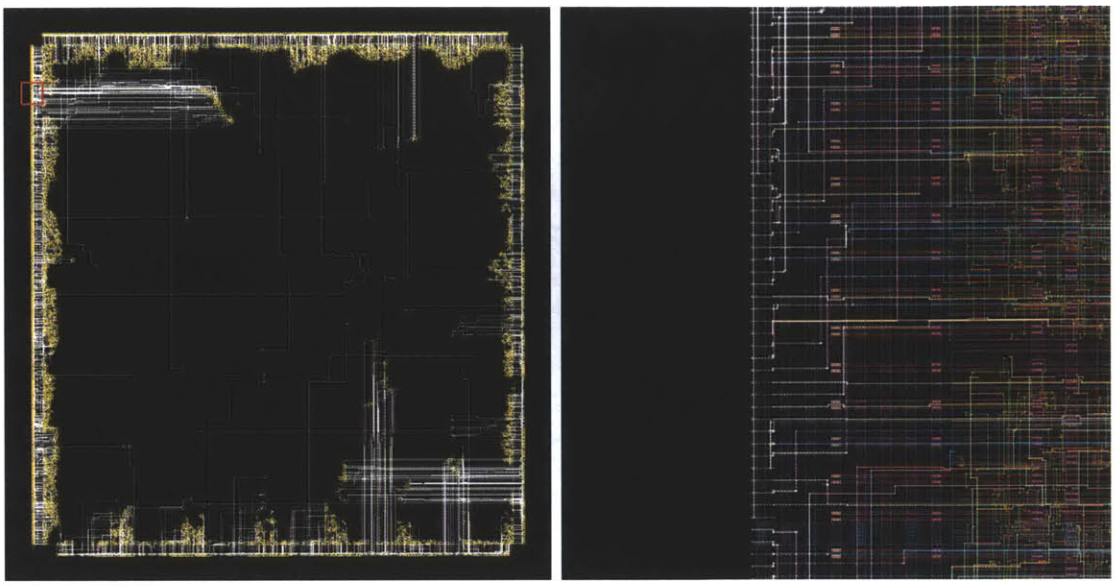


Figure 5-9: Tapeout-ready EM^2 processor layout

5.5 Design Iteration Using BAN on EM^2

From the final layout of the chip, we noticed a severe wire routing complexity for the router pins (Figure 5-10). Adding more router pins results in more design rule violations that are not easy to fix. Therefore, it is not straightforward to further increase the on-chip network bandwidth. Is the current total bandwidth sufficient to meet the needs of various applications? If not, how can we reduce the network congestion and improve the network performance degradation without adding more router pins?



(a) Tile-level view (b) Inside the red box, magnified

Figure 5-10: Wire connected to input and output network ports

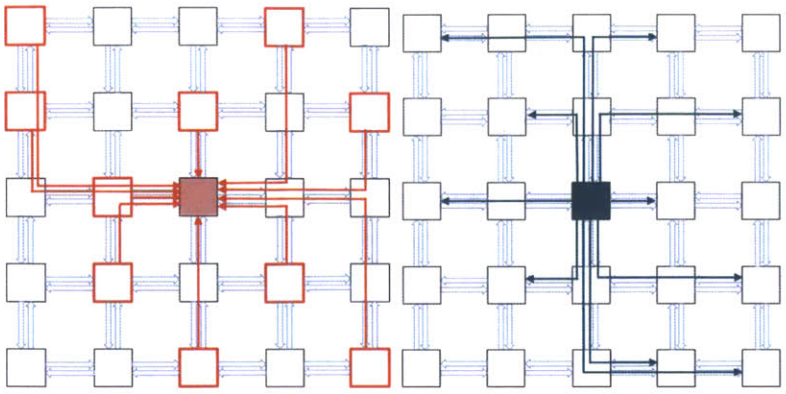
To evaluate how much network bandwidth applications need, we ran five different applications in migration-only mode[4] on a 64-core version of EM^2 using the Hornet simulator [73]. To assess migration patterns of applications, we defined the *peak* and *average concentrations* as follows:

Definition 1 *Application running time is divided into a set of 1000 time windows,* $W = \{W_1, W_2, \ldots, W_{1000}\}$. *There is also a set of cores* $C = \{C_1, C_2, \ldots, C_{64}\}$. *The destination popularity function* $P(c, w)$ *is defined as the number of threads that visit core* C_c *at least once in time window* W_n.

[4]Threads use only migration, not RA, to access remote cache.

	Barnes	**LU -contiguous**	**Ocean -contiguous**	**Radix**	**Water-n^2**
Peak concentration	5	18	15	64	5
Average concentration	2.2	1.6	6.8	4.1	2.1

Table 5.1: Peak and average migration concentration in different applications



(a) Concentrate-in (b) Concentrate-out

Figure 5-11: Migration traffic concentration

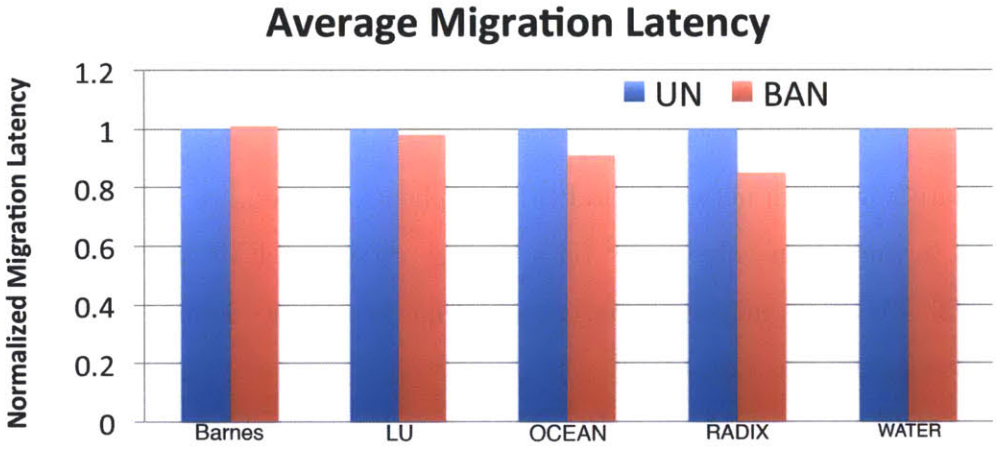


Figure 5-12: Average migration latency on BAN+EM^2

Definition 2 *Concentration* $F(w) = \underset{c=1...}{Max}\{P(c,w)\}$

Definition 3 *Peak concentration* $= \underset{w=1...1000}{Max}\{F(w)\}$

Definition 4 *Average concentration* $= \underset{w=1...1000}{Avg}\{F(w)\}$

Table 5.1 reveals that applications such as ocean-contiguous or radix have a very high degree of concentration. Note that an incoming thread always evicts a currently running thread to its native core. This evicted thread is likely to come back and compete for the same core again to access data it needs. Therefore, high-level concentration may cause severe ping-pong effects, and burden the network with a lot

of migration traffic. If we cannot simply increase total link bandwidth, how can we optimize the network to deal with the high bandwidth demand of highly concentrated applications?

In Figure 5-11 a lot of threads make frequent visits to the core in the middle. Because EM^2 uses DOR-YX routing, the migration traffic *to* the middle core is jammed more on the horizontal links (Figure 5-11(a)). When threads are moving out of the core, on the other hand, the vertical links get more congested as shown in Figure 5-11(b). As explained in Section 3.4, this is a perfect opportunity for BAN to take advantage of asymmetric network patterns. We applied BAN to the migration network of EM^2 and performed a simulation study. Figure 5-12 illustrates that without increasing total link bandwidth, BAN can improve the network performance for applications with high-level concentration by up to 16%.

Chapter 6

Conclusions

6.1 Thesis Contributions

This thesis makes the following contributions:

- A new path-diverse oblivious routing algorithm with both flexibility and simplicity,
- A novel adaptive network that uses oblivious routing and bidirectional links,
- The first deadlock free, fine-grained autonomous thread migration protocol,
- An extension of existing on-chip network simulation to flexible manycore system simulation, and
- An example on-chip network implementation from RTL to silicon.

6.2 Summary and Suggestions

In this thesis, we have taken a wide range of approaches to optimize on-chip network designs for manycore architectures. First, we have introduced PROM and BAN, optimization focused on throughput improvement. These techniques both improve the performance and maintain design simplicity as low complexity implementation is paramount in on-chip network design. Second, we have presented ENC, the first,

deadlock-free, fine-grained thread migration protocol. This research not only solves the specific problem of efficient cycle-level thread migration, but also encourages a design paradigm that relaxes the conventional abstraction and uses the resources of the network to support higher-level functionality. Finally, we have undertaken the arduous task of implementing a 110-core EM^2 processor on silicon. This effort has helped to address realistic implementation constraints and provided perspective for future research.

An important lesson that can be learned from this thesis is that, an on-chip network is not just about making physical connections between system components. The benefits and constraints that each component brings to the system are consolidated into a complex global design space by an on-chip network. Therefore, the on-chip network must take a role as an arbiter and tightly integrate the components to meet the design goals. For example, ENC is designed to take the burden of deadlock prevention off processor cores; shipping the context of a running thread out of the pipeline is an essential operation to solve the deadlock issue because it forces progress. In order to not lose the thread context, however, the evicted thread must be stored in another place immediately. And because the registers in processor cores are tightly utilized, it is better to store the context in a relatively underutilized resource – the network buffer. Therefore, ENC puts only the *minimum* amount of additional buffer in processor cores (for just one thread context at its native core), and lets the thread context utilize the ample network buffer until it arrives at its native core. This is a example of resource arbitration between system components, which is efficiently handled by the on-chip network. Future on-chip network design for manycore architecture must take this role into account and take charge in orchestrating all system components.

Bibliography

[1] Arvind, Nirav Dave, and Michael Katelman. Getting formal verification into design flow. In *FM2008*, 2008.

[2] Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with Pin. *Computer*, 43:34 41, 2010.

[3] H.G. Badr and S. Podar. An Optimal Shortest-Path Routing Policy for Network Computers with Regular Mesh-Connected Topologies. *IEEE Transactions on Computers*, 38(10):1362–1371, 1989.

[4] Arnab Banerjee and Simon Moore. Flow-Aware Allocation for On-Chip Networks. In *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 183–192, May 2009.

[5] Jesse G. Beu, Michael C. Rosier, and Thomas M. Conte. Manager-client pairing: a framework for implementing coherence hierarchies. In *MICRO*, 2011.

[6] Bluespec, Inc. *Bluespec SystemVerilog™ Reference Guide*, 2011.

[7] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2):105–128, 2004.

[8] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas B. Jablin, and David I. August. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, 28(1):12–20, 2008.

[9] Ge-Ming Chiu. The Odd-Even Turn Model for Adaptive Routing. *IEEE Trans. Parallel Distrib. Syst.*, 11(7):729–738, 2000.

[10] Myong Hyon Cho, Mieszko Lis, Keun Sup Shim, Michel Kinsy, and Srinivas Devadas. Path-based, randomized, oblivious, minimal routing. In *In Proceedings of the 2nd International Workshop on Network on Chip Architectures*, pages 23–28, December 2009.

[11] Myong Hyon Cho, Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. Deadlock-free fine-grained thread migration. In *NOCS*, 2011.

[12] Intel Corporation. Intel delivers new architecture for discovery with intel xeon phi coprocessors. Press release, November 2012.

[13] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Kepler GK110. Whitepaper, April 2012.

[14] Tilera Corporation. Tilera's tile-gx72 processor sets world record for suricata ips/ids: Industry's highest performance. Press release, July 2013.

[15] A Correia, M Pérez, JJ Sáenz, and PA Serena. Nanotechnology applications: a driving force for R&D investment. *Physica Status Solidi (a)*, 204(6):1611–1622, 2007.

[16] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 04(4):466 -475, 1993.

[17] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Computers*, 36(5):547–553, 1987.

[18] William J. Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[19] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, October 1974.

[20] A. DeOrio, A. Bauserman, and V. Bertacco. Post-silicon verification for cache coherence. In *ICCD*, 2008.

[21] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core Opteron processor. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 102–103, 2007.

[22] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-network cache coherence. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 321–332. IEEE, 2006.

[23] Noel Eisley, Li-Shiuan Peh, and Li Shang. Leveraging on-chip networks for data cache migration in chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 197–207, 2008.

[24] Peter Elenius and Lee Levine. Comparing flip-chip and wire-bond interconnection technologies. *Chip Scale Review*, pages 81–87, July/August 2000.

[25] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, 2011.

[26] Wu-Chang Feng and Kang G. Shin. Impact of Selection Functions on Routing Algorithm Performance in Multicomputer Networks. In *In Proc. of the Int. Conf. on Supercomputing*, pages 132–139, 1997.

[27] Christian Fensch and Marcelo Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 355 366. IEEE, 2008.

[28] International Technology Roadmap for Semiconductors. `http://www.itrs.net/Links/2012ITRS/2012Tables/ORTC_2012Tables.xlsm`, 2012.

[29] Samuel H. Fuller and Lynette I. Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, January 2011.

[30] Andre K Geim and Konstantin S Novoselov. The rise of graphene. *Nature Materials*, 6(3):183–191, 2007.

[31] Christopher J. Glass and Lionel M. Ni. The turn model for adaptive routing. *J. ACM*, 41(5):874 902, 1994.

[32] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *Design & Test of Computers, IEEE*, 22(5):414–421, 2005.

[33] P. Gratz, B. Grot, and S. W. Keckler. Regional Congestion Awareness for Load Balance in Networks-on-Chip. In *In Proc. of the 14th Int. Symp. on High-Performance Computer Architecture (HPCA)*, pages 203–214, February 2008.

[34] Boris Grot, Stephen W Keckler, and Onur Mutlu. Preemptive virtual clock: a flexible, efficient, and cost-effective qos scheme for networks-on-chip. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–279. ACM, 2009.

[35] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[36] James C. Hoe and Arvind. Scheduling and Synthesis of Operation-Centric Hardware Descriptions. In *ICCAD*, 2000.

[37] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz Mesh Interconnect for a TeraFLOPS Processor. *IEEE Micro*, 27(5):51–61, Sept/Oct 2007.

[38] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, 2010.

[39] Jingcao Hu and Radu Marculescu. Application Specific Buffer Space Allocation for NetworksonChip Router Design. In *Proc. IEEE/ACM Intl. Conf. on Computer Aided Design*, San Jose, CA, November 2004.

[40] Jingcao Hu and Radu Marculescu. DyAD: Smart Routing for Networks on Chip. In *Design Automation Conference*, June 2004.

[41] Wei Hu, Xingsheng Tang, Bin Xie, Tianzhou Chen, and Dazhou Wang. An efficient power-aware optimization for task scheduling on noc-based many-core system. In *Proceedings of CIT 2010*, pages 172–179, 2010.

[42] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., 1st edition, 2013.

[43] Natalie Enright Jerger and Li-Shiuan Peh. *On-Chip Networks*. Morgan and Claypool Publishers, 1st edition, 2009.

[44] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. Checking Cache-Coherence Protocols with TLA+. *Formal Methods in System Design*, 22:125–131, 2003.

[45] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos. Variable packet size buffered crossbar (CICQ) switches. In *2004 IEEE International Conference on Communications*, volume 2, pages 1090–1096, June 2004.

[46] Omer Khan, Mieszko Lis, and Srinivas Devadas. EM^2: A Scalable Shared Memory Multi-Core Achitecture. In *CSAIL Technical Report MIT-CSAIL-TR-2010-030*, 2010.

[47] Hasina Khatoon, Shahid Hafeez Mirza, and Talat Altaf. Exploiting the role of hardware prefetchers in multicore processors. *International Journal of Advanced Computer Science and Applications(IJACSA)*, 4(6), 2013.

[48] H. J. Kim, D. Park, T. Theocharides, C. Das, and V. Narayanan. A Low Latency Router Supporting Adaptivity for On-Chip Interconnects. In *Proceedings of Design Automation Conference*, pages 559–564, June 2005.

[49] Michel Kinsy, Myong Hyon Cho, Tina Wen, Edward Suh, Marten van Dijk, and Srinivas Devadas. Application-Aware Deadlock-Free Oblivious Routing. In *Proc. 36th Int'l Symposium on Computer Architecture*, pages 208–219, June 2009.

[50] Jae W. Lee, Man Cheuk Ng, and Krste Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 89–100, 2008.

[51] Kangmin Lee, Se-Joong Lee, and Hoi-Jun Yoo. SILENT: serialized low energy transmission coding for on-chip interconnection networks. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 448–451. IEEE Computer Society, 2004.

[52] Y-M Lin, Christos Dimitrakopoulos, Keith A Jenkins, Damon B Farmer, H-Y Chiu, Alfred Grill, and Ph Avouris. 100-GHz transistors from wafer-scale epitaxial graphene. *Science*, 327(5966):662–662, 2010.

[53] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas. Guaranteed in-order packet delivery using Exclusive Dynamic Virtual Channel Allocation. Technical Report CSAIL-TR-2009-036 (http://hdl.handle.net/1721.1/46353), Massachusetts Institute of Technology, August 2009.

[54] Mieszko Lis, Keun Sup Shim, Brandon Cho, Ilia Lebedev, and Srinivas Devadas. Hardware-level thread migration in a 110-core shared-memory processor. In *HotChips*, 2013.

[55] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Omer Khan, and Srinivas Devadas. Scalable directoryless shared memory coherence using execution migration. In *CSAIL Technical Report MIT-CSAIL-TR-2010-053*, 2010.

[56] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Omer Khan, and Srinivas Devadas. Directoryless Shared Memory Coherence using Execution Migration. In *PDCS*, 2011.

[57] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Pengju Ren, Omer Khan, and Srinivas Devadas. DARSIM: a parallel cycle-level NoC simulator. In *Proceedings of MoBS-6*, 2010.

[58] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *IPPS*, 1995.

[59] Steve Melvin, Mario Nemirovsky, Enric Musoll, and Jeff Huynh. A massively multithreaded packet processor. In *Proceedings of NP2: Workshop on Network Processors*, 2003.

[60] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 890–895. IEEE, 2004.

[61] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of HPCA 2010*, pages 1–12, 2010.

[62] Matthew Misler and Natalie Enright Jerger. Moths: Mobile threads for on-chip networks. In *Proceedings of PACT 2010*, pages 541–542, 2010.

[63] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[64] Robert D. Mullins, Andrew F. West, and Simon W. Moore. Low-latency virtual-channel routers for on-chip networks. In *Proc. of the 31st Annual Intl. Symp. on Computer Architecture (ISCA)*, pages 188–197, 2004.

[65] Ted Nesson and S. Lennart Johnsson. ROMM Routing: A Class of Efficient Minimal Routing Algorithms. In *in Proc. Parallel Computer Routing and Communication Workshop*, pages 185–199, 1994.

[66] Ted Nesson and S. Lennart Johnsson. ROMM routing on mesh and torus networks. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA '95*, pages 275–287, 1995.

[67] George Nychis, Chris Fallin, Thomas Moscibroda, and Onur Mutlu. Next generation on-chip networks: what kind of congestion control do we need? In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 12:1–12:6, 2010.

[68] Cheolmin Park, Roy Badeau, Larry Biro, Jonathan Chang, Tejpal Singh, Jim Vash, Bo Wang, and Tom Wang. A 1.2 TB/s on-chip ring interconnect for 45nm 8-core enterprise Xeon® processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 180–181, 2010.

[69] Sunghyun Park, Masood Qazi, Li-Shiuan Peh, and Anantha P Chandrakasan. 40.4 fJ/bit/mm low-swing on-chip signaling with self-resetting logic repeaters embedded within a mesh NoC in 45nm SOI CMOS. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1637–1642. EDA Consortium, 2013.

[70] Li-Shiuan Peh and William J. Dally. A Delay Model and Speculative Architecture for Pipelined Routers. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, pages 255–266, January 2001.

[71] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of ISCA 2009*, pages 93–104, 2009.

[72] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of ISCA 2009*, pages 302–313, 2009.

[73] Pengju Ren, Mieszko Lis, Myong Hyon Cho, Keun Sup Shim, Christopher W Fletcher, Omer Khan, Nanning Zheng, and Srinivas Devadas. Hornet: A cycle-level multicore simulator. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(6):890–903, 2012.

[74] R.J. Riedlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, and T. Grutkowski. A 32nm 3.1 billion transistor 12-wide-issue Itanium® processor for mission-critical servers. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 84–86, 2011.

[75] S. Rusu, Simon Tam, H. Muljono, D. Ayers, Jonathan Chang, B. Cherkauer, J. Stinson, J. Benoit, R. Varada, Justin Leung, R.D. Limaye, and S. Vora. A 65-nm dual-core multithreaded Xeon® processor with 16-MB L3 cache. *Solid-State Circuits, IEEE Journal of*, 42(1):17–25, 2007.

[76] Daeho Seo, Akif Ali, Won-Tack Lim, Nauman Rafique, and Mithuna Thottethodi. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *Proc. of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 432–443, 2005.

[77] Ohad Shacham. *Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms.* PhD thesis, Stanford University, May 2011.

[78] Kelly A. Shaw and William J. Dally. Migration in single chip multiprocessor. In *Computer Architecture Letters*, pages 12–12, 2002.

[79] K. S. Shim, M. H. Cho, M. Kinsy, T. Wen, M. Lis, G. E. Suh, and S. Devadas. Static Virtual Channel Allocation in Oblivious Routing. In *Proceedings of the* 3^{rd} *ACM/IEEE International Symposium on Networks-on-Chip*, pages 253–264, May 2009.

[80] Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. Thread migration prediction for distributed shared caches. *Computer Architecture Letters*, Sep 2012.

[81] Arjun Singh, William J. Dally, Amit K. Gupta, and Brian Towles. GOAL: a load-balanced adaptive routing algorithm for torus networks. *SIGARCH Comput. Archit. News*, 31(2):194–205, 2003.

[82] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The impact of technology scaling on lifetime reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186, 2004.

[83] Karin Strauss. *Cache Coherence in Embedded-ring Multiprocessors.* PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[84] Leonel Tedesco, Fabien Clermidy, and Fernando Moraes. A path-load based adaptive routing algorithm for networks-on-chip. In *SBCCI '09: Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design*, pages 1–6, New York, NY, USA, 2009. ACM.

[85] Brian Towles and William J. Dally. Worst-case traffic for oblivious routing functions. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 1–8, 2002.

[86] Brian Towles, William J. Dally, and Stephen Boyd. Throughput-centric routing algorithm design. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 200–209, 2003.

[87] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proc. 13th annual ACM symposium on Theory of Computing STOC'81*, pages 263 277, 1981.

[88] Sriram Vangal, Nitin Borkar, and Atila Alvandpour. A six-port 57gb/s double-pumped nonblocking router core. In *VLSI Circuits, 2005. Digest of Technical Papers. 2005 Symposium on*, pages 268–269. IEEE, 2005.

[89] Sriram R Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, et al. An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, 2008.

[90] Boris Weissman, Benedict Gomes, Jürgen W. Quittek, and Michael Holtkamp. Efficient fine-grain thread migration with active threads. In *Proceedings of IPPS/SPDP 1998*, pages 410–414, 1998.

[91] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown, and Anant Agarwal. On-chip interconnection architecture of the Tile processor. *Micro, IEEE*, 27(5):15–31, 2007.

[92] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[93] T. Wu, C. Y. Tsui, and M. Hamdi. CMOS Crossbar. In *Proceedings of the* 14^{th} *IEEE Symposium on High Performance Chips (Hot-Chips 2002)*, August 2002.

[94] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[95] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *MICRO*, 2010.